平成 25 年度 修士論文

DS-CUDA によるコンシューマ向け GPU を使用した GPGPU の信頼性向上システム

電気通信大学大学院　情報理工学研究科
情報・通信工学専攻　成見研究室
1231102　吉川 和幸
指導教員　成見 哲教授
副指導教員　沼尾 雅之教授

平成 26 年 1 月 27 日

概 要

本研究では，信頼性の面で問題があるコンシューマー向け GPU（グラフィックカード）を使用しても，見かけ上 GPGPU（GPU を用いた汎用計算）の信頼性を向上させることができるシステムを開発した．

コンシューマー向け GPU はまれに計算ミスをするということが知られており，通常は GPGPU 用途にはメモリエラーの検出・訂正機能があるメモリモジュールを搭載した GPGPU 専用モデルの GPU が使用される．しかしコンシューマー向け GPU は GPGPU 専用の GPU に比べると安価なため，GPGPU 向けに全く使用されないわけではない．例えば研究用途では，価格性能比に優れる点を活かして開発された GPU クラスタに使用されている．また GPGPU に対応した実用ソフトウェアも登場してきているため，学術用途以外で GPGPU に使用されることが増えると予想される．こういったグラフィックス用途以外でコンシューマー向け GPU を使用すると計算ミスの発生が大きな問題となる．そこで本研究では，信頼性と可用性の 2 つの視点からコンシューマー向け GPU を GPGPU に使用する際の問題の解決を目指した．

今回は GPU 仮想化ソフトウェアである DS-CUDA を使用して，計算ミスの検出・訂正を行う冗長計算機能と，GPU で実行中の計算を他の GPU に移行して計算を続行するマイグレーション機能を実装して評価した．また，既存の GPGPU アプリケーションを変更することなくシステムを利用できるようなインターフェイスにすることで，ユーザーがシステムの存在を意識することなく利用できることを目指した．

両機能ともまだ不完全な状態ではあるが，コンシューマー向け GPU の信頼性向上について一定の効果があることが確認され，将来的にコンシューマー向け GPU を GPGPU に問題なく利用できるようになる可能性を示した．

目次

1 はじめに

1.1 背景

1.1.1 GPGPU 技術の広まり

近年，元々は画面描画や動画再生を担当する PC パーツである GPU が持つ高い演算能力を，画像処理以外の汎用的な目的で使用する技術である GPGPU（General-Purpose computing on Graphic Processing Units）が広がりを見せている．

1 つのコアで様々な処理を行う CPU に対し，GPU はシェーディング（陰影処理）に特化した単純なプロセッサが多く集まった構造をしており，大量のデータに対する単純な演算を高速かつ並列に行う能力に優れる．この特徴を活かし，物理シミュレーション等の大規模な科学技術計算に代表される HPC（High Performance Computing）分野に用いられることが多い．また近年では CUDA[1] や ATI Stream[2]，OpenCL[3] といった GPGPU 開発環境の登場によって，動画エンコードソフトや画像処理ソフトにおいて GPGPU 技術を搭載したものが次々と登場している．多くの一般 PC ユーザーにとって GPU の用途は 3D ゲームや動画再生などに限られおり，GPU の計算資源は余りがちであるため，実用ソフトウェアの GPGPU 対応によって今後コンシューマーでの GPGPU 需要の増加が見込まれる．

1.1.2 GPU の計算信頼性

通常，GPGPU を行う際には GPGPU 専用モデルの GPU が使用される．GPGPU 専用モデルにはそもそも映像出力端子が存在せず，完全に GPGPU に用途が限定される．メモリエラーの検出・訂正機構や，コンシューマー向け製品よりも高い倍精度演算性能を持つなど高性能計算に特化した設計となっているが，コンシューマー向け GPU よりも非常に高価なものとなっている．

一方でコンシューマー向け GPU はグラフィック用途向けに設計されており，計算ミスの検出や訂正などの信頼性については考慮されていない．しかしその分価格は抑えられており，学術分野においても価格性能比に優れるコンシューマー向け GPU を使用した研究が存在する．

長崎大学の濱田らは，コンシューマー向け GPU360 台を使用した GPU クラスタを構築し，N 体問題および乱流解析を解く粒子法のアルゴリズム 2 つを同クラスタ上で実行するプログラムを開発した [4]．安価なコンシューマー向け GPU の使用により 1 ドルあた

り 124MFlops と非常に良好なコストパフォーマンスを発揮した．システム全体の価格も約 3800 万円と，それまでのスーパーコンピューターと比べ驚異的なほどの低価格であるにもかかわらず，最大 42TFlops という高性能であった．しかし，GPGPU 専用 GPU のようなメモリエラーの検出・訂正機能を持たないコンシューマー向け GPU を大量に使用したことでクラスタ全体ではエラーの発生確率がかなり高くなっており，計算を一週間続けて計測したところ多くの GPU で計算ミスが発生した．元々用意した GPU は 421 台であったが，この内 72 台の GPU で何らかの障害が発生し，実際にクラスタに使用したのは 380 台であった．

スタンフォード大学を中心に 2000 年から行われている分散コンピューティングプロジェクト Folding@home[5] では，様々な疾病の治療に役立つとされるタンパク質の折りたたみ構造（フォールディング：folding）の解析を行うプログラムを作成した．このプログラムをユーザーがダウンロードして実行し，結果をスタンフォード大学に送信することでプロジェクトへの参加が可能となっている．プログラムは CPU での実行だけでなく，GPGPU による処理も可能である．Haque らは同プロジェクト中で使用された GPU の情報を解析し，GPU のエラーレートについて報告した [6]．それによると，多くの GPU は Folding@home のプログラムの処理を 50000 回程度反復すると計算ミスが発生する確率が高かったという．またわずかではあるが，10000 回程度の反復で計算ミスが発生する確率が高い GPU もあったという．Haque らは原因として GPU コアのオーバークロックを挙げており，オーバークロックなどの不安定な条件下にある GPU では計算ミスの発生確率が増加すると言える．実際に，GPGPU 専用の GPU は同じコアを使用したコンシューマー向け GPU に比べ，動作周波数が低く設定されているものが多く，GPU コンピューティングはできるだけ安定な条件で行うことを推奨しているものと思われる．

1.2 目的

本研究では，コンシューマー向け GPU を GPGPU に使用した際に問題となる計算ミスの発生を自動的に検出・訂正するシステムを開発し，コンシューマー向け GPU による GPGPU の計算信頼性の向上を目指す．また大規模なシミュレーションなどでは計算が数日〜数週間にも及ぶことも少なくない．計算中にオペレーティングシステムのクラッシュや GPU の故障などが発生すると，初めから計算をやり直さなくてはならない．そこで開発するシステムには，これらの問題が発生した際にも計算を続行可能とする機能を搭載し，可用性の向上を目指す．また後述する先行研究においては，既存のプログラムに変更を加えなければ利用できないものが多い．そこで本研究では，ユーザーが既存のプログ

ラムのソースコードを変更することなくシステムを利用可能とすることで利便性の向上を図る．

1.3 構成

本論文の構成は以下の通りとなっている．

1 章 はじめに

GPGPU 技術の発展による需要の拡大，それに伴って起こる問題について触れ，本研究の背景・目的を述べる．

2 章 GPGPU

GPGPU 専用，コンシューマ向けの各モデルの GPU について GPGPU の観点から述べ，その違いや特徴を説明する．また，GPGPU 統合開発環境である CUDA について説明する．

3 章 既存技術・既存研究

GPU 計算の信頼性を向上させる既存の研究や技術を紹介し，本研究との差異を挙げて新規性について述べる．

4 章 システム概要

本研究で開発したシステムの概要を説明する．また，システムの開発に使用したソフトウェア DS-CUDA について解説する．

5 章 システム評価

開発したシステムの性能評価として，各機能の動作確認やパフォーマンスの測定を行う．

6 章 まとめ

本研究において得られた結果や課題点，今後の展望などについて述べる．

2 GPGPU

本研究では，GPU の中でも特に NVIDIA 社製の GPU を扱う．本章では同社製 GPU およびそれを用いた GPGPU 統合開発環境である CUDA についての解説を行う．

2.1 GPU

2.1.1 コンシューマー向けモデル

コンシューマー向けモデルの GPU は，ゲーミングや動画再生といった一般的な用途向けに設計されている GPU である．NVIDIA GeForce シリーズがこれに当たる．GeForce シリーズは，マイクロソフトが開発したゲームやマルチメディア処理用 API 群である Microsoft DirectX に対応・最適化されており，主にグラフィックス用途のモデルである．グラフィックス用途ではビット反転などのメモリエラーは画面のごく一部の色の僅かな違いとしてしか現れず，しかも瞬時に次のフレームの描画によって隠されるために人間が知覚することはほぼない．そのためメモリエラーなどに対する耐故障性はあまり重要視されていない．また，グラフィックス用途では倍精度浮動小数点数ほど高い計算精度が必要になる場面は少なく，GeForce シリーズは単精度浮動小数点数の処理能力に主眼が置かれている．

後述する GPGPU 専用モデルである NVIDIA Tesla シリーズが登場するまではこの GeForce シリーズで GPGPU を行っていたが，常に耐故障性・信頼性の問題があった．そのため，コンシューマー向けモデルの GPU を使用した GPGPU での信頼性を向上させる手法がいくつか提案されている．

Tesla シリーズ登場後も，価格性能比に優れる GeForce シリーズが GPGPU に使用される例はあり，信頼性向上手法は現在でも研究が続けられている．表 1 は，コンシューマ向けモデルと GPGPU 専用モデルの最上位機種の性能・価格を比較した表である．コンシューマ向けモデルからは GeForce GTX Titan を，GPGPU 専用モデルからは TeslaK40 を代表して掲載する．なお表中のピーク演算性能は各 GPU のスペックから計算される理論上の値である．

表 1　各モデル GPU の性能・価格比較

モデル名	単精度ピーク演算性能	倍精度ピーク演算性能	価格
GeForce GTX Titan	4.7TFlops	1.3TFlops （※）	約 10 万円
Tesla K40	4.3TFlops	1.4TFlops	約 80 万円

（※）設定によって変化する．通常の設定では表中の数字のおよそ 1/8 となる．

図 1　左が GeForce GTX Titan，右が Tesla K40
画像は http://www.nvidia.co.jp/titan-graphics-card および http://www.nvidia.co.jp/object/tesla-servers-jp.html からの引用

2.1.2　GPGPU 専用モデル

NVIDIA は 2008 年に GPGPU 専用モデルである NVIDIA Tesla シリーズを発表し，これによって GPGPU に使われ続けていた GeForce シリーズを置き換えた．Tesla シリーズは ECC メモリを搭載することで耐故障性を向上させている．また，コアクロックなどのスペック上は同程度の性能である GeForce シリーズの GPU よりも倍精度演算の性能が高くなっているなど，高性能計算向けのチューニングが施されている．しかしその分高価であるというのは先述の通りである．

なお業務用途として設計された，ワークステーション向けモデルである NVIDIA Quadro シリーズでは，上位モデルに限って ECC 機能が搭載されているため，Tesla シリーズ同様信頼性の高い GPGPU を行うことが可能である．

2.2　CUDA

2.2.1　CUDA の概要

CUDA（Compute Unied Device Architecture）[1] は NVIDIA が開発した，自社製 GPU による GPGPU の統合開発環境である．現在最新バージョンとして CUDA 5.5 が公開されており，無料で利用できる．C 言語を一部拡張した言語と，コンパイラやライブ

ラリなどから構成される．NVIDIA はこれ以前にも Cg という CG 描画のための GPU プログラミング言語を開発しているが，データ型に GPU 特有の型しか使えないなど，汎用的なプログラミングを行うのは困難であった．CUDA は C 言語の型をそのまま使えるなど，より汎用コンピューティングに適したものとなっている．CUDA プログラムの動作には対応する NVIDIA の GPU が必要だが，GPGPU 専用のものである必要はなく，コンシューマー向け GPU である GeForce シリーズや，ECC 非搭載の Quadro シリーズでも CUDA 対応のものであれば CUDA プログラムを動作させることが可能である．ただし前述の信頼性・耐故障性の問題のため，長時間に及ぶ，または負荷の高い計算に用いることは推奨されない．

2.2.2 CUDA 対応 GPU の構造

CUDA に対応する GPU は，CUDA コアと呼ばれる多数のストリーミングプロセッサ（SP）と，グローバルメモリと呼ばれるビデオメモリを持つ．CUDA コアとグローバルメモリは高速なメモリインターフェースで接続されており，CPU - メインメモリ間よりも非常に大きなメモリバンド幅を持つ．また CUDA コアは 32 個毎などでグループとしてまとめられており，これをストリーミングマルチプロセッサ（SM）と呼ぶ．SM 内の各 CUDA コアでは同時に同じ命令を違うデータに対して実行できる（SIMD: Single Instruction Multiple Data）．これらの特徴から，並列性がある多数のデータを扱う計算を高速に行うことが可能である．

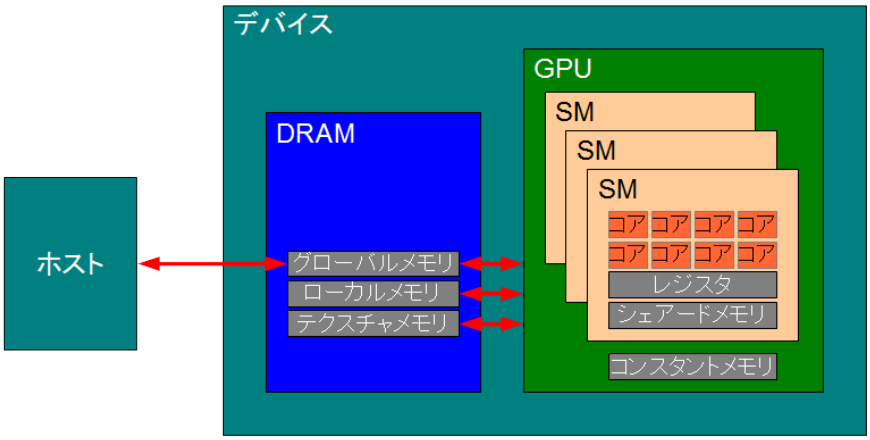

図 2　GPU 構造の概略

GPU には他にもシェアードメモリ，ローカルメモリ，コンスタントメモリ，テクスチャメモリ，レジスタが存在し，用途やアプリケーションのデータ構造によって使い分けられる．

2.2.3 CUDA アプリケーションの処理の流れ

CUDA では GPU をデバイスと呼び，デバイス上で実行される関数をカーネルという．一方，CPU ホストと呼ぶ．CUDA を利用したプログラムの一般的な流れは図 3 のようになる．

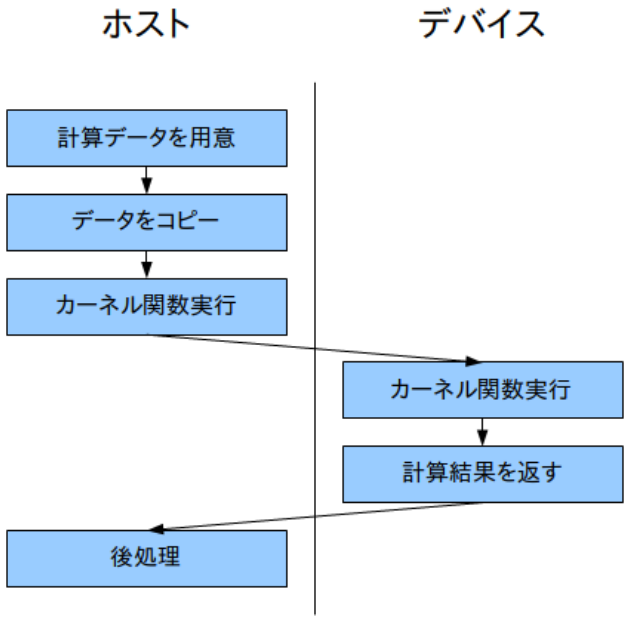

図 3　CUDA アプリケーションの一般的な処理の流れ

1. ホストからデバイスへ計算データをコピーする
2. ホストがカーネル関数を呼び出す
3. デバイスで計算する
4. デバイスからホストへ計算結果を返す

デバイスでのメモリ確保や、ホストからデバイスへのデータの転送には CUDA の API を使用する．CUDA は 2 系統の API を持ち，それぞれ CUDA Runtime API, CUDA Driver API と呼ばれる．Runtime API の関数には接頭語 cuda が付き（例：`cudaMalloc`），Driver API の関数には接頭語 cu がつく（例：`cuMemAlloc`）．2 系統の API ともデバイスコードの記述は同様であるが，ホストコードの書き方が異なる．

Runtime API は上位 API であるため，ホストコードの記述が Driver API よりも容易であるという特徴を持つ．Runtime API は CUDA Toolkit に付属する共有ライブラリに実装されている，一方下位 API である Driver API は，リソースの管理に気を付けなければならないが Runtime API よりも柔軟な処理の記述が可能である．またモジュール機能など，Runtime API にはない機能が含まれる．実装はライブラリではなく GPU ドライバに含まれる．

以下では CUDA Runtime API を使用する場合について述べる．

1. や 4. でのデータ転送には `cudaMemcpy` 関数が使用される．引数で転送方向を指定することでホストからデバイス (H2D: Host To Device)，デバイスからデバイス (D2D: Device To Device)，デバイスからホスト (D2H: Device To Host) へデータを転送する．以下ではこれを略して `cudaMemcpyH2D` などと呼ぶことにする．

2. でのカーネル関数の呼び出しには，`kernelName<<<grid, block>>>(args)` といったような CUDA 独自の記法を用いる．CUDA にはスレッド，ブロック，グリッドと呼ばれる実行単位がある．

スレッド
: カーネル動作時のプログラムの最小単位．CPU におけるスレッドよりも小さい処理を担当し，1 つのコアに対して多数のスレッドが割り当てられて実行される．

ブロック
: スレッドをまとめたもので，1 ブロック当たり最大 512 個のスレッドを持つ．ブロック中のスレッドは 3 次元的に管理でき，アプリケーションのデータ構造に合わせた表現が可能．

グリッド
: ブロックをさらにまとめたもので，ブロック同様 3 次元的に表現される．

カーネル呼び出し時に指定する `grid` および `block` は，それぞれグリッド中のブロックの配置を 3 次元的に表現したもの，ブロック中のスレッドの配置を 3 次元的に表現したものである．

2.2.4 CUDA アプリケーションのコンパイル

CUDA を使ったアプリケーションのコンパイルには `nvcc`（NVIDIA CUDA Compiler）を使用する．コンパイルの流れは図 4 のようになる．CUDA ソースコードは CPU 上で実行されるホスト用コードと GPU 上で実行されるカーネル用コードに分けられ，ホスト

用コードは C コンパイラに渡されてコンパイルされる．カーネル用コードは nvcc 自身によって PTX と呼ばれる中間コードにコンパイルされる．さらに PTX コードは実行時に CUDA ドライバによって，実際に実行される GPU 環境に合わせて JIT(Just In Time) コンパイルされ，GPU 上で実行される．

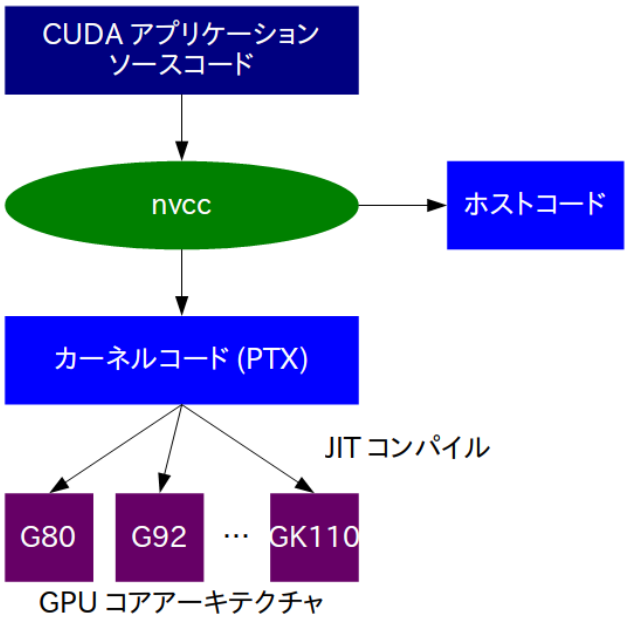

図 4　CUDA アプリケーションのコンパイル

2.2.5　Compute Capability

CUDA のバージョンとは別に，CUDA における GPU ハードウェアの機能を示す Compute Capability という指標が存在する現在は 1.0，1.1，1.2，1.3，2.0，2.1，3.0，3.5 の各バージョンが存在し，Compute Capability が 1.x である GPU を Tesla 世代（NVIDIA Tesla シリーズと同名であるが直接的な関係はない），同 2.x を Fermi 世代，同 3.x を Kepler 世代と呼ぶことがある．Compute Capability 1.3 以降，倍精度浮動小数点演算がサポートされ．これによってさらに高い精度での GPU コンピューティングが可能となった．ただし，先述の通り GeForce シリーズは単精度浮動小数点演算の性能に重きを置いており，倍精度浮動小数点演算の性能自体が高いレベルでチューニングされているというわけではない．

他の世代およびバージョンについては CUDA C Programming Guide[7] を参照されたい.

3　既存技術・既存研究

本章では，本研究の目的である信頼性や可用性の向上についての先行研究や，既存の技術について紹介する．

3.1　信頼性の向上

ここで言う信頼性とは，「計算やホスト - デバイス間の通信が正常に行われ，結果に誤りが含まれないこと」を指す．本研究においてはメモリエラーに限らず，通信路上でのノイズ混入等によって結果が異なった場合全てを総じて計算ミスと呼ぶ．以下では，計算ミスの検出・訂正を行うことにより GPU の信頼性を高める技術や研究を紹介する．

3.1.1　ECC メモリ

ECC（Error Check and Correct）メモリとは，メモリに誤った情報が記録されていることを検出し，その訂正を行うことができるメモリモジュールのことを言う．通常の DRAM にはハミング符号を用いた (72,64)SEC/DED(Single-Error-Correction/Double-Error-Detection) 符号 [8] が使用される．(72,64)SEC/DED 符号とは，元の情報 64 ビットに対し ECC 用の情報 8 ビットを付加し，全体として 72 ビットの情報から元の 64 ビットに含まれる 1 ビットの誤りを訂正，または 2 ビットの誤りを検出できる符号である．データ量が増加するため帯域幅を通常より多く使用し，誤り検出・訂正の実行時間も必要になるため，高いパフォーマンスのみが要求される場面で使われることは少ないが，サーバーなど継続して安定動作することが求められる用途で多く使用される．

3.1.2　ソフトウェア ECC

丸山らはコンシューマー向け GPU を GPGPU に使用した際に耐故障性が大きな問題になるとして，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手法を提案した [9][10]．ECC には通常の DRAM と同様にハミング符号を用い，被保護領域 4 バイトまたは 8 バイト毎に ECC 用の領域 1 バイトを用いる．この ECC 用領域は GPU 上で通常の領域にメモリが確保されたタイミングで別途確保される．被保護領域への書き込み時に ECC が計算され，ECC 用領域に保存される．被保護領域からの読み出し時にもメモリ内容から ECC を計算し，ECC 用領域に保存されたものと比較する．これによって，1 ビットの誤り訂正または 2 ビットの誤り検出を行うことが可能となる．

丸山らはプロトタイプとしてこの手法を CUDA のライブラリとして実装し，いくつかの CUDA アプリケーションに適用して性能評価を行っている．それによると行列積で最大 300%，N 体問題で 15% 程度の性能オーバーヘッドが発生した．メモリアクセスが発生する度に ECC を計算・検査するため，メモリアクセス頻度の高いアプリケーションではオーバーヘッドが高くなるという問題点があった．また ECC の計算・検査機能はライブラリとして実装されているものの，この機能を利用するにはユーザーが明示的に呼び出しを指示しなければならず，元のプログラムコードを変更する必要が生じる．

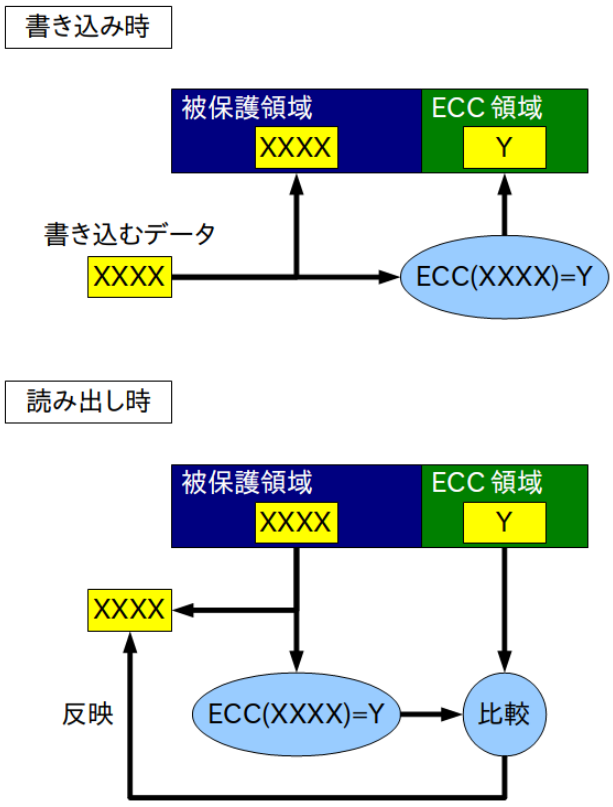

図 5　メモリアクセス時の ECC の計算・検査

丸山らはさらにオーバーヘッドを削減する手法として，チェックポイントとエラー検知を併用した耐メモリエラーフレームワークを提案している [11]．GPU カーネルを実行する際，GPU に与えた計算データをホストに保持することでエラー検知時にカーネル関数を再実行し，エラーからの復帰を可能としている．またそれによって ECC による誤り訂正は不要となり，符号を誤り検出に特化することでオーバーヘッドの低減を実現しながら

も通常の DRAM に搭載される ECC と同程度の信頼性を達成している．しかし CUDA アプリケーションへのフレームワークの適用は手動であり，元のプログラムコードの変更は必要なままである．また検知対象をメモリエラーに限っており，ハードウェア故障などには対応していない．

本研究ではメモリエラーに限らず，ハードウェア故障などにも対応する．また誤りの検出・訂正を元のプログラムを変更することなく，自動的に行えるシステムを提供する．

3.1.3 多重計算（冗長計算）

島田らは耐過渡故障技術の一つである多重計算（冗長計算とも言う）について，トレードオフとなっている性能・電力コストを評価した [12]．多重計算とは同じ計算を一度のプログラム実行で 2 回以上行い，その結果を比較することによって結果が正しいかを確認する手法である．連続計算と並列計算の二種類が存在し，それぞれ以下のような特徴を持つ．

連続計算

同一のデバイス上で同じ計算を 2 回連続して行い，結果を比較する．計算時間は単純に 2 倍となるが，単一の GPU で実行できる．

並列計算

同じ計算を同時に複数行う．単一の GPU 内で倍の数のスレッドを用いて計算する方法と，2 台の GPU を用いて計算する方法とがある．後者では実行時間の増加は最小限に抑えられるが，前者ではハードウェア資源の不足する場合があるため必ずしもそうはならない．また後者では GPU を 2 台使用することによって電力コストが増加する．

島田らは結果の比較方法と組み合わせて計 8 通りの多重計算手法を挙げ，それぞれについて実行時間，電力，エネルギー効率に関して比較を行っている．島田らの研究の主眼は耐故障性を向上させることによる性能・電力コストのトレードオフの評価であり，耐故障性の向上を実現する手法の提案ではない．そのため，多重計算は GPGPU アプリケーションに直接記述する形で行っている．本研究では耐故障性の向上を実現する方法として冗長計算を扱い，実際にユーザーが利用しやすい形のシステムとして提供する．

3.2 可用性の向上

可用性とは，システムにおける障害の発生しにくさや障害発生時の復帰能力のことを言う．以下に挙げる手法は直接的には可用性の向上を目的としたものではないが，本研究においては同手法を参考に可用性の向上を目指す．

3.2.1 マイグレーション

立薗らは長時間に及ぶ計算，あるいは遊休計算機上での計算の実行においては各計算期間の負荷分散が重要であるとして，仮想計算機を用いた負荷分散可能な MPI 実行環境を提案した [13]．仮想計算機として Xen[14] を用い，同計算機が持つマイグレーション機能を利用して，仮想計算機ごと MPI 実行環境をマイグレーションさせることによって MPI プロセスに透過的なマイグレーションを実現している．この研究では動的負荷分散を実現する手法としてマイグレーションを取り上げているが，本研究ではこれを可用性の向上に用いる．具体的には，ある GPU が故障により計算の続行が不可能になった，あるいは計算ミスの頻度が高く使用に耐えないと判断した場合，その GPU で行っていた計算を他の GPU に移して計算を続行できるようにすることを目指す．

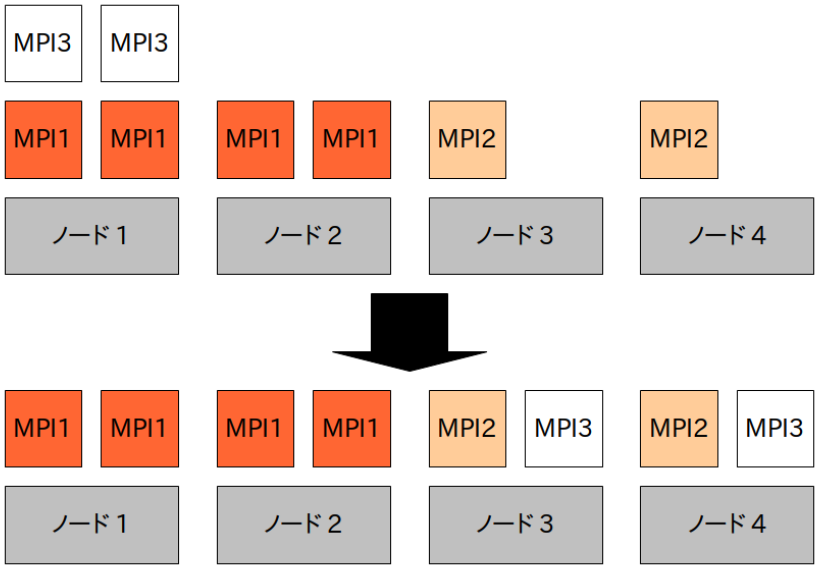

図 6　動的負荷分散による各ノードの処理量の均一化（参考文献 [13] 図 9 より）

4　システム概要

本章では実装したシステムの概要や，システムの構築に使用したソフトウェアについて説明する．

4.1　要求される機能

1 章で述べたように，GPGPU 専用モデルの GPU が登場したとは言え，未だ GPGPU 分野においては信頼性の向上が必要なシーンが多い．そこでまず，本システムでは GPGPU における，特にコンシューマー向け GPU の信頼性向上を目指す．具体的には，複数の GPU に同じ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計算ミスを検出する．

さらに，信頼性の向上のみでは不十分な場合も考えられる．例としては，数日から数週間に及ぶような大規模な計算中に GPU が故障し，計算のやり直しを余儀なくされることなどが考えられる．計算過程やメモリ内容などを記録し，他の GPU に状態を移して計算を続行させるマイグレーションによって解決できるが，一般的な PC 筐体には GPU は 2 台程度しか搭載できないため十分な可用性を確保できない場合がある．そこで本システムでは，次節で述べる GPU 仮想化ソフトウェアを用いてマイグレーション機能を実装し，ネットワーク上の GPU を使用することでこの制限を排して可用性を高めるとともにシステムの使用を容易にする．

また，冗長計算についても汎用性のため GPU 仮想化ソフトウェアを利用して実装する．

4.2 DS-CUDA

4.2.1 概要

DS-CUDA(Distributed-Shared CUDA)[15][16] は慶應義塾大学および電気通信大学で開発された，ネットワーク上の PC に搭載された GPU をローカルにある GPU と同様に扱えるようにするためにミドルウェアである．GPU としては NVIDIA 社製の CUDA 対応 GPU に，インターフェイスとしては CUDA Runtime API のみに対応しており，CUDA Driver API や Open CL を使用することはできない．本システムはこの DS-CUDA を機能拡張することにより実現されている．

4.2.2 ハードウェア構成

図 7 は典型的な DS-CUDA システムの概要図である．DS-CUDA バージョン 1.2.9 以前では各サーバーノード上ではあらかじめサーバープログラムである `dscudasvr` を動作させておく．一方，アプリケーションプログラムはクライアントノード上で実行する．アプリケーションプログラムが CUDA Runtime API を使って GPU にアクセスしようとすると，DS-CUDA システムによって要求がネットワークを通じてサーバープログラムへ送られて API やカーネル関数が実行される．

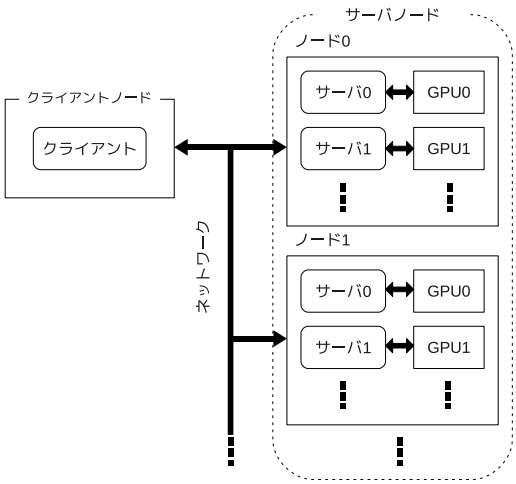

図 7　DS-CUDA システム

バージョン 1.3.0 以降ではサーバーデーモン機能が搭載された．複数の GPU を搭載するサーバーノードでも単一のデーモンプロセス dscudad を起動するだけで，クライアントからの要求に応じて自動的に dscudasvr が起動し，クライアントプログラムが終了すればサーバープログラムも終了する．

デーモン機能を利用するにはクライアント側で環境変数 DSCUDA_USEDAEMON に何らかの値を設定する．

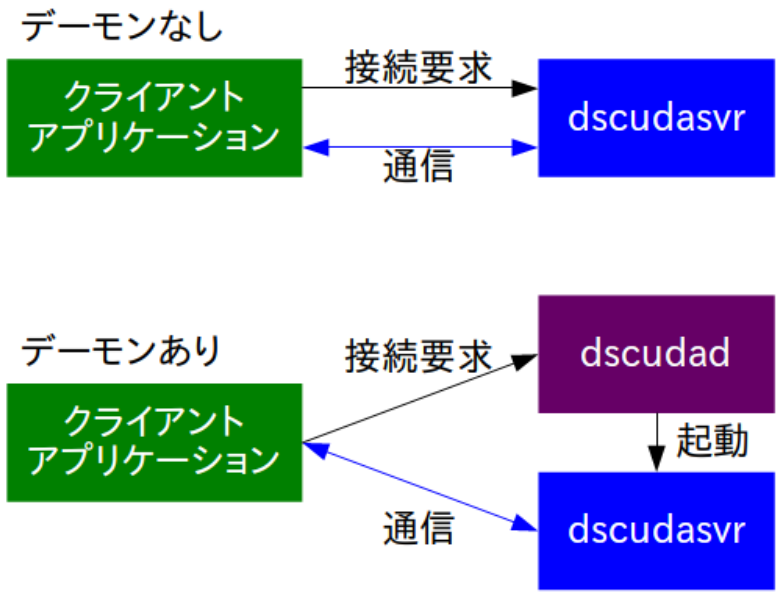

図 8　バージョンによるサーバーとの接続方法の違い

アプリケーションプログラムからはいずれのサーバーノード上の GPU もクライアントノード上に搭載されているかのように見える．例えば CUDA Toolkit に付属する，本来はローカル PC 上の GPU の情報を表示する deviceQuery プログラムを DS-CUDA システム上で起動すると，接続しているサーバー上の GPU の情報が表示される．

ユーザーからはネットワーク上の GPU が直接見えているわけではなく，DS-CUDA システムが用意する仮想的な GPU を通じてネットワーク上の GPU（物理 GPU と呼ぶ）にアクセスする．仮想 GPU と物理 GPU は必ずしも 1 対 1 で対応するわけではなく，2 つ以上の物理 GPU で後述する冗長化された仮想 GPU を構成する場合がある．

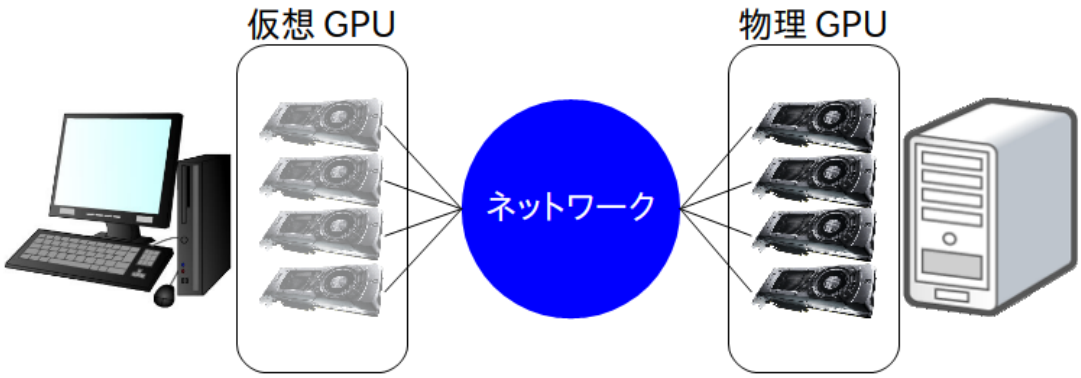

図 9　仮想 GPU と物理 GPU

4.2.3 ソフトウェア階層

DS-CUDA は図 10 に示すように階層化されている．クライアントノードで実行される CUDA アプリケーションは，CUDA Runtime API と互換の API を持つラッパーを介してサーバーノードにアクセスする．要求を受け取ったサーバープログラムは実際の CUDA Runtime API を使用して物理 GPU で処理を行う．通信部分はラッパー内に隠蔽されているため，ユーザーが通信処理を記述する必要はない．

ネットワークとしては InfiniBand を使用することを想定しているが，TCP ソケット通信にも対応しているため，通信速度は劣るが Gigabit Ethernet も使用可能である．ただし，本研究で実装した機能の一部は InfiniBand では使用できない．

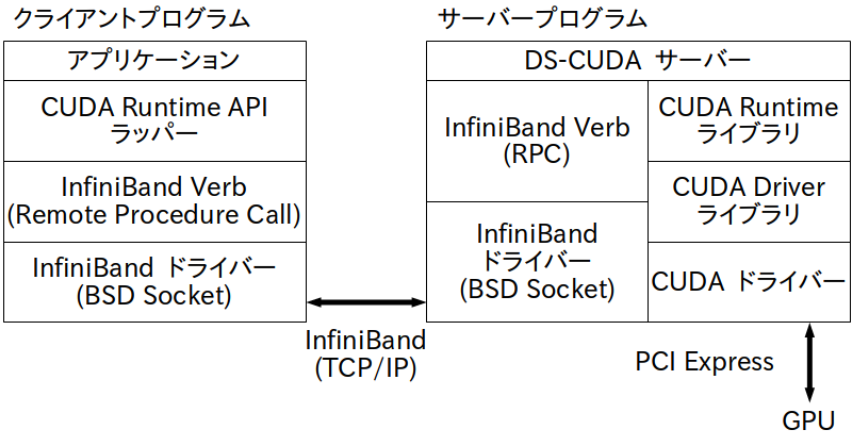

図 10　DS-CUDA のソフトウェア階層

4.2.4 コンパイラ

DS-CUDA では，実際のコンパイル処理のほとんどを `nvcc` で行っている．これは DS-CUDA が CUDA Runtime API と互換の API を持つライブラリを提供しているためである．しかし，カーネル関数の呼び出し部分は CUDA の拡張記法である三重ブラケット（例：`myKernel<<<g, b>>>()`）を使用するため，この部分についてはラッパーライブラリのみでは対応できない．そのため，カーネル呼び出し部分は DS-CUDA の専用プリプロセッサが前処理を行い，ネットワーク越しにカーネルモジュールの読み込みと起動を行う DS-CUDA の専用 API に置き換えている．

4.2.5 使用方法

接続するサーバーの指定は DSCUDA_SERVER 環境変数で行う．DS-CUDA サーバーの IP アドレスに続いて GPU 番号を指定することで，接続先のサーバーおよび使用する物理 GPU を切り替える．例として，svr0 と svr1 の 2 つのサーバーノードに各 2GPU が搭載されている環境で，

```
> export DSCUDA_SERVER="svr0:0 svr0:1,svr1:0 svr1:1"
```

のように指定すると，svr0 上の 0 番の物理 GPU が 1 つ目の仮想 GPU として，同様に svr1 上の 1 番の物理 GPU が 3 つ目の仮想 GPU として扱われる．カンマ区切りで記述された複数の GPU は単一の冗長な仮想 GPU として扱われ，それに含まれる物理 GPU では全て同じ内容の処理が実行される．この例では svr0 上の 1 番と svr1 上の 0 番の GPU で冗長計算が行われるように設定されている．またこの例のように，冗長デバイスはサーバーノード間を越えて構成できる．

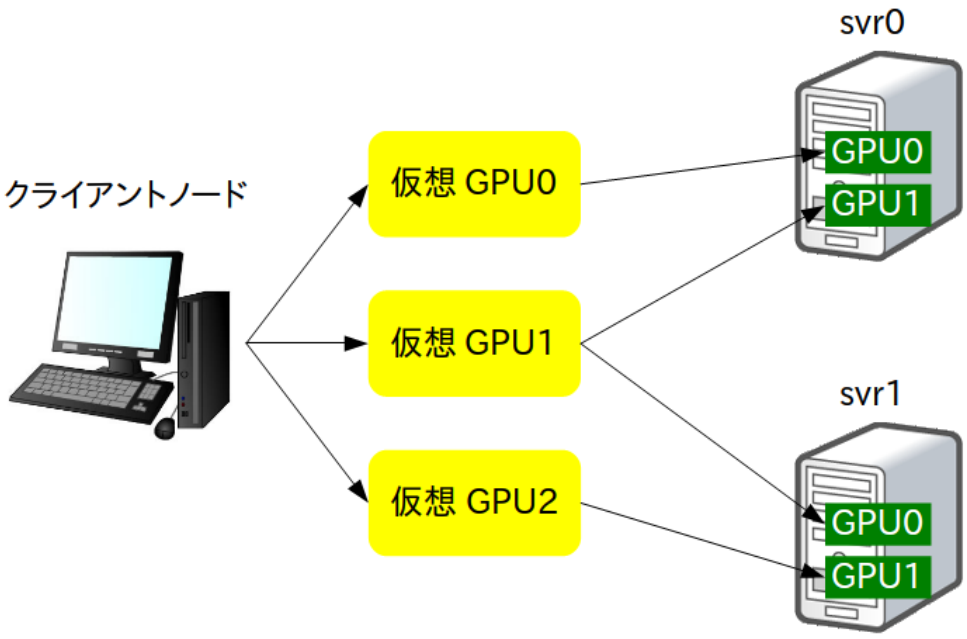

図 11　DSCUDA_SERVER 環境変数の設定例

複数の GPU を使うプログラムを記述は，図 12 のようになる．通常，一台の PC に搭載できるグラフィックカードの数は，マザーボードの PCI Express のスロット数や電源容量の制限から 2〜4 程度となる．それ以上の数を使いたい場合は MPI（Message Passing Interface）などを利用して実現する必要がある．しかし DS-CUDA ではローカルにある GPU を使う通常の CUDA アプリケーションと同様に，`cudaGetDeviceCount`, `cudaSetDevice` 関数を使って使用する GPU の切り替えを容易に行える．ここで言う GPU とは仮想 GPU を指しており，図 11 の例の環境では `cudaGetDeviceCount` 関数は物理 GPU の数である 4 ではなく，仮想 GPU の数である 3 を返す．

```
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 12　複数の GPU を使う CUDA コードの例

4.3 冗長計算機能

本節では冗長計算機能の実装について述べる．計算ミスの検出と，計算ミスの訂正を行う部分からなっており，各機能ごとに分けて解説する．

4.3.1 計算ミスの検出

冗長計算によって計算ミスを検出するには，計算に使用した複数の GPU から計算結果を回収して比較する必要がある．DS-CUDA は CUDA Runtime API のラッパーライブラリを持ち，CUDA Runtime API と互換の関数を呼び出すことでネットワーク上の GPU を使用している．本システムではホスト - デバイス間でデータ転送を行う `cudaMemcpy` のラッパー関数に，各 GPU からクライアントに転送されたデータを比較するコードを追加することで計算結果の比較を実現する．現在の実装では，最初の GPU の計算結果に対してそれ以外の GPU での計算結果が一致しているかを確認している．この方法では計算ミスの検出のみが可能であり，この時点での計算ミスの訂正はできない．

参考として，GPU を 3 つ以上使用する場合には多数決により正しい計算結果を推定することもできる．1 つの GPU で計算ミスが発生する確率よりも，当然 2 つ以上の GPU で同時に計算ミスが発生する確率の方が低いため，実用的に問題ないレベルの信頼性を得ることも可能だろう．ただし GPU の枚数や消費電力が元の 3 倍になり，本研究のように 2 枚使用するよりも更に効率は悪くなる．

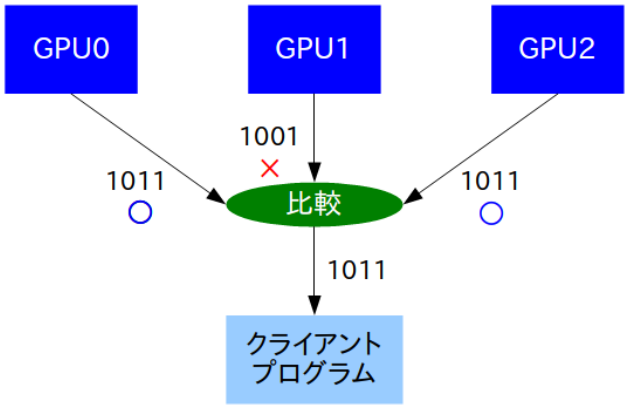

図 13　3GPU を使用した冗長計算の例

本システムにおいては計算ミス検出後のエラー状態からの復帰を確実にするために，計算ミスが含まれると思われる部分を自動的に再計算する．

4.3.2 自動再計算による計算ミスの訂正

前項で述べた計算ミスの検出機構では，cudaMemcpyD2H によって計算結果がデバイスからホストへ転送される度に比較を行う．あるタイミングで計算ミスが検出された場合，その計算ミスは 1 回前の cudaMemcpyD2H 呼び出し以降，現在実行中の cudaMemcpyD2H 以前に発生しているはずである．よって GPU のメモリ内容を 1 回前の cudaMemcpyD2H 時の状態に戻し，2 度の cudaMemcpyD2H 間に行われた処理を繰り返すことで正しい結果を得られると考えられる．ここでは，この期間のことを 1 ステップと呼ぶ．この自動再計算機能を加えた DS-CUDA システム上の CUDA アプリケーションの処理の流れは図 14 のようになる．

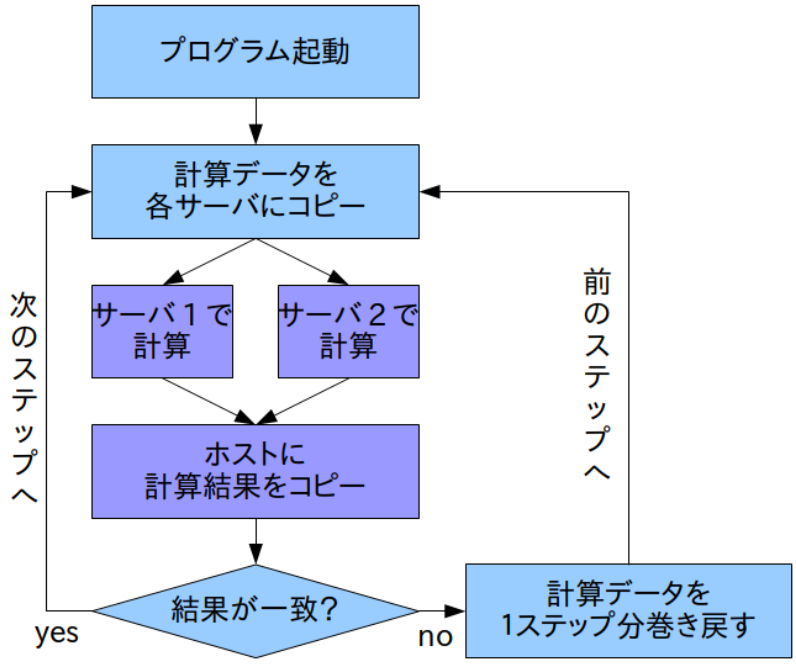

図 14　自動再計算を加えた処理の流れ

DS-CUDA システムでのネットワーク上の GPU の制御は，カーネル関数の実行を除いて全て CUDA Runtime API およびそのラッパー関数を通じて行われる．よって 1 ステップの間に実行された CUDA Runtime API およびカーネル関数の呼び出し履歴とそ

の引数を保存し，計算ミスの検出時にその履歴を辿ることで 1 ステップ分の処理を再実行できる．

実装としては，各ラッパー関数の実行終了時に関数識別用の ID と引数を保存する．例として cudaMemcpy のラッパー関数の一部を図 15 に示した．

```
(cudaMemcpy 関数の本体)
...
cudaMemcpyArgs args;
switch (kind) {
  case cudaMemcpyHostToDevice:
    args.dst = dst;
    args.src = (void *)src;
    args.count = count;
    args.kind = kind;
    dscudaVerbAddHist(dscudaMemcpyH2DId, (void *)&args);
    break;

  case cudaMemcpyDeviceToDevice:
...
```

図 15　cudaMemcpy ラッパーの実装の一部

関数本体の処理が終わった後に，cudaMemcpy **の引数**と関数識別用の ID を引数として dscudaVerbAddHist 関数を呼び出している．dscudaVerbAddHist 関数は ID によって保存する引数の型を判別して保存する．関数の履歴と引数は全てホストメモリに保存されるため，クライアントノード上で問題が発生した場合には対応できない．

4.4　マイグレーション機能

前節で解説した冗長計算機能では，サーバーノード自体の問題により計算が続行不可能になった場合に対応できない（例：サーバーノード上の GPU の故障，サーバーノードの OS のクラッシュ）．そこで本システムでは実行中のプログラムを他の計算機に移行し，状態を復元して計算を続行する動的マイグレーション機能を搭載することで可用性の向上を目指す．

本節ではマイグレーション機能について解説する．

4.4.1　サーバーの動的な再接続

本システムでは DS-CUDA を利用することにより，ネットワーク越しに GPU を使用することを可能としている．そのため使用する GPU の動的な変更自体は容易で，単にプログラム中で接続するサーバーノードを変更すればよい．しかしマイグレーション元の GPU で行っていた計算をマイグレーション先の GPU で続行したい場合，GPU 上で確保されているメモリアドレスの内容を保存しておき，その内容を元に再接続先でメモリ状態を復元する必要がある．図 16 に一連の処理の流れを示した．

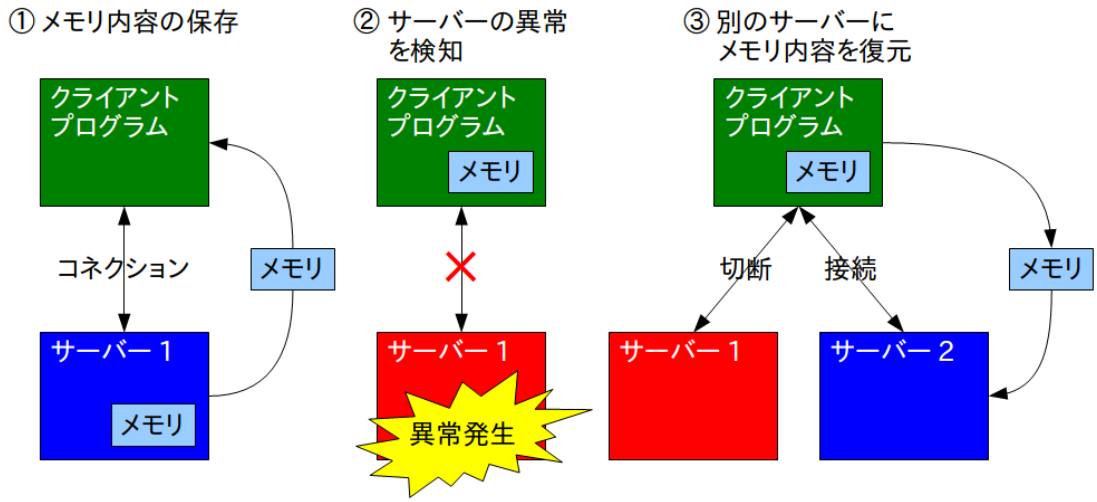

図 16　マイグレーションの流れ

しかし，GPU メモリの確保要求に対して返されるアドレスが毎回同じかは不明である．

そこで予備実験として，次の 3 つの実験を行った.

予備実験 1

確保するメモリサイズを固定的に変化させながら（1 回目は 10 バイト，2 回目は 20 バイト，3 回目は…）`cudaMalloc` 関数を 10 回呼び出すプログラムを作成し，繰り返し実行して確保されるメモリのアドレスを調査する.

予備実験 2

予備実験 1 のプログラムを同時に 2 つ以上起動し，同一の GPU 上で確保されるメモリアドレスのプロセス毎の変化を調査する.

予備実験 3

予備実験 1 と同様の操作を行い，全ての GPU メモリを開放してから同一のプロセス内でもう一度同じ操作を繰り返す.

予備実験 1 は，同じ順序・同じサイズでメモリを確保した場合に，確保されるメモリのアドレスに再現性があるかを確かめるものである．予備実験 2 は，同時に 2 つ以上のプロセスからメモリ確保要求があった場合に，確保されるメモリのアドレスに他のプロセスの動作が影響するかを調べるものである．予備実験 3 は，同一プロセスでの以前のメモリの使用状況ががその後に確保されるメモリのアドレスに影響を与えるかを確かめるものである.

図 17，図 18，図 19 に以上の 3 つの予備実験の出力を示した．結果として，予備実験 1,2,3 いずれも何度実行しても出力に変化はなかった．実験 1 の結果から，マイグレーション元の GPU 上に `cudaMalloc` 関数によって確保されたメモリサイズと各関数が呼び出された順序を記録しておけば，マイグレーション先で同じアドレスにメモリを確保することが可能である．これによってクライアントノードで動作するアプリケーションは，マイグレーションの前後で参照するメモリアドレスを変更する必要がないことがわかった.

また実験 2 の結果から，GPU でもプロセス毎にメモリ空間は独立しており，複数の DS-CUDA サーバープログラムが起動していても各プロセスで確保されるメモリアドレスに影響しないことがわかった.

さらに実験 3 の結果から，同一のプロセス内であっても正常にメモリが開放されていれば，同じ手順を辿ることで確保するメモリアドレスを再現できることがわかった.

これらの結果をまとめると，マイグレーション先のサーバーはメモリが全て開放された状態，あるいはサーバーが起動直後である状態が望ましい.

```
> ./preexp1
0x200200000
0x200200200
0x200200400
0x200200600
0x200200800
0x200200a00
0x200200c00
0x200200e00
0x200201000
0x200201200
```

図 17　予備実験 1 の出力

```
> ./preexp2.sh
process1:  0x200200000
process2:  0x200200000
process1:  0x200200200
process2:  0x200200200
process1:  0x200200400
process2:  0x200200400
process1:  0x200200600
process2:  0x200200600
process1:  0x200200800
process2:  0x200200800
```

図 18　予備実験 2 の出力

```
> ./preexp3
0x200200000
0x200200200
0x200200400
0x200200600
0x200200800
0x200200a00
0x200200c00
0x200200e00
0x200201000
0x200201200

0x200200000
0x200200200
0x200200400
0x200200600
0x200200800
0x200200a00
0x200200c00
0x200200e00
0x200201000
0x200201200
```

図 19　予備実験 3 の出力

バージョン 1.2.9 以前の DS-CUDA では，サーバーノード上で実際に CUDA Runtime API を実行する `dscudasvr` プロセスが継続して動作しているため，以前に他のクライアントからの接続を受けて動作していた際のメモリ操作が後のプログラムで確保されるメモリアドレスに影響する場合がある．逆にバージョン 1.3.0 以降では，デーモン機能によってサーバーノード上で常に動作するのは `dscudad` プロセスのみにすることができる．

dscudasvr はクライアントからの接続があったタイミングで起動され，クライアントプログラムの終了とともに dscudasvr も終了する．そのため，サーバーノードへの接続直後は常にメモリが全く確保されていない状態が保証される．よって本システムでは確実性のために，DS-CUDA のデーモン機能を使用している場合に限ってマイグレーションを行う．

4.4.2 サーバー検索機能

DS-CUDA の接続先は DSCUDA_SERVER 環境変数で指定するが，動的マイグレーションを行うには指定されたサーバーの他にマイグレーション先のサーバーの IP を知る必要がある．そこで本システムでは，他のサーバーノードで動作中の DS-CUDA サーバーを自動的に検索する機能を搭載した．

クライアントプログラムは起動後の初期化処理中に UDP プロトコルのブロードキャストメッセージを利用し，サーバーノード上で動作中のサーバープログラム dscudad に対してメッセージを送信する．メッセージを受け取ったサーバープログラムは応答メッセージを返すことでクライアントプログラムに自身の IP を知らせる．さらに応答を受け取ったクライアントでは，DSCUDA_SERVER 環境変数に既に何かしらの値が指定されていた場合，受け取った IP をマイグレーション先候補として保存する．一方 DSCUDA_SERVER 環境変数に何も指定されていなかった場合は，見つかったサーバーにすぐさま接続する．これは DS-CUDA システムからはクライアントプログラムがいくつの GPU を使用するかは実際に実行するまでわからないためである．見つかったサーバーを初期接続先とすべきかマイグレーション先候補とすべきか判断できないため，マイグレーション先候補としては扱わず初期接続先としている．

現在の実装では，サーバープログラムは自身のノード上で動作する GPU の数は通知しない．またクライアントノードと同一のサブネットに属するサーバーのみを検索する．

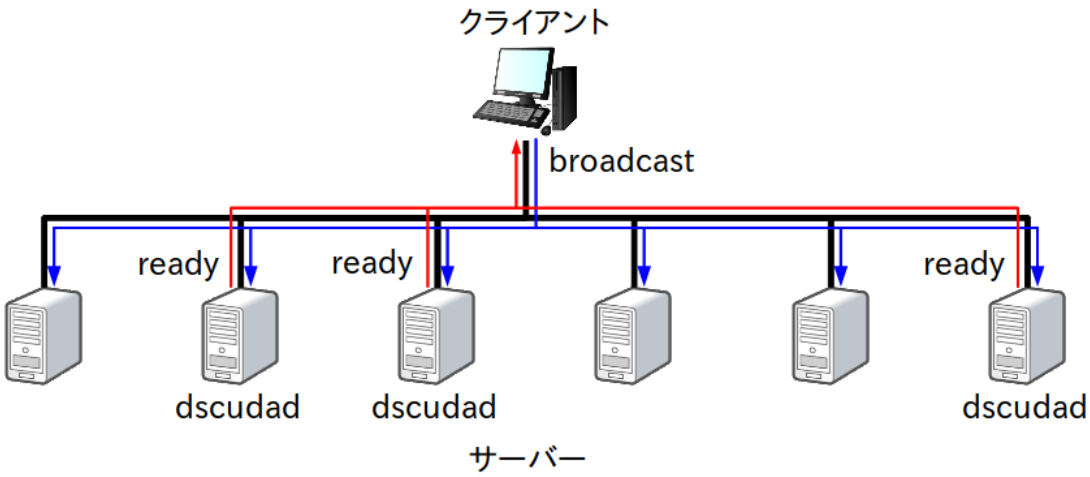

図 20　DS-CUDA サーバー検索のイメージ

5　システム評価

本章では実装したシステムの機能的な評価と，本システムを利用することによる実行効率の変化の評価を行う．

5.1　機能評価

実装した各機能が正常に動作するかを確認する．動作確認にはリアルタイム分子動力学シミュレーションプログラムである claret を使用する．

claret は NaCl 分子中の原子の動きを MD（Molecular Dynamics：分子動力学）によって計算するプログラムである．系に含まれる粒子数の変更や，GPU の使用・不使用などが実行中に変更できる．計算速度が表示されるため，パフォーマンス評価にも利用可能である．

claret では計算の初期条件にランダム性はなく一定のため，複数回実行しても同じ結果が得られる．この点を利用して冗長計算機能の評価を行う．

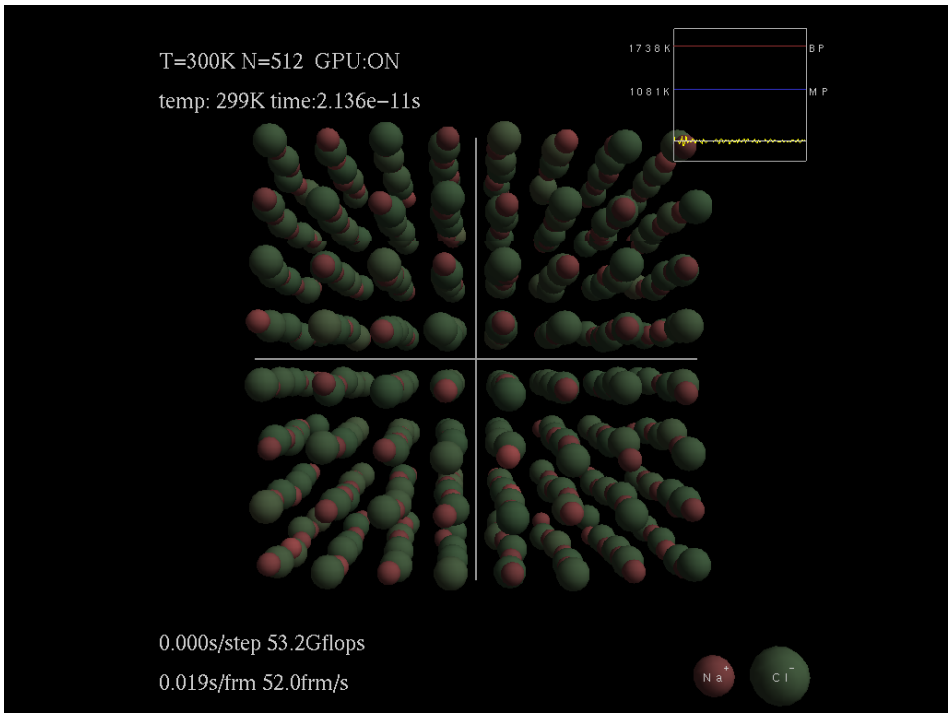

図 21　claret の実行中の様子

5.1.1 冗長計算機能

計算ミスが発生した際に，自動的に計算ミスが検出されるか，また検出後の自動再計算が正常に行われるかを確認した．しかし実際にコンシューマー向け GPU で計算ミスが発生する確率は非常に低いため，ここでは擬似的に計算ミスを発生させて機能の評価を行う．具体的には，1/1000 の確率で計算結果の先頭 1 バイトを不正な数値に変更する処理を追加した `dscudasvr` と，正常に動作する `dscudasvr` を使用して冗長計算を行う．

図 22 は正常な計算結果のグラフと計算ミスが含まれるグラフを重ねたものであり，横軸は計算ステップ，縦軸は NaCl 分子の系の温度を表す．また，その一部を拡大したものが図 23 である．

1000 ステップの動作中に，計算ミスを発生させるサーバープログラムが動作するターミナルには異常なデータが生成された旨を知らせるメッセージが数回表示された（図 24）．実際，正しい計算結果である緑色のグラフと，計算ミスが含まれる計算結果である赤色のグラフにずれが生じており，計算ミスが発生していることが確認できる．

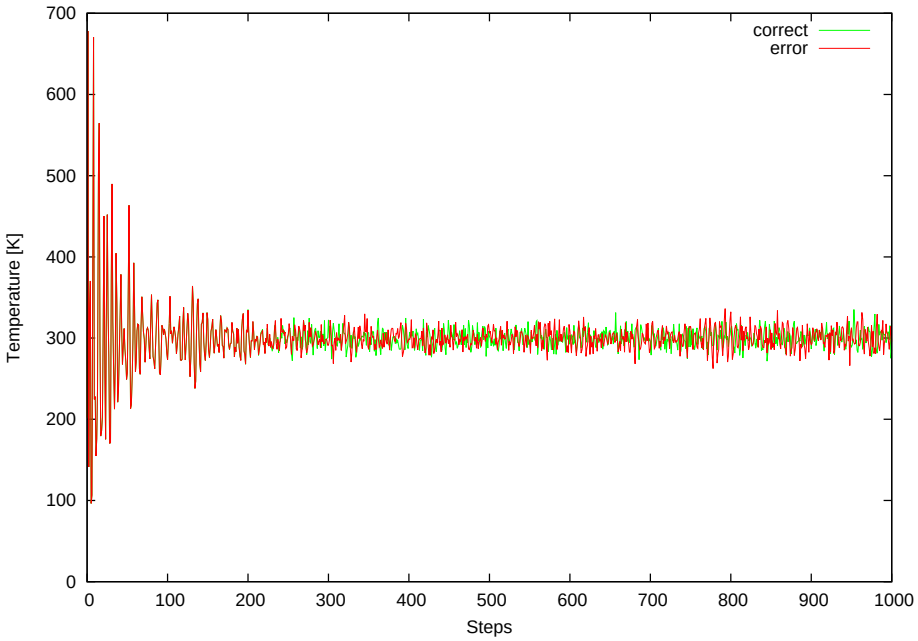

図 22　NaCl の系の温度の推移

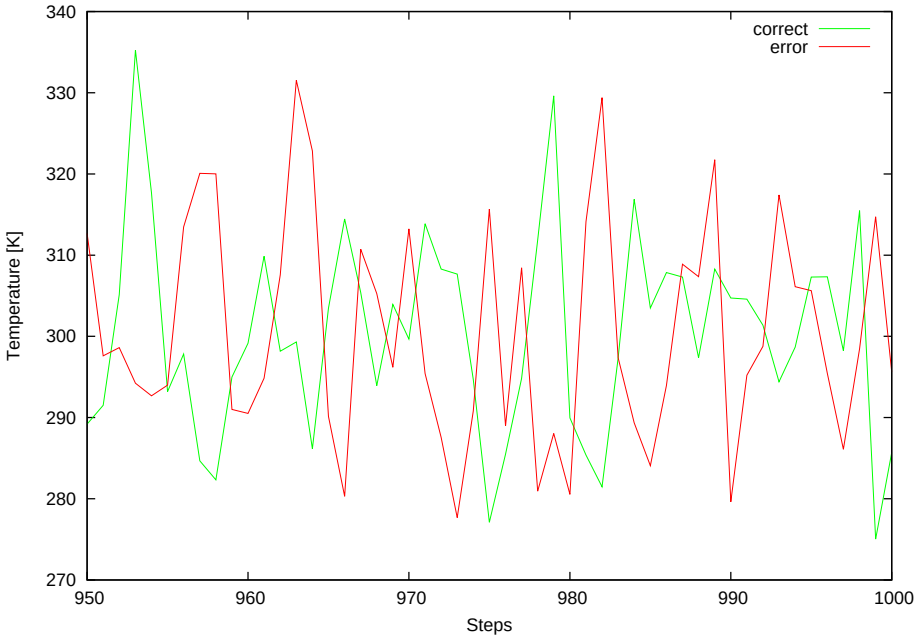

図 23　2 つのグラフのずれ

```
[yoshikawa@ds12 src]$ ./dscudasvr_rpc_ng
dscudasvr[0] : WarnLevel: 2
dscudasvr[0] : method of remote procedure call: RPC
dscudasvr[0] : TCP port : 65433 (base + 0)
dscudasvr[0] : ndevice : 1
dscudasvr[0] : real device       : 0
dscudasvr[0] : virtual device    : 0
dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.

dscudasvr[0] : ################## bad data generatad.
```

図 24　不正なデータの生成を知らせるメッセージ

図 25 の青色のグラフは，先ほどの正しくない結果を返す dscudasvr と正常な dscudasvr で冗長計算機能を使用した際のものである．

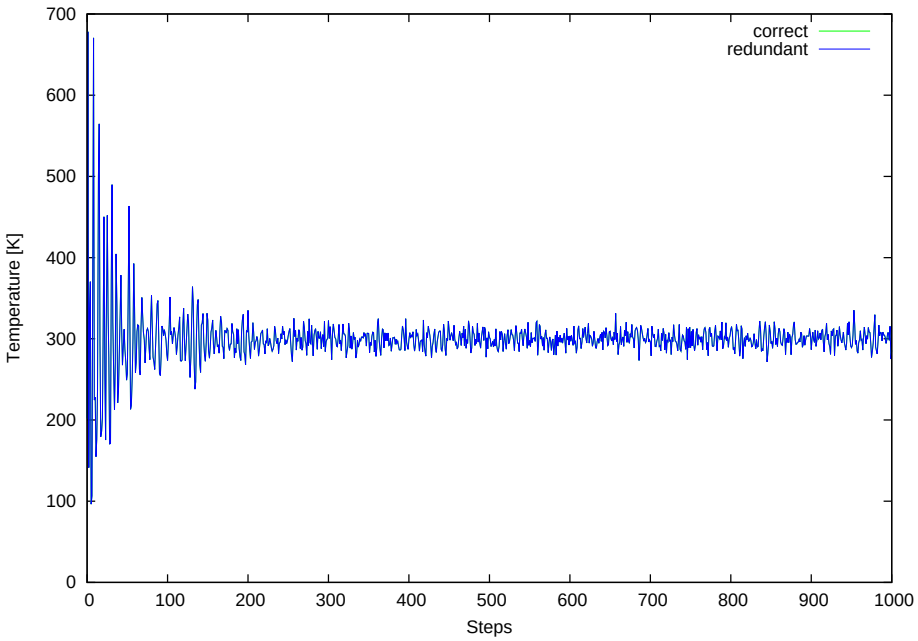

図 25　自動再計算による計算ミスの訂正の確認

先程と同様に，計算ミスを発生させるサーバープログラムは同様のメッセージを出力したが，クライアントプログラムが動作するターミナルには計算ミスを検出したというメッセージが表示された（図 26）．続けて自動再計算機能によって 1 ステップ分の再計算が行われているメッセージも表示された．

実際にグラフを見ると，2 つのグラフがちょうど重なっているために赤色のグラフは表示されず，青色のグラフのみが見える．冗長計算機能によって計算ミスが検出され，自動再計算によって正しい結果が得られて最後までクライアントプログラムが正常に動作したことが確認された．

```
158     281.427925
159     294.033663
160     310.316026
161     301.620516
162     289.881442
163     292.571422
164     282.269668
165     309.268979
166     327.729997


cudaMemcpy() data copied from device1 & device0 UNMATCHED.


recalling functions...
recalling done.
167     320.240569
168     290.803567
169     289.625249
170     275.472094
171     309.973746
172     309.800917
173     307.004115
174     312.565406
```

図 26　計算ミスが発生したステップの自動再計算

5.1.2 マイグレーション機能

クライアントプログラムの実行中に接続するサーバーを切り替え，正常に計算が続行されるかを確認した．検証には冗長計算機能と同様に claret を使用し，数ステップを最初のサーバーで計算した後別のサーバーへと接続を切り替えた．図 27 はマイグレーションを実行した際のシステムおよび claret の出力である．

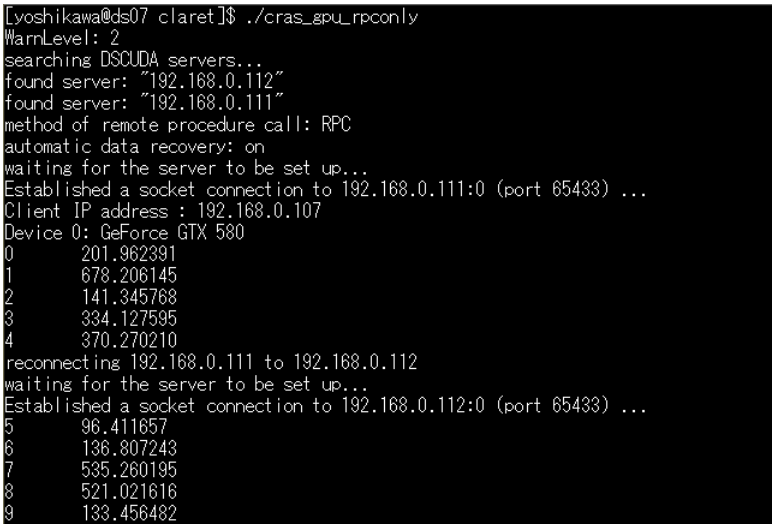

図 27　マイグレーション実行時の出力

「ステップ数　温度」となっている行が claret の出力，それ以外がシステムの出力である．5 ステップ分の計算を元のサーバーで行った後，マイグレーションの実行によって接続するサーバーを切り替えているメッセージが表示され，マイグレーション先のサーバーで計算が続けられている．図 28 はマイグレーション機能を使用せずに計算したものを correct，マイグレーション機能を使用して計算したものを migration として出力をグラフ化したものである．図 25 と同様に，correct と migration のグラフが一致しており，青色の migration のグラフのみが表示されている．これにより，クライアントプログラムの実行中にマイグレーションによって別のサーバーに移行しても計算が正しく続行できていることが確認された．

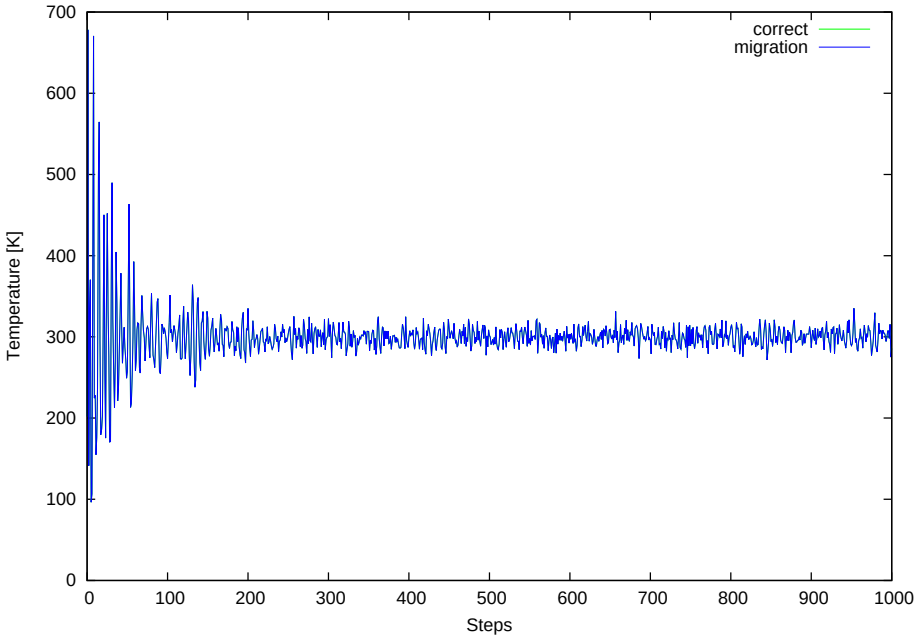

図 28　マイグレーション機能使用時の計算の正しさの確認

5.2 パフォーマンス評価

本章では，本システムの使用時のパフォーマンスを評価する．クライアントプログラムを本システムを使用せずに実行した際と，使用して実行した際の計算速度や計算にかかる時間を測定する．また，自動再計算とマイグレーションの実行コストについても測定する．

測定を行った環境は以下の通りである．

- CPU: Intel Corei7 920 2.67GHz
- メモリ: 8GB
- GPU: NVIDIA GeForce GTX580
- ネットワーク: Gigabit Ethernet 1000BASE-T

機能評価に続いて claret と，それに加えてベクトル加算のプログラムを使用して測定を行った．

5.2.1 ベクトル加算

ベクトルの長さ N を 512 から 131072 まで変化させ，通常の CUDA プログラムとして実行した場合，本システムを使用した場合，本システムを使用しさらに冗長計算機能を使用した場合の 3 通りで実行時間を測定した．図 29 にその測定結果を示した．図中の凡例 local が通常の CUDA プログラムとして実行した場合，remote が本システムを使用した場合，remote(with redundant) が remote に加えて冗長計算機能を使用した場合である．

ベクトル加算はベクトルの長さ N に対して，データ通信量も計算量も $\mathcal{O}(N)$ である．計算の内容は加算の繰り返しと単純なため，同じオーダーでもデータ通信に多く時間がかかると考えられる．今回の測定ではクライアントとサーバーで同じ GPU を使用しているため，カーネル関数の実行にかかる時間は本システムや冗長計算機能の使用の有無に関わらずおおよそ同程度と考えられる．そのため，データ通信にかかる時間が実行時間の差として現れることが予想される．

実際にグラフを見ると，本システムを使用した場合では使用しない場合に比べて実行時間はおよそ 30 倍にもなっている．これはカーネル関数の実行時間が非常に短いため，データ通信だけでなくシステム内の処理がオーバーヘッドとして現れてしまったのだと考えられる．

また冗長計算機能を使用した際にさらに実行時間が増加しているのは，クライアント -

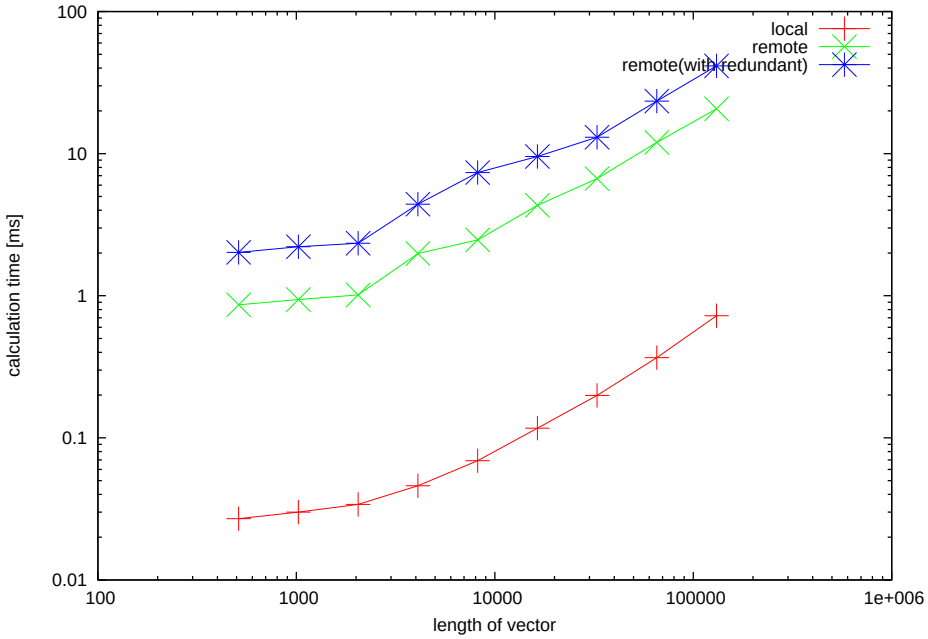

図 29　ベクトル加算プログラムの実行時間

サーバー間のデータの転送に加え，クライアント上での関数の履歴・引数の保存のためのメモリ転送と，結果の比較にかかる時間が加わるためであると考えられる．

5.2.2　N 体問題

ベクトル加算のプログラムでは，計算量の少なさのために本システムを使用した際のオーバーヘッドが非常に大きくなっていた．そこで，計算量が $\mathcal{O}(N^2)$ である N 体問題を計算するプログラム claret でも同様の測定を行った．図 30 がその測定結果で，凡例は図 29 と同様であるが，縦軸が計算速度であることに注意されたい．

本システムを使用した際の効率はベクトル加算と比べて全体的に向上しており，粒子数が増えて計算量が大きくなると local の 75% 程度のパフォーマンスを発揮している．local のグラフは粒子数 16000 前後で 950GFLOPS 程度で横ばいになっているのに対し，他 2 つのグラフはまだ上昇している．粒子数をさらに増やせば効率はさらに上昇するだろう．

冗長計算機能を使用すると，ベクトル加算の場合と同様の理由から，remote のグラフからさらに 1 割程度効率が低下している．

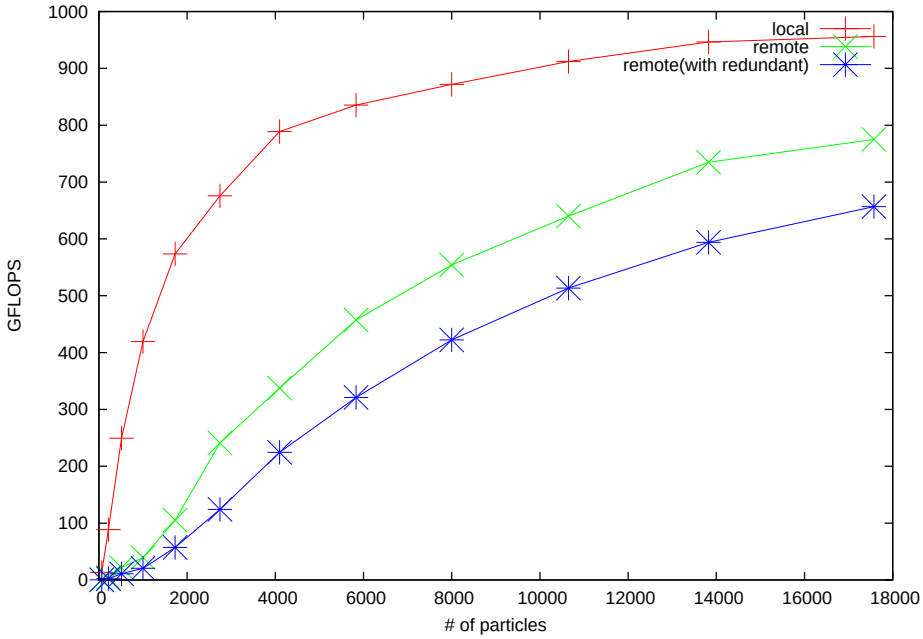

図 30　claret における計算速度

5.2.3　各機能の実行コスト

ここでは全体の実行パフォーマンスではなく，冗長計算機能，マイグレーション機能を実行した際の局所的なコストを測定した．測定には引き続き claret を使用した．

初めに，図 31 に冗長計算機能による自動再計算実行時の claret および本システムの出力を示した．出力の形式は「ステップ数　そのステップの計算時間」となっている．

自動再計算の前後で，1 ステップにかかる計算時間に有意な差は見られなかった．自動再計算実行時にはそのステップの開始時のメモリ状態を復元するために，クライアント上に保存していたメモリ内容をサーバーに転送してからカーネル関数を再実行する．そのため，そのステップの実行時間はほぼ倍となるはずである．しかし実際には他のステップと実行時間に変化はない．原因については現在調査中である．

図 32 に 1000 ステップ分の出力を示した．575 ステップ目以外にも自動再計算は実行されたが，いずれも実行時間に差は現れなかった．グラフ中で 2 点実行時間が増加しているステップがあるがこのステップでは自動再計算は実行されておらず，サーバーの GPU の状態など他の要因によるものと考えられる．

```
566     0.005892
567     0.005888
568     0.005847
569     0.005817
570     0.005847
571     0.005842
572     0.005868
573     0.006050
574     0.005771


cudaMemcpy() data copied from device1 & device0 UNMATCHED.


recalling functions...
recalling done.
575     0.005884
576     0.005896
577     0.005976
578     0.006072
579     0.005823
580     0.005895
581     0.005907
582     0.006037
```

図 31　自動再計算前後の 1 ステップの計算時間の出力

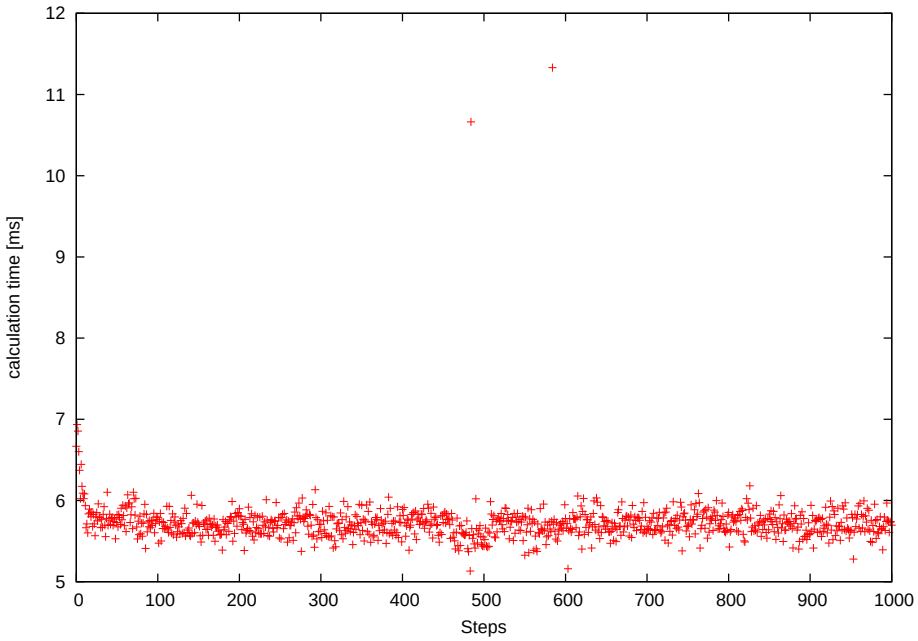

図 32　自動再計算実行時の各ステップの計算時間

次に，マイグレーションを実行した際の実行時間の増加を測定した．図 33 はマイグレーション実行前後の各ステップの計算時間の出力である．

498 ステップ目から 499 ステップ目の間でマイグレーションが実行されており，499 ステップ目の実行時間が他のステップよりも非常に長くなっている．マイグレーションには新しいサーバーとの通信の確立や，マイグレーション先でメモリ状態を復元するためのデータ通信などの比較的重い処理が含まれる．そのためマイグレーションの実行には大きなコストがかかる．しかし実際にサーバーでの計算の続行が不可能になり，マイグレーションが実行される確率を考えれば，実行時間の増加はプログラム全体から見るとそこまで大きなものではないだろう．

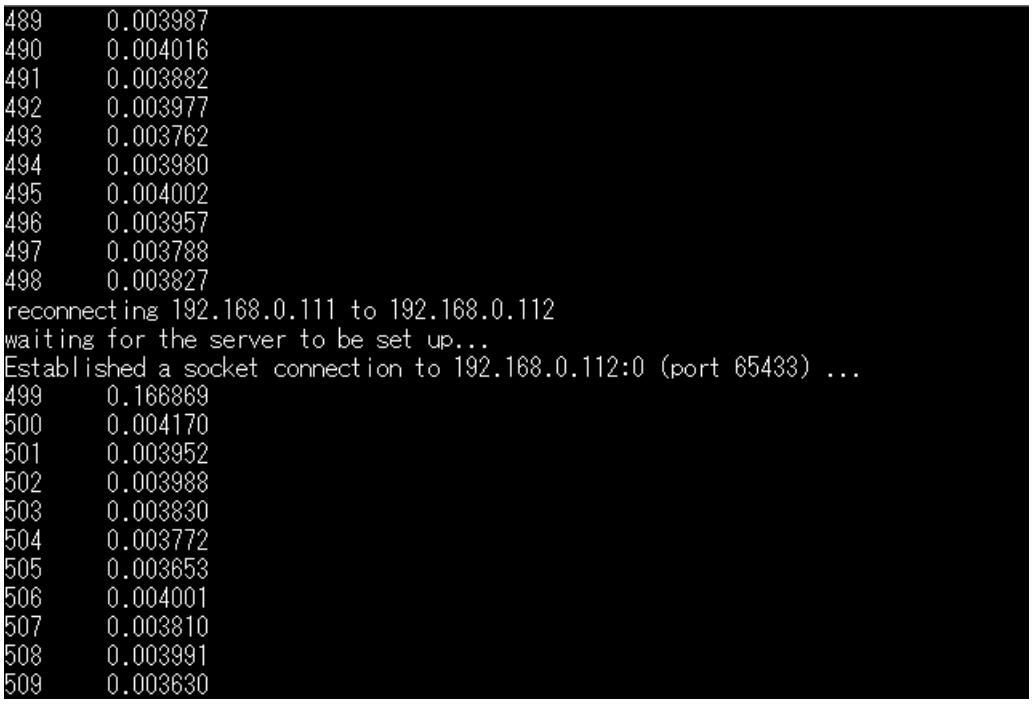

図 33　マイグレーション前後の 1 ステップの計算時間

6 まとめ

6.1 本研究で達成したこと

コンシューマー向け GPU はまれに計算ミスが発生するため，GPGPU に使用する際に信頼性の面で問題になることがあった．本研究では冗長計算によって計算ミスの発生を検出・訂正する機能と，GPU で問題が発生して計算の続行が不可能になった際に使用する GPU を変更して計算を続行させるマイグレーション機能を実装した．どちらも GPU 仮想化ソフトウェアである DS-CUDA を利用して，ネットワークで接続された PC 上の GPU を使用する形で実装した．現在の実装では機能は限定的であり汎用性に欠けるが，信頼性の向上について一定の効果があることを確認し，コンシューマー向け GPU を GPGPU に問題なく使用できるように可能性を示した．

6.2 本研究の問題点

本システムでは DS-CUDA を利用してネットワーク上の GPU を使用することによって機能の実装をしたため，GPU を使用する際にネットワーク通信が発生する．そのため，通常の CUDA プログラムとして実行した場合よりも実行時間が余計にかかってしまう．計算量の多いプログラムでは，本システムを使用せずに実行した場合の 75% 程度のパフォーマンスを発揮したが，計算量の少ないプログラムではオーバーヘッドが非常に大きくなってしまった．パフォーマンスを向上させる方法として，ネットワークに InfiniBand を使用する，冗長計算機能やマイグレーション機能におけるメモリコピーを非同期で行うなどの対策が考えられる．しかし，現在の実装ではネットワークは Gigabit Ethernet にしか対応していない．

参考文献

[1] “CUDA”, [Online] Available: https://developer.nvidia.com/category/zone/cuda-zone

[2] “ATI Stream”, [Online] Available: http://www.amd.com/jp/products/technologies/stream-technology/Pages/stream-technology.aspx

[3] “Open CL”, [Online] Available: http://www.khronos.org/opencl/

[4] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, M. Taiji: “42TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence”, SC ’09 Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, No. 62, USA, 2009.

[5] “Folding@home”, [Online] Available: http://folding.stanford.edu/

[6] I.S.Haque, V.S.Pande: “Hard Data on Soft Errors: A Large-Scale Assessment of Real-World error Rates in GPGPU”, CCGRID ’10 Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp.691-696, Australia, 2010.

[7] “CUDA C Programming Guide”, [Online] Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[8] E. Fujiwara: “Code Design for Dependable Systems: Theory and Practical Applications”, Wiley Interscience, 2006.

[9] 丸山直也，松岡聡，尾形康彦，額田彰，遠藤敏夫：“ソフトウェア ECC による GPU メモリの耐故障性の実現と評価”，電子情報通信学会技術研究報告．DC，ディペンダブルコンピューティング 108(181)，pp.9-15，2008.

[10] 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．[ハイパフォーマンスコンピューティング] 2009-HPC-123(14)，pp.25-30， 2009.

[11] 丸山直也，額田彰，松岡聡：“GPU 向け耐メモリエラーソフトウェアフレームワーク”，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] 2009-HPC-123(8)，pp.1-6, 2009.

[12] 島田大地，丸山直也，額田彰，遠藤敏夫，松岡聡：“GPU における耐故障性を考慮した数値計算の電力性能”，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] 2009-HPC-122(26)， pp.1-6，2009.

[13] 立薗真樹，中田秀基，松岡聡：“仮想計算機を用いて負荷分散を行う MPI 実行環境”，電子情報通信学会技術研究報告．[コンピュータシステム] 105(225)，pp.7-12，2005.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, H. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Wareld: “Xen and the art of virtualization”, SOSP ’03 Proceedings of the nineteenth ACM symposium

[15] A. Kawai, K. Yasuoka, K. Yoshikawa, T. Narumi: “Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability”, The Fourth International Conference on Future Computational Technologies and Applications, FUTURE CONPUTING 2012, Nice, France, 2012.

[16] M. Oikawa, A. Kawai, K. Nomura, K. Yoshikawa, K. Yasuoka, T. Narumi: “DS-CUDA: a Middleware to Use Many GPUs in the Cloud Environment”, International Workshop on Sustainable HPC Cloud at SC12, Salt Lake City, USA, 2012.

謝 辞

本研究を進めるにあたり，熱心にご指導いただいた指導教員の成見哲教授と副指導教員の沼尾雅之教授に感謝致します．また DS-CUDA を開発された慶應義塾大学の川井敦特任准教授には，DS-CUDA に機能を追加する際の設計や方針について相談に乗って頂き，システムを開発する上で大きな助けとなりました．ここに感謝の意を表します．