平成 28 年度修士論文
複数 GPU を用いた DS-CUDA による P2P 機能使用時の性能評価及び最適化

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

1531009　伊藤 一輝

指導教員　成見 哲　教授

副指導教員　沼尾 雅之　教授

平成 29 年 1 月 30 日

概要

GPU は数値流体シミュレーションやディープラーニングの分野に応用する GPGPU として多岐に渡る発展を見せ広く普及してきた． GPU は豊富な計算資源を所有しており大規模な並列演算が可能で年々性能が飛躍的に向上している． しかし, 大規模なシミュレーションや数値流体計算問題をアプリケーションプログラムとして実行するには単体の GPU ではメモリなどの計算資源が不足する． 通常は並列コンピューティングで用いられる MPI と GPGPU で用いられる CUDA を利用することで GPU 間あるいはノード間での通信を行い不足する計算資源を補う． また当研究室で扱っている DS-CUDA はネットワークに接続されたサーバ上の 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リモートの GPU の計算資源を用いた GPGPU によって同様の問題を解消することが可能である． しかし, 大規模なデータに対し高並列化のプログラムを実装するとレイテンシが大きくなり通信速度が向上しないという問題が新たに発生する． 対策としては DS-CUDA API の dscudaMemcopies()を利用することで, サーバ上の GPU 間の通信を Peer to Peer(P2P)で並列に処理することで高速化が可能になっている.

そこで本研究では, 3 次元 Euler 方程式から Rayleigh-Taylor 不安定性の成長シミュレーションを解く数値流体計算用のコードを複数 GPU を用いて最適化を行った． さらに DS-CUDA に搭載されている P2P 機能を用いてノード間の通信をサーバ側だけで行う通信の最適化を行った． アプリケーションプログラムは最大 8 つの GPU を用いて Native 時, DS-CUDA を利用した InfiniBand ネットワーク使用時, DS-CUDA を利用した P2P 機能使用時における性能評価を行った． 予備実験では, P2P 機能の通信速度を測定し転送データ量やパラメータ数, メモリアクセスの手法を変えることでどのくらいの転送速度が出るか測定した． その結果, 連続領域で 4 台のノード間における P2P 機能を使用した場合に最大で約 5.08 倍の通信時間が高速化されており, 本研究で利用する数値流体計算用のコードを想定した検証では, 8 台のノード間において最大で約 1.95 倍の通信時間高速化が見込まれることを示した.

実際のアプリケーションコードでは 8GPU における P2P 機能使用時に256^3グリッドサイズにおいて InfiniBand ネットワーク使用時と比較して約 2.56 倍高速化された． また, 通信時間については P2P 機能使用時512^3グリッドサイズにおいて同様の比較を行い約 72.51%の通信時間を削減した． これらにより, DSCUDA API の P2P 機能はサーバ側におけるノード間 GPU 通信において転送データ量を大きくすること, 高並列化することが通信時間削減に有効であることを示した.

目次

1.はじめに

1.1 背景

近年, GPU は大きな発展を遂げており一般にはゲームグラフィックにおける画像処理の部分を担当していることで知られている． オンラインゲームや e-sports といったゲーム分野においては高精度な映像, 快適な操作を実現するために高性能な GPU を搭載した PC が必要不可欠となっている.

また,GPU の使用用途は画像処理以外でも NVIDIA 社[1]が提供している並列コンピューティングアーキテクチャ CUDA[2]を利用することで数値流体シミュレーション[3]やディープラーニング[4]の分野に応用する GPGPU[5]として多岐に渡る発展を見せ広く普及してきた． GPU は豊富な計算資源を所有しており大規模な並列演算が可能で年々性能が飛躍的に向上している． しかし, 個人が所有する PC に搭載できる GPU はスロット数の関係上制限があり, 更に単体の GPU メモリ容量, 計算資源を超えた大規模なシミュレーションや計算問題は取り扱うことができない． 通常は並列コンピューティングで用いられる MPI[6]と GPGPU で用いられる CUDA を利用することで GPU 間あるいはノード間での通信を行い不足する計算資源を補う． しかし, MPI はメッセージ通信であるためデータの送信先, 受信先の計算機を意識して分散メモリの考え方で実装する必要がありプログラミングが難しいことや記述するコード量が接続先の台数に比例して増加するため開発者の負担が大きい.

当研究室では GPU 仮想化ソフトウェアである DS-CUDA[7]を利用してネットワーク上に分散配置された GPU を透過的に GPGPU として扱う． ノード間でのデータ通信に InfiniBand を介すことで高速化しているが, 多並列になるほどクライアント-サーバ間のデータ通信量の増加により処理性能の低下が発生する． このため, 仮想的に共有メモリの考え方でプログラミングでき, かつクライアント-サーバ間通信量を削減する dscudaMemcopies()という API が用意されている.

図 1.東京工業大学 TSUBAME 2.5(参考文献[8]の画像より引用)

1.2 目的

本研究では，まず単一 GPU 向けの数値流体計算用のコードを複数 GPU に対応する． 更に DS-CUDA に搭載されている P2P 機能を用いてノード間の通信をサーバ側だけで行うシステムを構築し，以下の手法で複数の GPU を用いた計算性能を評価し並列化効率を示す．

- ローカル時の複数 GPU の利用
- InfiniBand ネットワークを用いた DS-CUDA による仮想 GPU の利用
- InfiniBand ネットワークを用いた DS-CUDA による P2P 機能を使用した仮想 GPU の利用

1.3 本論文の構成

本論文の構成とその内容は以下の通りである.

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 GPU アーキテクチャや関連技術の並列コンピューティングについて紹介する

第 3 章 関連研究

本研究に関連した複数 GPU,並列コンピューティングの手法を扱う既存研究について紹介し,新規性について述べる.

第 4 章 システム構成

本研究で行った実験,評価に使用したシステム,ハードウェアやプログラムのアルゴリズムについて述べる

第 5 章 評価

DS-CUDA による P2P 機能の予備実験における性能の検証から, 計算問題の概要, 実験結果,性能評価の考察について述べる

第 6 章 まとめと今後の課題

本研究で達成したことと,今後の課題や展望について述べる

2. GPGPU (General Purpose computing on GPU)

GPGPU とは GPU を画像処理以外の用途に演算資源を利用することである． この章では GPU アーキテクチャの特徴について紹介し，本研究での GPU 利用について解説する．

2.1 GPU コンピューティング

GPU コンピューティングとは，計算科学や数値流体力学,機械学習などのアプリケーションを加速させ処理性能を飛躍的に向上させるために GPU と CPU を併用することを指す．この節では GPU の発展から現在に至るまでを紹介する．

2.1.1 GPU (Graphics Processing Unit)

GPU は画像処理を担当する主要なハードウェアである． 特徴としては画像処理専用プロセッサである GPU は, CPU と比較してコア数が遥かに多い． CPU は連続した計算に強いが GPU は並列した計算に強い． そのため問題に適したプログラムを構築するために複雑な処理は CPU に，並列で単純な処理は GPU に分けることでアプリケーションの処理性能を向上させることが可能になる． GPU を利用したプログラム一連の流れについては図 2 のようになる．

現在 GPU 開発ベンダーは NVIDIA 社と AMD 社が大部分を占めており, 2~3 年に 1 度の周期で性能を飛躍的に向上させている． NVIDIA 社の製品を例にあげるとコンシューマ向けに開発された DirectX に最適化した GeForce シリーズ，デザインや制作向けに開発された OpenGL に最適化した Quadro シリーズ, GPGPU 向けに開発された Tesla シリーズが存在する．

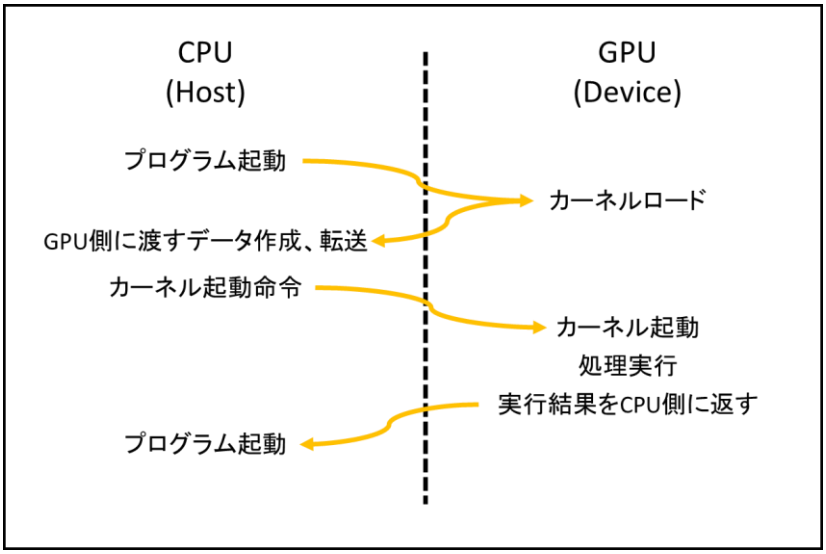

図 2.プログラムの一連の流れ

2.1.2 CUDA (Compute Unified Device Architecture)

CUDA とは NVIDIA 社が提供している GPU コンピューティングアーキテクチャのことである． CUDA は無料で使用することができ, CUDA でプログラムを記述すると CUDA をサポートするすべての GPU 上で動作させることができる． 開発者側は C, C++, Fortran など高級言語で記述することが可能であり CUDA 向けに制作されたソースコードは拡張子 *.cu がつく． ソースコードのコンパイルは nvcc により実行ファイルが図 3 のようにして作られる． ソースコードは CPU 上で動作するコードと GPU 上で動作するコードに分けられコンパイルされ, オブジェクトファイルを生成しリンカから実行ファイルを出力する.

複数 GPU を利用する場合には OpenACC[9], OpenMP[10], MPI のいずれかを組み合わせることで利用することができ, 2017 年 1 月現在はバージョン 8.0 まで公開されている.

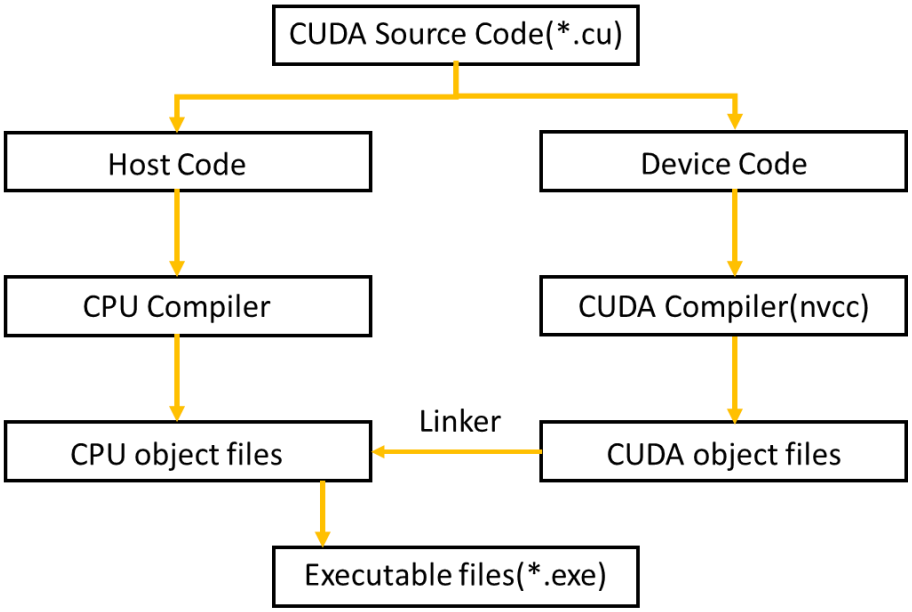

図 3.コンパイルから実行までの流れ

次に CUDA プログラムにおいて，スレッド，ブロック，グリッドという用語について説明する． 概念図については図 4 に示す．

・スレッド

カーネルを動作させたときの多数のプログラムの最小単位を指す． これは CPU においても使われる単語であるが，CPU ではそのコア数とほぼ同数のスレッドが動作するのに対し，GPU ではコアに対し数千～数万と圧倒的な数のスレッドが並列に動作することにより，高い性能を引き出す．

スレッド自体はホスト側から起動されて各スレッドプロセッサで同じ処理が行われるが，その実行タイミングはそれぞれ異なる． これはカーネルの呼び出しタイミングが同時ではなく１つあたり１クロックずつずれるのが原因である． すると最初のスレッドによる計算が終わった時点で最後のスレッドの計算処理が終わってないことが予測される． CUDA プログラミングにおいては多数の処理が並列に行われるが，その処理は全て非同期である．

同時に処理が実行されることを考慮した場合，計算結果の値を再利用する場合，計算のズレが問題となる． それを解決するために __syncthreads()， cudaThreadSynchronize()によって前の CUDA カーネルの実行が終了するまで待機させることが必要である．

・ブロック

スレッドをまとめたもので，1 つのブロック当たり最大 512 スレッドが格納される． x 方向，y 方向，z 方向に 8 スレッドずつ，1 ブロックに 8×8×8 スレッドと 3 次元的表現をとることができる．また 1 次元的，2 次元的にすることも可能である．

・グリッド

ブロックをさらにまとめたものがグリッドと呼ぶ． ブロック同様，xyz の 3 次元で表現されるが，現在 z 方向のブロック数は 1 でなければならないので実質 2 次元で管理される． 1 グリッド当たり，x 方向あるいは y 方向に配置できる最大ブロック数は 65535 個で，それを超えてブロックを配置することはできない．

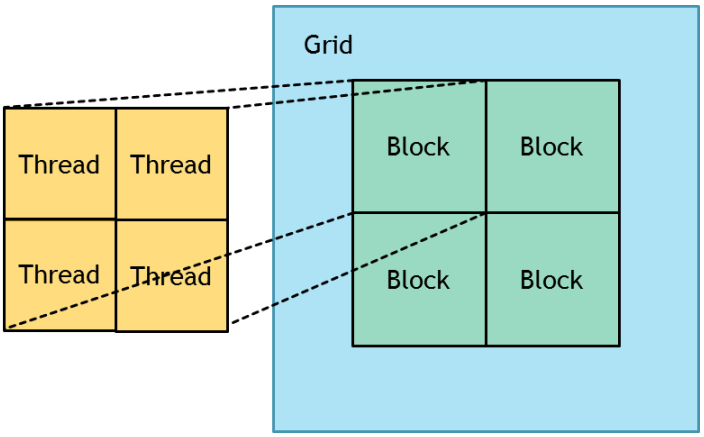

図 4.CUDA のグリッド,ブロック,スレッドの概念図

2.2 DS-CUDA (Distributed Shared-CUDA)

DS-CUDA 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 を用いるため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，クライアントノード，サーバノードから構成され，サーバノードに搭載された GPU をクライアントノードから利用できる．

図 5 は DS-CUDA システムの概略図である，各サーバノード上では，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 GPU にアクセスしようとすると，接続要求がネットワークを通じてサーバプログラムへと送信され，データ転送やカーネル関数の実行が行われる．

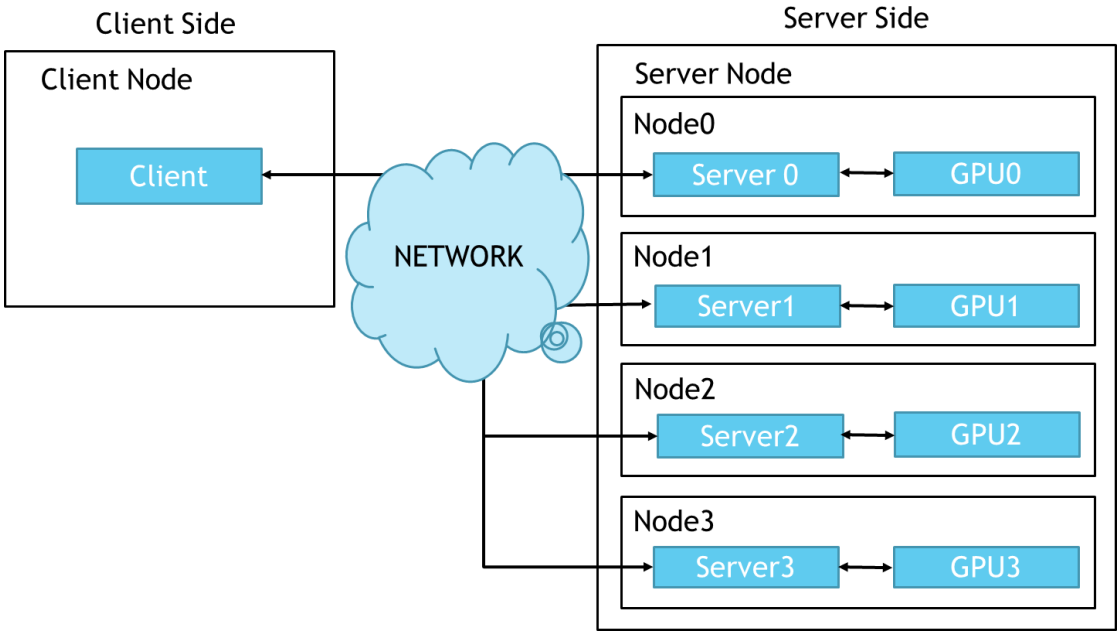

図 5.DS-CUDA システムの概略図

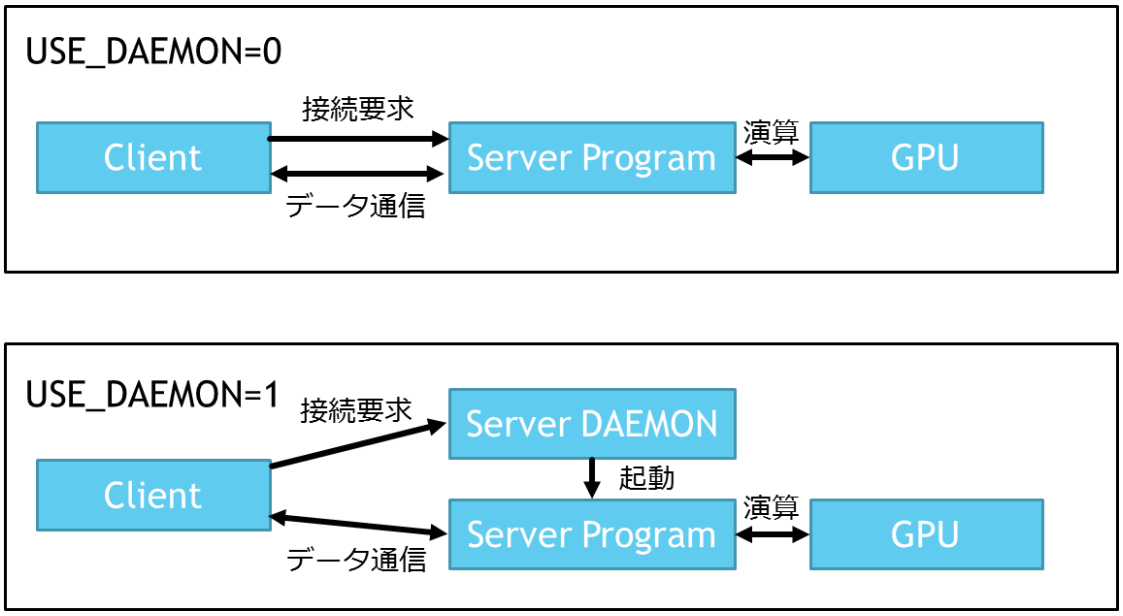

図 6.DS-CUDA デーモンの仕組み

DS-CUDA のバージョン 1.3.0 以降よりサーバデーモン機能が搭載された(図 6). サーバデーモン機能を使用した場合, 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ると, サーバプログラムも自動的に終了する. デーモンを用いるかどうかは環境変数の USE_DAEMON で指定する.

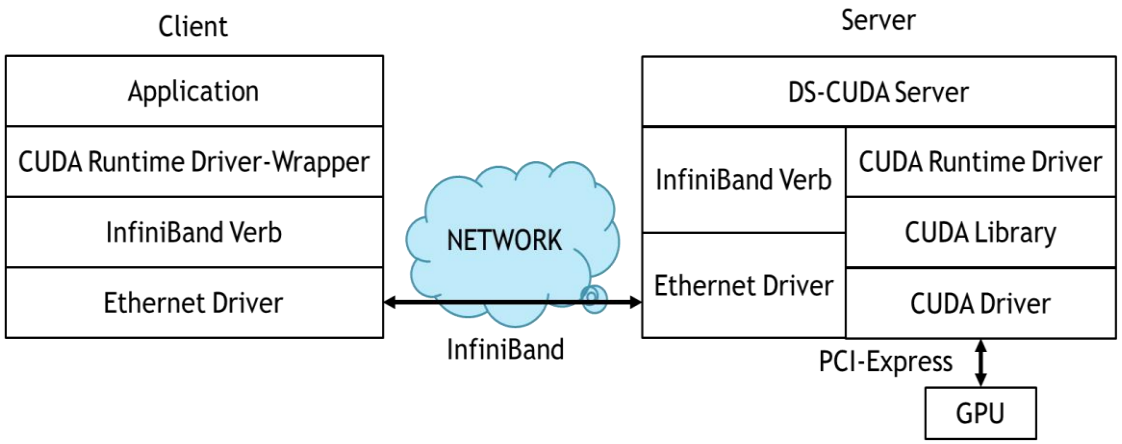

図 7.DS-CUDA システムの階層図

図 7 は DS-CUDA システムの階層図である． 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 ドライバといったクライアント-サーバ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, 開発者側はネットワークを介した GPU の利用を意識せずに, 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, Gigabit Ethernet 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開発者側はネットワークを意識せずに CUDA ソースコードを利用実装することができる.

2.2.1 P2P 機能

本研究で用いられる P2P 機能の使用前，使用後のデータ通信について図 8 に示す．通常はクライアント側から CPU の命令で各 GPU に計算領域として必要なデータをクライアント側へ送信してもらい，そのデータを該当する GPU へパラメータ毎に送信する(図 8 左)．本システムでは DS-CUDA API にある dscudaMemcopies()を用いてノード間における GPU データをサーバ側だけでやり取りする． また，データの転送は転送回数を削減するためにパラメータ毎にまとめて並列送信することが可能である．dscudaMemcopies()の引数と機能について以下で説明する．

$void\ \ dscudaMemcopies(void\ **\ dbufs, void\ **\ sbufs, int\ *\ counts, int\ \ ncopies)$
$dbufs$: 転送元アドレスのリスト
$sbufs$: 転送先アドレスのリスト
$counts$: データ転送量のリスト
$ncopies$: データ転送個数

i 個目のデータ転送の転送元アドレス,転送先アドレスは，それぞれ sbufs[i]，dbufs[i]で指定する． 転送量は counts[i]にバイトサイズで指定する． 転送元および転送先のデバイスは，アドレス(UVA; Unified Virtual Address)から自動的に判定され同時実行可能な複数のデータ転送は，複数のスレッド上で並列に実行される(図 8 右)．

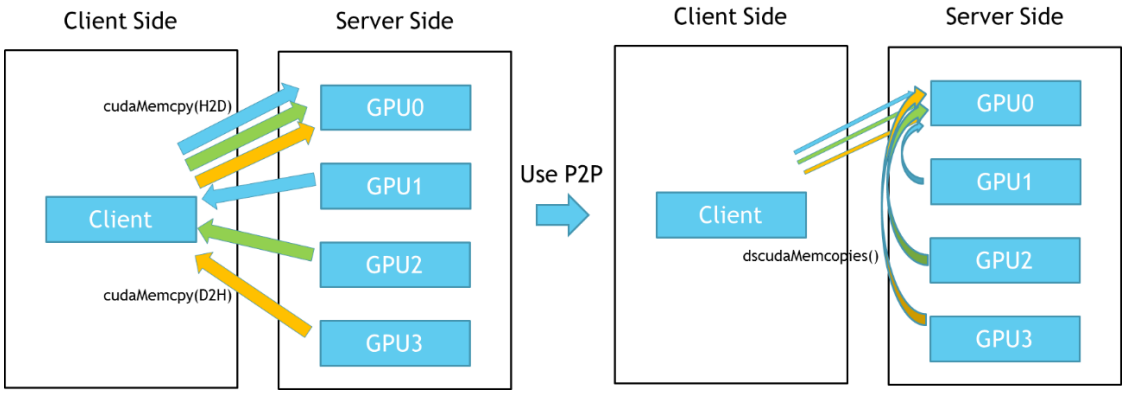

図 8.dscudaMempcopies()によるデータ通信の流れ

3.既存研究

この章では GPU による並列化に関連する既存研究について紹介し, 本研究と異なる点や実際に複数 GPU をどのように利用するか説明する.

3.1 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[11]

老川らはシングルノード向けレプリカ交換分子動力学シミュレーションに GPU 仮想化ソフトウェア DS-CUDA を適用し,マルチノード上の GPU に対応した計算並列化を行った. レプリカ交換分子動力学シミュレーションを使ってアルゴン原子群から構成されるバルクを対象とした固体-液体間の相転移温度を見積もっている. 1 つの分子系における温度を上下させることで相転移温度を見積もると, 低温領域において原子が局所的なエネルギー極小状態になり温度を逃さないようにするため最安定状態に緩和するに至るまで多くのシミュレーションタイムステップが必要となる. そのため計算量が膨大になり計算時間に影響を及ぼすためレプリカ交換法を用いて最安定状態を効率良く求めている. 更に計算並列化を行うために DS-CUDA を用いてスーパコンピュータ TSUBAME2.5 に含まれる計算ノードの一部を扱い, InfiniBand に接続された最大 343 台の計算ノードを動作させた. 各計算ノードは 6 コアの CPU が 2 基, PCI-Express に接続された GPU が 3 台搭載されており, 1 台はクライアントノードとしてその他のノードはサーバノードとして使用し, サーバノード上の GPU に対応した計算並列化を行うことによってノード間のデータ通信を少なくした. 最大 1,024 台の GPU を使用して計算速度の並列化効率を計測した結果, 1,024 並列時において並列化効率で 87%の結果を得ている.

本研究ではノード間における通信手法でデータ通信にクライアントノードを経由させない. クライアント側からの命令によってサーバ側のみでデータ通信を行わせる. また P2P 機能によりノード間データ通信を並列化させることで計算効率を良くするという点において異なる.

3.2 NVIDIA GPUDirect[12]

GPU Direct は NVIDIA 社が 2010 年 6 月に導入した異なる GPU 間のデータ転送を高速化させる機能である[13]. CPU 上の不要なオーバーヘッドを取り除くことにより、PCI-Express における GPU 間の高速通信を可能とする. GPU Direct ver.1 では InfiniBand によって CUDA ドライバによるノード間転送, ver.2 では P2P 通信によってノード内のホストメモリを経由しない GPU 間転送, ver.3 では InfiniBand 間での DMA 転送(RDMA)を実装している(図 9). しかし上記の機能を使用し処理性能を向上させるにはデータ転送時に MPI(CUDA-Aware MPI[14])の導入が不可欠となる.

本研究では MPI を使用せずに DS-CUDA API を使用することで複数のデータ転送を並列に行うという点で異なる.

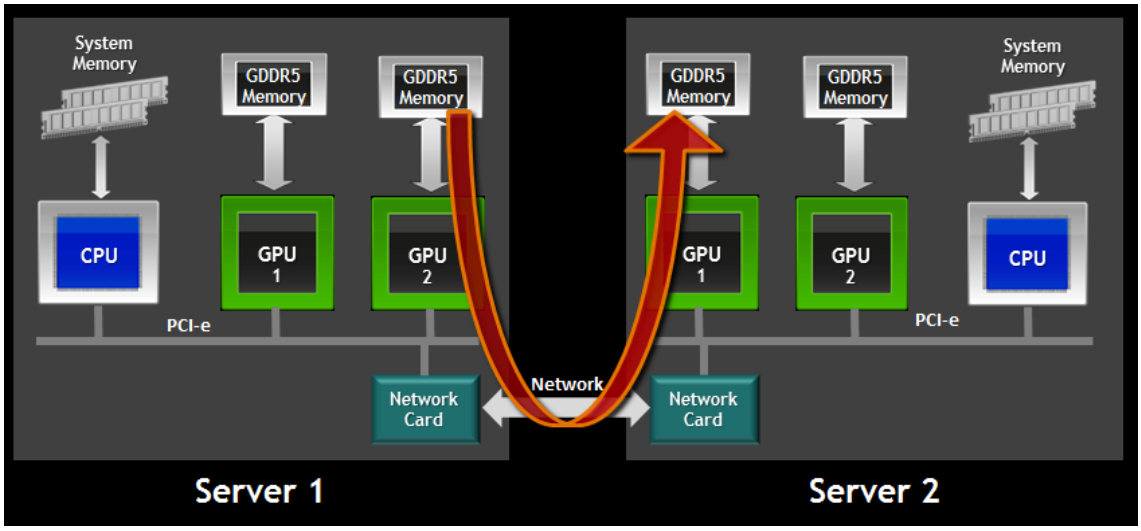

図 9. InfiniBand 間での DMA 転送(参考文献[13]の図より引用)

3.3 GPU および領域分割を用いた粒子法による流体シミュレーションの高速化[15]

佐々木らは粒子法による流体シミュレーションにおいて Uniform Grid 法を用いた領域分割による近傍の粒子探索の高速化と CUDA を用いた GPU コンピューティングによるシミュレーションの高速化を行った. Uniform Grid 法により計算領域をグリッドで分割し, 求めたい粒子が存在するグリッドとその周辺のグリッドに存在する粒子との粒子間距離の計算に限定することで 2 次元空間における計算を$O(N^2)$ から$O(N)$ の計算量に抑えている. Uniform Grid を用いた領域分割に加えて OpenMP を用いて並列化することで粒子数が 13 万個の時, 初期プログラムと比較し約 46 倍高速化している.

GPU コンピューティングによるシミュレーションの高速化では, GPU に搭載されているメモリを活用したシミュレーション用のソースコードを作成し, 最適化としてレジスタやシェアードメモリに格納された値を再利用するために, カーネル呼び出し回数を減らした. またグローバルメモリへのアクセスでは, 値の再配置により連続したメモリ領域をまとめて転送するコアレッシングを起きやすくすることで GPU へのアクセス効率を良くなるようにしている. 更に, シェアードメモリに周辺のグリッドに存在する粒子の座標を格納し, グローバルメモリとのアクセス回数を削減した. 結果, 3 次元シミュレーションで探索粒子数が大きく増加することにより, シェアードメモリによる高速化の効果が大きくなり, 2 次元シミュレーションより大きな高速化率が見込まれ, 最終的に粒子数が 13 万個の場合において計算時間を比較すると約 7 倍の高速化を達成できた.

本研究では領域分割による高速化や GPU におけるアクセス効率の最適化を図るだけでなく単体 GPU で扱うことのできない流体計算問題に対して複数のノード間におけるデータ通信の最適化を行う.

3.4 A High-productivity Framework for Multi-GPU computation of Mesh-based applications[16]

青木らは複数 GPU による格子に基づいたシミュレーションを簡便に高い生産性で開発することを可能にするマルチ GPU コンピューティング・フレームワークを提案した．フレームワークはユーザコードをノード間は MPI，ノード内の複数 GPU は OpenMP で並列化している．

異なるノード間の GPU 間通信は受信側の GPU メモリを直接参照することができないため GPU メモリからホストメモリへデータコピーを行い，MPI によるホストメモリを送受信しホストメモリから GPU メモリへのデータコピーを行うことでデータ通信を行っている．

ノード内の GPU 間通信は 1 つのスレッドが 1 つの GPU を担当している．各スレッドは GPU で確保された配列に対しデータ通信に利用する配列のポインタをフレームワーク内に登録する．登録されたポインタは OpenMP スレッドで異なるスレッドにフレームワーク内で登録された配列のポインタにアクセスする．通信を行う GPU 間は cudaMemcpy()，もしくは GPU Direct による P2P によって配列のポインタを指定しデータ通信を行っている．

また GPU 間通信を簡単に記述する複数スレッドによる並列実行を行うためのクラス，配列変数の GPU 間通信を行うクラスを C++言語から利用できるテンプレートクラスを提供している．これらを用い開発者側は制約の多い GPU アーキテクチャや GPU 間通信における実装を意識することなく最適化をすることが可能になっている．

評価実験においては圧縮性流体計算による Rayleigh-Taylor 不安定性の成長シミュレーションを 2 基の NVIDIA Tesla K20X GPU を使用してフレームワークを適用した結果，約 1.4 倍の高速化に成功し，複数 GPU 計算では良い弱スケーリングを得ている．

本研究とは複数 GPU 計算でもユーザコードを簡単に記述できるという点は相似しているが Tesla シリーズのような大容量のメモリで問題を扱うわけではなくコンシューマ向けの GeForce シリーズを利用するため汎用性の観点から異なる．また，ノード内やノード間におけるデータ通信においてホストメモリを介す点が異なる．本研究では DSCUDA API である dscudaMemcopies()を利用してノード間通信においてホストメモリを介さずサーバノード側のみでデータ通信を行っている．

4.システム構成

この章ではシステムの構成, 単一 GPU 向けの数値流体計算用コードを複数 GPU に対応および最適化した手法について紹介する.

4.1 動作環境

本研究では DS-CUDA を利用して最大 8 台の GPU サーバにアクセスするようアプリケーションプログラムを実装した. ノード間の通信には cudaMemcpy()や dscudaMemcopies()に対して高速な通信を実現するために InfiniBand を使用している. 各ノードの構成は表 1 の通りである. 各ノードに搭載する GPU は 1 つのみとし CPU, GPU による性能差が生じるのを防ぐ. (以降サーバノード X に対し dsXX と呼ぶ) クライアント側には ds11, サーバ側は ds02, ds03, ds04, ds05, ds06, ds07, ds13, ds15 の 8 台を利用し検証, 評価を行った.

表 1.ノードの構成

OS	Ubuntu 14.04.3 LTS / Fedora 14
CPU	Intel Core i7 920 2.67GHz
CPU Memory	8GB (4GB×2)
GPU	GeForce GTX 780
GPU Memory	3GB (3072MB)
Compiler	dscudapkg2.4.0 and CUDAToolkit 6.0

4.2 数値流体計算用のコードについて

本研究では東京工業大学の青木らが制作した圧縮性流体計算による Rayleigh-Taylor 不安定性の成長シミュレーションコードを複数 GPU で扱えるように z 軸で領域分割を行った. 計算時間の大部分を占める CPU の関数, GPU のカーネルを表 2 に示す.

表 2.関数,カーネル一覧と使用用途

関数	用途
malloc_variables()	CPU・GPUのメモリ領域確保
initial()	初期条件入力
cpu_euler_x()	CPUによるx軸方向の粒子計算
gpu_euler_x<<<>>>	GPUによるx軸方向の粒子計算
cpu_euler_y()	CPUによるy軸方向の粒子計算
gpu_euler_y<<<>>>	GPUによるy軸方向の粒子計算
cpu_euler_z()	CPUによるz軸方向の粒子計算
gpu_euler_z<<<>>>	GPUによるz軸方向の粒子計算
cudaThreadSyncronize()	各スレッドの計算結果同期
update()	ステップ毎の計算更新
cudaMemcpy()	データ通信
dscudaMemcpy()	DS-CUDA API によるP2Pデータ通信

4.2.1 拡散方程式による性能評価

拡散方程式は流体計算等で多く用いられる方程式で, 以下のように表される.

$$\frac{\partial f}{\partial t} = \kappa \nabla^2 f \quad (1)$$

ここで, fは物理変数,κは拡散係数である． 多くの拡散方程式は 1 方向に 3 点, 3 次元計算では 7 点の格子点を参照する. 隣接 GPU から送信されるデータを保持する拡張境界領域は, 1 格子点の厚さが必要となる.

今回本研究で扱う数値流体計算用のコードは圧縮性流体計算として 3 次元 Euler 方程式から Rayleigh-Taylor 不安定性の成長シミュレーションを次の方程式で解く.

$$\frac{\partial U}{\partial t}+\frac{\partial E}{\partial x}+\frac{\partial F}{\partial y}+\frac{\partial G}{\partial z}=S \quad (2)$$

$$U=\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ \rho e\end{bmatrix},\quad E=\begin{bmatrix}\rho u\\ \rho uu+p\\ \rho vu\\ \rho wu\\ (\rho e+p)u\end{bmatrix},\quad F=\begin{bmatrix}\rho v\\ \rho uv\\ \rho vv+p\\ \rho wv\\ (\rho e+p)v\end{bmatrix},$$

$$G=\begin{bmatrix}\rho w\\ \rho uw\\ \rho vw\\ \rho ww+p\\ (\rho e+p)w\end{bmatrix},\quad S=\begin{bmatrix}0\\ 0\\ 0\\ \rho g\\ pwg\end{bmatrix}$$

ここで, ρは密度, (u,v,w)は速度, pは圧力, eはエネルギーを表している． gは重力加速度である． 移流計算は 3 次精度風上手法で解き, 時間積分は低メモリ消費型の 3 段 3 次精度の TVD Runge-Kutta 法を用いる.

拡散計算では扱う変数がfのみであるが, 本研究では $\rho,\rho u,\rho v,\rho w,\rho e$ における 5 変数の時間発展を解く． ステンシル計算は 1 方向に 5 点, 3 次元計算では 13 点の格子点を参照する(図 10)． オレンジは実際に計算するグリッドの位置を表しており緑,黄色,青はオレンジから見て 3 次元計算における参照する格子点を表している.

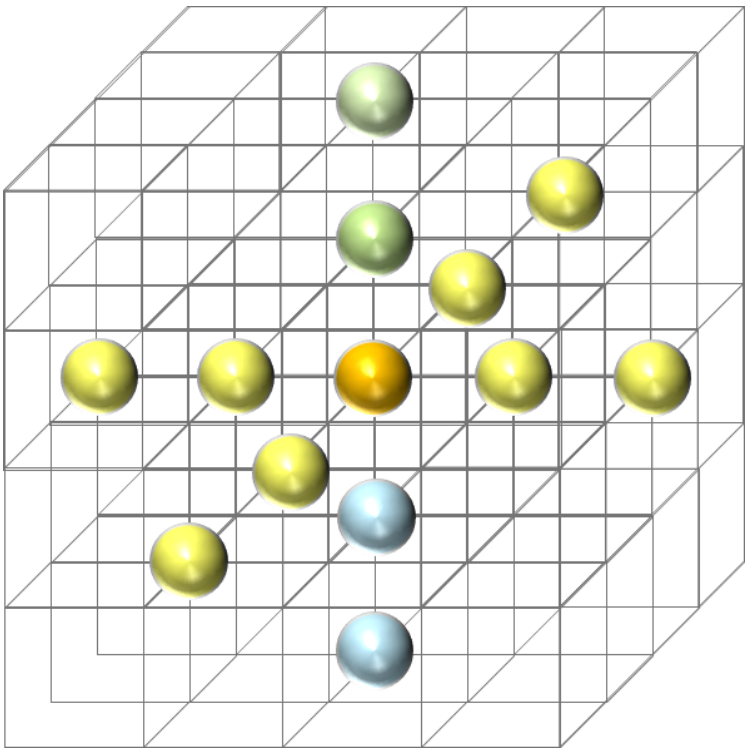

図 10.該当グリッドのステンシル計算

4.2.2 拡散方程式による境界条件

各計算ノードで計算を行うことができるのは接続されている計算ノード分の資源であるため有限の領域である． そのため計算空間の大きさを適当なところで切り出し，解析を行う必要がある． この切り出された空間のことを解析領域という．

解析領域の端を計算する場合，オレンジを計算する格子，黄色を参照する格子とすると各軸方向で前後 2 格子分が必要になる． そのため前後どちらかの格子は存在しない，ゆえに参照する値がないため計算を行うことができない(図 11 左)． そのため，解析領域の端では計算問題に対して各パラメータに値を与えることで解析領域をトーラス空間に見立てる必要がある． この値を決定する条件のことを境界条件という． 本研究では周期境界条件を x 軸, y 軸に適用する． 境界条件として x 軸の両端にある 2 格子分のグリッドを境界領域とすることでオレンジの格子計算はもう一端にある 2 格子分のグリッドを参照することで計算を行う(図 11 右)．

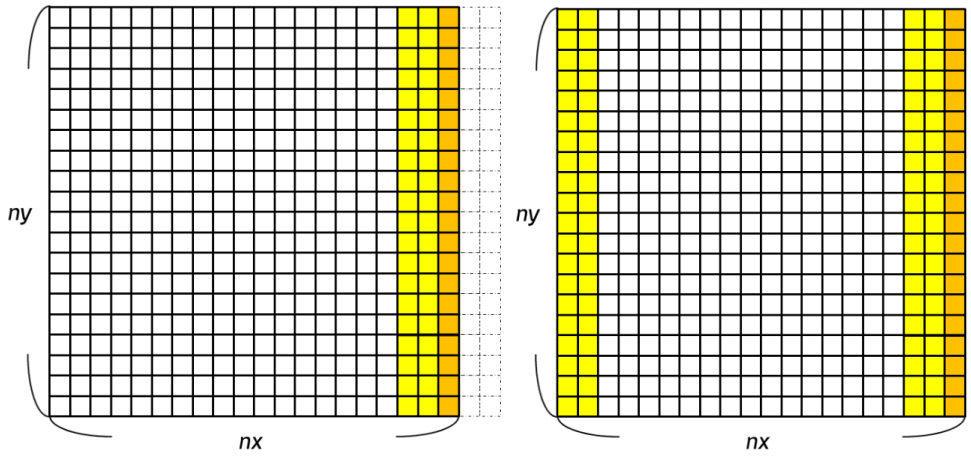

図 11. トーラス空間による x 軸の計算

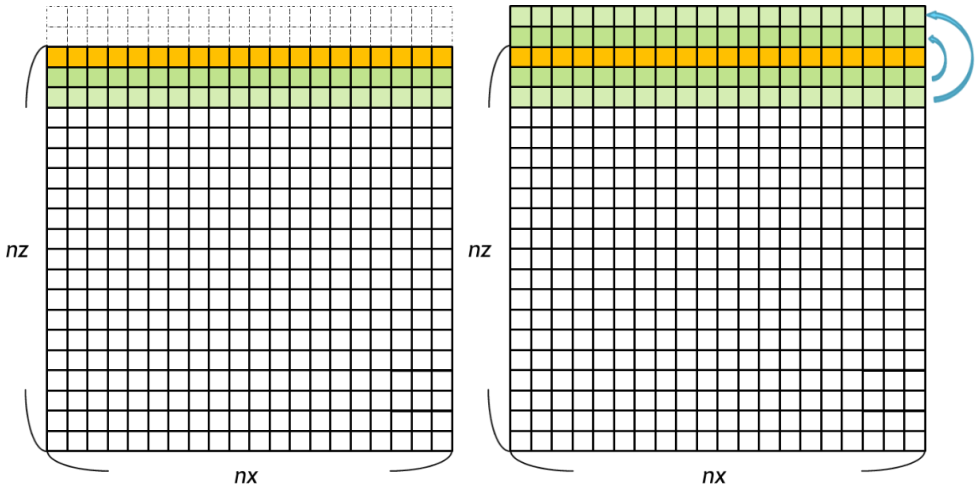

図 12.z 軸における上底格子の計算

z 軸の周期境界条件はオレンジを計算する格子, 緑を参照する格子とすると同様に前後 2 格子分が必要になる． z 軸の計算は解析領域の下底に向かうほど圧力pが大きくなる, そのため解析領域の上底, 下底の計算に対し逆端 2 格子分のグリッドの参照は圧力差が大きく周期性として利用することができない(図 12 左)． そこで z 軸の境界条件は解析領域の上底, 下底においてオレンジの格子計算に対し面対称条件を適用する． 参照する 2 格子分のグリッドを鏡像関係としてパラメータ値を与えることで計算を行う(図 12 右).

4.2.3 計算領域の分割及び複数 GPU による計算実行

本研究では, DS-CUDA を利用して接続した計算ノードに搭載されている台数分の GPU を使用する． 計算領域を GPU の台数分 z 軸方向に分割するため, 4.2.1 で説明した計算をするために z 軸方向で上下 2 格子分の領域を拡張しなければならない, この領域を境界領域とする． あらかじめ cudaMalloc()により計算領域と隣接 GPU による通信領域分のメモリを確保し, 各ステップに z 軸方向計算用の領域データをパラメータ分まとめて送信することで z 軸計算が行えるようにする． 各 GPU における計算領域を色分けし隣接 GPU に送信する境界領域を破線部分とする(図 13 左)． 各 GPU に上下 2 格子分の拡張された領域をデータ送信することで計算領域における z 軸計算を行う(図 13 右)

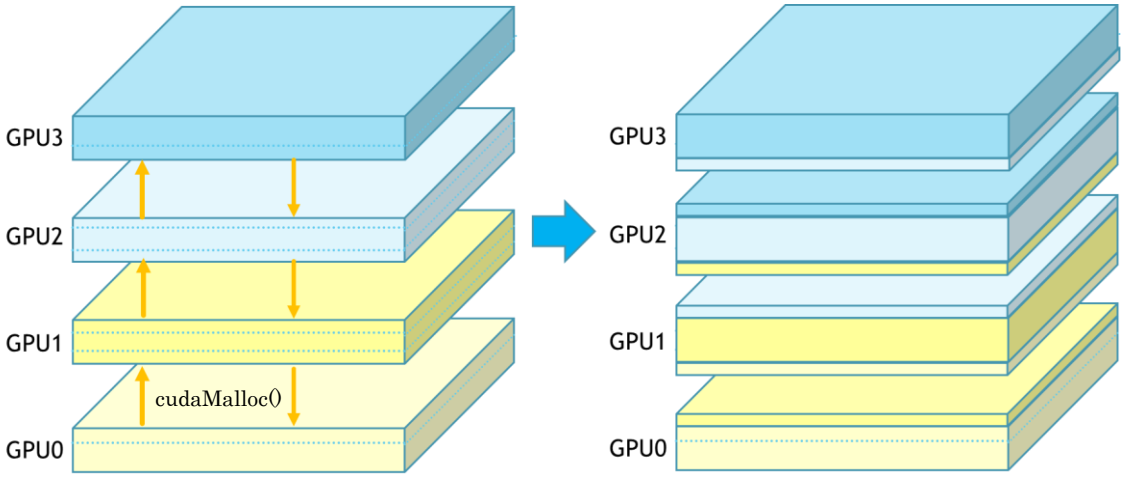

図 13.境界条件による GPU メモリ領域の拡大

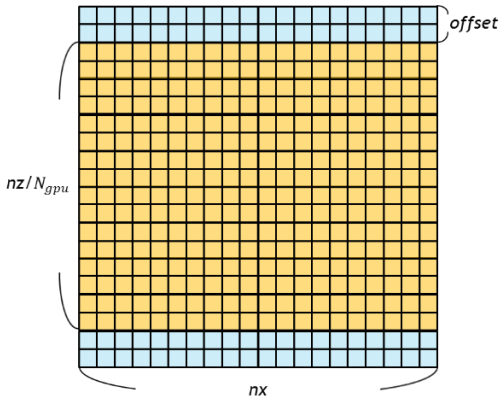

図 14. 1 つの GPU が計算を行う計算領域と境界領域の xz 断面

1GPU 分で必要な領域は計算ノードに搭載されている GPU 数をN_{gpu}, 隣接 GPU に送信する境界領域を offset とすると xz 断面においてオレンジが計算領域となり青の部分が隣接 GPU から送信された境界領域となる(図 14).

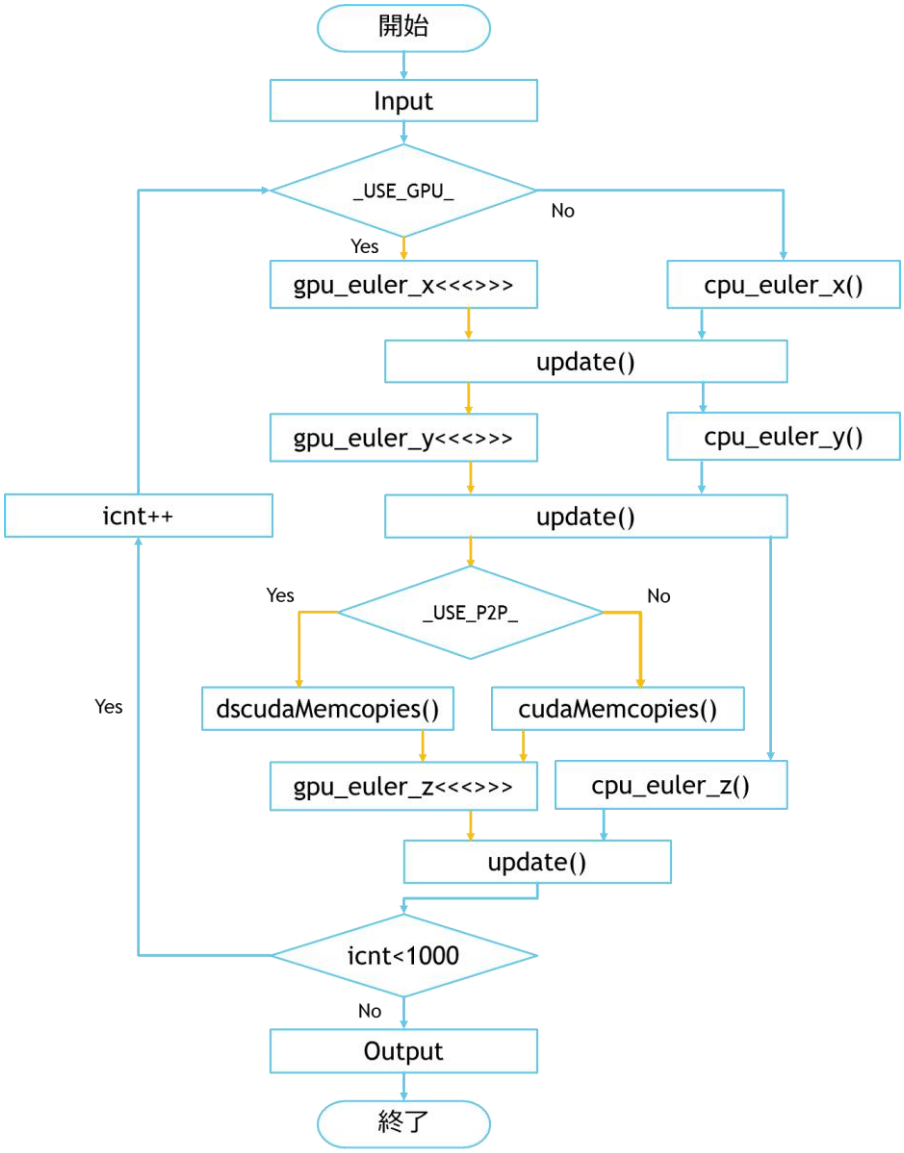

図 15.圧縮性流体計算のアルゴリズム

次に, 圧縮性流体計算のアルゴリズムについて説明する． 初めにグリッドサイズ数に応じて初期条件における各パラメータの値を決定する． 計算は x 軸, y 軸, z 軸の順に 1000 ステップ行われ, 計算終了後は update()で$\rho, \rho u, \rho v, \rho w, \rho e$の 5 変数における値の更新を行う． 複数 GPU を利用する場合, z 軸計算前は隣接 GPU へデータ送信を行うため P2P 機能を使用するか条件分岐を行い, dscudaMemcopies(), cudaMemcopies()で送信する． ステップ終了後は処理性能, 通信速度, 通信時間, 計算時間を出力し終了処理を行う(図 15).

複数台の GPU を扱う場合，境界条件の他にデータ転送時間の隠蔽が重要になる． GPU における Kernel 処理と CPU 命令におけるデータ送受信は同時に行うことが可能であるため隣接 GPU に転送するための計算領域と単体 GPU 内で計算可能な内部計算領域を分けて以下の順序で計算を行うことで最適化を行う(図 16).

(i) 境界領域を含まない z 軸方向の前後 2 格子分を GPU で計算開始
CPU 命令より同時に境界領域をデータ転送
(ii) (i)の計算終了後，先に境界領域の通信が終わった z 軸方向の前後 2 格子分を GPU で計算開始
(iii) (ii)の計算終了後，後に境界領域の通信が終わった z 軸方向の前後 2 格子分を GPU で計算開始
(iv) cudaThreadSynchronize()で CUDA カーネル実行終了((iii)の計算終了後)まで待機させる
(v) 次ステップへのパラメータ値更新

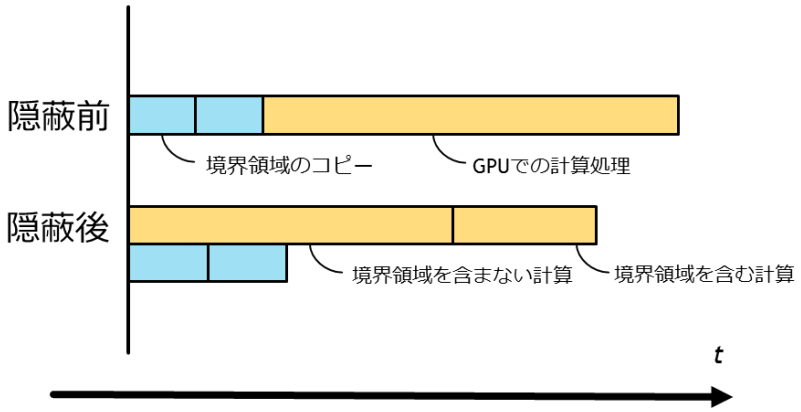

図 16.通信の隠蔽による処理の流れ

5.評価

本評価実験では前章で説明した数値流体計算用コードを用いて多並列時における処理性能, 通信時間を評価する. そのために DS-CUDA の P2P 機能がデータ通信時, 並列処理でどのくらい通信時間を削減できる見込みがあるか確認する必要がある. そこで本格的な実験, 検証をする前に予備実験を行い通信速度の高速化を予測する. そして数値流体計算用コードで P2P 機能使用時にどのくらい高速化できるか測定し, 通信時間や計算時間の計測結果から予備実験の結果を踏まえて考察をする.

5.1 予備実験

予備実験では 2 つの実験を行う. 予備実験 1 では連続領域にあるデータに対してノード間転送の手法を変更し, 計測を行う. 以下の 3 つの手法で比較し通信速度がどのくらい高速化可能か予測する.

5.1.1 予備実験の概要

予備実験 1
(a) クライアントを介した GPU 間データ転送
(b) UVA によるデータ転送
(c) P2P 機能によるデータ転送

予備実験 2 では 5 つのパラメータに対し連続領域でないところからデータ転送を行う. 本評価で使う数値流体計算用コードは 4.2.1 で示したように 5 つのパラメータを境界領域分転送しなければならない. また, 各パラメータは連続領域でないところからデータ転送が行われる. そこで 5 つのパラメータを P2P 機能使用時にノード間の通信台数を増やした場合, 複数のデータ転送を並列実行することでどのくらい通信速度が高速化されるか以下の手法で比較し予測する.

予備実験 2
(a-2) UVA による連続領域にないパラメータ数 5 のデータ転送
(b-2) P2P 機能による連続領域ではないパラメータ数 5 のデータ並列転送
(c-2) P2P 機能による連続領域ではないパラメータ数 5 のデータ 4 台分単方向並列転送
(d-2) P2P 機能による連続領域ではないパラメータ数 5 のデータ 8 台分単方向並列転送

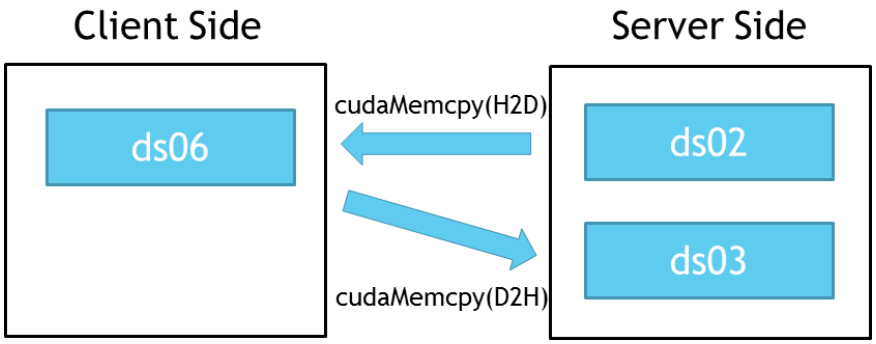

図 17.(a)クライアントを介した GPU 間データ転送

クライアントを介した GPU 間データ転送は 2.2.1 で述べたようにクライアント側における CPU の命令で送信側の GPU は計算領域として必要なデータをクライアント側へ送信してもらい(cudaMemcpy(H2D)), CPU のメモリ上にデータを確保する. 次にデータを受信側の GPU へ送信する(cudaMemcpy(D2H))ことで GPU 間におけるデータ通信が成立している(図 17).

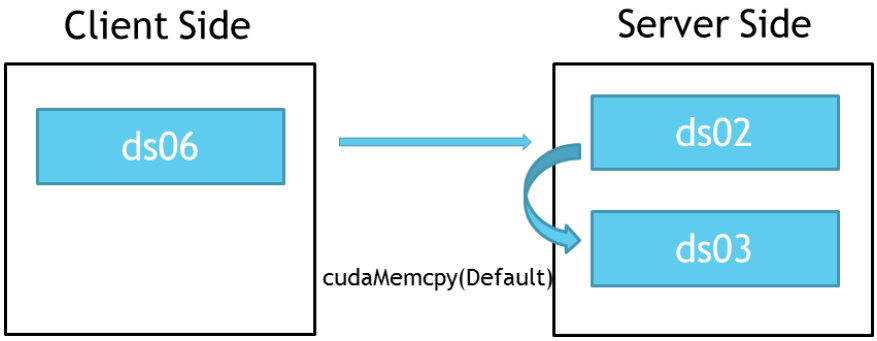

図 18.(b)UVA によるデータ転送

UVA では, CPU と GPU のメモリを, 一つの 64 bit アドレス空間中に配置する. これにより, メモリのアドレスを参照することで, どのデバイスに属するメモリか判定できる. 実際のメモリ転送用 CUDA API である, cudaMemcpy(), cudaMemcpyAsync()では, 通常, ホスト→デバイス, デバイス→ホストなどの転送方向を, cudaMemcpyHostToDevice(H2D), cudaMemcpyDeviceToHost(D2H)などの定数を用いて, 指定する.

ここで, UVA が有効であるとメモリのアドレスから対応するデバイスが判明し転送方向を判定できる. これにより, 転送方向に関わらす"Default"を指定することでクライアントを介してデータ送信する必要がなくなる(図 18).

P2P 機能として DS-CUDA API である dscudaMemcopis()では同様に UVA でメモリのアドレスから対応するデバイスが判明し送信側から受信側へのデータ転送方向を判定できる. cudaMemcpy()の場合は通常 1 回の呼び出しで 1 ペア GPU のデータ転送だが dscudaMemcopies()は 1 回の呼び出しで複数ペアの GPU におけるデータ転送が可能である. 各ペア GPU 転送は並列に行うため転送にかかる時間を短縮することができる(図 19).

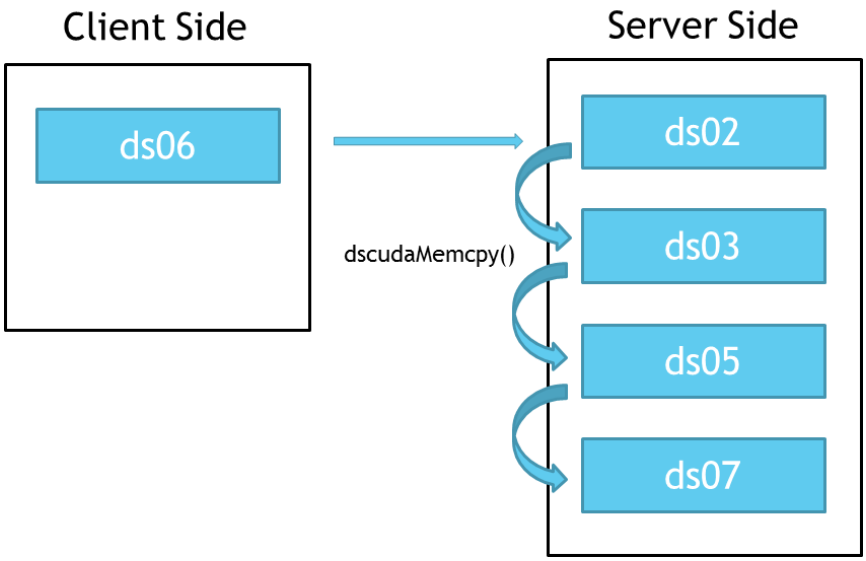

図 19.(c)dscudaMemcopies()によるデータ転送

5.1.2 予備実験 1 評価結果

予備実験 1 の結果を図 20 に示す． x 軸が 1 回の転送量, y 軸が通信速度を表している．1 回の転送量は 1kbyte から 2 倍ずつ増やし 128Mbyte までを計測し，通信速度は対数目盛でとり評価を行った． 図 20 より(b)は(a)と比較して 128Mbyte の時点において約 1.86 倍の高速化している． これは cudaMemcpy()において(a)の場合，(b)より 2 倍の回数 cudaMemcpy()を呼び出すことになるためデータ量が多くなればなるほど 1 回における呼び出し時間が長くなってしまう，その分がオーバーヘッドとなってしまうため(b)の方が通信速度を上回ることになる． 反対に転送量が少ないほど(b)の場合, UVA によるアドレス判定で転送方向を決定する待機時間がボトルネックになるため転送速度が下がっていると考察する． (c)は(b)と比較し 128Mbyte で約 5.08 倍高速化する結果となった． これは dscudaMemcopies()における 4 台分のデータ転送処理が並列で実行されているために高速化されていると推測される．

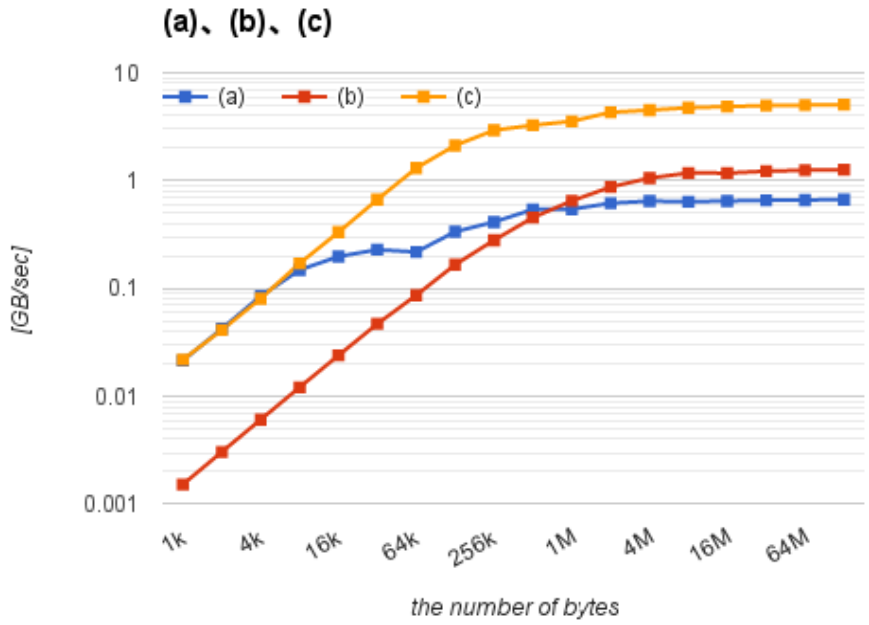

図 20.(a),(b),(c)におけるデータ転送通信速度

5.1.3 予備実験 2 評価結果

予備実験 2 の結果を図 21 に示す． 各軸と計測方法は 5.1.1 と同様である． 図 21 より(a-2)と(b-2)は同程度であることが判明した．このことから CUDA API の cudaMemcpy()と DSCUDA API の dscudaMemcopies()における 2 台における転送速度に差異はあまりないことが判明した．しかし，全体的に dscudaMemcopies()を使用する(b-2),(d-2)において 8kbyte~1Mbyte において転送速度が不安定であることがうかがえる．さらに 4 台における転送速度は 1kbyte のデータ転送において 2 台の(a-2), (b-2)の約 1/10 の転送速度しか出ていない．しかし(d-2)における通信速度は 128Mbyte において(a-2)~(c-2)の約 1.95 倍高速化である． この結果から，予備実験 1 のように転送領域が一つの場合は dscudaMemcopies()が有効(5.08 倍の加速)であるが，予備実験 2 のように 5 つに分かれている場合はメリットが減ってしまう(1.95 倍の加速)ことが分かった．

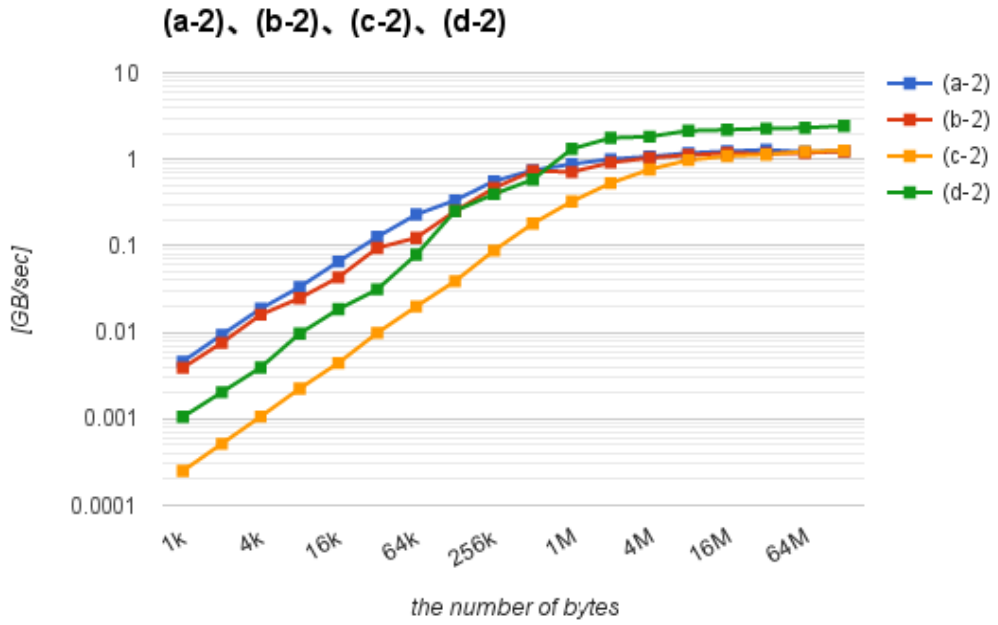

図 21. (a-2),(b-2),(c-2),(d-2)におけるデータ転送通信速度

5.2 評価検証

本研究では予備実験の結果を踏まえ, データ転送における時間がどのくらい削減できるか推測した上で以下の 3 つの方法で評価を行った.

・Native: DS-CUDA を使用しない場合(1GPU, 2GPU)の性能評価(GPU 数に対する計算性能のみ)
・InfiniBand: DS-CUDA を利用した InfiniBand ネットワークによる(1GPU~8GPU)性能評価
・P2P: DS-CUDA を利用した P2P 機能による(2GPU~8GPU)性能評価

評価項目は以下の通りである.
・GPU 数に対する計算性能
・GPU 数に対する処理時間
・InfiniBand 使用時と P2P 機能追加時の性能比
・GPU 数に対する通信時間
・グリッド数に対する通信時間
・グリッド数に対する計算時間
・P2P による通信時間削減率

本研究で使用した数値流体計算用のコードは GeForce GTX 780 で使用可能な計算資源(~3GB)から 1 台分で格子サイズを16^3から256^3まで変化させることが可能である. そこで 1GPU から 4GPU までは16^3から256^3のグリッドサイズで 8GPU は32^3から512^3のグリッドサイズで評価を行った. 1 回あたりの実行におけるステップ数は 1000 回とし, 1 回目の実行結果は GPU の処理に急な負荷をかけることから予期せぬ性能評価が出る可能性があるため無視し, 2 回目以降から計測を 5 回行った. また, 性能評価結果に大きな差が生じてしまった場合, その性能差に再現性があるか確認のため 20 回の計測を行い評価と確認をした.

5.3 評価結果

5.3.1 処理性能

Native 時における処理性能はグリッド数128^3で 2GPU の方が処理性能において向上するという結果になった(図 22). GPU による計算時間は 2GPU 分で計算することになるのでグリッドサイズ数が小さいほど計算処理時間による差が小さくなるためデータ転送の cudaMemcpy()による通信時間がオーバーヘッドになっているのが原因で性能が下がっている. そしてグリッド数128^3以上では 2GPU の計算処理で削減できた分が cudaMemcpy()の通信時間よりも大きくなったことから処理性能が向上している.

InfiniBand 使用時は cudaMemcpy()によって GPU 数が増加するとデータ転送によるレイテンシが大きくなるため全体的に処理性能が下がる傾向にある(図 23). しかし GPU の計算資源は並列数が増えるほど通信時間の影響が減るためグリッドサイズを増加させることで処理性能が向上する.

P2P 機能使用時は InfiniBand 同様に dscudaMemcopies()によって GPU 数が増加するとレイテンシが大きくなるためグリッド数64^3までは処理性能が下がっている(図 24). しかし図 25 より InfiniBand 使用時と比較してグリッド数が増加するほど処理性能の向上は大きくグリッド数128^3において性能が上回っていることから通信時間の並列化効率が大きく関わっていることが考察される. 1 ステップあたりの処理時間については図 26~図 28 に示す.

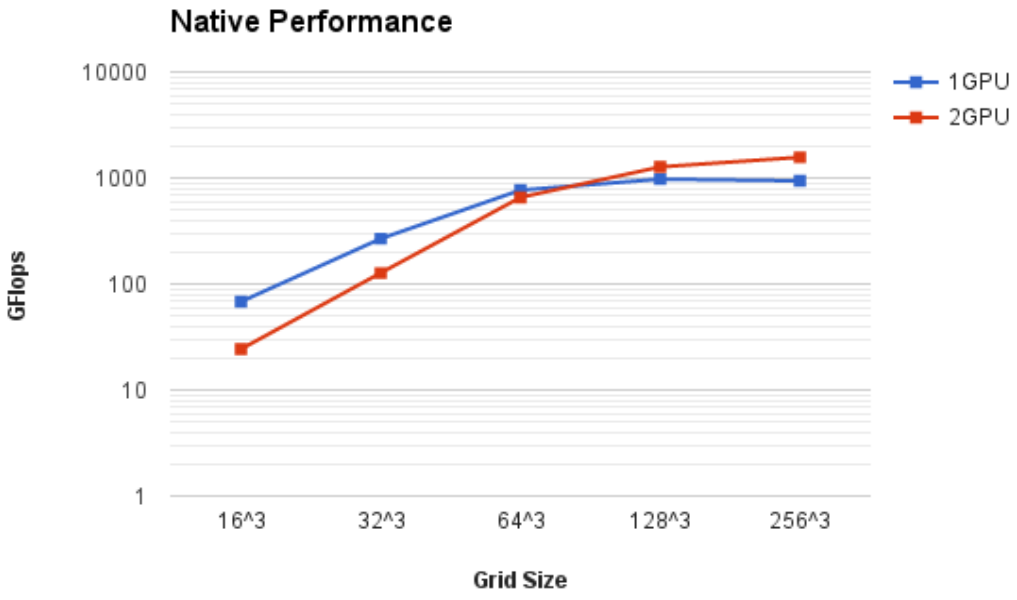

図 22.GPU 数に対する計算性能(Native)

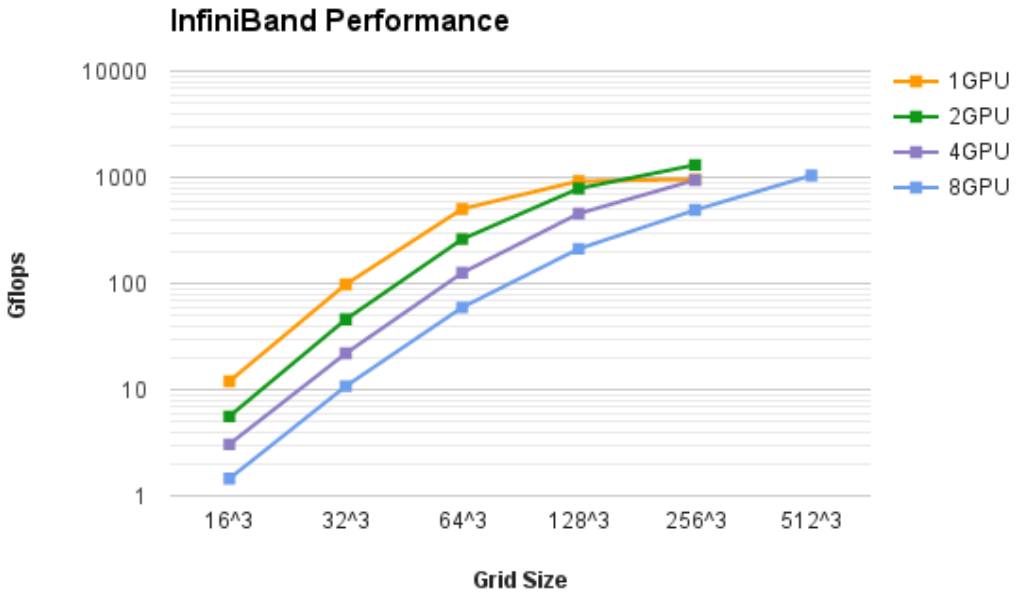

図 23.GPU 数に対する計算性能(InfiniBand)

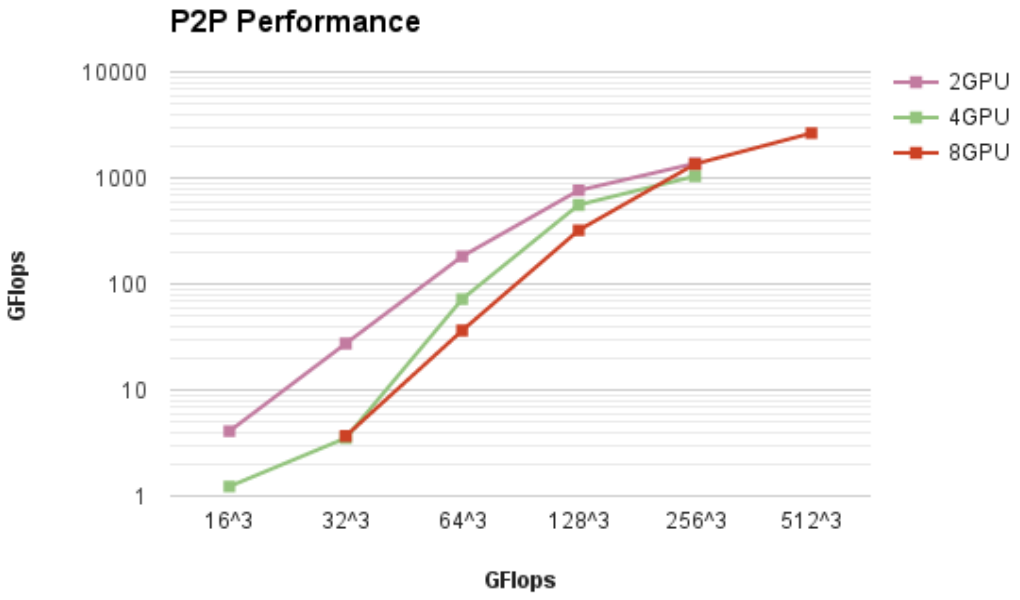

図 24.GPU 数に対する計算性能(P2P)

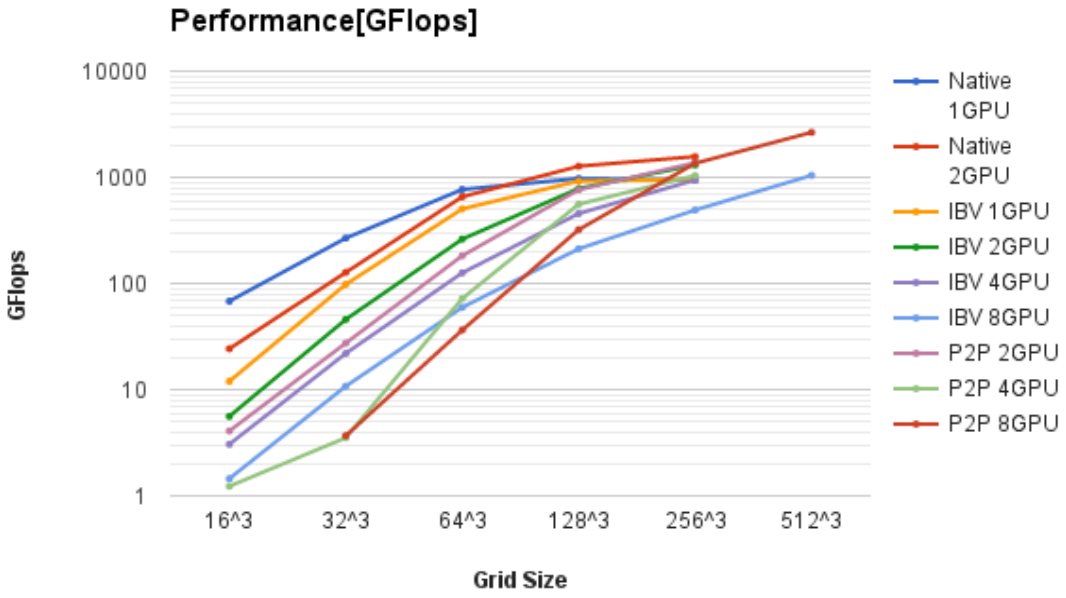

図 25.GPU 数に対する計算性能(全体)

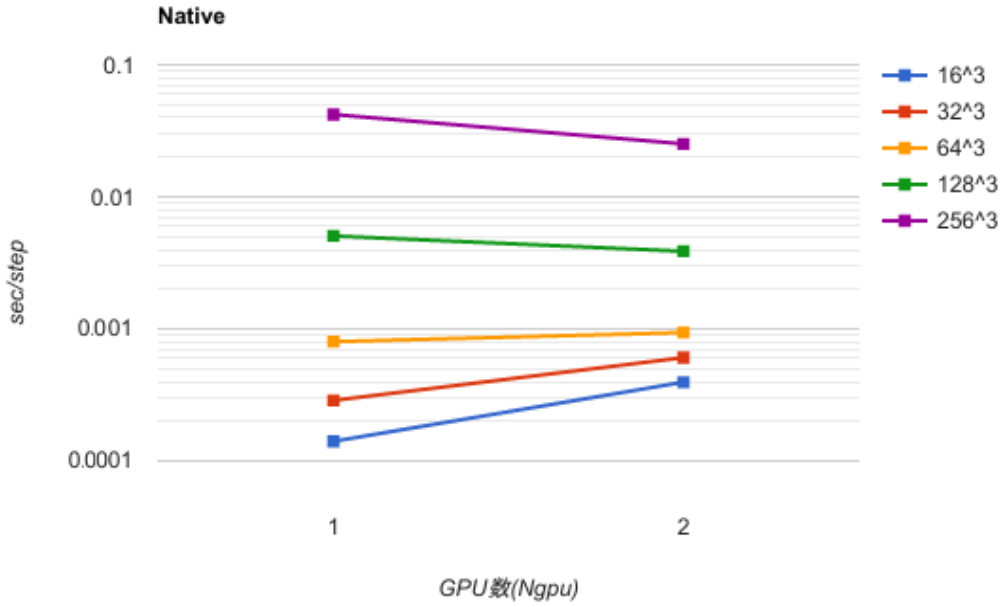

図 26.GPU 数に対する処理時間(Native)

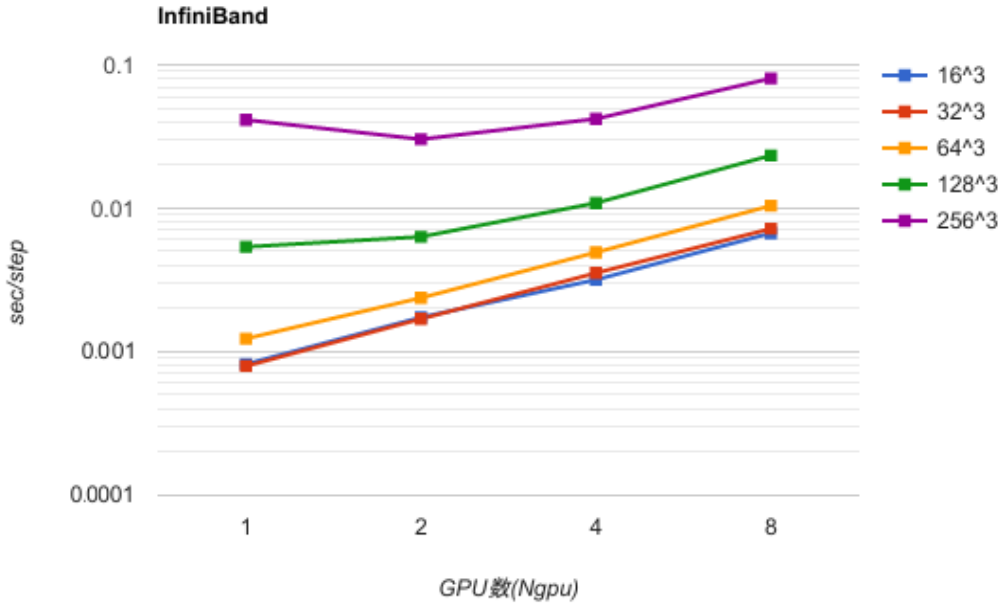

図 27.GPU 数に対する処理時間(InfiniBand)

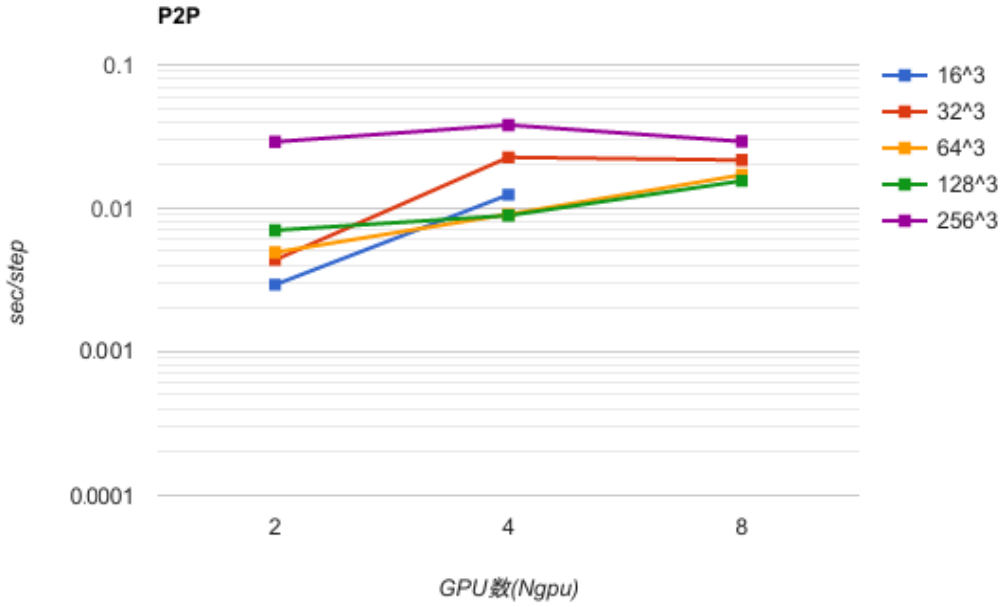

図 28.GPU 数に対する処理時間(P2P)

5.3.2 InfiniBand との性能比

性能評価として最後に InfiniBand 使用時と P2P 機能使用時とで比較を行う. InfiniBand の性能を 1.0 とした時の P2P の処理性能を図 29.30.31 に示す. 本研究では, 通常の DS-CUDA の使い方ではなく, P2P 機能を使うことでどの程度そのメリットがあるかを評価することが目的であるため, InfiniBand と P2P の性能比較が重要になる. グリッド数が小さい場合, 特に 4GPU のグリッドサイズ32^3においては InfiniBand と比較して約 0.18 倍まで性能が落ちるという結果になった. しかし並列数とグリッド数が増加するほど処理性能比は増加しており, 特に 8GPU 時のグリッドサイズ256^3において最大 2.75 倍の処理性能を出すことに成功した.

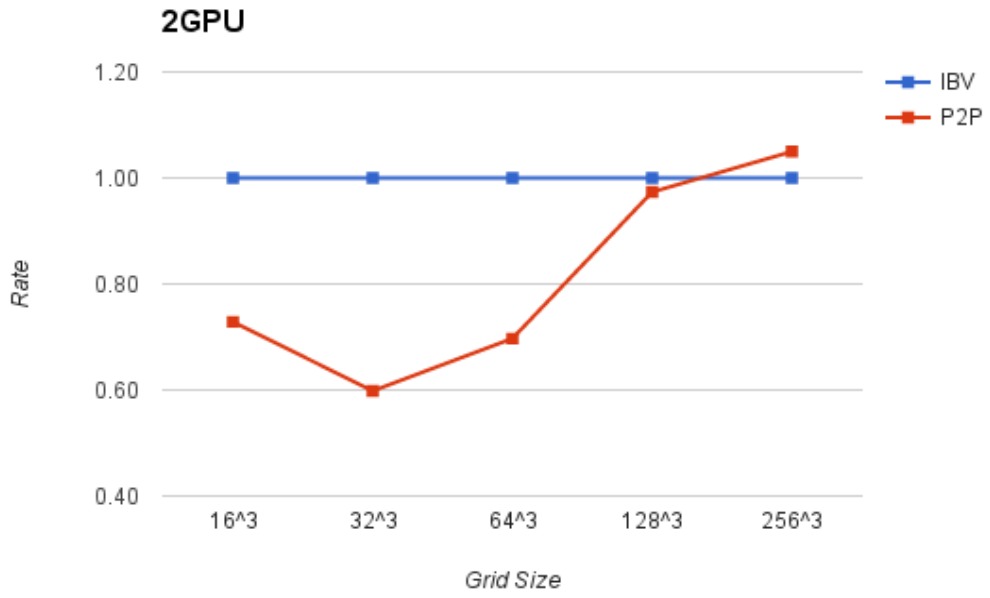

図 29.InfiniBand に対する性能比(2GPU)

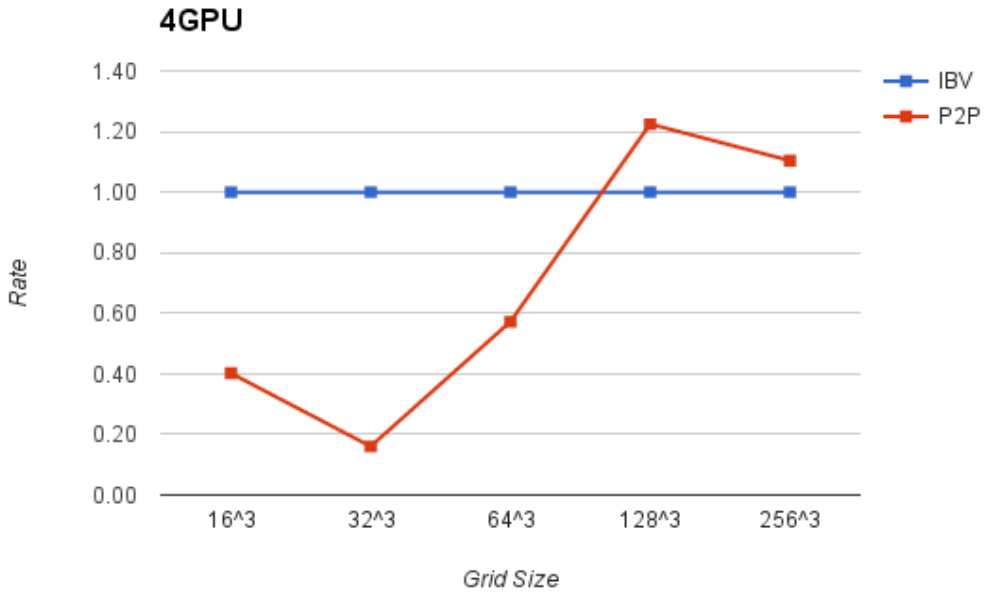

図 30.InfiniBand に対する性能比(4GPU)

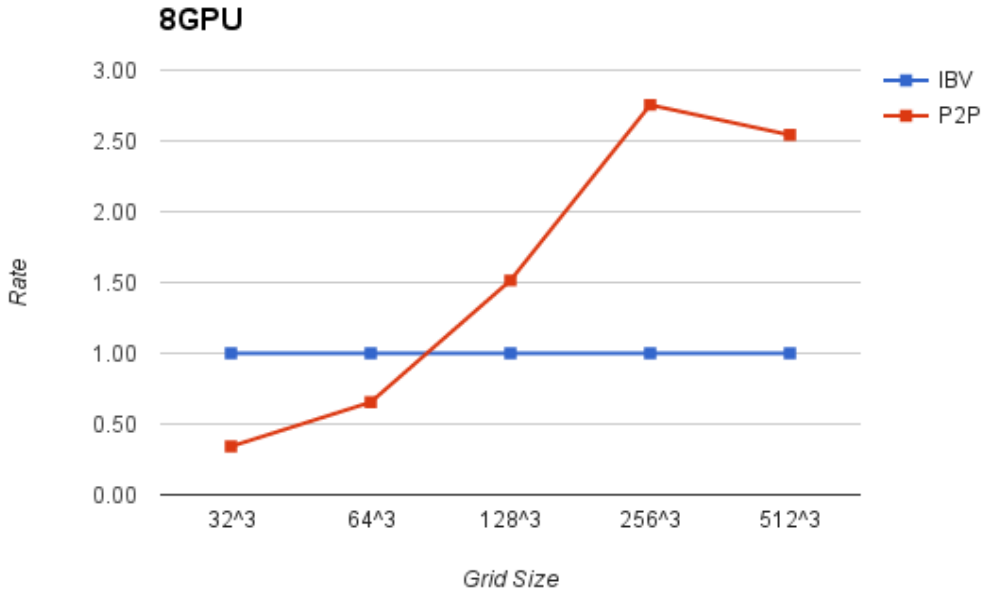

図 31.InfiniBand に対する性能比(8GPU)

5.4 計算時間,通信時間

この節では予備実験で行った転送速度をもとに各環境における処理性能が妥当であるか GPU 数における計算時間と通信時間，グリッド数に対する計算時間と通信速度から考察する． 図 32.33 では InfiniBand 使用時と P2P 機能使用時における GPU 数に対する計算時間を示している． InfiniBand 使用時，データサイズが小さいかつ GPU 数が増えるほど各 GPU で一度に行う並列処理数が減少しレイテンシが大きくなることが判明し，128^3や256^3のような計算も GPU における計算時間の減少がゆるやかになり，GPU 数が増加するほどやがて16^3，32^3グリッドサイズで計算時間が単調増加になることが見られる．またこの結果は P2P 機能使用時でも同様であることがうかがえる．

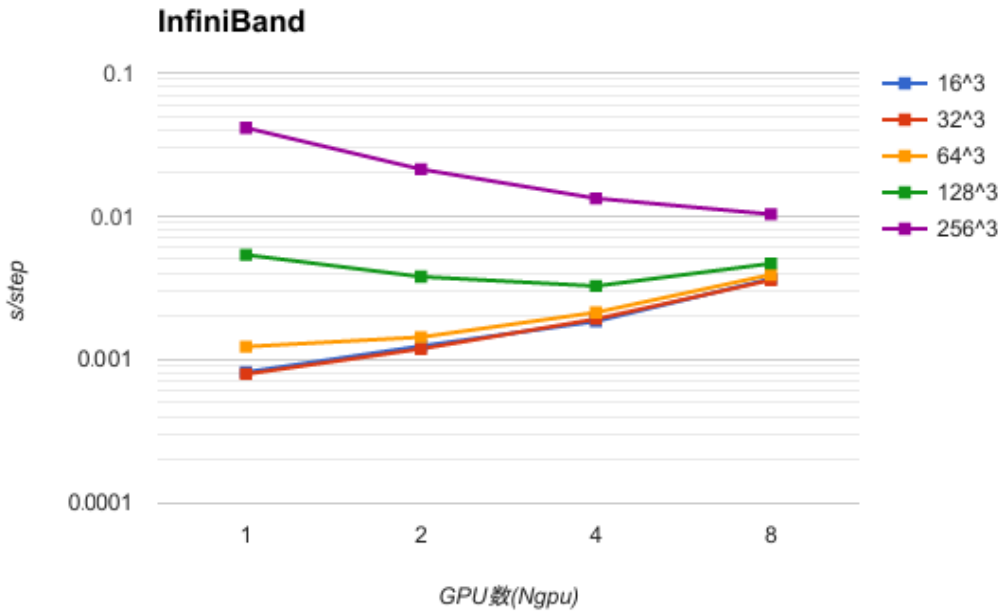

図 32.GPU 数に対する計算時間(InfiniBand)

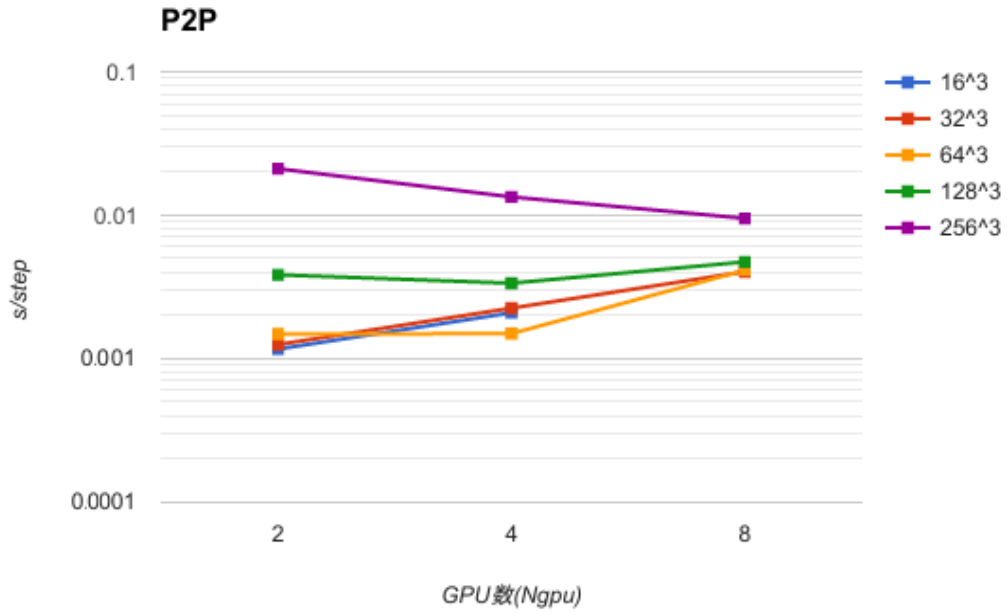

図 33.GPU 数に対する計算時間(P2P)

次に GPU 数に対する通信時間を図 34.35 に示す． InfiniBand 使用時における 1 回の転送量は GPU 数が増加しても変わらない． GPU 数が増えるほど cudaMemcpy()によって呼び出す回数が比例して増加することになるのでデータ通信する時間が全てにおいて単調増加となる． 一方, P2P における GPU 数に対する通信時間は 4GPU において増加傾向にある．dscudaMemcopies()によってデータ通信が並列に実行される為，本来は通信時間が InfiniBand 使用時と比較して緩やかになることが期待される．

しかし，5.1 予備実験の(c-2)で示したように 4 台で連続領域でないメモリから読み込んで通信した場合，2 台で P2P 機能を使用した(b-2)と比較して通信速度が落ちている．また，8kbyte~1Mbyte において転送速度が不安定であることが通信時間における増加の原因と考察する． 8GPU 時においては(c-2)と比較し(d-2)が約 2 倍の通信速度であることから通信時間の削減につながっていると考察する．

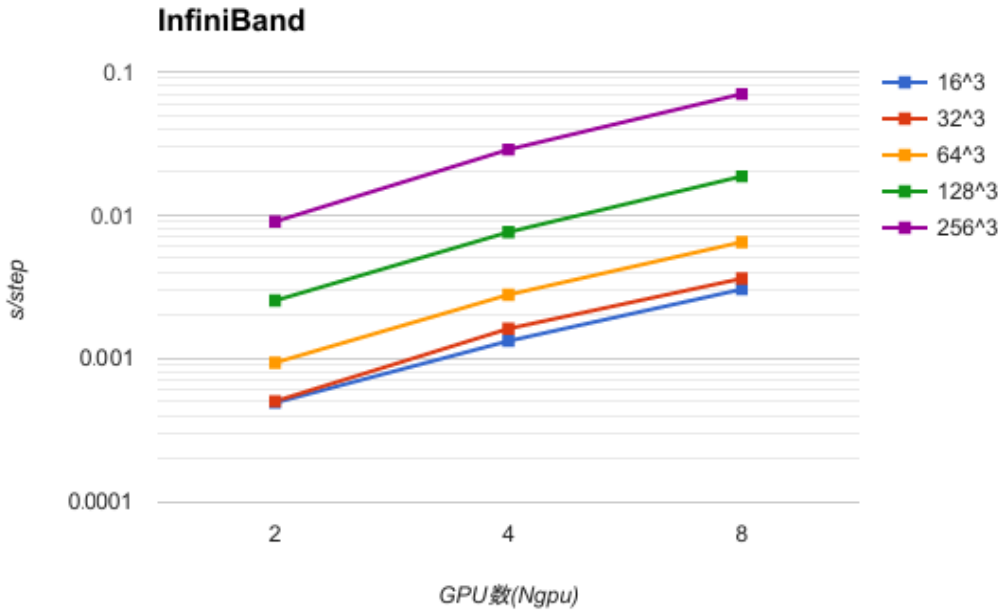

図 34.GPU 数に対する通信時間(InfiniBand)

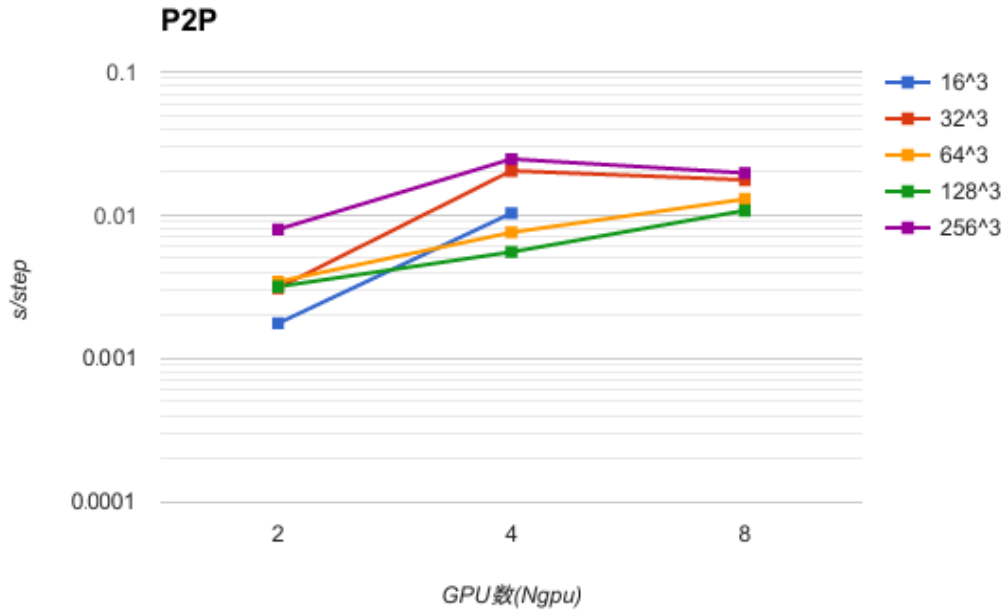

図 35.GPU 数に対する通信時間(P2P)

5.4.1 通信速度

次にグリッド数に対する通信速度を図 36, 図 37 に示す. InfiniBand 使用時における通信速度は cudaMemcpy()における転送量が一定の為 2GPU, 4GPU, 8GPU において同程度の通信速度を出すことが考察される. P2P 機能使用時に対してはプログラムで実際に使用したグリッドサイズにおける転送量を表 3 に示す. そこから予備実験の結果より(b-2), (c-2), (d-2)と比較するとほぼ検証通りの結果となっている. また, 高並列時にアプリケーションプログラムにおいて各 GPU における計算量の処理量が均等でないとロードインバランスが生じ待機時間による処理が長くなる問題が発生する. そこで InfiniBand 使用時と P2P 機能使用時とで 2GPU, 4GPU, 8GPU における計算時間の比較を図 38~図 40 に示した. その結果, 4GPU における計算時間において 10%以上の計算時間に誤差があることが判明した. これは z 軸分割において 2GPU の場合は互いに境界領域が一端分しかないため処理量が均等であることから計算時間に変化がなく, 8GPU においては高並列になるほど境界領域が両端分になるため処理量が均等に近づいていくため計算時間に影響がなくなっていくと考察される.

4GPU の場合においては 2 つの GPU が境界領域の両端分, 残りの 2 つが境界領域の一端分となるため 3 つの中だと処理量が一番均等ではない, 更にグリッドサイズが小さいほどその処理量に差がつくことが計算時間に影響したと考察される.

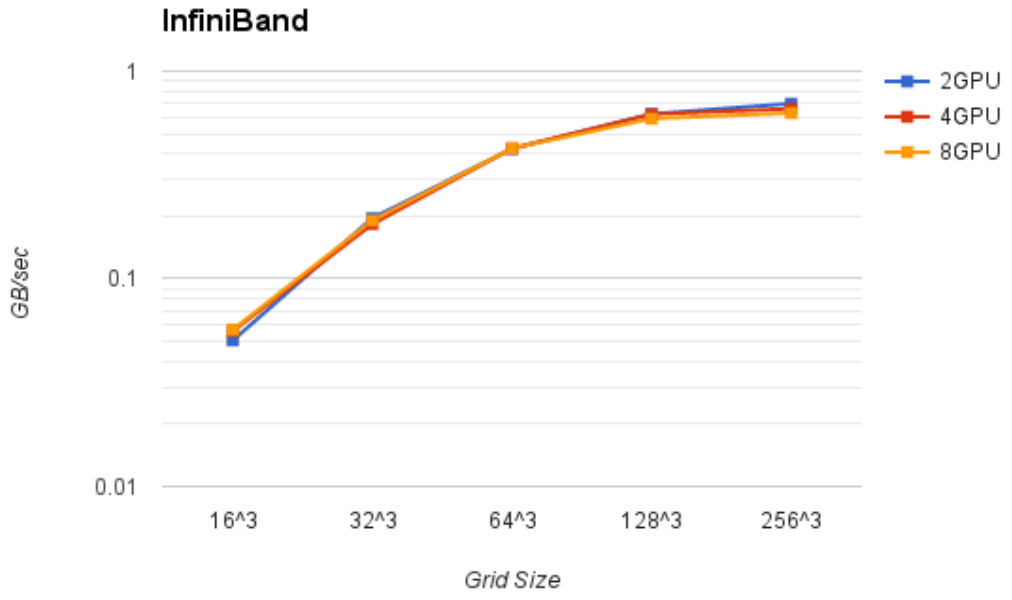

図 36.グリッド数に対する通信速度(InfiniBand)

表 3.グリッドサイズに対する 1 回のノード間転送量

Grid Size	16^3	32^3	64^3	128^3	256^3
byte	10KB	40KB	160KB	640KB	2.56MB

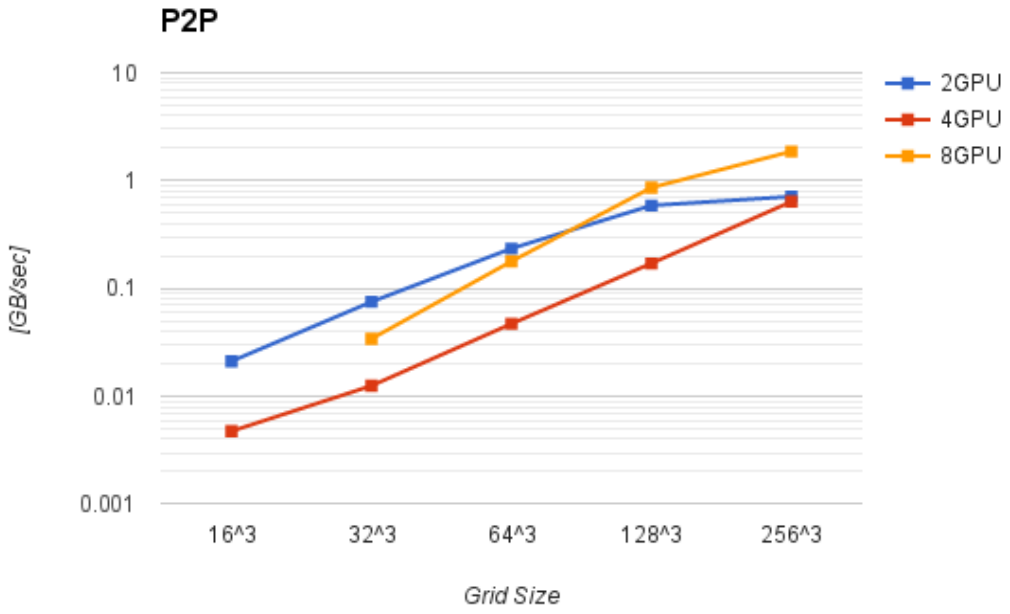

図 37.グリッド数に対する通信速度(P2P)

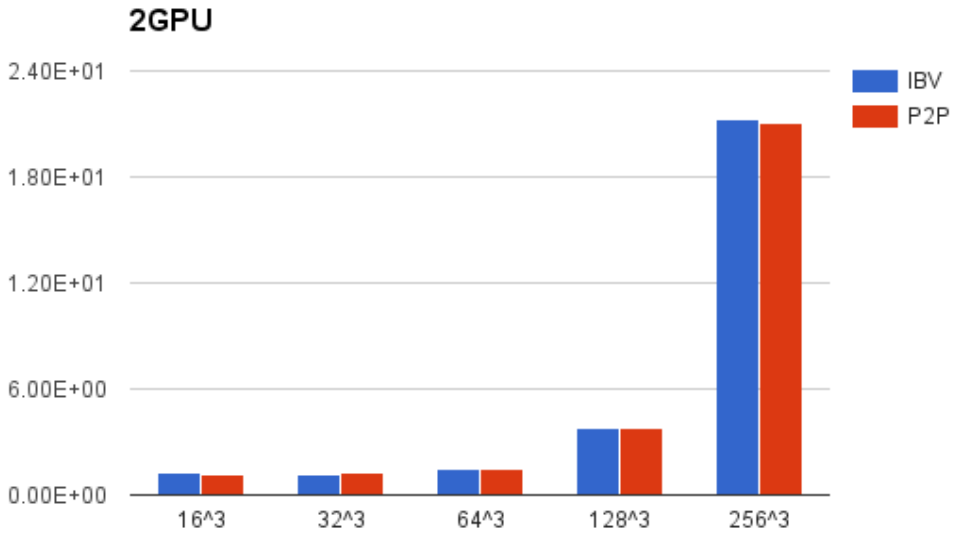

図 38.グリッド数による計算時間の比較(2GPU)

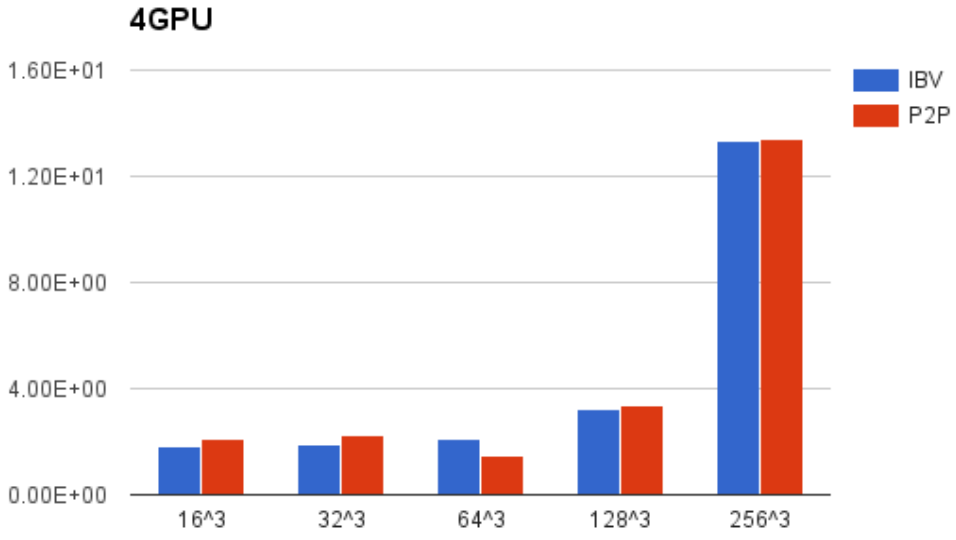

図 39. グリッド数による計算時間の比較(4GPU)

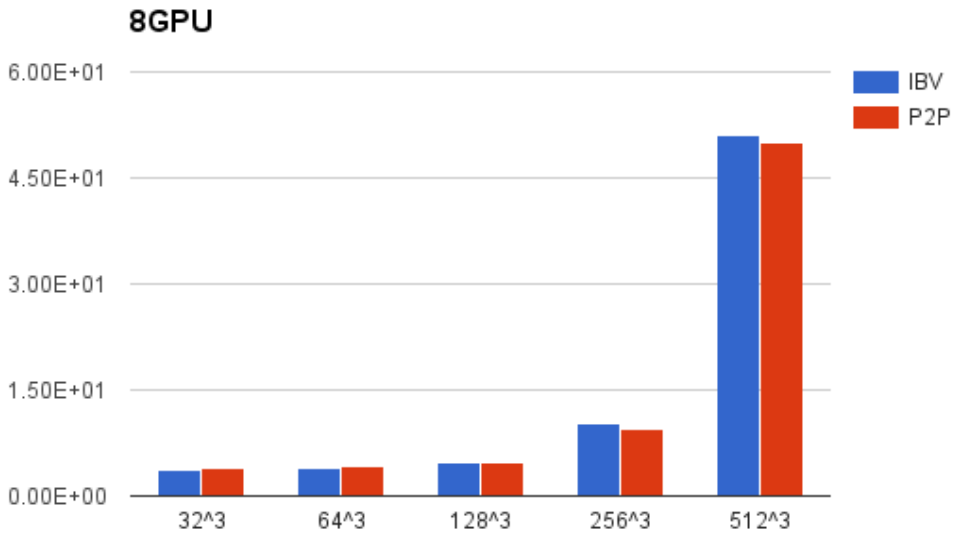

図 40.グリッド数による計算時間の比較(8GPU)

最後にこのアプリケーションプログラムにおいて P2P 機能使用時に InfiniBand 使用時と比較してどれだけ通信時間を削減できたのか表 4 に示す． 負の値は InfiniBand 使用時と比較して通信時間の増加を，正の値は通信時間の削減を表している． 通信時間 P2P 機能はグリッドサイズが大きく高並列であるほど有効であり，8GPU における P2P 機能使用時には512^3グリッドサイズにおいて最大 72.51%の通信時間の削減に成功した．しかし 2GPU における128^3グリッドサイズやそれ以下のグリッドサイズにおいては通信時間が増加し，InfiniBand 使用時よりも処理性能を下げる原因となっている． 故に P2P 機能を使用する場合において転送データ量と高並列化を視野に入れ開発を行うことが重要になってくる．

表 4.P2P による通信時間削減率

P2P	128^3	256^3	512^3
2GPU	−25.31%	11.79%	−
4GPU	27.35%	14.10%	−
8GPU	42.27%	71.90%	72.51%

6. まとめと今後の課題

本研究では, 単一 GPU 向けの数値流体計算用のコードを複数 GPU に対応した. 更に DS-CUDA に搭載されている P2P 機能を用いてノード間の通信をサーバ側だけで行うシステムを構築した. その結果, P2P 機能を使うことにより通信時間を削減して複数 GPU 使用時の性能を向上することが出来た。

評価をする予備実験の段階においては P2P 機能の通信速度を測定し転送データ量やパラメータ数, メモリアクセスの手法を変えることでどのくらいの転送速度が出るか測定した. 予備実験 1 では連続領域で 4 台のノード間における P2P 機能を検証した結果, 最大で約 5.08 倍の通信時間高速化が確認された. 本研究で利用する数値流体計算用のコードを想定した予備実験 2 においては 8 台のノード間において最大で約 1.95 倍の通信時間高速化が確認された.

実際の評価では Native 時, DS-CUDA を利用した InfiniBand ネットワーク使用時, DS-CUDA を利用した P2P 機能使用時における性能評価を行い, 処理性能の点において 8GPU における P2P 機能使用時に256^3グリッドサイズにおいて InfiniBand ネットワーク使用時と比較して約 2.56 倍の高速化を達成できた. また, 通信時間においては P2P 機能使用時512^3グリッドサイズにおいて InfiniBand ネットワーク使用時の通信時間より約 72.51%の削減を達成できた. このため, DSCUDA API の P2P 機能である dscudaMemcopies()はサーバ側におけるノード間 GPU 通信において転送データ量を大きくし高並列化することが通信時間削減に有効であるといえる.

今後の課題として, 本研究では数値流体計算を題材として 1 つの問題に対し P2P 機能を利用し通信時間の削減を図ることで高速化したが数値流体計算分野においては解析が困難とされている問題も無数に存在し, それらに対しても高速化が図れる汎用性が求められる. また, 予備実験で行った結果より連続領域による転送を行うことで通信時間をさらに高速化可能と見られるため, 各パラメータに対し連続領域における最適化を行いノード間におけるデータ転送時間がより短縮されることが期待される.

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.

参考文献

[1] “NVIDIA のビジュアルコンピューティングにおけるリーダーシップ”
http://www.nvidia.co.jp/page/home.html (最終アクセス日 2017 年 1 月 5 日)
[2] “開発者向けの CUDA 並列コンピューティングプラットフォーム”
http://www.nvidia.co.jp/object/cuda-parallel-computing-platform-jp.html
(最終アクセス日 2017 年 1 月 5 日)
[3] Jonathan M. Cohen , M. Jeroen Molemaker , “A Fast Double Precision CFD Code using CUDA” ,
http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/DoublePrecision-CFD-Cohen-parCFD09.pdf (最終アクセス日 2017 年 1 月 29 日)
[4] NVIDIA, “GPU-Based Deep Learning Inference: A Performance and Power Analysis”,
http://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf
(最終アクセス日 2017 年 1 月 29 日)
[5] 大島 聡史, “これからの並列計算のための GPGPU 連載講座”
http://www.cc.u-tokyo.ac.jp/support/press/news/VOL12/No1/201001gpgpu.pdf
(最終アクセス日 2017 年 1 月 5 日)
[6] “MPI Solutions for GPUs”, https://developer.nvidia.com/mpi-solutions-gpus
(最終アクセス日 2017 年 1 月 5 日)
[7] Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability Atsushi Kawai, Kenji Yasuoka, Kazuyuki Yoshikawa, Tetsu Narumi
FUTURE COMPUTING 2012 : The Fourth International Conference on Future Computational Technologies and Applications ISBN: 978-1-61208-217-2
[8] “TSUBAME-KFC is Ranked No. 1 on both Green500 and Green Graph 500 Lists”
http://www.titech.ac.jp/english/news/2013/024456.html (最終アクセス日 1 月 19 日)
[9] OpenACC Home http://www.openacc.org/ (最終アクセス日 2017 年 1 月 5 日)
[10] OpenMP: Home http://www.openmp.org/ (最終アクセス日 2017 年 1 月 5 日)
[11] 老川 稔,野村 昴太郎,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, 1-14 2014/12
[12] “NVIDIA GPUDirect™ Technology”,
http://developer.download.nvidia.com/devzone/devcenter/cuda/docs/GPUDirect_Technology_Overview.pdf (最終アクセス日 2017 年 1 月 5 日)
[13] “NVIDIA GPUDirect”, https://developer.nvidia.com/gpudirect (最終アクセス日 2017 年 1 月 5 日)
[14] “An Introduction to CUDA-Aware MPI”, https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/ (最終アクセス日 2017 年 1 月 5 日)

[15] 佐々木 卓雅, 電気通信大学, GPU および領域分割を用いた粒子法による流体シミュレーションの高速化, 2016
[16] Takashi Shimokawabe, Takayuki Aoki, Naoyuki Onodera, Tokyo Institute of Technology,
A High-productivity Framework for Multi-GPU computation of Mesh-based applications
ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集 2013-12-31, 78 – 86, 2014