修　士　論　文　の　和　文　要　旨

研究科・専攻	大学院　情報理工学研究科　情報・通信工学専攻　博士前期課程		
氏　　　名	瀬戸口　幸寿	学籍番号	1431065
論　文　題　目	多数の仮想 GPU を用いた際の計算性能モデルの構築		

要　旨

GPU を科学計算などの汎用的な目的で使用する技術は GPGPU(General-Purpose computing on Graphics Processing Units)として知られている.DS-CUDA(Distributed Shared CUDA)はネットワークを通じたサーバ上の GPU を仮想化するミドルウェアで,クライアント側でソフトを書き換えることなく GPU 資源を用いた GPGPU が可能である.ただし，クライアントとサーバー間の通信がボトルネックになり易い.そこで本研究では MD シミュレーションと行列の乗算を題材に,最大 8 つの GPU を用いて計算時間の測定を行い,DS-CUDA による仮想 GPU を用いた場合と,物理 GPU を PCI Express 拡張 Box を用いて直接扱う場合とで,性能モデルを構築し実際の時間と比較した.

MD シミュレーションを通じて,DS-CUDA は PCI Express 拡張 Box と比較して,GPU 内の演算である kernel 関数の発行による通信に多くの時間がかかることがわかった.このことが,計算データ量小,GPU 並列数大の条件下(1000 粒子,8GPU)では,仮想 GPU と物理 GPU との計算時間に高速なネットワーク(Infiniband)で 1.5 倍,低速なネットワーク(GigabitEthernet)では 6 倍の差が生じることがわかった.一方で GPU 内での計算量が増大するに従って相対的に通信時間が占める割合は小さくなり,計算性能の向上のために GPU 並列数を大きくすることの有効性を示すことができた.

行列の乗算においても同様の傾向が得られた一方で,計算データ量に対する通信量が多いことから通信スループットの影響により,GPU 並列数を増やすことによる計算性能の向上は通信性能が十分,かつ計算データ量が十分大きい場合に限られることを示した.低速な GigabitEthernet ネットワークでは GPU 並列数を大きくすることによる計算性能の向上を,測定実験の計算データ量領域内(最大 8,190 次行列,8GPU)で確認することはできず,構築したモデルもその事実を示した.

以上の測定実験とモデル構築から,計算データ量が十分大きく GPU 並列数が小さい場合は,仮想 GPU がパフォーマンスを発揮できることを示した.

平成27年度修士研究論文
多数の仮想GPUを用いた際の計算性能モデルの構築

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
1431065　瀬戸口幸寿
指導教員　成見 哲　教授
副指導教員　沼尾 雅之　教授

平成28年3月14日

概要

GPUを科学計算などの汎用的な目的で使用する技術はGPGPU(General-Purpose computing on Graphics Processing Units) として知られている. DS-CUDA(Distributed Shared CUDA) はネットワークを通じたサーバ上の GPU を仮想化するミドルウェアで, クライアント側でソフトを書き換えることなく GPU 資源を用いた GPGPU が可能である. ただし，クライアントとサーバー間の通信がボトルネックになり易い. そこで本研究では MD シミュレーションと行列の乗算を題材に, 最大 8 つの GPU を用いて計算時間の測定を行い, DS-CUDA による仮想 GPU を用いた場合と, 物理 GPU を PCI Express 拡張 Box を用いて直接扱う場合とで, 性能モデルを構築し実際の時間と比較した.

MD シミュレーションを通じて,DS-CUDA は PCI Express 拡張 Box と比較して, GPU 内の演算である kernel 関数の発行による通信に多くの時間がかかることがわかった. このことが, 計算データ量小,GPU 並列数大の条件下 (1000 粒子,8GPU) では, 仮想 GPU と物理 GPU との計算時間に高速なネットワーク (Infiniband) で 1.5 倍, 低速なネットワーク (GigabitEthernet) では 6 倍の差が生じることがわかった. 一方で GPU 内での計算量が増大するに従って相対的に通信時間が占める割合は小さくなり, 計算性能の向上のために GPU 並列数を大きくすることの有効性を示すことができた.

行列の乗算においても同様の傾向が得られた一方で, 計算データ量に対する通信量が多いことから通信スループットの影響により, GPU 並列数を増やすことによる計算性能の向上は通信性能が十分, かつ計算データ量が十分大きい場合に限られることを示した. 低速な GigabitEthernet ネットワークでは GPU 並列数を大きくすることによる計算性能の向上を, 測定実験の計算データ量領域内 (最大 8,190 次行列,8GPU) で確認することはできず, 構築したモデルもその事実を示した.

以上の測定実験とモデル構築から, 計算データ量が十分大きく GPU 並列数が小さい場合は, 仮想 GPU がパフォーマンスを発揮できることを示した.

目 次

第1章　はじめに

1.1　背景

1.1.1　GPGPU 技術の広まり

近年,GPU(Graphics Processing Unit) が科学シミュレーションなどの汎用計算に用いられている．この技術は GPGPU(General-Purpose computing on GPU) と呼ばれており, HPC の分野に大きく貢献している．GPU は本来, 画像処理に特化したプロセッサであるが, 近年の GPU は大規模なデータに対し単純な演算を並列に行うことが可能である．また, 並列計算に対する親和性が大きく, 計算規模の増加に対して GPU の並列数を大きくすることで対応できる．この特徴から, 大規模行列に対する演算や分子動力学計算などのシミュレーションに用いられることが多い．

通常, 科学計算には高価な GPGPU 専用モデルの GPU を利用することが多い．GPGPU 専用モデルの GPU は ECC(Error Check and Correct) メモリを搭載しており, 倍精度演算機能を持つなど, 高性能計算に対して最適化されている．しかし,GPGPU モデルの GPU は高価であること, また GPU の台数分のノードを用意しなければならないことから, 研究機関外で複数台の GPU を利用することは困難が伴う．

1.1.2　仮想 GPU

これらの困難を解決する仕組みとして, ソフトウェアを書き換えることなくネットワークを介して複数の GPU を使用できるようにする GPU 仮想化ツールである DS-CUDA[1] がある．GPU の仮想化により, 場所の制限なしに複数の GPU を使用するプログラムを簡単に記述することが出来る．しかし, ネットワークを介してサーバに命令を送信し, 演算結果を受信してから出力等行う為, クライアントノード-GPU サーバ間の通信がボ

トルネックになる．その為, 最適なパフォーマンスを発揮する為の GPU の並列数等を導く指標やモデルの構築が待たれる.

1.2 目的

本研究の目的は,DS-CUDA を用いて複数台の GPU を仮想化した場合の計算における性能モデルを構築し, パフォーマンスを発揮するための指標について議論を行うことで, 今後の研究に貢献することである．また, ネイティブな GPU を複数台利用した場合のモデルと比較することで, 特徴と課題を議論することである.

目的の達成のために, 本研究では以下の手法で複数の GPU を用いた計算性能の比較を行った.

- Infiniband ネットワークを用いた DS-CUDA による仮想 GPU の利用
- GigabitEthernet ネットワークを用いた DS-CUDA による仮想 GPU の利用
- PCI Express 拡張 Box によるネイティブな GPU の直接利用

1.3 本論文の構成

本論文の構成と, その内容は次の通りである.

第 1 章 はじめに
本研究の背景と目的について述べる.

第 2 章 GPGPU
本研究に関連する既存の GPGPU における関連技術について紹介する.

第 3 章 関連研究
本研究に関連する既存の技術, 研究について紹介し, 本研究の新規性について述べる.

第 4 章 システム
本研究で行った実験で使用したシステム, ハードウェアについて述べる.

第 5 章 MD シミュレーション
: 分子動力学計算について, 計算の概要, 実験結果, モデル構築について述べる.

第 6 章 行列の乗算
: 行列の乗算について, 計算の概要, 実験結果, モデル構築について述べる.

第 7 章 まとめ
: 本研究で達成したこと, 今後の課題について述べる.

第2章　GPGPU

本章では, 本研究に関連する既存の GPGPU における関連技術について紹介する.

2.1　GPU

大島 (2010) によれば,GPGPU の歴史は以下の通りである [3].

GPU(Graphics Processing Unit) は画像処理の為の画像処理用の LSI チップである. 実用上では, ビデオメモリと呼ばれる DRAM とセットで運用され, その影響を大きく受ける. GPU の本来の役割は, グラフィックス表示の為の様々な処理を行い, 処理の結果をディスプレイに表示することである. 具体的には,CPU から画像の頂点やピクセルなどの描画情報を受け取り, 頂点情報を画面上のピクセルに対応付けする為の処理を行い, テクスチャや照明計算等を経て画面上のピクセルの色を決定し, ピクセルの色情報をフレームバッファへと書き込み, 画面へと出力する. この一連の流れは, パイプライン処理される.

描画処理技術の発展に従い, 様々な描画手法が提案されるようになると, 提案手法を実装した GPU の開発では実用化, 性能効率の点で問題が生じるようになった. そこで, ユーザが描画処理をソフトウェアレベルで変更できる GPU が登場した. これらは, 既存の描画処理を行う固定機能シェーダに対して, プログラマブルシェーダと呼ばれ, 普及することになった. プログラマブルシェーダにおけるプログラミングの制限は, GPU の世代が進むたびに緩和が進んだ. この緩和は GPGPU として汎用演算に使われるのに貢献することとなる.

また, 代表的な 3D グラフィックス処理である, 頂点処理やピクセル処理は, 高い並列性をもった処理である. これに対応するために,GPU は内部の演算機 (シェーダユニット, コア) の数を増やすことで並列性を向上させ, 性能を向上させた. この点は, 汎用計算においても高い並列性を発揮する点で貢献している.

CUDA に対応する GPU の構造の概略を図 2.1 に示す．CUDA 対応の GPU は多数の CUDA コア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,CPU-メインメモリ間よりも大きなメモリバンド幅を持ち, 高速である. また複数の CUDA コアはストリーミングマルチプロセッサ (SM) と呼ばれるグループでまとめられている．SM 内の各 CUDA コアは並列に同じ命令を違うデータに対して実行が可能である．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, 高速に行うことができる.

2.2 CUDA

CUDA(Compute Unified Device Architecture)[2] は NVIDIA 社が開発した GPGPU の為の統合開発環境であり, 拡張言語, コンパイラ,C 言語のライブラリから構成されている． GPGPU 向けの拡張言語は他に OpenCL[4] や DirectCompute[5],Openacc[6] などが存在する. それらと比較すると CUDA は

- NVIDIA 社製の GPU にしか対応してないが, 最適化されている

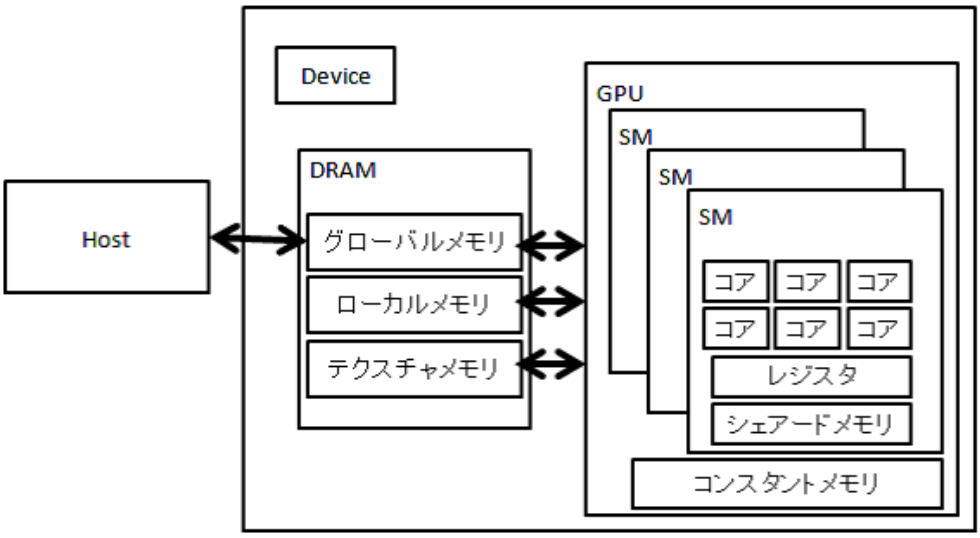

図 2.1: GPU の構造

- CPU-GPU 間のデータ管理が厳密に求められるが, 細かい記述が可能である

のような特徴があり, 高速化の為のチューニングがしやすく, 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,CUDA に対応する NVIDIA の GPU が必要である. CUDA 対応の GPU には,GPGPU 専用モデルの Tesla シリーズだけでなく, コンシューマ向け GPU である Geforce シリーズ等がある.

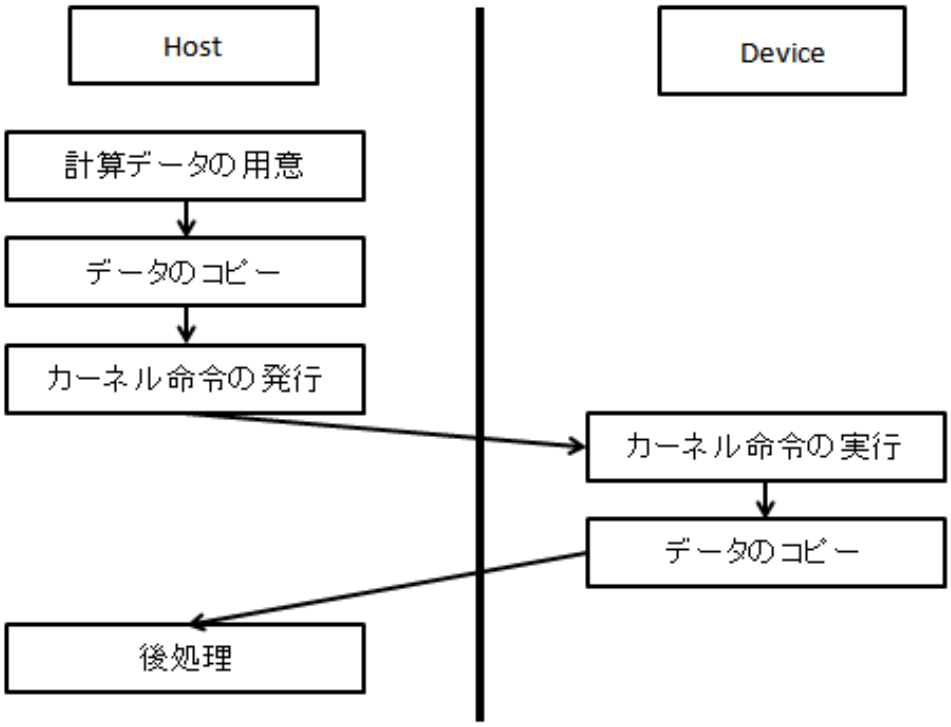

図 2.2: CUDA プログラムの流れの一例

CUDA では,GPU を `device`,CPU を `host` と呼び, GPU 上で実行される関数をカーネルという. CUDA プログラムの一般的な流れを図 2.2 に示す. 本研究で用いたアプリケーション (付録 1,3 に疑似コード記載) もこの流れの繰り返しで成り立っている. 付録上の `MemcpyH2D` は CPU から GPU へのデータ移動を, `MemcpyD2H` は GPU から CPU へのデータ移動を表す. カーネル関数の呼び出しには `kernel<<<グリッド,ブロック>>>(args)` という 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であり, 1 つのコアに対し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あり, ブロック中のスレッドは 3 次元的に管理できる.

- グリッド
  ブロックをさらにまとめたもので, グリッド中のブロックを 3 次元的に管理できる.

カーネル呼び出し時に指定するグリッド, ブロックはアプリケーションのデータ構造に合わせて 3 次元的に表現される.

2.3 DS-CUDA

DS-CUDA(Distributed-Shared CUDA)[1] は慶応義塾大学の川井らによって開発された, ネットワークを介してリモート GPU を用いる為のミドルウェアである. DS-CUDA のシステムは DS-CUDA コンパイラ, client ノード,(複数の)server ノードから構成され, server ノードに搭載された GPU を client ノードから利用できる.

図 2.3 は DS-CUDA システムの概略図である. 各 server ノード上では, あらかじめサーバプログラムを動作させておく. 一方,DS-CUDA コンパイラによってコンパイルされたアプリケーションプログラムは,client ノード上で実行する. アプリケーションプログラムが 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プログラムへと送られて, データ転送や kernel 関数の実行が行われる.

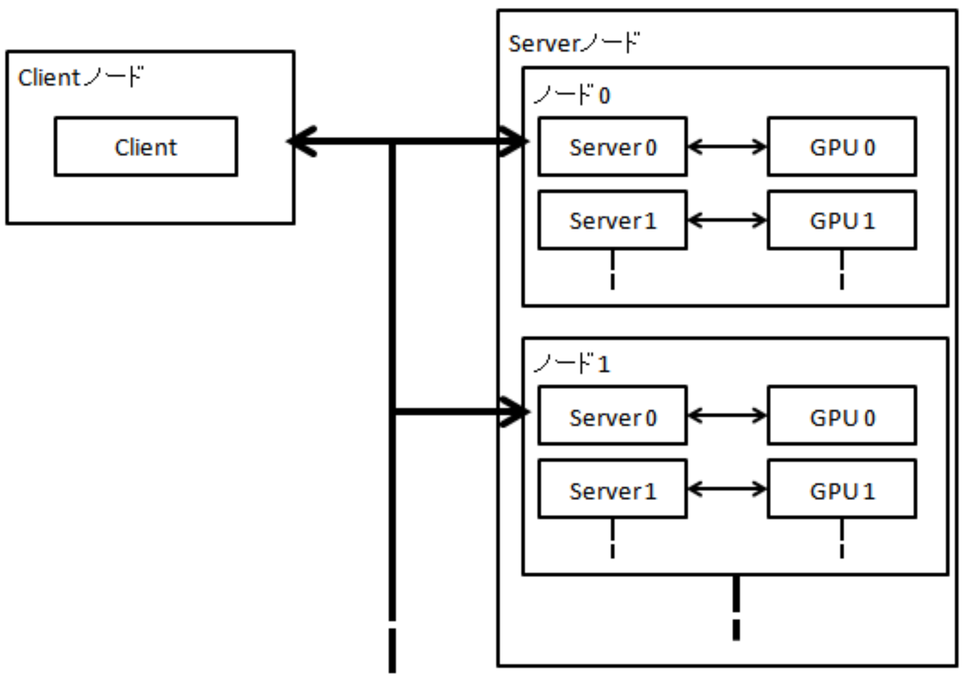

図 2.3: DS-CUDA システムの概略

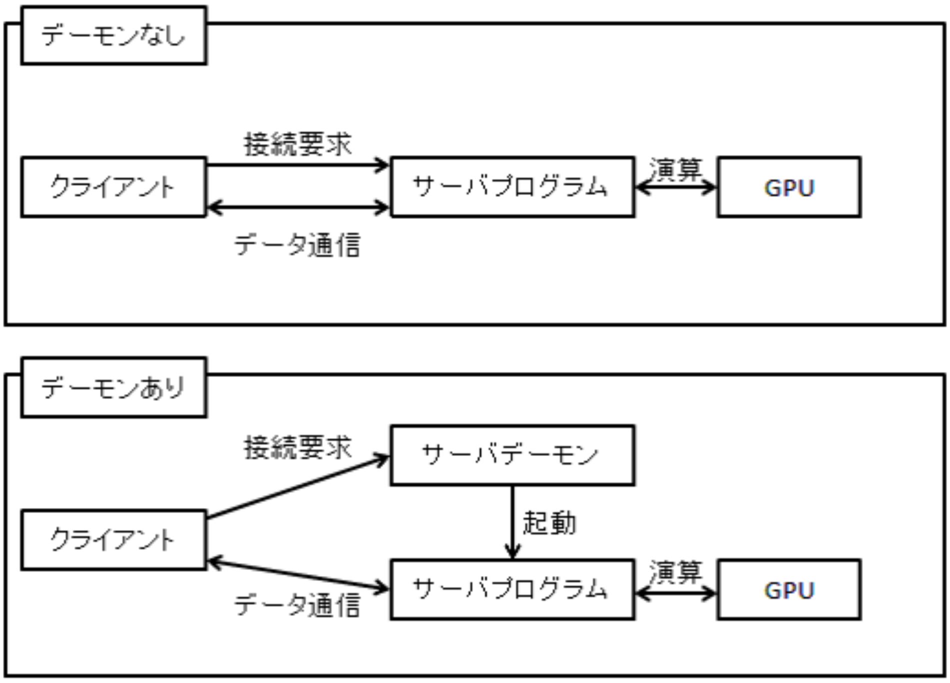

図 2.4: DS-CUDA デーモンの仕組み

DS-CUDA のバージョン 1.3.0 以降では, サーバデーモン機能が搭載された図 2.4. 複数の GPU を搭載する server ノードでも, 単一のサーバデーモンを起動することで, 複数の GPU を要求するクライアントに対し, 自動的にサーバプログラムを起動する. クライアントプログラムが終了すれば, サーバプログラムも自動的に終了する.

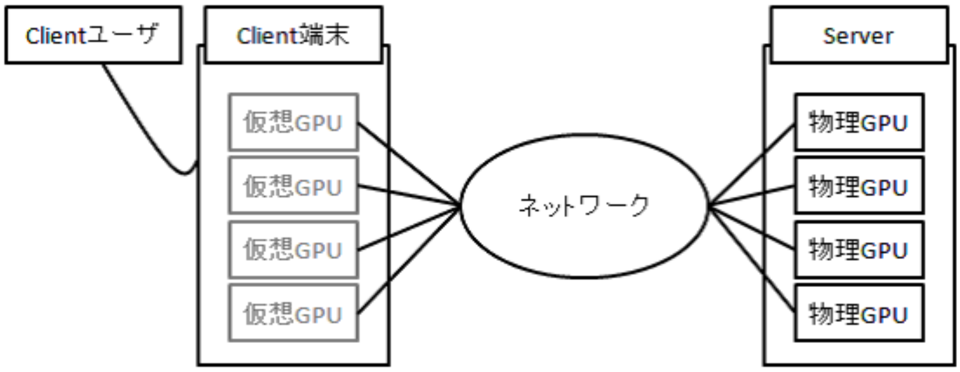

図 2.5: 仮想 GPU の仕組み

client ノード上のアプリケーションプログラムからは, いずれの server ノード上の GPU もクライアント上に搭載されているように見える. ネットワーク上の server ノードに搭載されている物理 GPU に対して, DS-CUDA システムが用意する仮想 GPU を通じてアクセスする (図 2.5). これにより, ユーザはローカルな GPU を用いるかのようにネットワーク上の GPU にアクセス, 実行ができる.

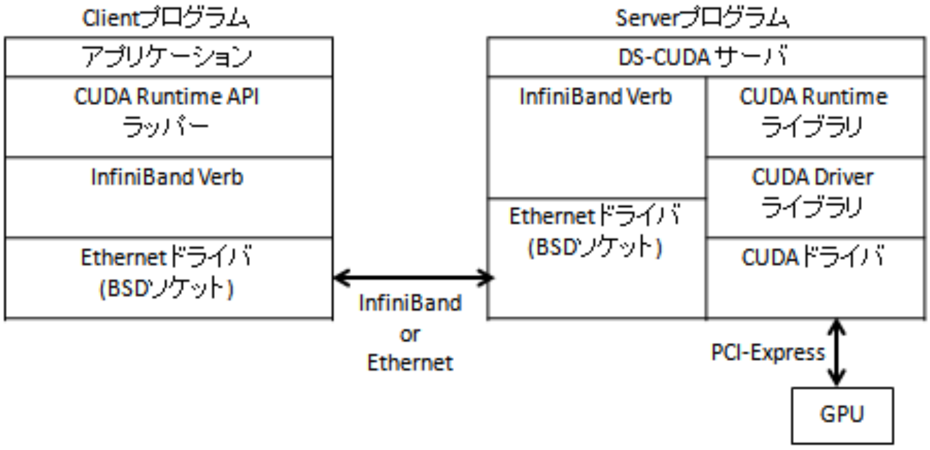

図 2.6: DS-CUDA システムの階層図

DS-CUDA システムの階層図を図 2.6 に示す. DS-CUDA コンパイラはクライアントプログラムに対し, CUDA API ラッパー, InfiniBand Verb, InfiniBand ドライバといった client-server 間通信の為ラッパーを付加してコンパイルを実行する. 通信部分はラッパーに記述, 隠蔽される為, プログラマはネットワークを介した GPU の利用を気にせずに, 同一の CUDA ソースコードを利用できる.

DS-CUDA は Infiniband ネットワークの利用を想定しているが, GigabitEthernet ネットワークの利用にも対応しており, コンパイラは同様の処理を施すことによってプログラマはネットワークを意識せずに同様の CUDA ソースコードを利用できる.

2.4　PCI Express 拡張 Box

DS-CUDA と比較の為に,client ノードから直接複数の GPU を繋ぐ仕組みが必要である. 本研究では PCI Express 拡張 box[7] を利用した.PCI Express 拡張 box は PCI-Express バスに挿すホストカード, GPU コンテナ box, 両者を繋ぐケーブルから構成される (図 2.7).

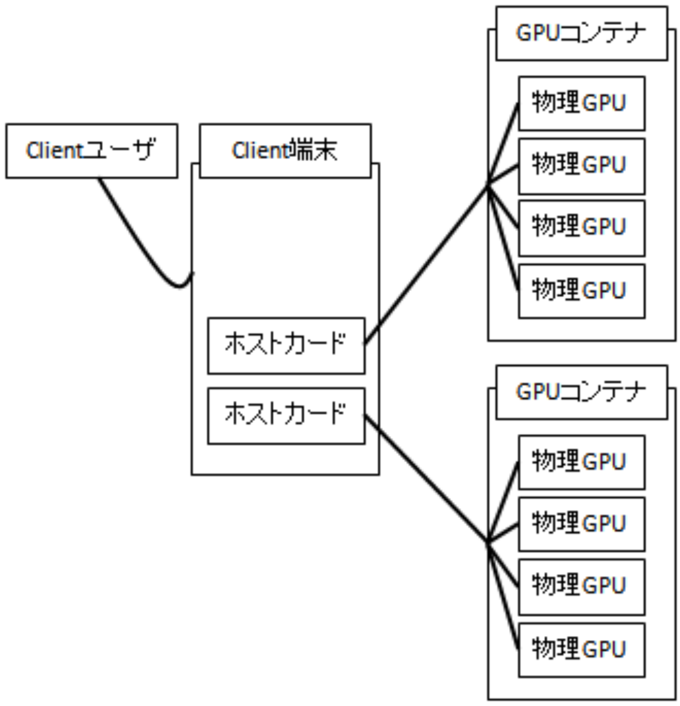

図 2.7: 本研究における PCI Express 拡張 Box の接続

第3章　関連研究

本章では複数の仮想GPUを用いた性能検証についての関連研究について述べる.

3.1　DS-CUDAの提案と性能評価の研究

川井らは, Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability の中でDS-CUDAの開発,提案と共に仮想GPUの性能の評価を行っている.[1]

通信手法としてPCI Express(ネイティブなCUDA), Infiniband,Gigabit Ethernet,WANを取り上げ, 転送速度の評価を行った. また,GPUの並列数に対する並列効率

$$並列効率 = \frac{\text{GPU1台の時の計算時間}}{\text{GPU並列数}\ (N_{gpu}) \times \text{GPU}N_{gpu}\text{台の時の計算時間}} \tag{3.1}$$

の評価を行い, 複数台の仮想GPUを用いたGPGPUの有用性を提案した.

川井らはレプリカ交換分子動力学シミュレーション (Replica-Exchange MD, REMD) を題材に, 通信の性能とGPUの並列数に対する並列効率を主に性能評価した. 一方で, 通信と計算に関わるデータ量 (粒子数) についてはサンプルが少なく, またREMDの性質上粒子数が固定ならGPUの並列数に拘らず1GPUが担当する計算の負担は一定 (Weak Scaling) という性質がある.

本研究では, より一般的な計算にモデルを与えることを目的として, GPU並列数と共にデータ量 (粒子数n:MDシミュレーション, 行列の次数m:行列の乗算) を変化させていったときのモデルを対象とする. 計算時間がそれぞれ$O(n^2)$,$O(m^3)$である計算を複数のGPUに分割したときの評価として一般化した. これらの計算は,GPUの並列数を大きくすることで, GPU1台あたりの計算負荷が小さくなる (Strong Scaling) という特徴を持っている.

3.2 GPU 仮想化技術のデータ転送速度比較の研究

Antonio J. Peña らは, A Complete and Effcient CUDA-Sharing Solution for HPC Clusters の中で rCUDA[9] や DS-CUDA 等のリモート GPU を扱う技術に対する, 通信スループットとレイテンシの比較を行っている [8].

本研究では, 通信によるボトルネックが計算性能に及ぼす影響を, モデル式と実測値を比較しながら議論してゆく.

3.3 REMD における大並列仮想 GPU 利用の研究

老川らは, 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にて, レプリカ交換分子動力学シミュレーション (Replica-Exchange MD, REMD) において, 最大 1,024 個の仮想 GPU を使用して演算を行い, 性能を評価している [10]. REMD では, 温度等のパラメータが異なる複数の MD 系 (レプリカ) を構成し, 同時に MD Step を実行, 条件に応じて最適解を求める為にレプリカの交換を行う. 実験の結果,1,024 並列時において効率 87% の結果を得, また GPU 並列数大の際に, ノード間通信のレイテンシが支配的になることを明らかにした.

REMD は GPU 内での計算量が大きく,GPU-GPU 間 (レプリカ間) の通信量が比較的小さいという特徴があり, 仮想 GPU の性能が発揮できることの提案を研究の主旨としている. 本研究では, より汎用的な計算に対する仮想 GPU の利用に貢献するための指標を提案するために, GPU 並列数や計算量を盛り込んだモデルの構築を目的としている.

第4章　システム

本章では目的である計算性能モデルの構築の為の実験で利用したシステム, ハードウェアについて述べる.

4.1　Client 端末

本研究の実験で DS-CUDA を用いた仮想 GPU, または PCI Express 拡張 Box を用いた物理 GPU に対して計算命令を行う client(host) 端末の環境は以下の通りである.

- CPU:Intel(R) Core(TM) i7 920 2.67GHz
- メモリ:5.9GiB
- ネットワーク:Infiniband または Gigabit Ethernet
- OS:Fedra 14 (カーネル:Linux 2.6)

4.2　GPU

本研究では両手法にて用いる GPU として, ELSA GeForce GTX 780[11] を最大 8 台まで用いた.

4.3　CUDA

本研究で用いた CUDA のバージョンは,CUDA6.0 である.

4.4 DS-CUDA

本研究では,DS-CUDA のバージョン 2.4.0 を用いて, 1 台の GPU を搭載した server ノードを 8 つ用意し, 8 台の GPU(server ノード) と client ノードに対して Infiniband ネットワークを構築し実験を行った. また同様に,Gigabit Ethernet ネットワークを同様に構築し, 実験を行った.

4.5 PCI Express 拡張 box

本研究では,2 機の PCI Express 拡張 box[7] を Client 端末と接続して, 図 2.7 のように 8 台の GPU を用いる.

第5章　MDシミュレーションのモデル化

本章では分子動力学 (Molocular Dynamics:MD) シミュレーションにおける GPGPU の性能測定結果と, そのモデル化について述べる．MD シミュレーションの題材として用いる Claret の説明を始めに述べる.

5.1　Claret

Clret は福井大学の古石によって作られた融解塩の分子動力学 (MD) シミュレーションプログラムである [12]. 例として, 図 5.1 は粒子数 $n = 1000$ でのシミュレーションの様子である．本研究では Claret を用いて性能比較, モデル構築を行う．オリジナルコードに対し, 複数 GPU への対応を行い, OpenGL による画面描画機能は単純化の為無効化した.

測定に利用した Claret プログラムの疑似コードを付録 1 に, 付録 1 内のソースコードから呼び出している Kernel 関数を付録 2 に記載する.

付録 1 を元に, 簡単に Claret の仕組みを説明する．8 行目の `CPUCalc1(...)` では, 初期化処理や粒子情報の管理が行われる.

12 行目の `MemcpyH2D(vec[n], n)` は host(client 端末) から GPU に対する転送命令であり, 粒子の座標と種類のデータである `vec[n]` を全粒子数分 (n) 転送している．10 行目から 13 行目のループは複数 GPU 対応の為追加した部分である, `SetDevice(d)` は d 番目の GPU に対し, データ転送命令を行うよう切り替える命令である．これをループさせることで全ての GPU に転送を行っている．このことは,`kernel` 命令についても同様である.

17 行目の `Kernel<<<n/N_gpu, スレッド数>>>`は GPU に対する Kernel 実行命令である．15 行目から 18 行目のループは複数 GPU 対応の為追加した部分であり, GPU 一つあたり n/N_{gpu} 粒子分の計算を行うスレッドを発行している.

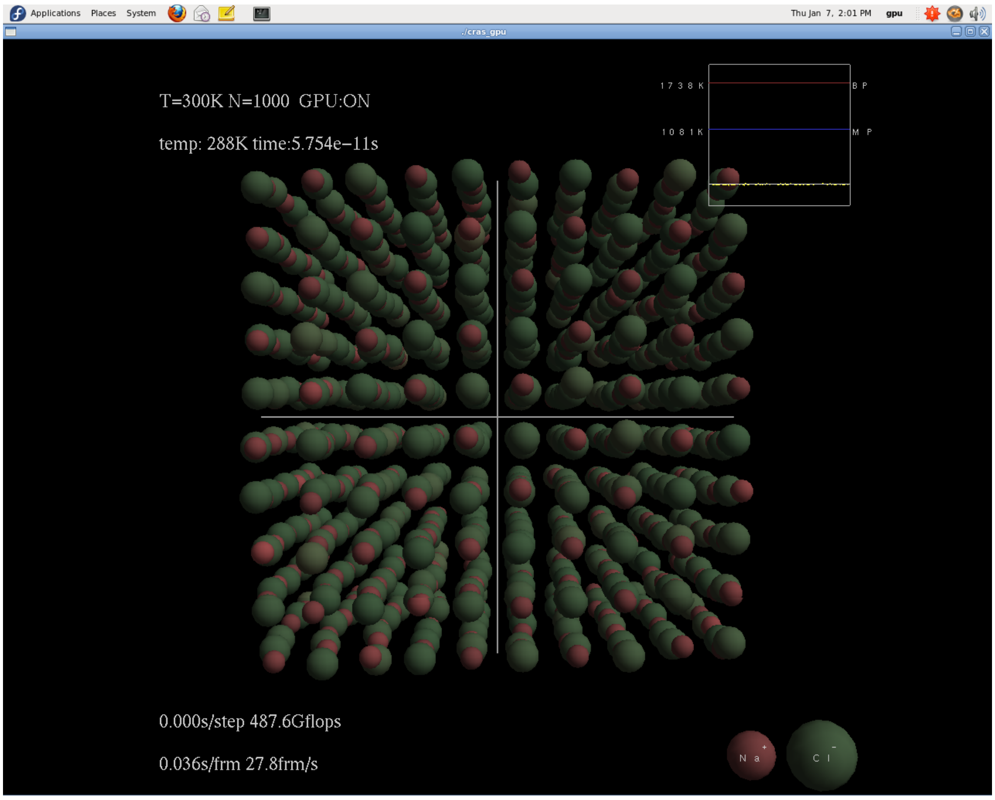

図 5.1: Claret のシミュレーション例 (粒子数 $n = 1000$)

22 行目の `MemcpyH2D(force[n], n/N_gpu)` は GPU から host への転送命令であり, 粒子にかかる力のデータである `force[n]` を転送している. 20 行目から 23 行目のループは複数 GPU 対応の為追加した部分である. 1 つの GPU あたり n/N_{gpu} 粒子分のデータを転送し, 全ての GPU 合わせて全粒子数 (n) のデータを集めている.

粒子数を n とすると 1 回の MDstep に対して CPU-GPU 間の通信量は $O(n)$,GPU での計算量は $O(n^2)$ である. また,GPU-GPU 間の通信は行われない. CPU から全ての GPU に対し全粒子の座標と種類の情報を送り, 計算結果である粒子に加わる力の情報を, 計算を行った GPU から CPU に返す.

5.2 実験結果

粒子数 n,GPU の並列数 N_{gpu}, そして GPU の利用手法を変えながら MD シミュレーションを実行した．シミュレーション結果の評価方法として 1MDStep にかかる計算時間 (sec/step) と, 計算時間と粒子数 n から算出した単位時間あたりの浮動小数点計算の回数として計算性能 (Gflops) を用いている．ただし,1MDStep の計算時間は, 複数 Step のシミュレーションに所要した計算時間から, Step 回数で割ることで求めている. その際, 初回には初期設定が入る為, 2 回目以降の MDstep に対して測定をしている.

計算性能 (Gflops) の計算法を示す．付録 2:11 行の `calcforce` 内での浮動小数点計算の数は 78 回である．従って, ある粒子 (i) に加わる粒子間力の浮動小数点計算の回数は $78n$ 回, 全粒子の浮動小数点計算の回数は $78 \times n^2$ 回である．従って

$$\text{計算性能} = \frac{78 \times n^2}{\text{計算時間 (sec)}} \times 10^{-9}(\text{Gflops}) \tag{5.1}$$

となる．

GPU の並列数 N_{gpu} に対する計算性能 (Gflops) を図 5.2(Infiniband による DS-CUDA と PCI Express 拡張 box との比較), 図 5.3(Infiniband と Gigabit Ethernet(以下, 図中では GbEthernet) との比較) に示す．粒子数が多い時には GPU 数に応じて性能が向上しているが, 粒子数が少ない時は逆に低下している．

GPU の並列数 N_{gpu} に対する 1step あたりの計算時間を図 5.4(Infiniband による DS-CUDA と PCI Express 拡張 box との比較), 図 5.5(Infiniband と Gigabit Ethernet との比較) に示す．十分粒子数が大きい時は, 計算時間は GPU の並列数 N_{gpu} に反比例する．粒子数が 27,000 の時には GPU が 2〜4 の間で計算時間の性能は打ち止めになり , GPU を 8 台用いた場合には計算時間が大きくなる．

粒子数が小さい領域で,PCI Express 拡張 box を用いた物理 GPU に対し, DS-CUDA を用いた仮想 GPU の計算時間が大きいことがわかる．また, 粒子数が小さい領域では,GPU の並列数が大きい程, DS-CUDA の計算時間が大きくなることが分かる．

また, 同じ仮想 GPU でも Infiniband と GigabitEthernet との性能差は, Infiniband と PCI Express 拡張 box との差よりも大きいことがわかる．

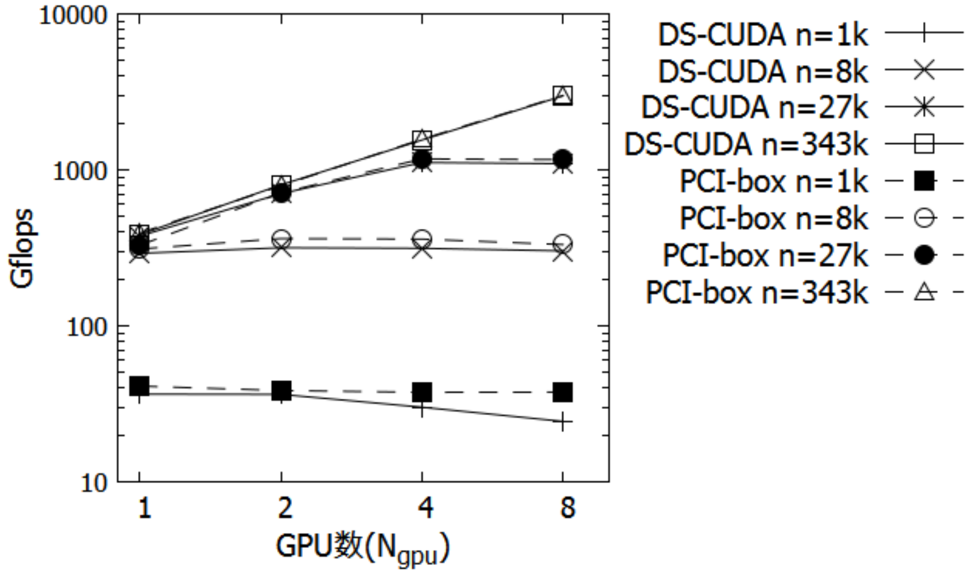

図 5.2: GPU 数に対する計算性能 (DS-CUDA(Infiniband)-PCI Express 拡張 box)

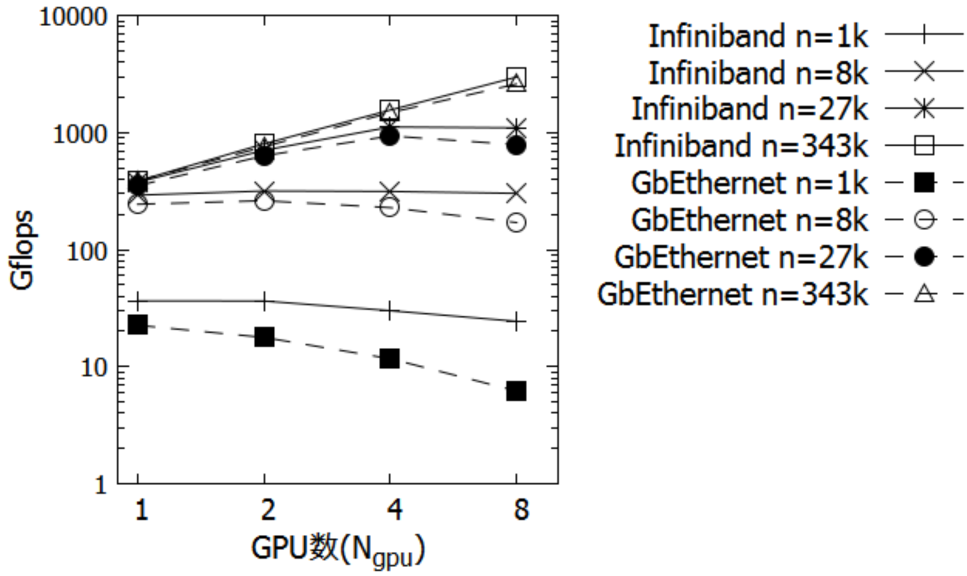

図 5.3: GPU 数に対する計算性能 (Infiniband-Gigabit Ethernet)

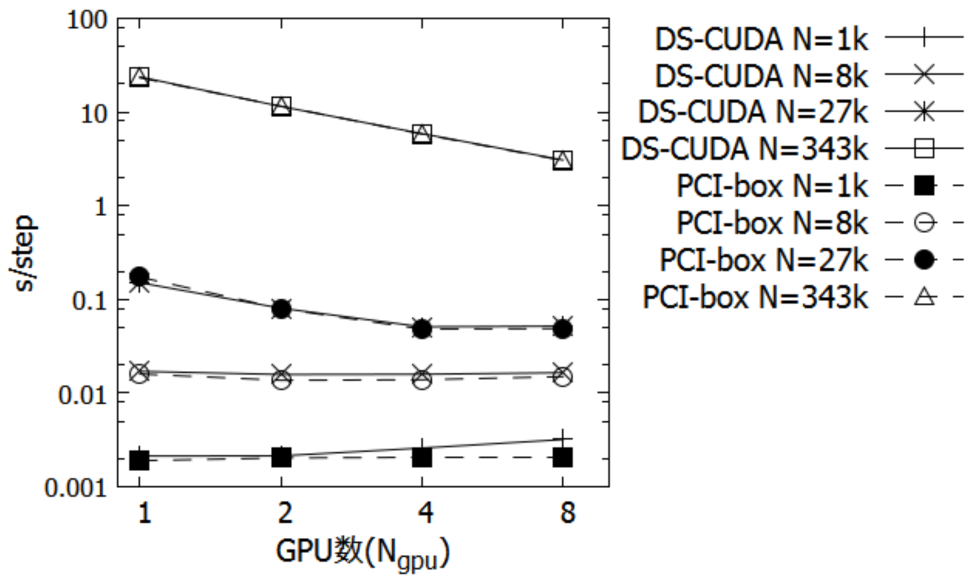

図 5.4: GPU 数に対する計算時間 (DS-CUDA(Infiniband)-PCI Express 拡張 box)

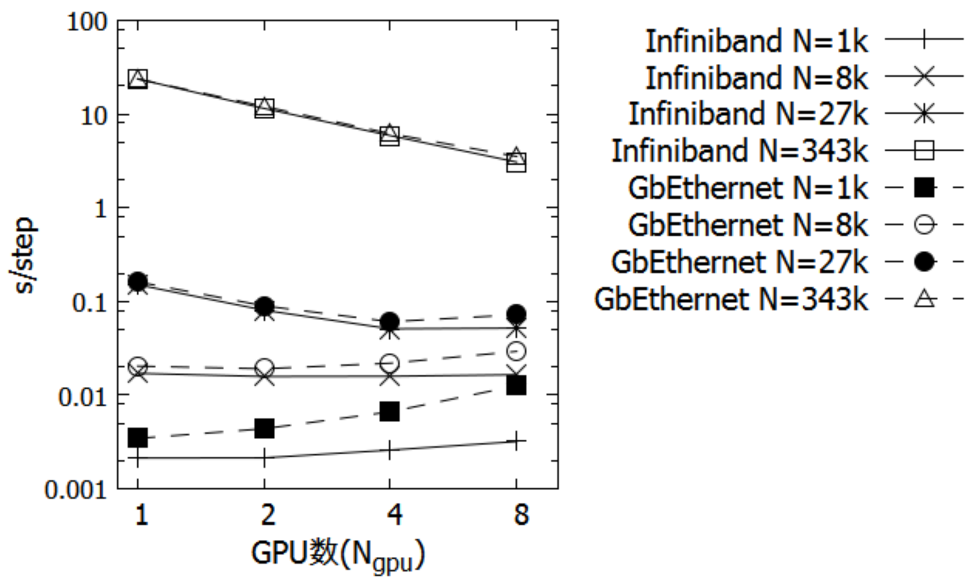

図 5.5: GPU 数に対する計算時間 (Infiniband-Gigabit Ethernet)

粒子数 n に対する計算性能を図 5.6(Infiniband による DS-CUDA と PCI Express 拡張 box との比較), 図 5.7(Infiniband と Gigabit Ethernet との比較) に示す. GPU の並列数 N_{gpu} によって計算性能のピークとピークに達する粒子数 n が共に増加しているのが分かる.

また, 粒子数 $n = 1,000$ に注目すると, PCI Express 拡張 box による物理 GPU よりも DS-CUDA による仮想 GPU の方が性能が低く, GPU 並列数による性能低下への影響も大きいことが分かる.

粒子数 n に対する計算時間を, 図 5.8(Infiniband による DS-CUDA と PCI Express 拡張 box との比較), 図 5.9(Infiniband と Gigabit Ethernet との比較) に示す. 粒子数 n が小さい場合, 計算時間は粒子数に比例して大きくなるが, 十分粒子数 n が大きい場合, 計算時間は粒子数の 2 乗に比例して大きくなる. $O(n)$ から $O(n^2)$ に転じる閾値は,GPU の並列数 N_{gpu} に比例している.

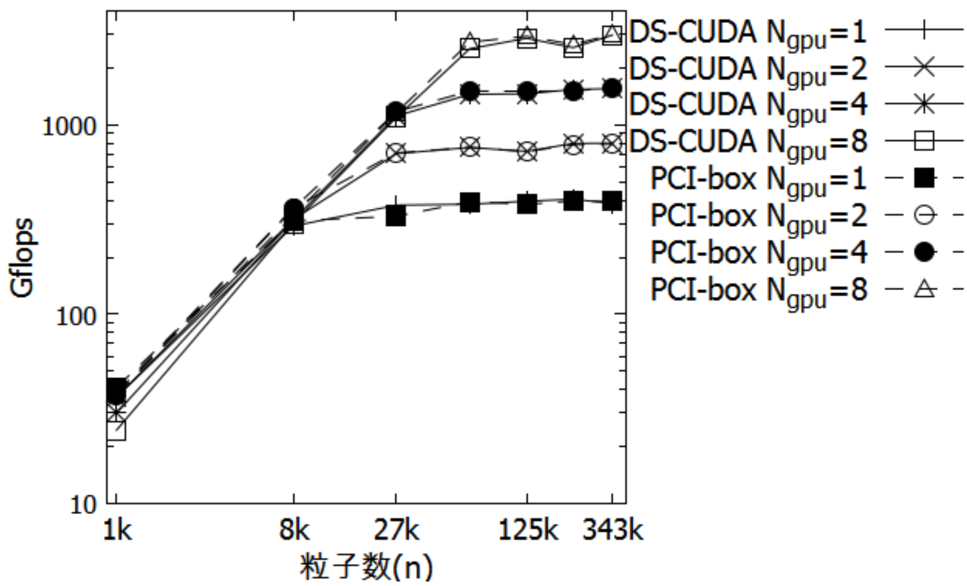

図 5.6: 粒子数に対する計算性能 (DS-CUDA(Infiniband)-PCI Express 拡張 box)

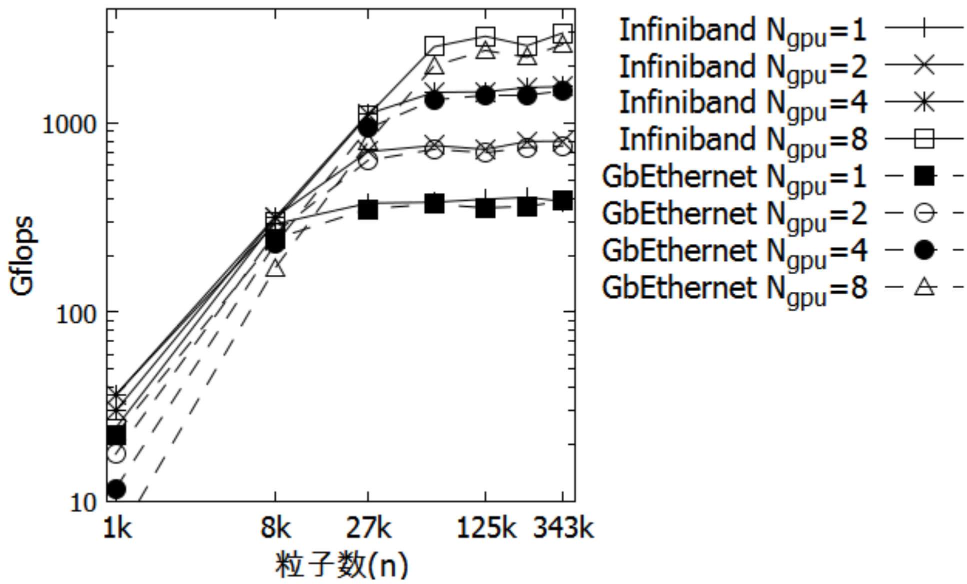

図 5.7: 粒子数に対する計算性能 (Infiniband-Gigabit Ethernet)

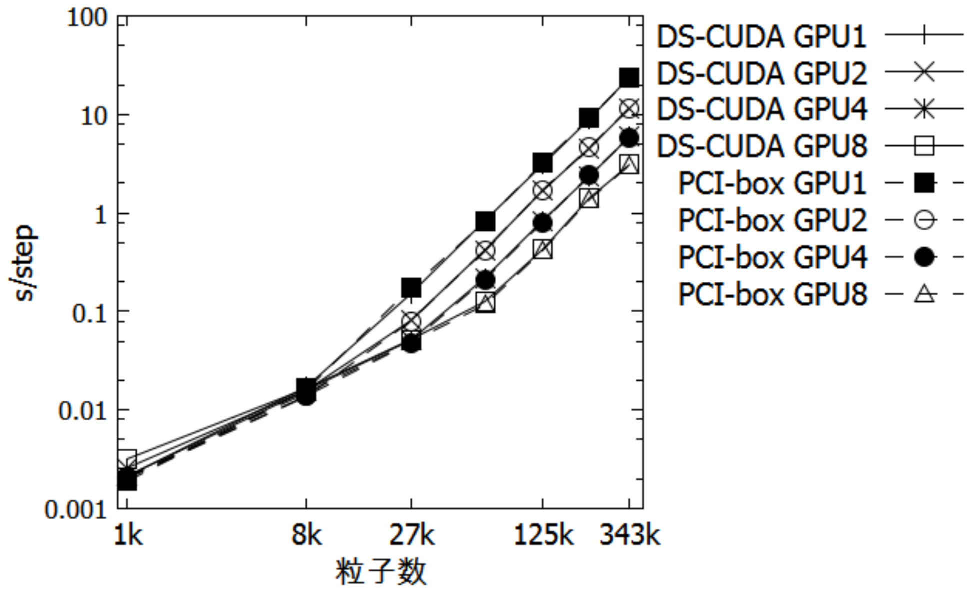

図 5.8: 粒子数に対する計算時間 (DS-CUDA(Infiniband)-PCI Express 拡張 box)

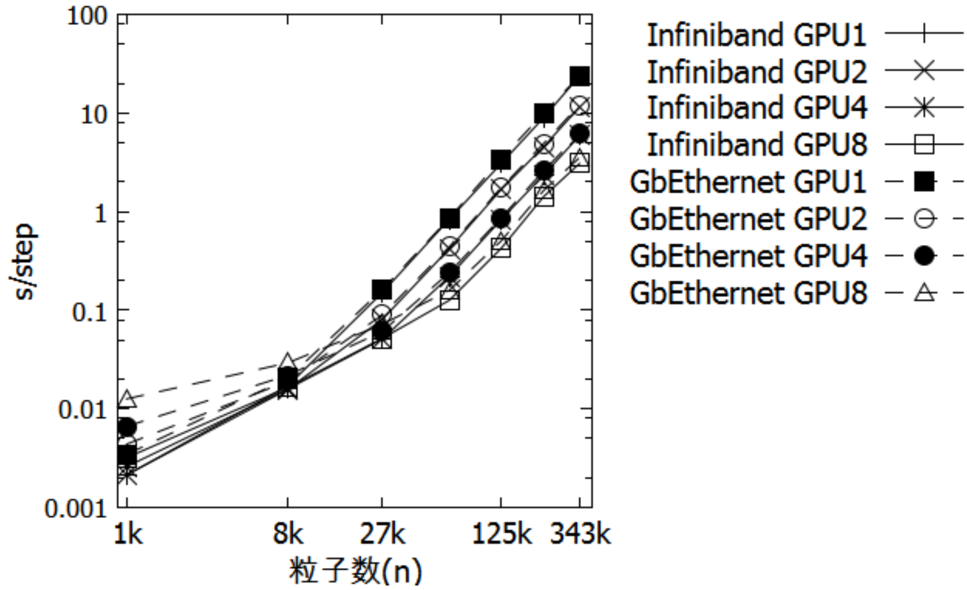

図 5.9: 粒子数に対する計算時間 (Infiniband-Gigabit Ethernet)

5.3 Claretにおけるモデルの構築

本節では計算性能モデル構築の為に必要な各係数を求めるための予備実験の説明と結果を述べ, 実測値とモデルの比較を行い仮想GPUの利用について議論を行う.

1stepの計算時間TをGPUで要する計算時間(T_{gpu}),CPUで要する計算時間(T_{cpu}), server-client間の通信時間(T_{com})の和でモデル化する.

$$T = T_{gpu} + T_{cpu} + T_{com} \tag{5.2}$$

以下の小節で, 各項のモデル式の決定と各定数の決定について述べる.

5.3.1 T_{gpu}のモデル化

T_{gpu}はGPU内での演算に要する計算時間であり, DS-CUDAによる仮想GPUを利用するか, またはPCI Express拡張Boxによる物理GPUを利用するかを問わず,GPUの性能によって等しい値になることを想定している.

T_{gpu}を以下のようにモデル化した. ただし,$\lceil\rceil$は整数値への切り上げを意味する.

$$T_{gpu} = \lceil \frac{n}{kcN_{gpu}} \rceil nkct_{gpu} \tag{5.3}$$

$$\begin{cases} n & : \text{粒子数} \\ N_{gpu} & : \textit{GPU}\text{並列数} \\ t_{gpu} & : 1\text{つの}\textit{GPU}\text{が}1\text{ペアの粒子間の力の計算にかかる時間 (sec)} \\ c & : \textit{GPU}\text{あたりのコア数 (今回は2304)} \\ k & : 1\text{コアが並列処理できるスレッド数} \end{cases}$$

ここでkcN_{gpu}は全GPUで1度に処理できるスレッド数を表している. $c = 2304$は今回用いたGPUであるGeForce GTX 780による. 粒子数nがkcN_{gpu}より小さい場合, 式5.3は

$$T_{gpu} = kct_{gpu} \qquad (n < kcN_{gpu}) \tag{5.4}$$

と表せる. このことは, 粒子数nがkcN_{gpu}より小さい場合, 並列GPUに対し1回のKernel命令が発効されること, そしてKernel関数内にはnに

関する for ループが一つある為 (付録 2 疑似コード 9〜12 行), Kernel 関数の処理時間は $O(n)$ に比例することに由来する.

一方, 粒子数が十分大きく,Kernel 命令の発行回数が $O(n)$ に比例すると見なせる場合, 式 5.3 は以下の式で近似出来る.

$$T_{gpu} = \frac{n^2 t_{gpu}}{N_{gpu}} \qquad (n >> kcN_{gpu}) \tag{5.5}$$

ここで, 決定すべき係数は k,t_{gpu} である.

t_{gpu} の決定は次の方法による. 十分大きな粒子数 n に対し, 1 台の GPU に対して Claret シミュレーションを実行する. 付録 1 の 17 行目の kernel 実行の後にスレッド同期命令を挿入し, kernel 実行の前からスレッド同期命令終了までの時間を測定する. 幾つか粒子数 n を取り, また DS-CUDA と PCI Express 拡張 Box の両手法を用いた場合においてデータをプロットし, 式 5.5 に基づいて最小二乗法で t_{gpu} を決定した.

k の決定は式 5.4 より次の方法による. 粒子数 n の小さい領域に対して同様にシミュレーションを実行する. 所要した実行時間を粒子数 n で割った値が十分近い領域に対して, さらに先に求めた t_{gpu},c を割り, 最も近い自然数として $k = 4$ を決定した. 決定した係数を表 5.2 に示す.

t_{gpu}	k
1.94×10^{-10}	4

表 5.1: T_{gpu} のモデル式において求めた係数

5.3.2 T_{cpu} のモデル化

T_{cpu} を以下のようにモデル化する.

$$T_{cpu} = nt_{cpu} \tag{5.6}$$

CPU では, 主にデータの初期化と,kernel で得られた各粒子にかかる力を元に座標情報の更新を行っている. 従って, 計算時間は粒子数 n に比例すると考えられる. 未知の係数は比例定数 t_{cpu}(sec) である. 付録 1 の 12,22 行目の client-server 間のデータ転送と,17 行目の Kernel 関数をコメントアウトした Claret プログラムに対し, 十分大きな粒子数 n を幾つか取って

演算時間を測定し, プロットしたものから最小二乗法によって t_{cpu} を決定した.

本研究では同一の client 端末を用いている為, T_{gpu} と同様に DS-CUDA と PCI Express 拡張 Box の両手法の違いによらない等しい値になることが望ましい. しかし, コンパイルオプションを同一のものにすることができなかった為, t_{cpu} の値が異なっている. 一方で,DS-CUDA を用いた手法のうち, 通信に Infiniband を用いた場合と GigabitEthernet を用いた場合とでは有意差は見られなかった為, 同一の値を用いる. T_{cpu} はモデルが比較的簡単な形で信頼性が高いこと, 総和 T の議論において本質的ではなく除きうることからこのまま議論を進めている.

5.3.3 T_{com} のモデル化

T_{com} を以下のようにモデル化する.

$$\begin{aligned} T_{com} &= \left(\frac{16nN_{gpu}}{B_{h2d}} + N_{gpu}L_{h2d}\right) \\ &+ \left(\frac{12n}{B_{d2h}} + N_{gpu}L_{h2d}\right) + t_{kernel}N_{gpu} \end{aligned} \tag{5.7}$$

T_{com} は client(host) から server(device) へのデータ転送 (付録 1:12 行目) にかかる時間 T_{h2d}, device から host へのデータ転送 (付録 1:22 行目) にかかる時間 T_{d2h}, そして各 GPU への kernel 命令の発行 (付録 1:17 行目) にかかる時間 T_{kernel} の和として, 以下のように表せる.

$$T_{com} = T_{h2d} + T_{d2h} + T_{kernel} \tag{5.8}$$

ここで T_{h2d} は転送データ量 S_{h2d}(bytes), 転送スループット B_{h2d}(sec/bytes), 転送レイテンシ L_{h2d}(sec), GPU 並列数 N_{gpu} を用いて

$$T_{h2d} = \frac{S_{h2d}}{B_{h2d}}N_{gpu} + L_{h2d}N_{gpu} \tag{5.9}$$

	DS-CUDA	PCI-Ebox
t_{cpu}	7.95×10^{-8}	5.04×10^{-8}

表 5.2: 実験で求めた t_{cpu}

と表せる. ここで付録 1 における転送データ `vec` は粒子一つあたり `float` 変数による xyz 座標と `int` 変数による粒子の種類による 16byte の大きさを持つ. したがって $S_{h2d} = 16n$ を代入することで, 式 5.7 の第一項を得る.

$$T_{h2d} = \left(\frac{16nN_{gpu}}{B_{h2d}} + N_{gpu}L_{h2d} \right) \tag{5.10}$$

T_{d2h} においても同様に, 付録 1 における転送データ `force` は粒子一つあたり `float` 変数による力の xyz 成分をもち. 12byte の大きさを持つ. 各 GPU からは $\frac{n}{N_{gpu}}$ 個分のデータを転送する為, $S_{d2h} = 12n/N_{gpu}$ を代入することで, 式 5.7 の第二項を得る.

$$T_{d2h} = \frac{12n}{B_{d2h}} + N_{gpu}L_{h2d} \tag{5.11}$$

また,T_{kernel} について,kernel 命令を GPU の数だけ発行することから, kernel 命令を 1 回発行するのに必要な時間を t_{kernel} とすれば

$$T_{kernel} = t_{kernel}N_{gpu} \tag{5.12}$$

と表せる.

求めるべき係数は B_{h2d},L_{h2d}, B_{d2h},L_{d2h},T_{kernel} である. 以下導出方法を述べる.

- B, L の導出

一例として,DS-CUDA における B_{h2d},L_{h2d} の算出方法を述べる. $x$bytes のデータを host から device へ転送する CUDA プログラムを実行し, 転送時間をプロットする. プロット結果から最小二乗法で一次関数を求める (図 5.10). 切片を L_{h2d}、傾きを $1/B_{h2d}$ とする. L_{d2h}, B_{d2h} について, また PCI Express 拡張 Box についても同様である.

- T_{kernel} の導出

付録 1:17 行目で呼び出す kernel 関数の代わりに, 引数だけ受け取りほぼ何も実行しないダミー kernel 関数を実行する. t_{gpu} の導出と同様に, スレッド同期命令挿入し, ダミー kernel 関数の実行からスレッド同期命令の終了までの時間を測定する. 得られた時間を t_{kernel} とする.

得られた各係数を表 5.3 に示す.

	DS-CUDA(Infiniband)	DS-CUDA(GigabitEthernet)	PCI-Ebox
B_{h2d}	1.84×10^9	1.89×10^8	4.98×10^9
L_{h2d}	9.36×10^{-6}	1.52×10^{-4}	6.12×10^{-6}
B_{d2h}	1.33×10^9	1.12×10^8	3.56×10^9
L_{d2h}	1.72×10^{-5}	2.78×10^{-4}	1.04×10^{-5}
t_{kernel}	1.02×10^{-4}	1.01×10^{-3}	1.63×10^{-5}

表 5.3: T_{com} における各係数

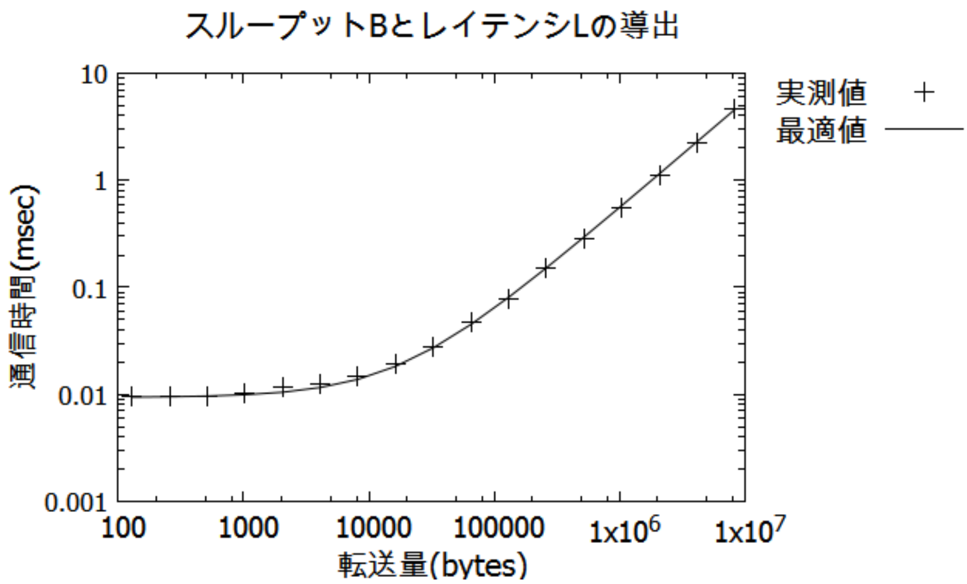

図 5.10: DS-CUDA における B_{h2d}, L_{h2d} の導出例

5.4 Claret のモデル式の考察

Infiniband ネットワーク使用時 DS-CUDA のモデルと実測値の比較を図 5.11 に, GigabitEthernet ネットワーク使用時 DS-CUDA のモデルと実測値の比較を図 5.12 に, PCI Express 拡張 Box のモデルと実測値の比較を図 5.13 に示す. 粒子数の少ない時は粒子数に比例して計算時間が上昇していくが, 粒子数が多い時は 2 乗で増えている.

モデル化の過程で L,$1/B$,t_{kernel}, いずれも DS-CUDA の方が大きく, 小粒子, 並列大領域における PCI Express 拡張 box との差を決定づけていることが分かった.

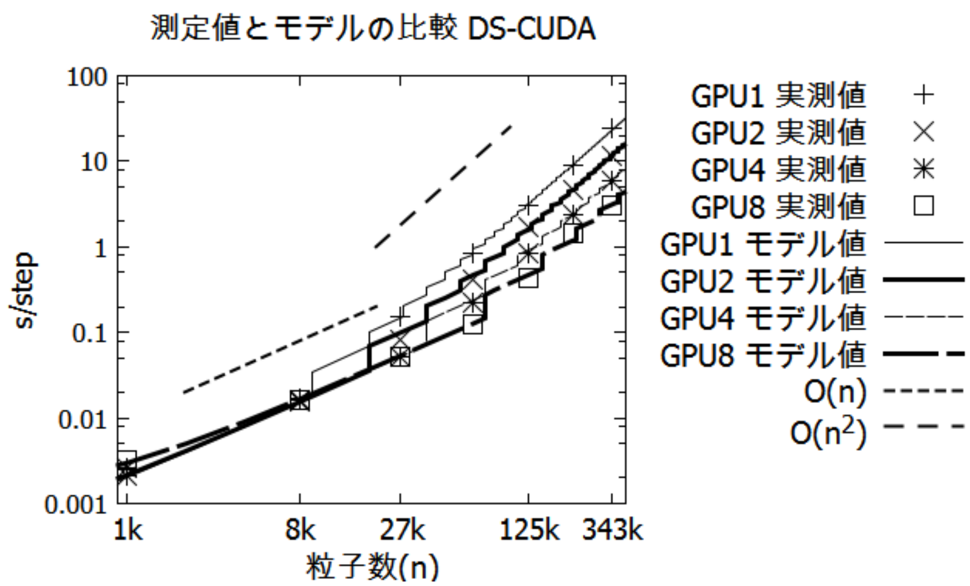

図 5.11: 1step の計算時間の実測値とモデルの比較 (Infiniband の場合)

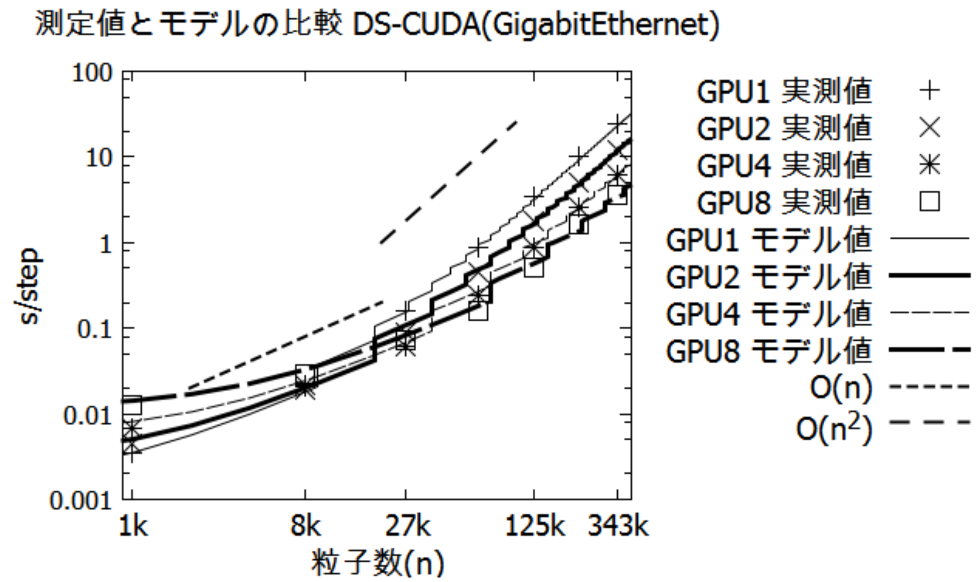

図 5.12: 1step の計算時間の実測値とモデルの比較 (GigabitEthernet 拡張 Box の場合)

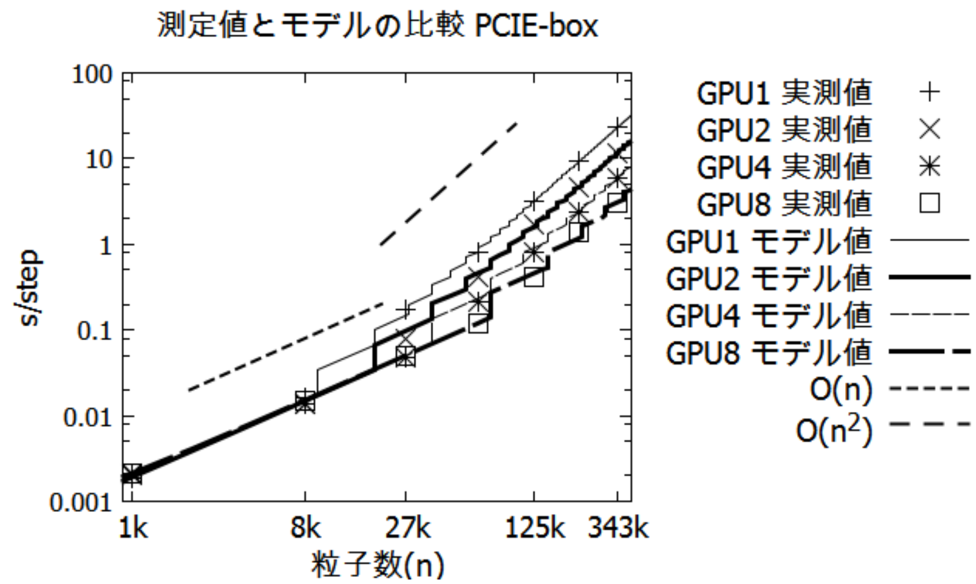

図 5.13: 1step の計算時間の実測値とモデルの比較 (PCI-Express 拡張 Box の場合)

また, 実測値 T_{ex} とモデル式の値 T_{md} とした時, 誤差を

$$\text{error} = \frac{T_{ex} - T_{md}}{T_{ex}} \tag{5.13}$$

と定義したとき,Infiniband ネットワークを用いた DS-CUDA における誤差 error を図 5.14 に, GigabitEthernt ネットワークを用いた DS-CUDA における誤差 error を図 5.15 に, PCI Express 拡張 Box における誤差 error を図 5.16 に示す. 誤差は 25%以下である. また, 全ての手法において $(N_{gpu}, n) = (2, 27k), (2, 125k), (8, 216k)$ で同様の大きな誤差を示していることがわかる. その為, モデル式, パラメータ双方で調整不足であることは否定できない.

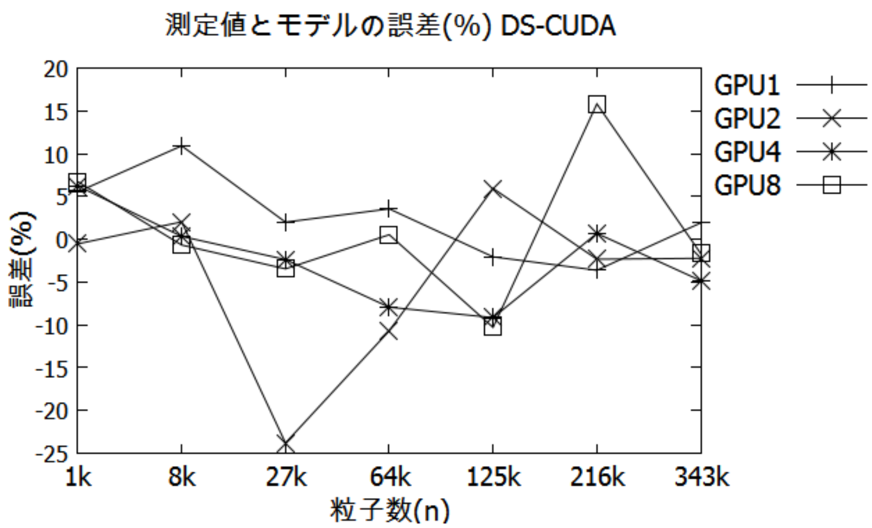

図 5.14: 1step の計算時間の実測値とモデルの誤差 (Infiniband の場合)

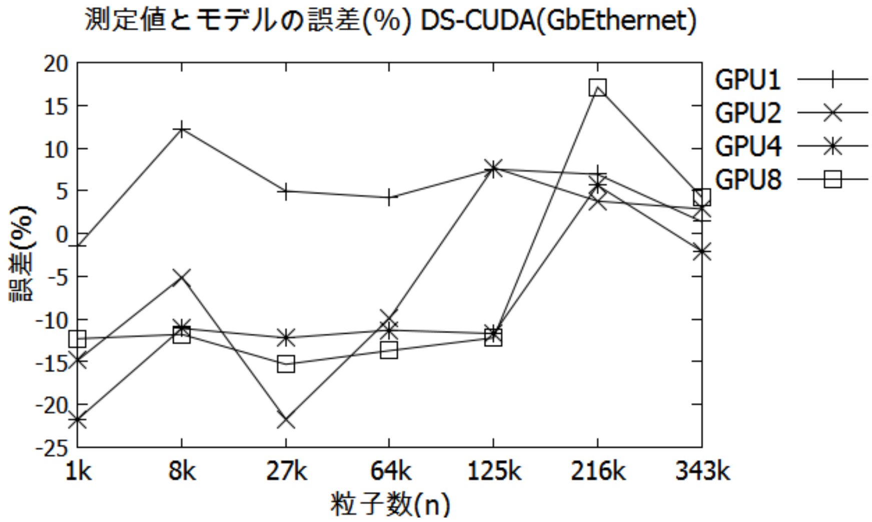

図 5.15: 1step の計算時間の実測値とモデルの誤差 (GigabitEthernet の場合)

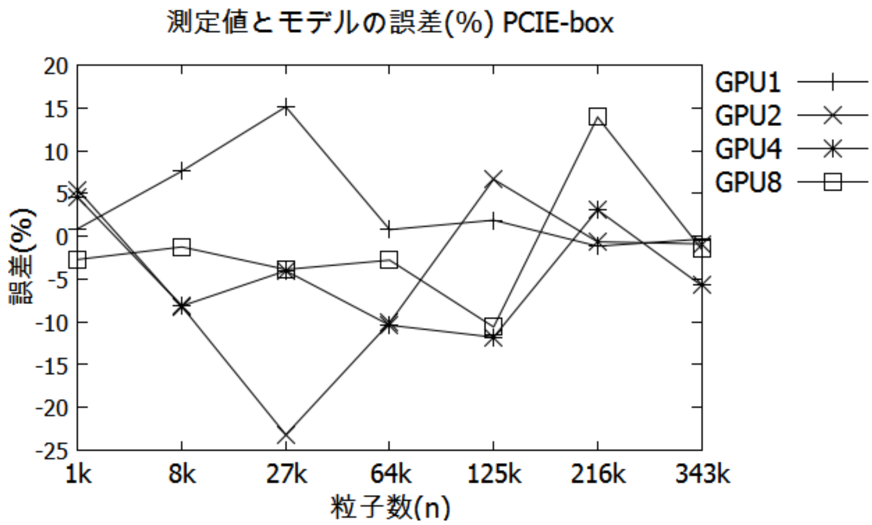

図 5.16: 1step の計算時間の実測値とモデルの誤差 (PCI-Express 拡張 Box の場合)

またモデルにおけるT_{gpu},T_{cpu},T_{com}の比率を図5.17(Infinibandの場合),5.18(GigabitEthernet の場合), 図 5.19(PCI Express 拡張 box の場合) に示す．横軸は GPU の並列数 ($N_{gpu} = 1, 8$), 粒子数 ($n = 1000, 343000$) の違いである．GPU 数が増えると通信時間が増えていること, 粒子数が増えると通信時間が減ることが分かる．

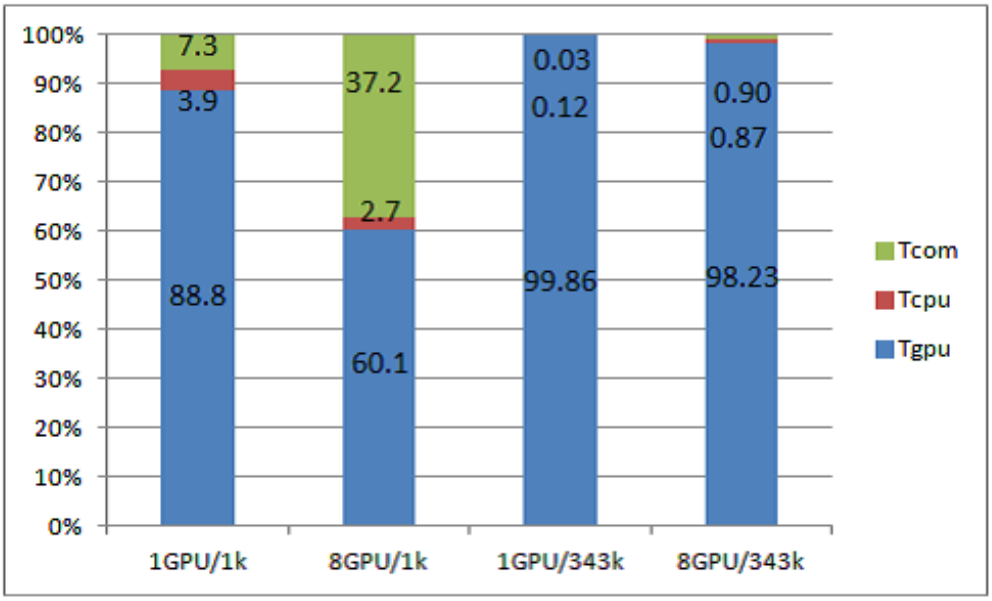

図 5.17: モデルにおける T_{gpu},T_{cpu},T_{com} の比率 (Infiniband)

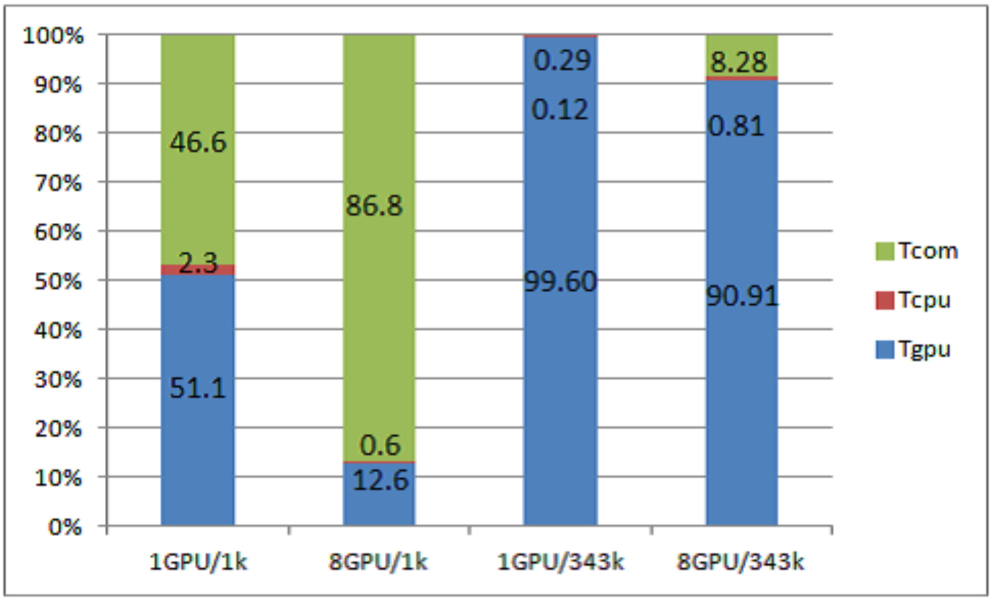

図 5.18: モデルにおける T_{gpu},T_{cpu},T_{com} の比率 (GigabitEthernet)

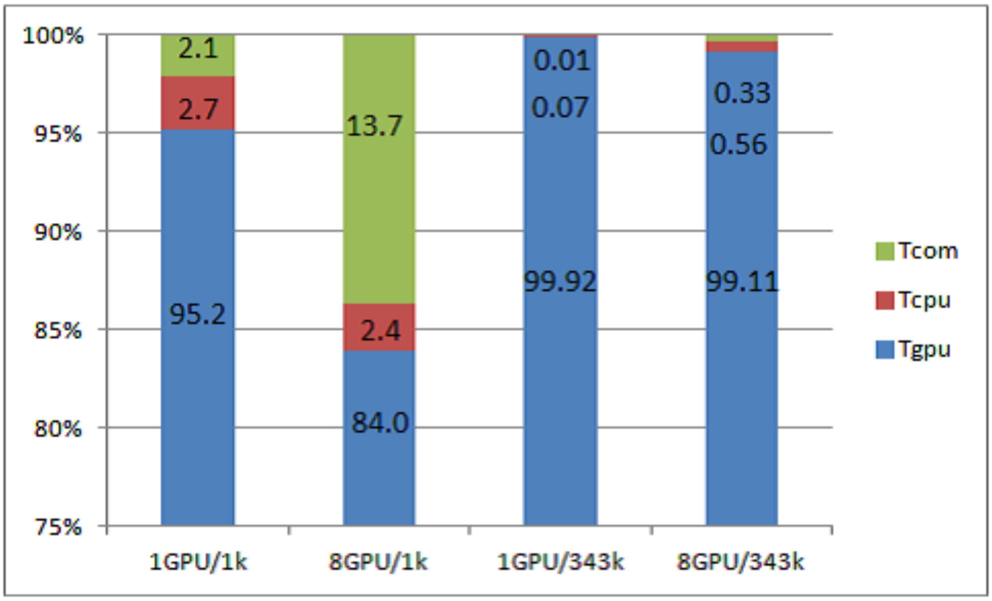

図 5.19: モデルにおける T_{gpu},T_{cpu},T_{com} の比率 (PCI Express 拡張 box

第6章　行列の乗算のモデル化

本章では, 行列の乗算を行うアプリケーションにおける GPGPU の性能測定結果を示し, 前章で述べたモデルと同様に, モデル化を行う.

6.1　行列の乗算

本研究の実験では,$m \times m$ 行列の乗算について扱う. $m \times m$ 行列の乗算を行うプログラムの疑似コードを付録 3 に, その Kernel 関数を付録 4 に記述する.

プログラムの流れについて説明する. host から GPU への転送 (付録 3:9 行目から 13 行目) は全 GPU に $m \times m$ 行列 `A[m*m]`,`B[m*m]` 全要素分行う. Kernel 関数 (付録 3:17 行目) のスレッド数は GPU 一つあたり $m \times m/N_{gpu}$ であり, スレッド一つに行列積 `C[m*m]` の要素一つを充てる. GPU から host への転送 (付録 3:22 行目) では GPU 一つあたり $m \times m/N_{gpu}$ 個の要素を解として受け取る.

$m \times m$ 行列の行列積を求める計算は, CPU-GPU 間の通信量が $O(m^2)$,GPU での計算量は $O(m^3)$ である.

6.2　実験結果

Claret による MD シミュレーションと同様に, 正方行列の次数 m,GPU の並列数 N_{gpu}, そして GPU の利用手法を変えながら行列の乗算を行うアプリケーションを実行し, 要した計算時間を測定した. 評価方法として 1Step にかかる計算時間 (sec/step) と, 計算時間と行列の次数 m から算出した単位時間あたりの浮動小数点計算の回数として計算性能 (Gflops) を用いている.

GPU の並列数によって, 記憶領域の関係から扱える行列の次数 m の最大値が異なり, $N_{gpu} = 1$ では $m = 4,094$, $N_{gpu} = 2$ では $m = 5,192$, $N_{gpu} \geq 3$ では $m = 8,190$ となっている.

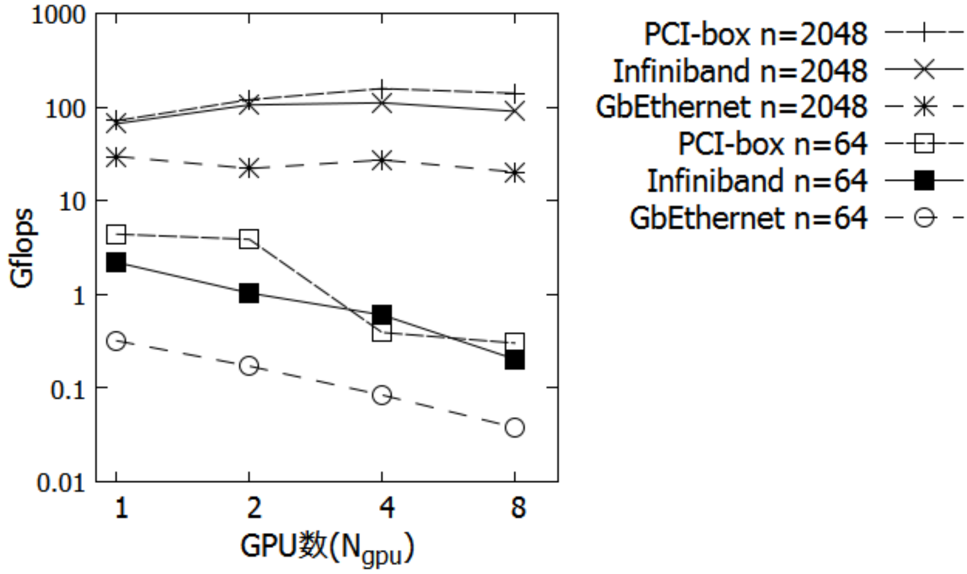

図 6.1: GPU 数に対する計算性能

計算性能 (Gflops) の計算法を示す．付録 4:13 行の要素の積を求める演算の浮動小数点計算の数は 2 回である．従って, ある要素 (i, j) の浮動小数点計算の回数は $2m$ 回, 全要素の浮動小数点計算の回数は $2 \times m^3$ 回である．従って

$$計算性能 = \frac{2 \times m^3}{計算時間\,(\mathrm{sec})} \times 10^{-9}(\mathrm{Gflops}) \tag{6.1}$$

となる．

GPU の並列数 N_{gpu} に対する計算性能 (Gflops) を図 6.1 に示す．$m = 64$ のとき,PCI Express 拡張 Box, DS-CUDA(Infiniband, GigabitEthernet) によらず, GPU の並列数を大きくすると計算性能が小さくなることが分かる．$m = 2,048$ では PCI Express 拡張 Box を用いた場合で $N_{gpu} = 4$ まで性能向上がみられるのに対し, Infiniband による DS-CUDA では $N_{gpu} = 2$ までとなっており, GigabitEthernet による DS-CUDA では性能の向上がみられない.

次数 $m = 64$ における PCI Express 拡張 Box の性能低下には不規則性が見られるが、その要因は分かっていない．後述のモデルと実測値の誤差の原因にもなっており, 正確な測定が求められる.

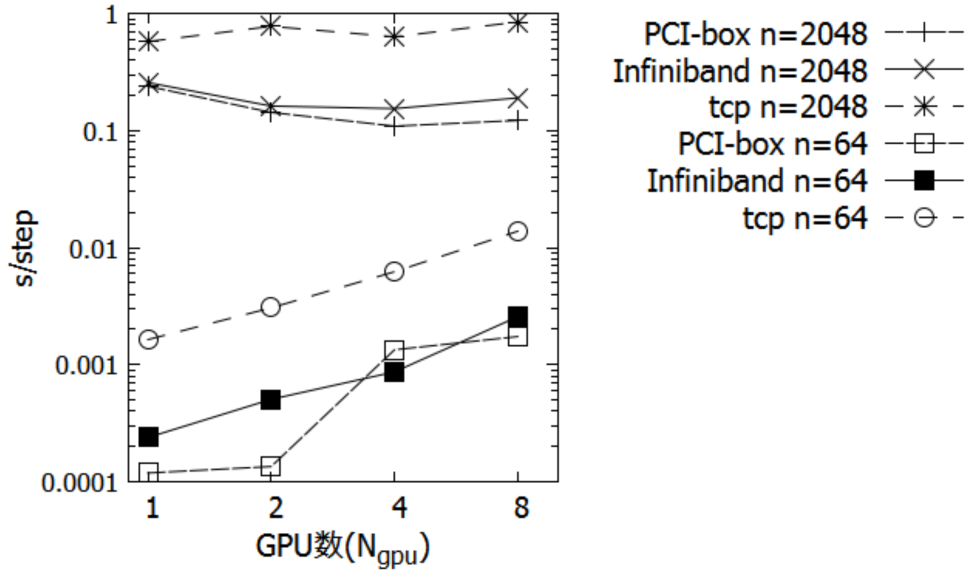

図 6.2: GPU 数に対する計算時間

GPU の並列数 N_{gpu} に対する計算時間 s/step を図 6.2 に示す．PCI Express 拡張 Box の不規則性を除けば，次数 $m = 64$ における計算時間は，GPU の並列数 N_{gpu} に対して線形的に大きくなっている．このことから，並列数による計算能力の増大の影響を受けず，また GPU 内での計算時間の影響に対して，並列数が増加することによる通信時間の増大の影響が大きいことが分かる．GigabitEthernet の場合,Infiniband,PCI Express 拡張 box に見られる傾向とは異なり，次数 m が大のときも並列数 N_{gpu} による性能向上が見られない．

行列の次数 m に対する計算性能を図 6.3,6.4 に，示す．図 6.3 は Infiniband を用いた DS-CUDA と PCI 拡張 Box とによる，仮想 GPU と物理 GPU との比較である．次数 m が大きくなるにつれて，計算性能の傾きが小さくなり，並列数 $N_{gpu} = 1$ では打ち止めになっている．また，次数 m が大きくなるにつれて，GPU の並列数 N_{gpu} が大きくなることによって性能が向上しているのがわかる．

図 6.4 は Infiniband を用いた DS-CUDA と GigabitEthernet を用いた DS-CUDA とによる，仮想 GPU の通信手法の違いによる比較である．

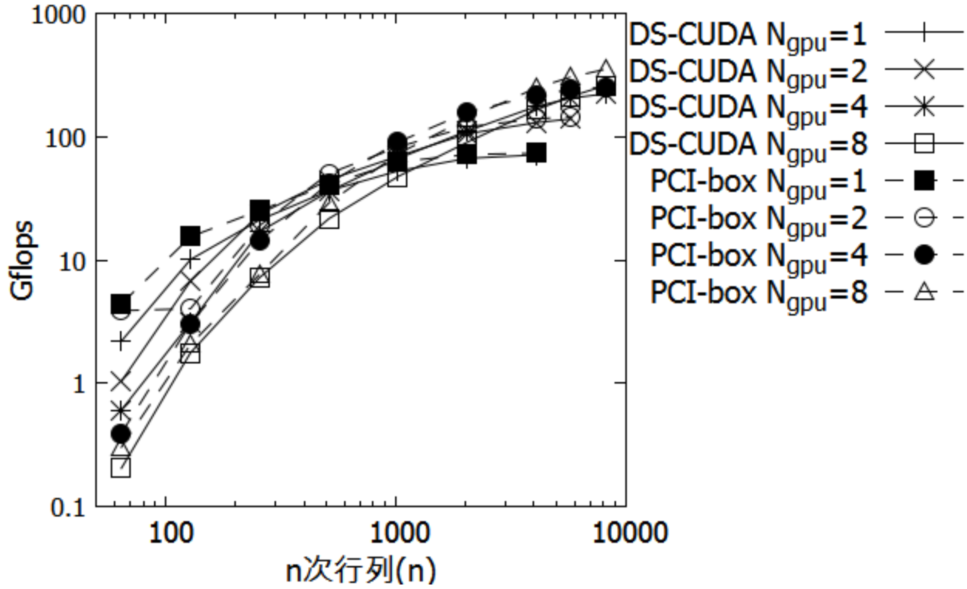

図 6.3: 次数に対する計算性能 (DS-CUDA(Infiniband)-PCI Express 拡張 Box)

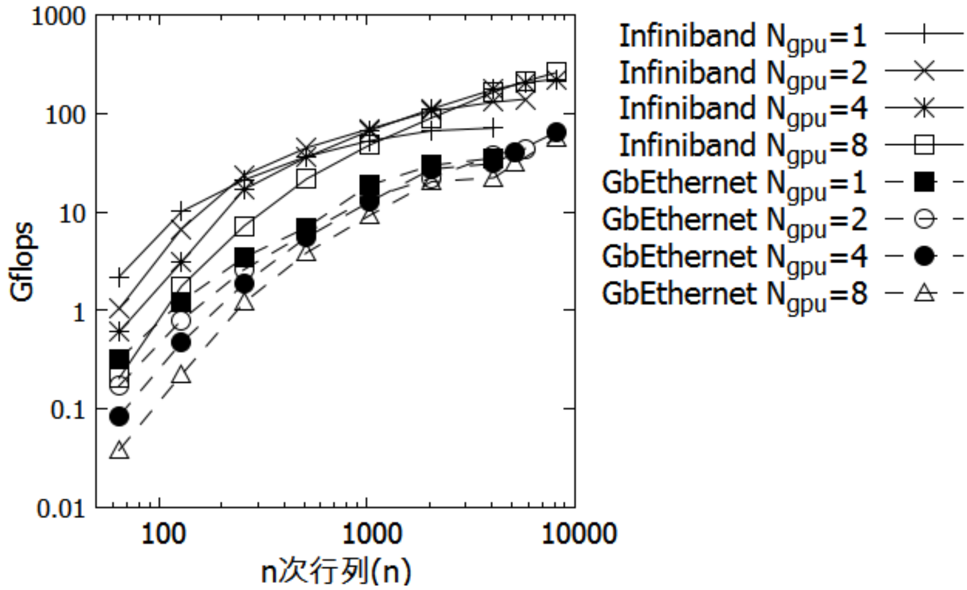

図 6.4: 次数に対する計算性能 (DS-CUDA(Infiniband-GigabitEthernet))

行列の次数 m に対する計算時間を図 6.5,6.6 に, 示す. 行列の次数 m が小さい場合, 次数の増加に対する計算時間の増大が穏やかである特徴が分かる. 行列の次数 m が十分大きい場合, 計算時間は $O(m^3)$ に比例している.

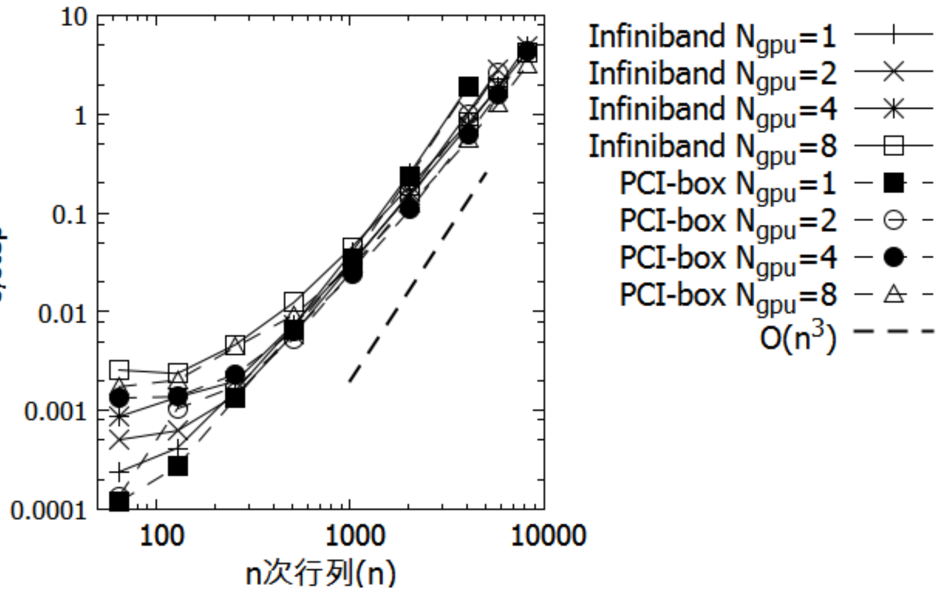

図 6.5: 次数に対する計算時間 (DS-CUDA(Infiniband)-PCI Express 拡張 Box)

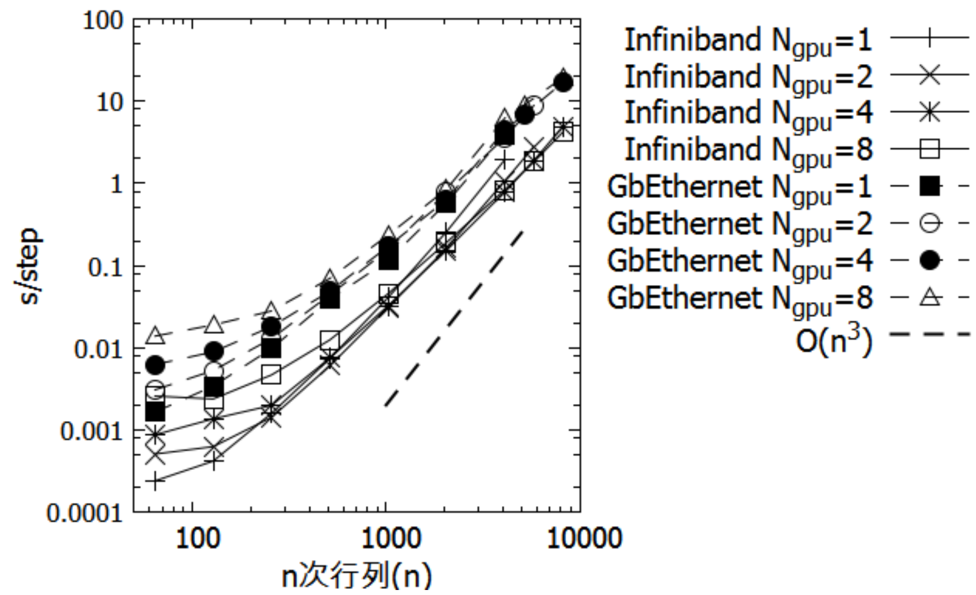

図 6.6: 次数に対する計算時間 (DS-CUDA(Infiniband-GigabitEthernet)

6.3 行列の乗算におけるモデル構築

本研究で構築するモデルには, 一定の普遍性が求められる. 本節では行列の乗算を行うアプリケーションに対し, MD シミュレーションで構築したモデルと同様の手法でモデルを構築し, 実測値とモデルの比較を行い, モデルの普遍性を考慮した議論を行う.

MD シミュレーションと同様に,1step の計算時間 T を GPU で要する計算時間 (T_{gpu}),CPU で要する計算時間 (T_{cpu}), server-client 間の通信時間 (T_{com}) の和でモデル化する (式 5.2). ただし,$\lceil\rceil$ は整数値への切り上げを意味する.

6.3.1 T_{gpu} のモデル化

T_{gpu} を以下のようにモデル化した.

$$T_{gpu} = \lceil\frac{m^2}{kcN_{gpu}}\rceil mkct_{gpu} \tag{6.2}$$

$$\begin{cases} m & : & \text{正方行列の次数} \\ N_{gpu} & : & GPU\ \text{並列数} \\ t_{gpu} & : & 1GPU\ \text{で 1 要素の最小単位の計算 (乗算 1 回と加算 1 回) にかかる時間 (sec)} \\ c & : & GPU\ \text{あたりのコア数 (今回は} 2,304) \\ k & : & \text{1 コアが並列処理できるスレッド数 (今回は 4)} \end{cases}$$

式 6.2 の導出方法は, MD シミュレーションにおける T_{gpu} のモデル式 (式 5.3) と同様である. 違いとして, 切り上げを行う分子が,m から m^2 になっている. これは,1 スレッドが担当するデータが, 1 粒子 (m 個) であるか, 行列の 1 要素 ($m \times m$ 個) であるかの違いである. 要素数 $m \times m$ が kcN_{gpu} より小さい場合, 式 6.2 は

$$T_{gpu} = mkct_{gpu} \qquad (m^2 < kcN_{gpu}) \tag{6.3}$$

と表せる. 粒子数が十分大きく, Kernel 命令の発行回数が $O(m^2)$ に比例すると見なせる場合, 式 6.2 は以下の式で近似出来る.

$$T_{gpu} = \frac{m^3 t_{gpu}}{N_{gpu}} \qquad (m^2 >> kcN_{gpu}) \tag{6.4}$$

ここで,決定すべき係数はt_{gpu}である．t_{gpu}の決定方法はMDシミュレーションと同様である．即ち,付録3の17行目のkernel実行にかかる時間を測定し, 式6.5の値になった.

$$t_{gpu} = 2.47 \times 10^{-11} \tag{6.5}$$

6.3.2　T_{cpu}のモデル化

T_{cpu}を以下のようにモデル化する.

$$T_{cpu} = m^2 t_{cpu} \tag{6.6}$$

モデル式6.7の導出方法,T_{cpu}の決定方法はMDシミュレーションと同様である．MDシミュレーションではnに対する比例係数を求めていたが, 行列の乗算では$m \times m$の比例係数を求める．ただし,行列の乗算を行うアプリでは, DS-CUDA(Infiniband, GigabitEthernet), PCI Express拡張boxとの間で有意差が見られなかった．これは, コンパイルオプションが共通のものであることが影響していると考えられる. したがって, 全ての手法において共通の値を導き, 下記の値を用いることとする.

$$_{cpu} = 3.88 \times 10^{-09} \tag{6.7}$$

6.3.3　T_{com}のモデル化

T_{com}を以下のようにモデル化する.

$$\begin{aligned} T_{com} &= \left(\frac{8n^2 N_{gpu}}{B_{h2d}} + 2N_{gpu}L_{h2d}\right) \\ &+ \left(\frac{4n^2}{B_{d2h}} + N_{gpu}L_{h2d}\right) + t_{kernel}N_{gpu} \end{aligned} \tag{6.8}$$

$$\left\{ \begin{array}{rcl} B & : & \text{転送スループット (byte/sec)} \\ L & : & \text{1転送命令のレイテンシ (sec)} \\ t_{kernel} & : & kernel\ \text{呼び出しにかかる時間 (sec)} \end{array} \right.$$

式 6.8 の導出法は MD シミュレーションと同様である (式 5.7). host から device への転送命令は GPU1 つあたり 2 回 (付録 3.11,12 行) であり, device から host への転送命令は 1 回 (付録 3.22) である. また行列の要素 1 つあたりのデータ量は `sizeof(float)`$= 4$(bytes) である.

転送スループット B, 転送レイテンシ L, はアプリケーションを問わず共通のものである. 従って,t_{kernel} だけを新たに求めた. 導出法は MD シミュレーションと同様である求めた t_{kernel} を表 6.1 に示す.

	DS-CUDA(Infiniband)	DS-CUDA(GigabitEthernet)	PCI-Ebox
t_{kernel}	1.40×10^{-4}	8.16×10^{-4}	2.19×10^{-5}

表 6.1: T_{com} における各係数

6.4 行列の乗算のモデル式の考察

通信に Infiniband を用いた DS-CUDA のモデルと実測値の比較を図 6.7 に, GigabitEthernet を用いたモデルと実測値の比較を図 6.8 に PCI Express 拡張 Box のモデルと実測値の比較を図 6.9 に示す. 粒子数が多い時は 3 乗で増えていることがわかる.

MD シミュレーションと異なり, 時間増加の傾きが急に変わる傾向はみられず, 緩やかに傾きが大きくなっている. また GPU 並列数が大きくなるほど, 傾きの変化が緩やかである. これは,MD シミュレーションと比較してデータ通信にかかる時間 ($O(n^2)$) の比率が大きい為と考えられる.

GigabitEthernt 通信による DS-CUDA の場合と PCI Express 拡張 Box の場合では, モデルと実測値の誤差が大きくなっている. 特に小粒子,GPU 並列数大の領域にて誤差が大きいことから, 通信時間 T_{com} が誤差の原因になっているものと思われる. 式 5.13 による誤差は Infiniband が最大 32%, GigabitEthernt が 91%,PCI Express 拡張 box が 82%となった. このことから, モデルは曲線の傾向は近いものの, 正確とは言えないという結果となった.

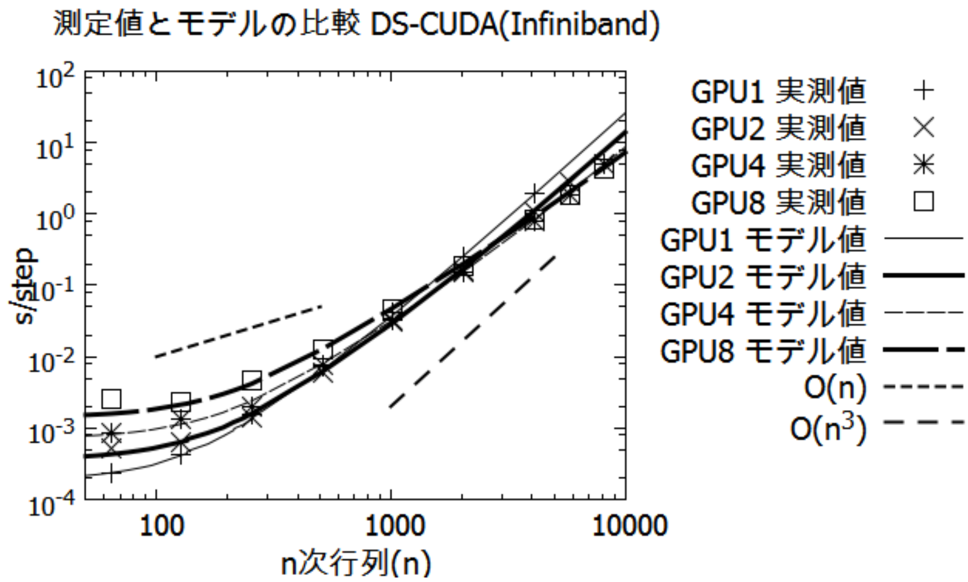

図 6.7: 1step の計算時間の実測値とモデルの比較 (DS-CUDA(Infiniband) の場合)

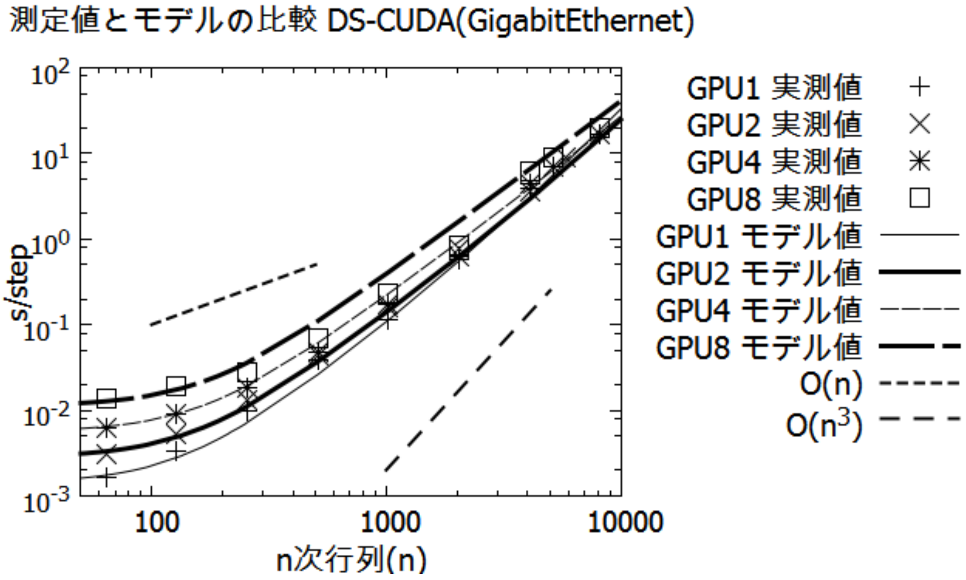

図 6.8: 1step の計算時間の実測値とモデルの比較 (DS-CUDA(GigabitEthernet) の場合)

またモデルにおける T_{gpu},T_{cpu},T_{com} の比率を図 6.7(Infiniband の場合),6.8(GigabitEthernet の場合), 図 6.9(PCI Express 拡張 box の場合) に示す. 横軸は GPU の並列数 ($N_{gpu} = 1, 8$), 行列の次数 ($n = 64, 2048$) の違いである. いずれも MD シミュレーションの時と比べて T_{com} の比率が大きく, また GPU の並列数が大きくなることによる T_{com} の増加がより顕著になっていることが分かる.

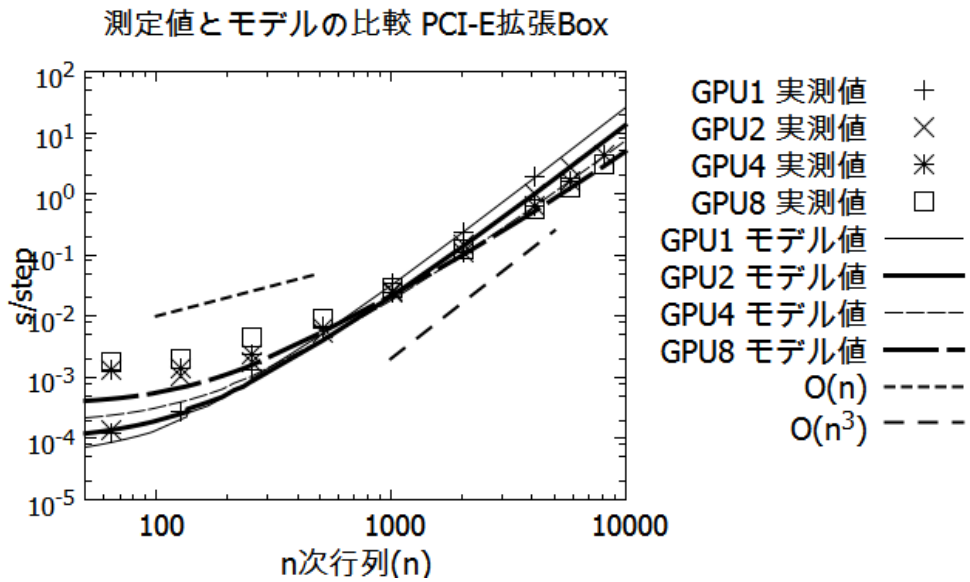

図 6.9: 1step の計算時間の実測値とモデルの比較 (PCI-Express 拡張 Box の場合)

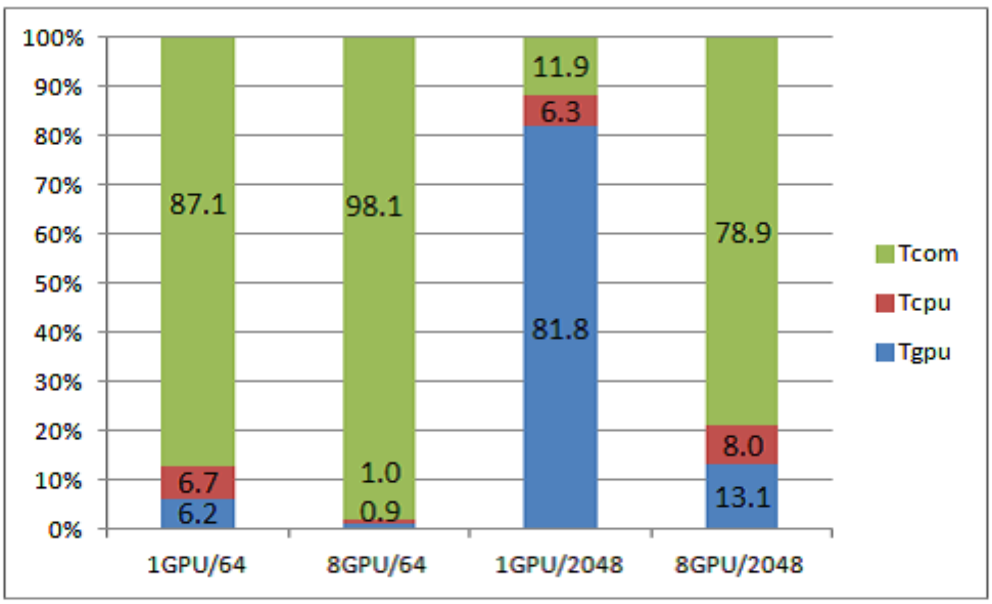

図 6.10: モデルにおける T_{gpu},T_{cpu},T_{com} の比率 (Infiniband)

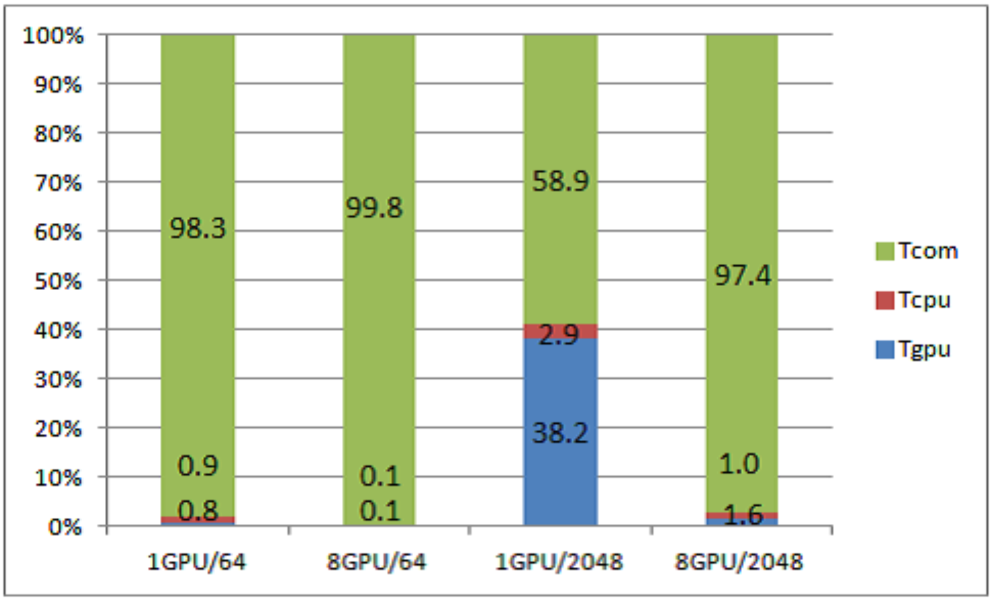

図 6.11: モデルにおける T_{gpu},T_{cpu},T_{com} の比率 (GigabitEthernt)

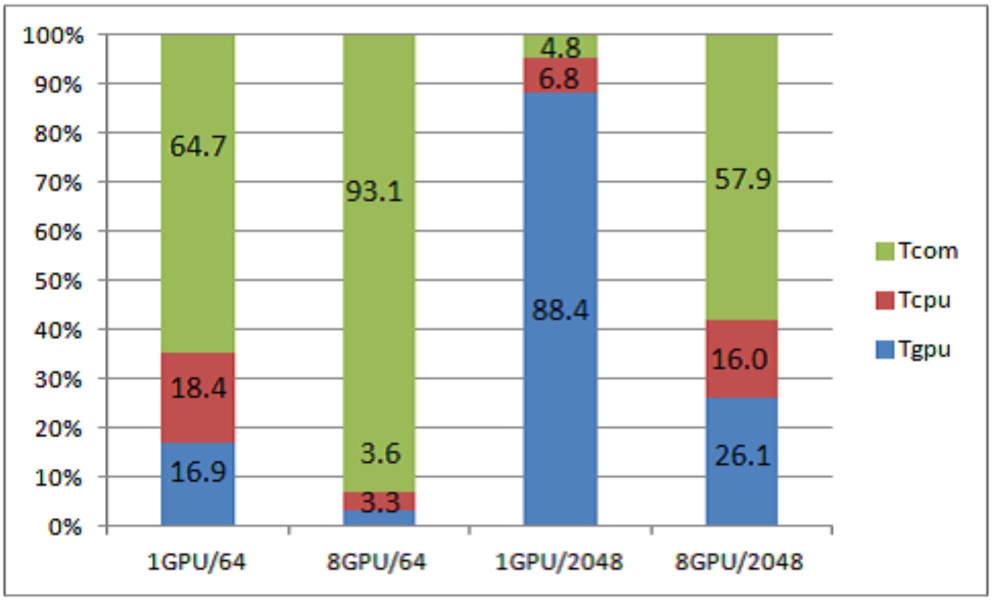

図 6.12: モデルにおける T_{gpu},T_{cpu},T_{com} の比率 (PCI Express 拡張 box

第7章　まとめ

7.1　本研究で達成したこと

本研究では, 複数台の仮想GPUと物理GPUを用いた計算時間の比較を行い, 性能モデルを構築することを通じて, 仮想化による通信時間の影響とパフォーマンスを発揮するGPUの運用の為の指標を提案しようとした.

測定とモデル化の結果から,DS-CUDA (Infiniband, GigabitEthernet), PCI Express 拡張 box いずれの手法においても, データ量小,GPU 並列数大のときに, 通信時間の割合が大きくなることが確認出来た.

通信に関するパラメータはPCI Express拡張boxと比較してDS-CUDA (Infiniband, GigabitEthernet) はいずれも劣ることが分かり, 実測値とモデルの比較からも性能差を明らかにすることができた. 一方でInfiniband ネットワークを用いたDS-CUDA の性能は PCI Express 拡張 box と比べて, kernel 命令の時間に差が見られたものの, 通信スループット, 通信レイテンシ共に大きく劣っているとは言えず, データ量が十分大きい時には計算時間が僅差であることが実測値, モデルの両者から確認できた. GigabitEthernet ネットワークを用いたDS-CUDA においても, 通信性能は大きく劣るもののMDシミュレーションにおいては粒子数大領域において最終的な計算時間の差を小さくすることが出来た.

定量的には仮想 GPU の計算性能は物理 GPU に対し優れているとは言えないが, PCI Express 拡張 box を用いた複数の物理 GPU による測定実験は度々システムのダウンが発生し, 中断を余儀なくされることがあった. 一方でDS-CUDA を用いた仮想GPU による測定実験は, 中断することなく測定することができた. これは,1 つの server ノードに対し 1 つのGPU を用いることでシステムが安定している為と思われる. また, 一つのGPU server がダウンしても, 全てがダウンするわけではないため, 複数台のGPU を用いていれば実用上は計算が続行不能になることはないという利点がある. 実際,DS-CUDA にはマイグレーション機能が搭載されている [13]. このことから, 対故障性等の運用の面ではDS-CUDA が優れてお

り, データ量大のときには優れた実用性を備えていると言える. 今回の実験では,PCI Express バスが 2 個ある host 端末では PCI Express 拡張 box で扱える物理 GPU の数は 8 個までであるが, DS-CUDA では server ノードの数が許す限り運用できる. したがって, データ量が多くなるにつれて GPU 内での計算時間が支配的になり, GPU 並列数を増やすことによる遅延の増加よりも計算時間の短縮が優位になる DS-CUDA による GPU の仮想化は有効であるといえる.

7.2 今後の課題と展望

モデルと実測値との間には, 特性に共通点が見られたものの誤差は決して無視できるものでは無い. モデル構築の為の各パラメータには誤差が含んでいると考えられ, 必ずしも適切な指標を提案出来たとは言えない. より正確な測定, またより大きなデータ量,GPU 並列数に対して測定を行うことで, 汎用的なモデル構築が求められる.

今後の展望として, 各モデルに対して GPU 内での計算時間が通信時間に対して十分支配的であるか, すなわち効率的な (仮想)GPU の運用をするためのデータ量と GPU 並列数の関係を導くことで, 仮想 GPU を用いた効率的な計算の運用に貢献することが出来ると考えられる.

謝辞

本研究を進めるにあたり, ご指導頂いた指導教員の成見哲教授, 副指導教員の沼尾雅之教授, および相談に乗って頂いた成見研究室の先輩, 同期, 後輩の皆様に感謝いたします.

関連図書

[1] 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.

[2] CUDA Zone:

https://developer.nvidia.com/cuda-zone

[3] 大島聡史, "これからの並列計算の為の GPGPU 連載講座 (1) GPU と GPGPU の歴史と特徴", 東京大学情報基盤センター

http://www.cc.u-tokyo.ac.jp/support/press/news/VOL12/No1/201001gpgpu.pdf

[4] OpenCL:

http://www.khronos.org/opencl/

[5] DirectCompute:

https://developer.nvidia.com/directcompute

[6] Openacc:

http://www.openacc.org

[7] ELSA VRIDGE X200 Tri:

http://www.elsa-jp.co.jp/html/products/pes/vridge_x200_tri/index.htm

[8] Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, José Duato: " A Complete and Effcient CUDA-Sharing Solution for HPC Clusters", Parallel Computing Vol40, Issue10, December 2014, Pages 574-588.

[9] rCuda:remote CUDA: http://www.rcuda.net

[10] 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分子動力学シミュレーションの並列化” 情報処理学会論文誌コンピューティングシステム Vol.7 No.4 1-14(Dec. 2014)

[11] ELSA GeForce GTX 780:

http://www.elsa-jp.co.jp/products/products-top/graphicsboard/geforce/ultra_high_end/geforce_gtx780/

[12] Claret:

htpp://atlas.riken.go.jp/∼koishi/claret.html

[13] 吉川 和幸, 成見 哲, 沼尾 雅之:

“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, 2014 年

付録1 Claretの疑似コード

Claret の疑似コード (1MDStep)

```
/*シミュレーション条件の設定*/
N_gpu ← GPU 並列数
n ←粒子数
/*変数の設定*/
vec[n] ←各粒子の座標, 種類
force[n] ←各粒子にかかる力
/*CPU での前演算*/
CPUCalc1(...);
/*各 GPU へのデータ転送*/
for d ← 0 to (N_gpu-1) do
 SetDevice(d);
  MemcpyH2D(vec[n], n);
end
/*カーネル実行*/
for d ← 0 to (N_gpu-1) do
 SetDevice(d);
  Kernel<<<n/N_gpu/ブロック数, ブロック数>>>(...);
end
/*クライアントノードへのデータ転送*/
for d ← 0 to (N_gpu-1) do
 SetDevice(d);
  MemcpyD2H(force[n], n/N_gpu);
end
/*CPU での後演算*/
CPUcalc2(...);
```

付録2 ClaretのKernel関数の疑似コード(1GPUの場合)

ClaretのKernelの疑似コード

```
/*引数の設定*/
n←粒子数
vec[n]←各粒子の座標,種類
force[n]←各粒子にかかる力
/*変数の設定*/
i←粒子番号,スレッドに割り当てられる番号
f←0
/*i粒子にかかる力の計算*/
for j←0 to (n-1) do
 /*i-j粒子間の力の計算*/
 f = f + calcforce(vec[i],vec[j]);
end
/*力の総和の出力*/
force[i] = f;
```

付録3 行列積を求めるプログラムの疑似コード

行列積の疑似コード

```
/*シミュレーション条件の設定*/
N_gpu ← GPU 並列数
m ←正方行列の次元数
/*変数の設定*/
a[m*m] ←行列 A
b[m*m] ←行列 B
c[m*m] ←出力
/*各 GPU へのデータ転送*/
for d ← 0 to (N_gpu-1) do
 SetDevice(d);
 MemcpyH2D(a[m*m], m*m);
 MemcpyH2D(b[m*m], m*m);
end
/*カーネル実行*/
for d ← 0 to (N_gpu-1) do
 SetDevice(d);
 Kernel<<<n*n/N_gpu/ブロック数, ブロック数>>>(...);
end
/*クライアントノードへのデータ転送*/
for d ← 0 to (N_gpu-1) do
 SetDevice(d);
 MemcpyH2D(c[m*m], m*m/N_gpu);
end
```

付録4 行列積を求めるプログラムのKernel関数の疑似コード(1GPUの場合)

行列積の Kernel 関数の疑似コード

```
/*引数の設定*/
m ←正方行列の次元
a[m*m] ←行列 A
b[m*m] ←行列 B
c[m*m] ←行列積
/*変数の設定*/
thd ←スレッド番号
i ← thd/m /*i 行*/
j ← thd%m /*j 列*/
ans ← 0
/*(i,j) 列成分の計算*/
for k ← 0 to (m-1) do
 ans = ans + a[i*m+k]*b[k*m+j];
end
/*力の総和の出力*/
c[i*m+j] = ans
```