GPU クラスタにおける
アプリケーション高速化に関する研究

2016 年 3 月

藤田 典久

GPU クラスタにおける
アプリケーション高速化に関する研究

藤田 典久

システム情報工学研究科
筑波大学

2016 年 3 月

概要

近年，アクセラレータを搭載したシステムが HPC (High Performance Computing; 高性能計算) 分野で広く使われており，数多くのアクセラレータ搭載のシステムが TOP500 リストに掲載され，GPGPU (General-Purpose computing on GPU) と呼ばれる GPU (Graphics Processing Units) を汎用計算に用いる手法が広く用いられている．アクセラレータを持つシステムでは，CPU が全体の性能に占める割合が少ないため，アクセラレータを効率良く利用することが重要である．本論文の目的は，GT5D と QUDA という 2 つのアプリケーションの GPU コードの開発を通じて，どのような要素が GPU アプリケーションの性能に影響を与えるのかを明らかにし，GPU アプリケーションの高速化を行うことである．GPU アプリケーションでは，計算をカーネル関数と呼ばれる単位に分割する．カーネル関数は GPU の動作単位であり，ホストから GPU に対してどのカーネル関数を実行するのかを指示 (オフロード) する．カーネル関数内での計算に必要なデータおよび結果出力については注意が必要であり，カーネル関数の入力と出力を実行の度に CPU〜GPU 間通信でやりとりするのではオーバーヘッドが大きく性能が十分出ないため，データは基本的に GPU 側に置き，CPU〜GPU 間通信を最小化する．また，GPU を含むアクセラレータは PCIe (PCI Express) バスを通じてホストなど外部と通信を行うため，通信コストが CPU のみの環境と比べると大きく，強スケーリング時に性能が悪化しやすい．そこで，アクセラレータ間の通信を，ホストを経由せずに直接通信を行うことで低レイテンシなアクセラレータ間通信を達成し，そのような通信環境下で強スケーリングの性能について評価を行う．本論文では，既存システム用に開発された 2 つの実アプリケーションコードを題材とすることにし，核融合シミュレーションコード GT5D の GPU 化と性能評価および，Lattice QCD (Quantum Chrono-Dynamics) フレームワーク QUDA の TCA (Tightly Coupled Accelerators) アーキテクチャへの適用と性能評価を行う．GT5D は核融合炉におけるプラズマ乱流をシミュレーションするアプリケーションであり，いくつかの大規模な PC クラスタでの動作実績があるが，GPU をはじめとするアクセラレータ対応コードはない．本論文では，GT5D の GPU クラスタ向けコードの開発を行い，どのような要素が GPU の性能に影響を及ぼすのか評価を行う．GT5D は PIC (Particle-in-Cell) コードであり，ステンシル計算における袖領域の交換のための通信を含む．PIC コードおよびステンシル計算は科学技術計算の分野で一般的なものであり，PIC コードおよびステンシル計算の GPU 化を行うことは，今後様々なアプリケーションを GPU 化していく上で重要である．TCA アーキテクチャは筑波大学計算科学研究センターで開発されているアクセラレータ間の通信を低レイテンシで行うためのアーキテクチャである．一般的な環境では，アクセラレータ間で通信を行う際にホストメモリを経由して通信しなければならず，通信に時間がかかるという問題があるが，TCA アーキテクチャを用いる場合では，ネットワークインターフェイスがアクセラレータのメモリに直接アクセスしデータを

転送することで，ホストを経由する通信よりも低レイテンシにアクセラレータ間の通信が行える．そして，TCA アーキテクチャの FPGA (Field-Programmable Array) による実装として，PEACH2 (PCI Express Adaptive Communication Hub Ver.2) が開発されており，NVIDIA GPU 間の直接通信を行える．TCA アーキテクチャを実際のアプリケーションである QUDA に対して適用し，TCA/PEACH2 の実証開発環境である HA-PACS/TCA 環境を用いて，TCA によってアプリケーションの強スケーリング性能が向上するか評価を行う．QUDA は MPI の peer-to-peer 通信を利用しているため，QUDA の RMA (Remote Memory Access) 通信に対応するコードを開発し，TCA および MPI-3 RMA 通信の適用を行う．TCA と MPI の間の性能比較だけでなく，MPI の peer-to-peer と RMA の 2 つの通信手法の間で比較することで，GPU のメモリに直接アクセスできるネットワーク環境において，片方向の RMA (Remote Memory Acccess) 通信が GPU アプリケーションに与える影響についても評価を行う．

目次

図目次

表目次

第 1 章

序論

1.1 研究の背景

近年，アクセラレータを搭載したシステムが HPC 分野で広く使われており，数多くのアクセラレータ搭載のシステムが TOP500 リスト [1] に掲載されている．GPU (Graphics Processing Units) がアクセラレータとして広く用いられており，GPU を汎用計算に用いる事を GPGPU (General-Purpose computing on GPU) と言う．また，規模の小さいコアを多数集積することで高いピーク性能を実現する Intel Xeon Phi[2] といったメニーコアプロセッサも広く利用され始めている．

一般的な PC クラスタ環境では，アクセラレータはホスト CPU と組み合わせて利用される．また，CPU とアクセラレータ間は PCIe (PCI Express) バスによって接続されている．PCIe は汎用のシリアルバスであり，アクセラレータだけでなくネットワークインターフェイスやディスクインターフェイスなどにも用いられている．

PCIe の接続における性能は世代とレーン数によって区別される．世代は gen.1 から gen.3 までの 3 つの世代があり，世代によって 1 レーンあたりの性能が決定され，レーン数は何本の PCIe リンクをまとめて利用するかを示す．例えば，NVIDIA K20X GPU では，最大 gen.2 の 16 レーンによってシステムと接続されており [3]，片方向あたり最大で 8GB/s，両方向では最大 16GB/s の帯域でシステムと接続される．

アクセラレータはホストと独立したメモリを持つ．ホスト CPU とアクセラレータはそれぞれ独立したメモリを持ち，それぞれメモリ空間が独立しており，計算に必要なデータや結果のデータは PCIe バスを利用して CPU と GPU 間で転送される．アクセラレータのメモリは CPU と比べると広帯域のバスで接続されており，前述した K20X GPU では GDDR5 メモリが 250GB/s の帯域で接続されている．一方で，gen.2 16 レーンの PCIe バスの帯域は 16GB/s であり，CPU メモリやアクセラレータのメモリ帯域と比べて低く，CPU〜アクセラレータ間のデータ通信がソフトウェアのボトルネックとなりやすい．

GPU プログラミングでは，計算処理をカーネル関数と呼ばれるある程度まとまった単位で行う．カーネル関数は GPU での実行単位であり，GPU アプリケーションではホストから GPU に対してどのカーネル関数を実行するのかを指示し，計算を行う．GPU プログラミングにおけるデータ管理の方法として，カーネル関数を実行する度に，カーネル関数の入力を GPU に送り，出力のデータを GPU から取り出すという方法も考えられるが，CPU〜GPU 間通信は PCIe バスを通じて行われるため，このような方法を取るとオーバーヘッドが大きく高いアクセラレータの性能を発揮できない．したがって，カーネル関数の実

行単位で入出力を行うのではなく，アプリケーション全体を通してデータ転送を最小化することが重要である．アプリケーションの主たる計算について全て GPU 化を行い，データの初期値を CPU から GPU に送り，計算を行なった後に GPU から結果を取り出すのが理想的である．

従来の環境では，ノードをまたぐアクセラレータ間でデータ通信を行おうとすると，ノードをまたぐアクセラレータ間の通信は直接できないため，ホストメモリに一旦データをコピーしてから通信を行う必要がある．したがって，CPU 間の通信と比較すると余分にデータをコピーしなければならず，通信レイテンシが CPU 間通信と比べると増大する．ノードをまたぐアクセラレータ間の通信性能を改善するために，ネットワークインターフェイスが直接アクセラレータのメモリにアクセスし，アクセラレータ間の直接通信を可能にする技術が提唱されている．

1.2 本論文の目的

本論文の目的は，GT5D と QUDA という 2 つのアプリケーションの GPU コードの開発を通じて，アクセラレータ間の通信手法の最適化を行うことである．ノードをまたぐアクセラレータ間の通信の最適化だけでなく，CPU〜GPU 間の通信の最適化も行う．

GT5D は PIC (Particle-in-Cell) コードであり，ステンシル計算における袖領域の交換のための通信を含む．GT5D は CPU 版のコードがこれまでに開発されているが，アクセラレータに対応するコードはないため，本研究では GT5D の時間発展部分の計算を含めてフル GPU 化を行う．PIC コードおよびステンシル計算は科学技術計算の分野で一般的なものであり，PIC コードおよびステンシル計算の GPU 化を行うことは，今後様々なアプリケーションを GPU 化していく上で重要である．GT5D の時間発展部分のフル GPU 化を行うことで，全ての計算を GPU で行うことが可能になり，計算に必要なデータを全て GPU に置けるようになり，CPU〜GPU 間のデータ転送を最小化できる．

GPU アプリケーションでは計算部分の最適化に加えて，通信の最適化が重要であり，計算と通信のオーバーラップ等を行い通信隠蔽を行う．しかしながら，アクセラレータは世代を増すごとに性能が向上しており，それに伴って計算時間が短くなるため，通信隠蔽が困難になりつつある．アクセラレータは PCIe バスを通じてホストなど外部と通信を行うため，通信コストが CPU のみで行うアプリケーションと比べると大きく，強スケーリング時に性能が悪化しやすい．

そこで，アクセラレータ間の通信をホストを経由せずに直接通信を行うことで低レイテンシな通信を行える通信機構を用いる．筑波大学計算科学研究センターで開発されている TCA (Tightly Coupled Accelerators) アーキテクチャはアクセラレータ間の低レイテンシな通信を実現するものであり，TCA アーキテクチャを用いることで，GPU 間の通信を低レイテンシで行えるようになり，GPU アプリケーションにおける強スケーリング性能が改善される．

QUDA は NVIDIA GPU を計算に利用する Lattice QCD フレームワークであり，QUDA に対して TCA アーキテクチャを適用することで QUDA の通信の高速化を行い，強スケーリング時の性能を評価する．QUDA も GT5D と同じくステンシル計算を行うものであり，袖領域の交換の通信に対して TCA を適用する．TCA を適用した QUDA の通信性能が改善されれば，他のステンシル計算のアプリケーションに対しても TCA を適用すれば性能改善が見込める．

1.3 本論文の構成

本論文の各章の構成は以下の通りである．第 2 章では，NVIDIA GPU に関する一般的な事柄について述べる．GPU の演算コアがどのような構成になっているのか，および，NVIDIA GPU で性能について論じる上で重要になる占有度 (Occupancy) の意味および計算方法について述べる．また，NVIDIA GPU プログラミングはどのように行うのか，MPI を併用してプログラミングを行う際にどのようにプログラミングを行うのかについて述べる．

第 3 章では，核融合シミュレーションコード GT5D の GPU 化と性能評価について述べる．GT5D はいくつかの大規模な PC クラスタでの動作実績があるが，GPU をはじめとするアクセラレータ対応コードはなく，アクセラレータを利用した際の性能については未知数である．CPU クラスタ向けに開発されているアプリケーションである GT5D の GPU クラスタ向けコードの開発を行い，どのような要素が GPU の性能に影響を及ぼすのか評価を行う．GPU の計算カーネルの最適化だけでなく CPU〜GPU 間通信の最適化についても述べる．通信隠蔽を行う際に，計算と MPI 通信のオーバーラップだけでなく，計算と CPU〜GPU 間通信のオーバーラップも行い性能を評価する．

第 4 章では，筑波大学計算科学研究センターで開発されている TCA アーキテクチャについて述べる．TCA に関する研究は筑波大学 計算科学研究センターとの共同研究であり，本論文の成果の範囲ではないが，第 5 章において TCA を利用しているため，第 5 章に必要な TCA と PEACH2 に関する情報を本章で述べる．TCA アーキテクチャでは，ネットワークインターフェイスがアクセラレータのメモリに直接アクセスし，通信時にホストの処理能力やメモリを利用せずに通信を行う．アクセラレータのメモリに直接アクセスすることで，ホストを経由する通信よりも低レイテンシにアクセラレータ間の通信が行える．TCA アーキテクチャの FPGA による実装として，PEACH2 (PCI Express Adaptive Communication Hub Ver.2) が開発されている．PEACH2 は NVIDIA GPU をターゲットとして開発されており，PCIe バスを通じて GPU のメモリにアクセスする．また，ノード間の通信についても PCIe プロトコルを利用しており，GPU 間通信を PCIe という 1 つのプロトコルのみで行える．

第 5 章では，実際のアプリケーションである QUDA に対して TCA アーキテクチャを適用し，TCA によってアプリケーションの強スケーリング性能が向上するか評価を行う．QUDA は NVIDIA GPU を計算に利用する Lattice QCD フレームワークであり，既に MPI の peer-to-peer 通信を用いる並列計算をサポートしている．本論文では，MPI による並列化コードをベースにして，コアとなる通信について，RMA (Remote Memory Access) 通信対応および TCA 化を行う．TCA 化を行う QUDA の通信は，科学技術計算で一般的なものであるステンシル計算における袖領域通信である．性能評価では，MPI peer-to-peer, MPI-3 RMA, TCA の 3 つの通信実装について性能を比較する．RMA 対応は TCA で QUDA を実装するために必要なものであるが，GPU メモリへのアクセスの仕組みは，CPU メモリへのアクセスの仕組みと異なっているため，MPI peer-to-peer 実装だけでなく，MPI-3 RMA 実装とも比較を行うことで，GPU を用いるステンシル計算アプリケーションにおける RMA 通信の効果を評価できる．最後に，第 6 章でアクセラレータを用いる際の通信に関するまとめを，第 7 章で本論文全体の結論と今後の課題について述べる．

第 2 章

研究の背景

2.1　NVIDIA GPU

NVIDIA GPU は多数の計算コア (CUDA Core) と，いくつかの階層からなるメモリから構成される．CUDA Core はいくつかの単位でグループを形成しまとめて管理される．Fermi アーキテクチャでは CUDA Core は 32 個で 1 つの SM (Stream Multiprocessor) と呼ばれるグループを形成しており [4], Kepler アーキテクチャでは 192 個の CUDA Core が 1 つの SMX と呼ばれるグループを形成している [5]. 1 つの GPU に搭載されている SM の数はモデルによって異なっており，CUDA Core の動作周波数および SM の数の差がモデル間の性能差となる．

SM および SMX の内部構造の概要を図 2.1 に示す．SM は CUDA Core だけでなく，レジスタ，シェアードメモリといった記憶領域や，制御機構であるワープスケジューラ，三角関数などの複雑な演算を行う場合に用いられる SFU (Super Function Unit) が含まれている．GPU のプログラムの実行の最小単位はスレッドであり，1 つのスレッドは 1 つの CUDA Core によって実行される．スレッドをまとめたものをブロック，ブロックをまとめたものをグリッドと呼ぶ．プログラムはブロックを単位として分散処理され，1 つのブロックは GPU 内の 1 つの SM に割り当てられる．また，1 つのブロックのスレッドは 32 個単位で分割されワープと呼ばれる．ワープは SM における実行時の制御単位であり，分岐処理や実行スケジューリングはワープ単位で行われる．

GPU には 4 種類のメモリが実装されており，GPGPU プログラミングではメモリ毎の特性を理解し，メモリへのアクセスパターンに適した種類のメモリを使用することが，GPU コンピューティングで高い性能を達成する上で重要である．GPU のメモリ階層を図 2.2 に示す．各メモリの特徴を以下に示す．

グローバルメモリ

GPU にとってのメインメモリであり，全ての CUDA Core の間で共有される．容量は大きいがアクセスに時間がかかる．

コンスタントメモリ

GPU からは読み込みしかできないメモリであり，書き込むためには CPU 側から書き込む必要がある．容量は少ないがコンスタントメモリ専用のキャッシュをもち，グローバルメモリと同様に全てのコアの間で共有される．

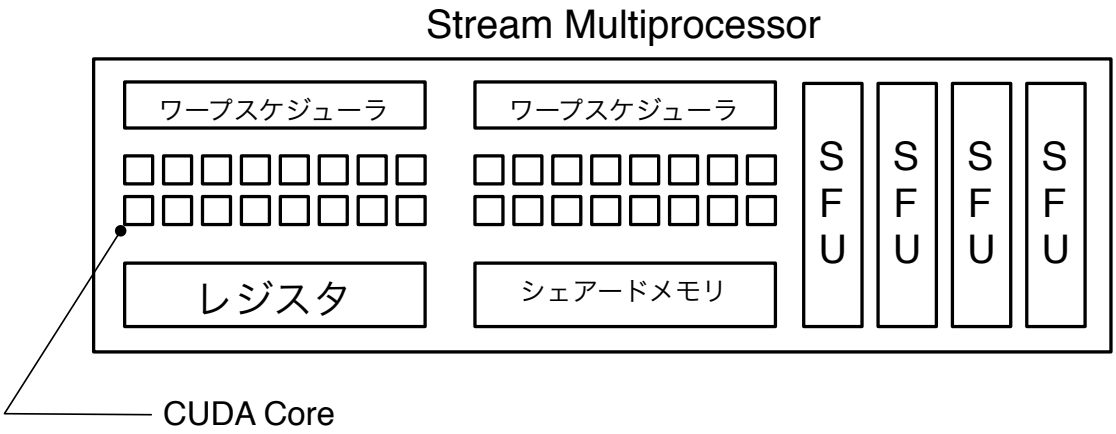

図 2.1 SM, SMX の概念図．ただし，図中央に示す四角は CUDA Core を，SFU は Super Function Unit を示す．

テクスチャメモリ

グローバルメモリの一部をテクスチャメモリとして扱い，テクスチャユニットを経由して参照する．テクスチャユニットによるフィルタ処理が可能であり，少量のキャッシュをもつ．

シェアードメモリ

最も高速にアクセスできるメモリだが，容量は少なく，SM 毎に独立している．

グローバルメモリは L1 および L2 キャッシュを通して参照される．L1 キャッシュは SM に所属し SM 毎に独立しているが，L2 キャッシュはグローバルメモリに付随するため GPU 全体で共有されている．キャッシュラインサイズは L1, L2 共に 128 バイトとなっている．

シェアードメモリは，先頭から順に 4 バイト毎にバンク 1，バンク 2，···，バンク 16，バンク 1，バンク 2··· と，バンクと呼ばれる単位に分割されている．1 つのバンクは同時に 1 つの要求しか処理を行えないため，ワープ内のスレッドのアクセスパターンによっては，バンクコンフリクトと呼ばれる状態が発生する．また，コンフリクトしたバンクの数とあわせて，n-way バンクコンフクリトとも呼ばれる．バンクコンフリクトが発生すると，ワープ内のスレッドのシェアードメモリへのアクセス要求を一度に処理できず，バンクコンフリクトが発生した回数分に分けて処理を行うため，バンクコンフリクトのないアクセスよりも性能が劣化する．バンクコンフリクトが発生しない状況ならば，シェアードメモリは，1 バンク当り 4 バイト/クロックのデータの転送を行える帯域をもつ．バンクコンフリクトはワープ内の 2 つ以上のスレッドが 1 つのバンクにアクセスすることによって発生する．ただし，シェアードメモリはデータの放送処理 (Broadcast) をサポートしており，バンクコンフリクトを軽減できる場合がある．シェアードメモリへのアクセス時に，同じアドレスへのアクセスを纏めて 1 つにして処理され，バンクコンフリクトは発生しない．

シェアードメモリと L1 キャッシュは同じメモリ領域を共有しており，64KB の容量を分け合う．API でサイズ割り当ての比率を変更でき，シェアードメモリ 16KB-L1 キャッシュ 48KB の設定と，シェアードメモリ 48KB-L1 キャッシュ 16KB の設定の 2 通りの容量設定がある．

CPU のアーキテクチャと同様に，CUDA Core の実行する命令のデータはレジスタから入出力される．

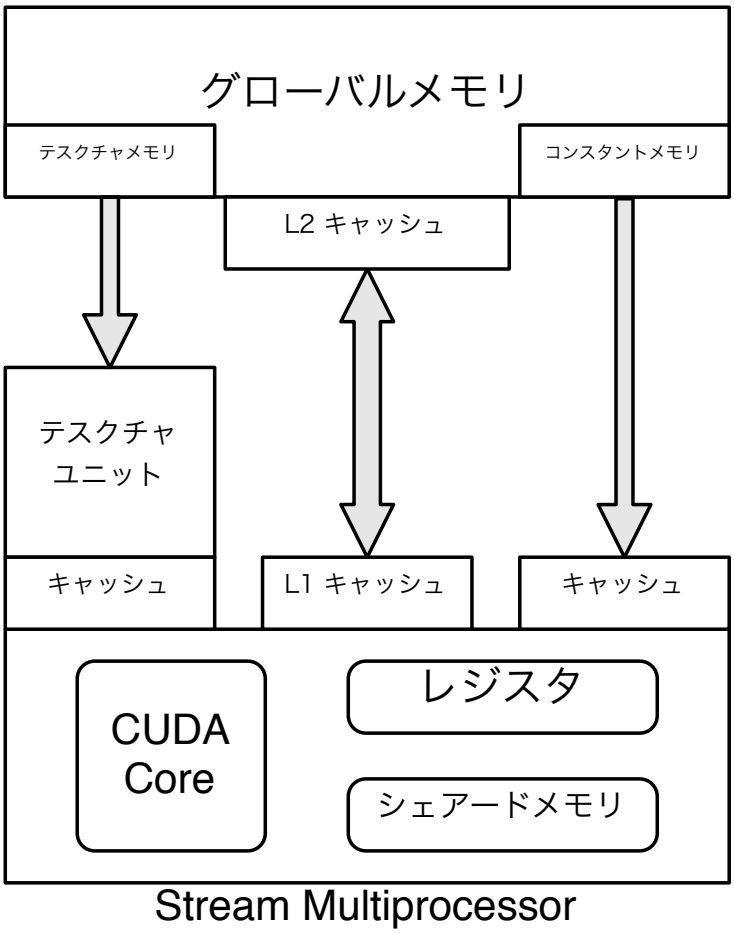

図 2.2　GPU のメモリ概念図．矢印はメモリからのデータの流れを表す．テクスチャメモリ，コンスタントメモリは読み込みのみ，グローバルメモリは読み書き可能である．

図 2.1 からもわかるように，NVIDIA GPU におけるレジスタは各 CUDA Core に実装されているのではなく，SM にレジスタが実装されており，どの CUDA Core がどのレジスタを利用するのかについては，実行時にレジスタの割り当てが行われる．CUDA におけるレジスタはある程度の大きさを持つメモリと捉えることもできるが，割り当ては CUDA コンパイラおよびドライバによって自動に行われ，プログラマは意識することはない．

2.2　ブロックの割り当てと占有度 (Occupancy)

以下の Occupancy の計算に関する各種パラメータは Fermi アーキテクチャにおけるものであり，Kepler アーキテクチャでは異なる．2 つのアーキテクチャで差異が発生する理由は，レジスタ数やワープ最大数といった各パラメータが Fermi と Kepler とで異なる事からであり，計算の手法そのものは同じ計算となるため．Kepler アーキテクチャにおける計算の詳細は割愛する．

1 つの SM は，最大 48 個のワープ (スレッド換算で 1536 個) を制御する能力をもつ．しかしながら，シェアードメモリやレジスタのリソースには限りがあるため，常に 48 個のワープを実行できるとは限ら

ない．何割のワープを制御できているかを示す割合を占有度と呼ぶ．占有度が高いということが，高いパフォーマンスを発揮できることに繋るわけではないが，GPU プログラミングは多くのワープを動かすことで，メモリなどのレイテンシを隠蔽しているため，一般的に低い占有度は性能が劣化することに繋る．

1 つの SM に割り当てられるワープの数は以下の 4 つの制限の中で最も厳しいものとなる．

1. SM の実行可能なワープの最大数.
2. (実行対象のプログラムの 1 ブロック当りのワープ数) × (SM の実行可能なブロックの最大数).
3. (SM のレジスタの本数) / (実行対象のプログラムの 1 ブロック当りのレジスタ使用本数).
4. (SM のシェアードメモリの容量) / (カーネルのブロック当りのシェアードメモリの使用容量) × (カーネルのブロック当りのワープ数).

例えば，シェアードメモリ 16KB の設定で，1 ブロック当り 256 スレッド，1 ブロック当り 4KB のシェアードメモリを使い，1 スレッドあたり 10 レジスタを使用するプログラムがあるとする．この条件で上述の制限を求めると，1 から順に 48, 64, 102, 32 となり，1SM 当りのワープ起動数は 32，占有度は 0.667 となる．

2.3 CUDA 開発環境と CUDA プログラミング

CUDA (Compute Unified Device Architecture) [6] は NVIDIA 社の GPU で汎用計算を行うための開発環境である．CUDA Toolkit には，C/C++ コンパイラ，ドライバ，ランタイムライブラリ，プロファイラ，CUDA 用 BLAS (Basic Linear Algebra Subprograms) ライブラリである CUBLAS，CUDA 用 FFT (Fast Fourier Transform) ライブラリ CUFFT などが含まれる．

CUDA プログラミングにおいて，GPU で行う処理は関数単位で記述し「カーネル」や「カーネル関数」と呼ばれる．CPU と GPU はメモリ空間が分れているため，CPU から GPU のメモリ，あるいは GPU から CPU のメモリへ直接アクセスできない．したがって，計算や通信に必要なデータは，`cudaMemcpy` といった API を用いて CPU と GPU の間でデータを転送する．図 2.3 の様に，計算用のデータを GPU へ送り，カーネルを起動して計算を行い，結果を GPU から転送するという手順が基本的な CUDA プログラミングの流れとなり，CPU プログラミングには存在しない転送に伴うオーバーヘッドが存在する．

カーネルは CUDA における基本的な実行単位であり，ホストから GPU に対してどのカーネルを実行するのかを指示する．カーネルは CPU コードにおける関数に相当するものであり，ある程度まとまった処理を記述する．一般的に，CPU コードにおける多重ループや関数を 1 つの単位としてカーネル関数を記述する．しかしながら，カーネルの起動や終了待機の処理は PCIe バスを通じて GPU とやりとりがあるため，CPU における関数呼び出しと比べるとオーバーヘッドが大きい．したがって，1 つのカーネル関数の計算量が少なすぎると，GPU 制御のオーバーヘッドによって性能が低下するため，カーネル関数の粒度には注意が必要である．

カーネル関数の例を図 2.4 に示す．配列 `x` の各要素を `alpha` 倍し，その値を配列 `y` に足すという単純なものであり，このコードは GPU 上で実行される．`__global__` は CUDA 独自の関数に対する属性であり，`__global__` な関数はホストから呼び出し可能な GPU 上で実行される関数を意味する．`__device__` 属性もあり，`__device__` 関数は `__global__` と同様に GPU 上で実行されるが，ホストからは呼び出せない

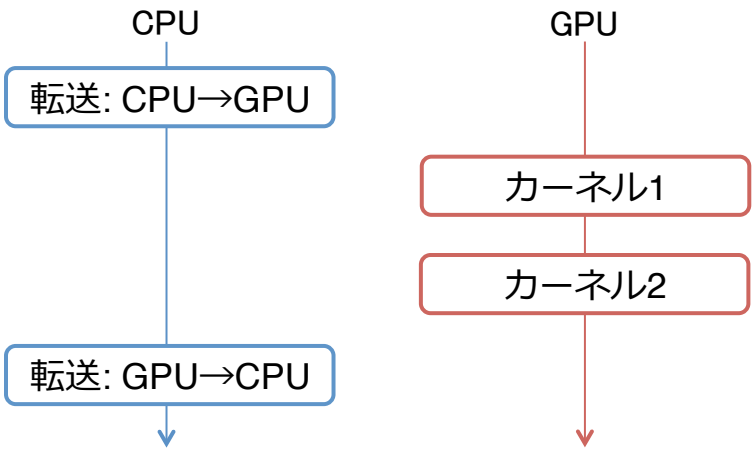

図 2.3　一般的な CUDA プログラミングの流れ.

```
__global__ void saxpy(float alpha, float* x, float* y) {
  int i = threadIdx.x;

  y[i] = alpha * x[i] + y[i];
}
```

図 2.4　CUDA カーネルのコード例.

関数を意味する．また，カーネルを呼び出すホスト側のコードの例を図 2.5 に示す．なお，`cudaMalloc` 関数は GPU 上のメモリを確保する CUDA API，`cudaMemcpy` 関数は CPU〜GPU 間のデータ転送を行う CUDA API であり，それぞれ CPU コードにおける `malloc`, `memcpy` 関数に相当するものである．図 2.4 と図 2.5 からわかるように，CPU 上で動くコードと GPU 上で動くコードは分断されており，別々に記述しなければならない．カーネル関数を呼び出す際は，CUDA の拡張構文を用いる (図 2.6)．通常の C++ の関数呼び出しと同じような構文であるが，CUDA 独自の記法である “<<<>>>” を用いて GPU 上でのブロック・スレッド実行数を指定する．

全ての GPU に対する操作は PCIe バスを経由して行われるため，PCIe バスを経由するオーバーヘッドが存在する．したがって，GPU プログラムではカーネル関数の呼び出し回数や CPU〜GPU 転送の回数を少なくすることが望ましい．実際のアプリケーションを記述する際は，初期のデータを GPU に書き込んだ後は全ての計算を GPU で行い，最後に結果を GPU から読み出すという処理の流れが一般的である．

2.4　CUDA プログラミングと MPI 通信

本節では CUDA による GPU プログラミングと MPI による通信を併用する場合のプログラムの構造について述べる．

まず，典型的な MPI 実装によるプログラミングについて述べる．通常の MPI では，通信の対象とで

```
const int N = 256;
const int SIZE = N * sizeof(float);

/* allocate memory on GPU */
float *x, *y, *z;
cudaMalloc((void**)&x, SIZE);
cudaMalloc((void**)&y, SIZE);
cudaMalloc((void**)&z, SIZE);

/* copy initial data to GPU */
cudaMemcpy(x, initial_x, SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(y, initial_y, SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(z, initial_z, SIZE, cudaMemcpyHostToDevice);

/* call saxpy kernel */
saxpy<<<1, N>>>(2.0f, x, y);
saxpy<<<1, N>>>(-10.0f, x, z);
```

図 2.5 CUDA カーネルを呼び出すコード例.

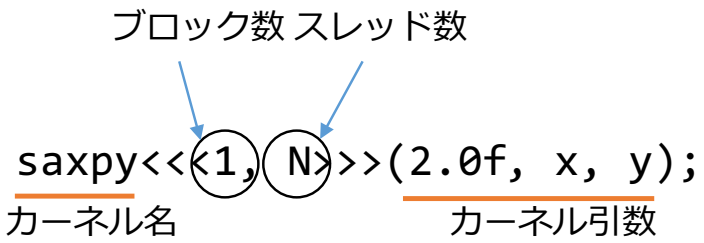

図 2.6 CUDA カーネル関数呼び出しの文法.

きるメモリは CPU のものに限られており，アプリケーションによる明示的な CPU〜GPU 転送が必要である．図 2.7 にコード例を示す．図 2.7 では，計算結果のデータを CPU 上の配列 `x` から CPU 上の配列 `x_host` に一旦コピーし，それから `MPI_Send` 関数を呼び出すことで MPI を用いてデータの送信を行う．受信側も同様に `MPI_Recv` でデータを受信した後に，`cudaMemcpy` 関数で GPU にデータを書き込む．

図 2.7 の例では，CPU〜GPU 間通信と MPI 通信は同時には実行されず，CPU〜GPU 間通信の後に MPI 通信が行われる．通信の最適化として，データを細切れにし，パイプラインで同時に CPU〜GPU 間通信と MPI 通信を行うことは可能であるが，コードが煩雑になってしまうという問題がある．

MPI の実装によっては，CUDA プログラミングを直接サポートするものがあり，(CUDA-aware MPI と呼ばれる)．CUDA-aware な MPI では，MPI ライブラリがポインタの指す先が CPU メモリなのか GPU

```
const int N = 256;
const int SIZE = N * sizeof(float);
const int DEST = 1;
const int TAG = 0;

/* allocate memory on GPU */
float *x, *y;
cudaMalloc((void**)&x, SIZE);
cudaMalloc((void**)&y, SIZE);

/* allocate memory on CPU */
float *x_host;
x_host = (float *)malloc(SIZE);

/* copy initial data to GPU */
cudaMemcpy(x, initial_x, SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(y, initial_y, SIZE, cudaMemcpyHostToDevice);

/* calculation: update x */
saxpy<<<1, N>>>(2.0f, x, y);

/* copy from GPU to CPU */
cudaMemcpy(host_x, x, cudaMemcpyDeviceToHost, SIZE);

/* MPI communication */
MPI_Send(host_x, N, MPI_FLOAT, TAG, DEST, MPI_COMM_WORLD);
```

図 2.7　CUDA プログラミング + MPI 通信のコード例.

メモリなのかを判断し，それぞれに対して最適な通信が行われる．CUDA-aware な MPI のコード例を図 2.8 に示す．図 2.7 と図 2.8 の違いは `MPI_Send` の第一引数に対して CPU のポインタを渡すか GPU のポインタを渡すかの差である．CUDA-aware な MPI では MPI 実装が GPU 上のメモリポインタを認識できるため，図 2.8 のようなコードを書いてもエラーにならず問題なく動作する．また，図 2.8 では明示的に CPU〜GPU 間通信を行う必要はなくなる．

具体的な通信アルゴリズムは MPI の実装に依存するが，CUDA-aware な MPI を利用すれば，アプリケーションのプログラマは CPU〜GPU 間の通信を意識することなく，ノード間の通信を行える．CPU 上の一時メモリ領域や，その領域に対する転送といった本質的ではないコードを書かなくて済むため，コードの見通しも良くなる．また，CPU〜GPU 間の通信に際してデータ長，通信相手，利用する通信デバイ

```
const int N = 256;
const int SIZE = N * sizeof(float);
const int DEST = 1;
const int TAG = 0;

/* allocate memory on GPU */
float *x, *y;
cudaMalloc((void**)&x, SIZE);
cudaMalloc((void**)&y, SIZE);

/* copy initial data to GPU */
cudaMemcpy(x, initial_x, SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(y, initial_y, SIZE, cudaMemcpyHostToDevice);

/* calculation: update x */
saxpy<<<1, N>>>(2.0f, x, y);

/* MPI communication */
MPI_Send(x, N, MPI_FLOAT, TAG, DEST, MPI_COMM_WORLD);
```

図 2.8 CUDA プログラミング + CUDA-aware な MPI の通信のコード例.

スの種類といった情報に依存した最適化を行う自由度を MPI 実装に与えられるため，通信性能も改善すると考えられる.

2.5 GPUDirect RDMA による直接通信

前節では CUDA-aware な MPI による通信プログラミングについて述べたが，CUDA-aware な MPI を利用したとしても，アプリケーションプログラマからは見えないだけで CPU を経由する通信であることに変りはない．CPU を経由しない GPU 間の直接通信が理想であり，直接通信を実現するために NVIDIA は GDR (GPUDirect RDMA) と呼ばれる技術を公開している．GDR を用いることで，GPU メモリのアドレス空間が PCIe アドレス空間へマッピングされ，PCIe アドレス空間を共有するサードパーティーの PCIe デバイスが PCIe バスを通じて GPU のメモリに直接アクセスできるようになる.

MPI プログラミングで GDR を用いて GPU 間直接通信を行うには，MPI ソフトウェア，通信ハードウェア，GPU ハードウェアの 3 つの条件が揃わなければいけない．GDR を利用するには，CUDA 5.0 以降のソフトウェアと，NVIDIA Kepler 世代以降の Tesla 版 GPU を持つ環境が必要である．また，一例として，GDR をサポートする MPI 実装としては MVAPICH2-GDR [7] が，GDR をサポートする通信ハードウェアとしては Mellanox 社 [8] の InfiniBand アダプタ [9] が挙げられる.

2.6　主な GPU 化コードおよび GPU 向け通信機構

NVIDIA の Micikevicius による論文 [10] では，FDTD (Finite Difference Time Domain) 法における GPU の演算最適化および袖領域通信の最適化について述べられている．同一ノード内でのマルチ GPU 利用であるが，フラット MPI による並列化を用いており，最大で 4 GPU までの性能評価が行われている．通信隠蔽が十分に行えている場合は 4 GPU まで良いスケールを示しているが，問題サイズが小さい場合は通信隠蔽が十分に行えないため，4 GPU 利用時で 1 GPU の 2.28 倍の速度しか得られていない．また，4 GPU 利用時に 1 GPU の 4.46 倍というスーパーリニアな性能が得られているが，これは 1 GPU が担当する計算領域が小さくなり TLB (Table Lookup Buffer) のヒット率が上昇するためと述べられている．

下川辺らは論文 [11] において，樹枝状凝固成長シミュレーションの大規模な GPU を用いた計算を行なっている．東京工業大学に設置されていた大規模 GPU クラスタである TSUBAME 2.0 [12] を用いた大規模並列計算を行い，4000 GPU と 16000 CPU コアを用いて 1.017 PFLOPS の性能を達成している．MPI 通信と GPU 計算のオーバーラップだけでなく，CPU も計算リソースの一部として利用するハイブリッド計算を行なっており，強スケーリング，弱スケーリングどちらの性能評価でも，ハイブリッド手法を用いる方が GPU のみの場合よりもスケーリング性能が向上しており，弱スケーリング時に 1.017 PFLOPS の性能を達成している．

TSUBAME 2.0 を用いた大規模計算としては他にも気象シミュレーションコード ASUCA の GPU 化 [13] が挙げられる．ASUCA のフル GPU 化は気象コードとしては初の GPU 化であり，TSUBAME 2.0 の 528 GPU を用いて 15.0 TFLOPS の性能を達成した．ASUCA も同じく通信と計算のオーバーラップを実装しており，袖領域通信を x 軸方向と y 軸方向の 2 つに分割し，通信と計算のオーバーラップを行なっており，通信隠蔽によってスケーリング性能が改善することが示されている．また，計算時間と MPI 通信時間のそれぞれ単体の時間を比較すると，オーバーラップ時の方が遅くなっているが，計算全体の時間で比べるとオーバーラップを有効にした方が高速であり，GPU アプリケーションにおける通信の最適化が重要であることがわかる．

Rietmann らが論文 [14] で Titan [15] を用いて地震波のシミュレーションを行なった際の性能評価について述べている．Titan は Cray XK6 [16] を用いた MPP (Massively Parallel Processing) マシンであり，ORNL (Oak Ridge National Laboratory) に設置されている．前述した 2 つのアプリケーションと同様に通信と計算のオーバーラップによる最適化が行われ，弱スケーリングでの性能評価では並列化効率が 96% と述べられており，良好な弱スケーリングの結果が示されている．しかしながら，強スケーリング・128 ノードで実行した際は，CPU の並列化効率が約 95% なのに対して，GPU は約 50% の性能となっており，強スケーリング時は性能が十分ではなく，低レイテンシな通信手法が必要とされている．

Mellanox [8] の InfiniBand HBA (Host Bus Adapter) は GPUDirect RDMA をサポートしている [9]. GDR をサポートしている環境では，InfiniBand の低レイヤー API である Verbs API に GPU のポインタを直接渡すことができる．また，MVAPICH2 [7] や OpenMPI [17] といった MPI 実装が InfiniBand を用いる GDR 通信をサポートしている．

GPU 間の直接通信ができるネットワークとしては，前述の InfiniBand が有名であるが，InfiniBand 以外にも GPU 間の直接通信が可能なネットワークが研究されている．APEnet+[18] は FPGA による

3D トーラスネットワークであり，GPU との直接通信を可能としている [19]．EXTOLL [20] は DEEP (Dynamical Exascale Entry Platform) プロジェクト [21] の一環として開発されているネットワーク機構であり，InfiniBand と同様に Intel Xeon Phi や GPU 間の直接通信を行える．EXTOLL は DEEP アーキテクチャに採用されており，EXTOLL は APEnet+ と同様にトーラス型のネットワークを構築するが，DEEP アーキテクチャにおける EXTOLL が特徴的なのは，アクセラレータの利用方法である．EXTOLL によって結合されたアクセラレータは，仮想的に 1 つのマシンに接続されているように扱え，あたかも数十や数百の GPU が 1 つのシステムに接続されているかのように見える．

第 3 章

核融合シミュレーションコード GT5D の GPU 化

3.1　GT5D の概要

核融合シミュレーション用プログラム GT5D (conservative global gyrokinetic toroidal full-f five-dimensional Vlasov simulation) [22] は，旋回平均された速度分布関数の時間発展を計算するコードであり，トカマクプラズマ中の乱流現象を記述する．プラズマ中の乱流現象は，プラズマ輸送などのより大きな時間・空間スケールの現象にも影響を及ぼし，例えば，異常輸送や，乱流駆動不安定性などの原因となる．

GT5D の扱う空間を図 3.1 と図 3.2 に示す．GT5D はトーラス配位の実空間 3 次元 (ρ, χ, ξ) (図 3.1) と，粒子の速度空間 2 次元 $(v_\parallel, v_\perp)$ を位相空間変数としている．ここで，$v_\parallel, v_\perp$ はそれぞれ磁力線に平行方向の速度，垂直方向の速度である．荷電粒子は磁力線に巻き付くように運動するが，磁力線を旋回する速度は GT5D が対象とする乱流現象に比べて十分速い．このため，旋回平均によって速度空間変数から旋回位相を消去できる．

GT5D の計算量は，シミュレーションの対象とする装置の規模に依存する．小規模な装置のシミュレーションは計算量が少くて済むが，ITER[23] や DEMO といった次世代の実験炉の乱流現象を計算するためには，現在のスーパーコンピュータでは計算能力が不足しているため，より高速な計算機が求められている．

GT5D は日本原子力研究開発機構で開発されており，GT5D の GPU 化に関する研究は同機構との共同研究である．GT5D はアクセラレータ対応のコードはまだなく，本研究が初のアクセラレータ対応コードとなる．

GT5D の計算は，プラズマ粒子の分布関数を求めるものであり，プラズマ分布の移流方程式およびポアソン方程式を解くものである．移流方程式の非線形項については additive semi-implicit Runge-Kutta (ASIRK) 法 [24] を用いて解く．その際に必要とされる連立一次方程式は，反復法である一般化共役残差 (Generalized Conjugate Residual; GCR) 法を用いる．また，ポアソン方程式は 2 次元の有限要素法が用いられ，こちらの計算では LAPACK を用いて LU 分解による直接法によって解かれる [25].

GT5D のプログラム構造は大きく分別して，初期化部，時間発展部，後処理部から成る．初期化部では，

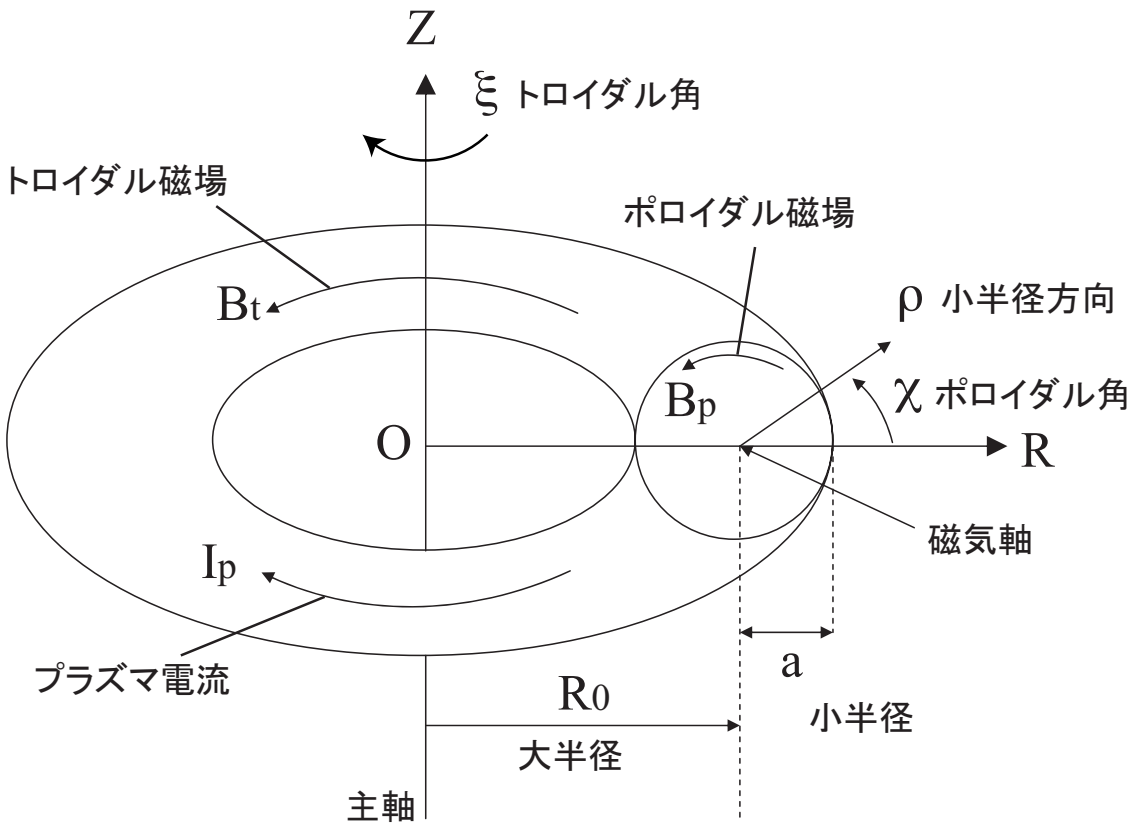

図 3.1 GT5D が扱う問題空間.

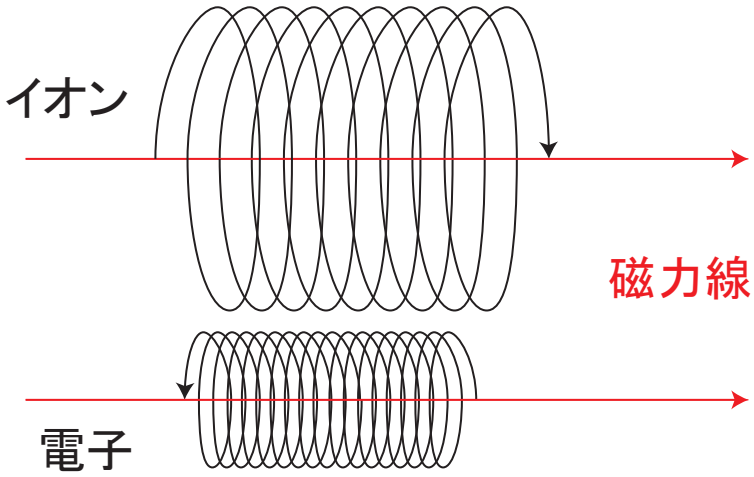

図 3.2 プラズマ粒子の運動.

初期値の計算やリスタートの処理などを行い，時間発展部でシミュレーションを行い，そして，後処理部で各種リソースの解放などを行う．初期化部は時間発展部の反復回数に依らず，一定の時間がかかるが，時間発展部は時間発展の反復回数に比例して計算時間が延びる．したがって，本章では GT5D の時間発展部分を GPU 化の対象とし，時間発展部分のフル GPU 化を目的とする．計算のフル GPU 化を行うことで，計算に必要なデータをすべて GPU メモリに配置し，CPU〜GPU 間のデータ転送については，初期値および結果の転送や，通信に必要なデータの転送のみを行い，CPU〜GPU 間のデータ移動を最小化する．

GT5D の GPU 化の対象となる関数の中で，注目するべき関数は 2 つある．1 つは計算カーネルの `l4dx_s` 関数であり，もう 1 つは `bcdf` 関数である．`l4dx_s` 関数は時間発展部分で最も多い回数呼び出される関数であり，計算時間の多くを占める．したがって，GPU 版の `l4dx_s` 関数の最適化が GPU 版 GT5D の性能を上げる上で重要となる．`l4dx_s` 関数は 4 次精度の無散逸保存型差分を用いて計算を行う関数であり，17 点のステンシル計算を含み，GT5D の計算の中で核となる部分である．もう 1 つは `bcdf` 関数であり，この関数はステンシル計算の袖領域の交換および計算を行うための関数である．ノードをまたぐ GPU 間で通信を行う場合は，一旦 CPU メモリにデータを引き上げ，CPU 間通信を行なった後にデータを宛先の GPU に書き戻すという工程が必要になるため，特に通信レイテンシの面で性能が悪化しやすい．`bcdf` 関数では，通信の結果を必要とする計算，必要としない計算の 2 つに計算を分け，通信を行なっている最中に通信の結果を必要としない計算を行うことで通信隠蔽が行えるため，通信と計算のオーバーラップが行える．GPU を用いて計算を行う場合は，CPU のみで計算を行う場合と比べて袖領域の交換に必要な通信時間が伸びるため，性能に与える影響が大きい．したがって，`bcdf` 関数における通信の最適化は，通信性能を上げる上で重要になる．

3.2　GT5D の GPU 化

3.2.1　PGI CUDA Fortran

GT5D は Fortran で記述されているが，NVIDIA 社の提供する GPGPU 用開発環境 CUDA では C 言語および C++ 言語のコンパイラのみ提供されており，Fortran で書かれたプログラムを GPU 上で実行できない．しかしながら，PGI 社が CUDA コードを生成できる Fortran コンパイラを開発しており，本章では PGI CUDA Fortran コンパイラ [26] を用いて GT5D の GPU 化を行う．

PGI CUDA Fortran は，CUDA C/C++ のように Fortran 2003 の仕様に CUDA のために文法を拡張したコンパイラと，CUDA ランタイムライブラリを Fortran から呼び出すためのライブラリから構成される．PGI CUDA Fortran コンパイラは Fortran コードを C コードに変換し，バックエンドとして CUDA C/C++ コンパイラを呼び出し，GPU 向け実行ファイルを作成する．PGI CUDA Fortran のソースコード例を図 3.3 に示す．CUDA C/C++ における `__global__` と同等の意味を持つ `attributes(global)` や，Shared Memory に領域を確保することを示す `shared` 属性，カーネル起動時のスレッド，ブロックの次元数を指定する `<<< >>>` といったものが Fortran に対する CUDA 拡張である．また，ほとんどの CUDA API について，Fortran 側から呼べるようにバインディングが提供されている．

```
attributes(global) &
subroutine saxpy_kernel(alpha, x, y)
  real, value :: alpha
  real :: x(256), y(256)
  real, shared :: tmp(256)

  tmp(threadIdx%x) = y(threadIdx%x)
  y(threadIdx%x) = &
    alpha * x(threadIdx%x) + tmp(threadIdx%x)
end subroutine saxpy

subroutine saxpy(alpha, x, y)
  real :: alpha
  real, device :: x(256), y(256)

  call saxpy_kernel<<<1, 256>>>(alpha, x, y)
end subroutine saxpy
```

図 3.3 PGI CUDA Fortran の例.

3.2.2 MPI プロセス毎の GPU の割り当ての方針

GPU クラスタでは，1 ノードに複数の GPU が搭載されている環境が一般的であり，そのような環境への対応は必須である．CUDA では，1 ノード内に複数の GPU がある環境で複数の GPU を制御する方法が大きく分別して 2 つある．1 つは操作対象の GPU を切り替えながら GPU を制御する方式であり，もう 1 つは 1 ノード内に GPU と同じ数の MPI プロセスを起動し，プロセス毎に別々の 1 つの GPU を制御する方式である．切り替える方式は効率が良いがプログラムが複雑になりがちであり，MPI プロセス毎に GPU を割り当てる方式では，ノード内のデータ交換でさえ MPI 通信を必要とし，オーバーヘッドがある．本章では，1 MPI プロセスにつき 1 つの GPU を制御するモデルを採用する．オーバーヘッドがあるものの，CPU から見たメモリと GPU のローカリティの確保が容易なためである．

GT5D の GPU 化にあたり，1 つの MPI プロセスがいくつの GPU を制御するかを考える．本章では，1 プロセスあたり 1 GPU を制御する方式を利用し，割り当ての詳細を図 3.5 に示す．また，スレッド数もコアを全て使い切れるように設定する．例えば，8 つのコアを持つ CPU に対して 2 つの GPU が接続されている環境では，OpenMP のスレッド数を 1 プロセスあたり 4 に制限する．スレッド数の設定は `OMP_NUM_THREADS` 環境変数を通じて行う．

HA-PACS は NUMA 構成となっているため，他の CPU にあるメモリへのアクセスは速度面でペナルティがある．また，GPU も同様に，他の CPU の配下にある GPU へのデータ転送は，避けなければなら

```
$ numactl --cpunodebind=0 --localalloc -- ./GT5D
```

図 3.4　numactl コマンドの例.

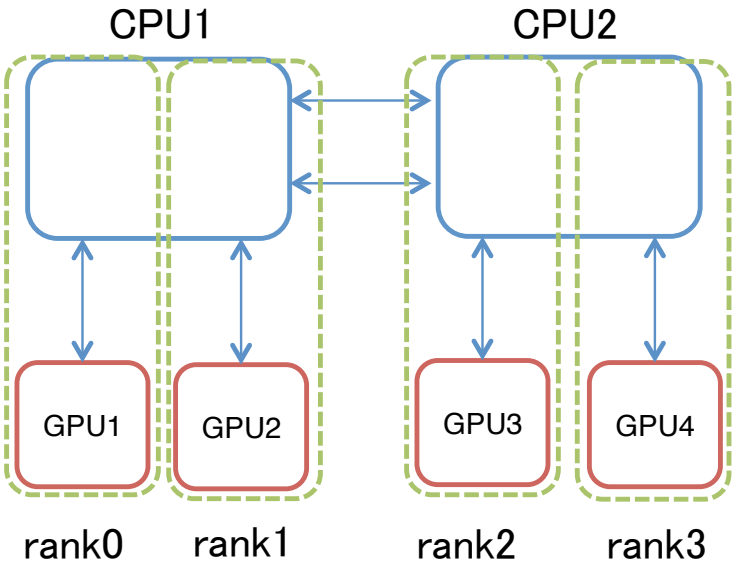

図 3.5　MPI プロセス毎の CPU コアと GPU の割り当ての方法.

ない．前述した 2 つの条件を満たすために，`numactl` コマンドを使用し，あるプロセスが実行される CPU を固定する．`numactl` は NUMA 環境でのリソースを制御するために用いるコマンドであり，プロセスが利用する CPU コアとメモリを限定できる．例えば，図 3.4 の様にコマンドを実行すると，GT5D をノード 0 番の CPU で実行し，0 番 CPU に接続されているメモリ (ローカルメモリ) を利用するという意味になる．

本方針では，GT5D が持つ既存の MPI 並列化をコードを再利用でき，開発が容易であること，また，プロセス毎にデータ参照の局所性があり，NUMA の対応を取りやすいこと，あるプロセスが操作する GPU が，`numactl` コマンドによって設定された CPU と直接接続されていることを保証できること，といった利点があるが，一方で，同じノードに接続されている GPU 間のデータ交換でさえ，MPI を経由せねばならず，オーバーヘッドが発生するという欠点を持つ．

時間発展計算の部分のフル GPU 化を行うが，1 つの巨大な GPU カーネルを作成するのではなく，CPU 版のコードにおけるループや関数を 1 つの単位として，それに対応する GPU カーネルを作成する．ベースとなる CPU 版のコードの構造を保ったまま GPU 化を行うことで，コードの見通しの向上や，一部の計算のみ GPU 版と差し替えられるため動作検証の行いやすいというメリットがある．時間発展部の計算は全て GPU で行うため，計算に必要なデータは全て GPU 上に置き，CPU〜GPU 間の通信を最小限に留める．しかしながら，ノード間通信に必要なデータは CPU 上になければ通信ができないため，通信の前後で発生する CPU〜GPU 間の通信を行いデータ移動を行わなければならない．

表 3.1 GT5D の時間計測結果.

関数名	時間 [ms]	割合 [%]	回数
Others	4447	28.6	
l4dx_s	5668	36.4	30
lfp	3252	20.9	2
l4dx_nl	997	6.4	2
l4dx_l	577	3.7	2
l4dx_r	295	1.9	2
fld_sfls	239	1.5	2
drift_nl	59	0.4	2
dn3d	33	0.2	2
bcv	1	0.0	2
Total	15568		

3.2.3 時間発展部の流れ

GPU 化の方針を立てるため，まずオリジナルの GT5D コード上で CPU のみを用いて実行時間を測定する．時間発展 1 回の時間と呼び出し回数測定結果を表 3.1 に示す．時間発展中で，最も時間のかかる関数は l4dx_s であり，以降，lfp，その他と続くことがわかる．

時間発展部の処理の流れの概要図を図 3.6 に示す．時間発展の中には，内部ループ (図 3.6 の波線部) が 2 つあり，収束判定が満たされるまで繰り返される．例えば l4dx_s 関数は内部ループ内で呼ばれているため，実行パラメータによって呼び出し回数が変化する．また，l4dx_s, l4dx_r, l4dx_l, l4dx_nl の 4 つの関数は，計算のみを含んでおり，MPI 通信を行わない関数であるが，lfp 関数は MPI 通信を含んでいる．したがって，l4dx_s, l4dx_r, l4dx_l, l4dx_nl 関数の方が GPU のみで計算が完結し，GPU 化が行いやすい．また，時間発展中に，関数として分離されていない，小さな DO ループがいくつかあり，それらのループが表 3.1 のその他の部分に該当する．

図 3.6 の波線部からわかるように，内部ループに bcdf という関数が含まれている．表 3.1 からわかるように，bcdf 関数は内部ループに含まれているため，呼び出される回数が多い．bcdf 関数は袖領域の交換のための関数であり，MPI 通信を含んでいる．MPI 通信に用いるデータは CPU のメモリに存在しなければならない．したがって，関数を GPU 化することはできず，必ず CPU で実行しなければならないため，前後の CPU〜GPU 間の通信を回避できない．

GT5D の MPI 並列の分割数は n_R, n_Z, n_μ の 3 変数で表され，n_R と n_Z はそれぞれ図 3.1 における R 方向と Z 方向への分割数であり，n_μ は $v_\perp$ の分割数である．bcdf 関数は，R 方向と Z 方向の袖領域を交換する関数であり，$n_R = 1$ の場合は R 方向への通信は行われず，$n_Z = 1$ の場合は Z 方向への通信を行わない．

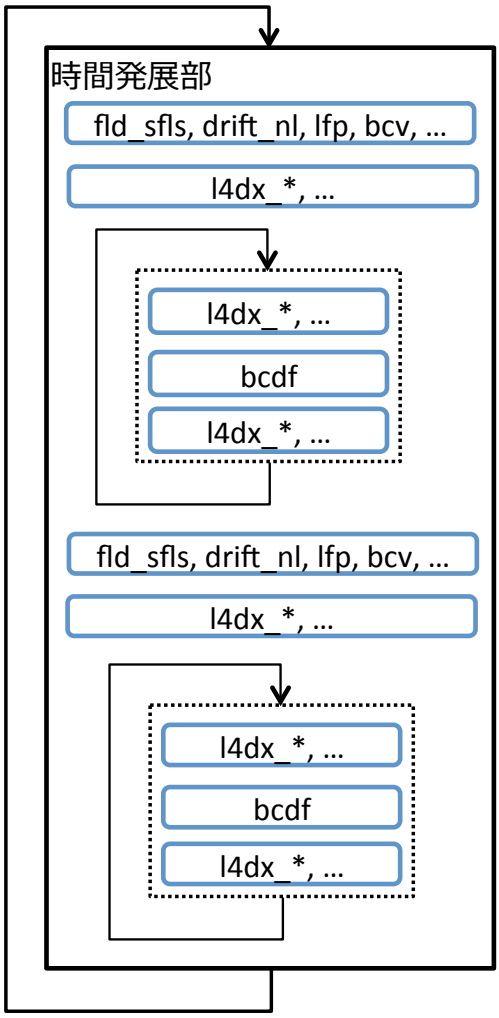

図 3.6　GT5D の時間発展部の概要図．ただし，波線部は内部ループを表す．

3.2.4　GPU 化する範囲

時間発展部分の中で，特に時間のかかる関数である `l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl` の 4 関数を中心に考える．`l4dx_r`, `l4dx_l`, `l4dx_nl` の 3 つの関数は，合計で実行時間の 8.26% しか占めないが，MPI 通信を含まないため GPU のみで処理が完結すること，`l4dx_s` 関数と処理内容が似ており GPU 化するコストが低いこと，CPU〜GPU 間のデータ転送を削減できることといった理由により GPU 化の対象とする．CPU と GPU の間の通信速度は GPU の演算性能に対して相対的に低く，時間発展全体の高速化のために CPU と GPU の間のデータ移動は，必要最低限に抑えなければならない．

前節で述べた通り，`l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl` の呼出の周辺に，小さな DO ループがいくつかあり，時間発展の中で約 40% の時間を占めている．個々のループは計算量としては多くないが，CPU で計算するとなると，`l4dx_r` 関数の場合と同様に CPU〜GPU 間のデータ転送が発生し，性能に悪影響を及ぼすため，図 3.7 のように関数化 (`timedev1`〜`timedev9`) し，GPU で計算する．ある計算を GPU

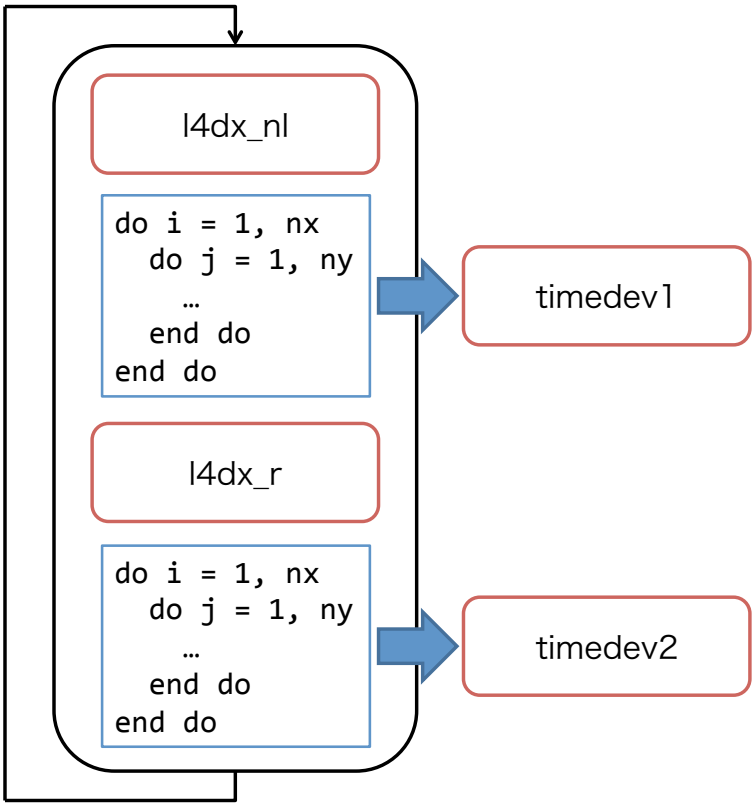

図 3.7 GT5D の時間発展部分の GPU 化の概要図．ただし，青で囲まれた部分は CPU で処理を行うことを示し，赤で囲まれた部分は GPU で処理を行うことを示す．

化するかどうか決定する際は，計算量だけでなく，CPU〜GPU 間のデータ転送量も重要となる．また，GPU よりも CPU で計算する方が速い処理の場合でも，データ移動の時間を含めて比較検討しなければならない．

本節では，表 3.1 の項目の内，`l4dx_s`, `l4dx_r`, `l4dx_l`, `l4dx_nl`, `lfp`, `bcv`, `dn3d`, `drift_nl`, その他の範囲について GPU 化と性能評価を行う．

3.2.5 カーネルの実装方法

GT5D の時間発展部分の GPU 化は，関数を基本単位として行う．GPU で CPU の関数と同じ計算をする関数を作り，関数を差し替えることで GPU 化を行う．1 つの CPU の関数と GPU のカーネル関数が一対一に対応するとは限らず，2 つ以上のカーネル関数から構成される場合もある．例えば，GPU から MPI 通信は制御できないため，関数内で MPI 通信を行なっている場合は通信の前後でカーネルを分割することが必須である．また，前節で述べた通り `timedev` 関数は GPU 固有の関数である．

それぞれの関数内では，並列処理可能なループを 1 つのまとまりとして捉え，GPU のカーネルに変換を行う．ループの反復回数が多い方が処理の並列度が高くなるため，多重ループをカーネル化する場合は，最外ループだけでなく内部のループも併せて並列化を行う．ループの処理を GPU で行う場合は，GPU のスレッド・ブロックをどのようにループ処理に割り当てるかを考えなければならない．ループの 1 イテ

```
do i = 1-nb, nx+nb
  do j = 1-nb, ny+nb
    do k = 1-nb, nz+nb
      do l = 1-nb, nv+nb
        f1(l,k,j,i) = f(l,k,j,i) - fc(l,k,j,i)
      end do
    end do
  end do
end do
```

図 3.8　4 重ループの例.

レーションを 1 つの GPU スレッドに割り当てるため，ループの反復回数がそのまま処理の並列度となる．

GT5D での多重ループの例を図 3.8 に示す．図 3.8 のループは 4 重ループであり，配列 f と配列 fc の各要素の差を配列 f1 に格納する処理を行うが，それぞれのループの反復回数は nx, ny, nz, nv, nb の 5 つの変数で与えられている．

図 3.8 で示す 4 重ループを GPU で計算させる場合のブロック・スレッド割り当て方法はいくつか考えられる．ただし，nx, ny の 2 つは 50 未満, nz, nv の 2 つは 100 前後，nb は 2 を想定する．割り当て方法の 1 つは，最外の i ループ 1 回を GPU のスレッド 1 つに割り当てる方法である．しかしながら，その場合はスレッドが 50 程度しか起動されず，50 スレッドという数は GPU の性能を引き出すには不十分であり，適切な割り当てとは言い難い．

次に最内ループの l ループ 1 回を 1 スレッドに割り当てる場合を考える．この場合では nx × ny × nz × nv 個以上のスレッドが必要とされるが，1 ブロックに含められるスレッドの最大数は 1536 であり，1 つのブロックだけで処理できないため，スレッドだけでなくブロックの割り当ても考慮しなければならない．そこで，i, j, k のループをそれぞれブロックの x, y, z 次元に割り当て，l のループをスレッドの x 次元に割り当てる場合を考える．グリッドの各次元は 65535 個までブロックを含められるため，ブロック数の制限に影響されることはない．また，各ブロックのスレッド数も 100 程度であり，制限値の 1024 を下回る．以上の割り当て規則を用いて実装したカーネル関数を図 3.9 に示す．また，カーネル関数を呼び出す CPU 側のコードを図 3.10 に示す．4 つのループ全てをブロックおよびスレッドで並列化したため，カーネル関数および呼び出し側から，見掛け上ループがなくなっていることがわかる．

以上の様に，各ループについて並列度や計算の依存性，プログラムの記述の容易さなどからブロック・スレッドの割り当てを決定する．

3.3　GPU 向け最適化

MPI 通信と計算をオーバーラップさせて，通信の時間を隠蔽する最適化手法は CPU で実行するプログラムの場合でも一般的に行われている．GPU 上にあるデータを直接 MPI で通信できないため，MPI 通信を行う際は CPU〜GPU 間通信も行いデータを CPU 側に転送する必要があり，CPU のみで計算を行う場

```
attribute(global) subroutine kernel(f1, f, fc)
  real, dimension(1-nb:nv+nb,1-nb:nz+nb, &
    1-nb:ny+nb,1-nb:nx+nb) :: f1, f, fc
  integer :: i, j, k, l

  i = blockIdx%x - nb
  j = blockIdx%y - nb
  k = blockIdx%z - nb
  l = threadIdx%x - nb
  f1(l,k,j,i) = f(l,k,j,i) - fc(l,k,j,i)
end subroutine kernel
```

図 3.9 図 3.8 をカーネル関数化した例.

```
call kernel<<< &
  dim3(nx+2*nb, ny+2*nb, nz+2*nb), nv+2*nb >>>(f1, f, fc)
```

図 3.10 図 3.9 を呼び出すコードの例.

合よりも通信オーバーヘッドが大きくなる．したがって，通信と計算のオーバーラップを行う際の効果も大きい．本節では，bcdf 関数内で行われている通信に着目し，通信と計算のオーバーラップを行う．

3.3.1 bcdf 関数における通信と計算のオーバーラップ

bcdf 関数はデータの袖領域の交換を行うための関数であり，MPI による通信を行う．CPU で実行する際は転送および計算に用いるデータは CPU メモリ上にあり，InfiniBand 等のネットワーク機構を通じて高速に通信できるが，GPU 等のアクセラレータを用いる環境では，CPU〜GPU 間通信や GPU 上で計算カーネルを起動するコストなどがあるため，bcdf 関数におけるノード間通信，CPU〜GPU 間のデータ転送，および計算カーネルの起動に関する最適化を行うことは重要である．

bcdf 関数が扱うデータは $(R, Z, \zeta, v_{\parallel})$ の 4 つの次元で構成されている．4 つの次元それぞれに袖領域処理が必要であるが，MPI プロセスをまたいでの分割の有無で，MPI による分割がある (R, Z) の 2 次元と，MPI による分割がない $(\zeta, v_{\parallel})$ の 2 次元に分けて考える．そして，(R, Z) はそれぞれ n_R, n_Z 個のプロセスに分割されている．各次元の袖交換作業はそれぞれ独立しているため，MPI による通信を行なっている最中に，プロセス内で閉じている作業を並行して行えるため，通信時間を隠蔽できる．

袖領域の処理だけでなく，前後で行う計算部についても，通信に影響を受ける部分の計算と受けない部分の計算に分離すれば，通信と計算のオーバーラップが行える．図 3.11 は 2 次元の計算を表した図であり，計算を (R, Z) 次元の袖領域の値を必要とする領域 (境界) と，(R, Z) 次元の袖領域の値を必要としない領域 (内点) に分ける．内点の部分の計算は MPI 通信を必要とする袖領域のデータを使わずとも計算できるため，MPI 通信と計算のオーバーラップが行える．

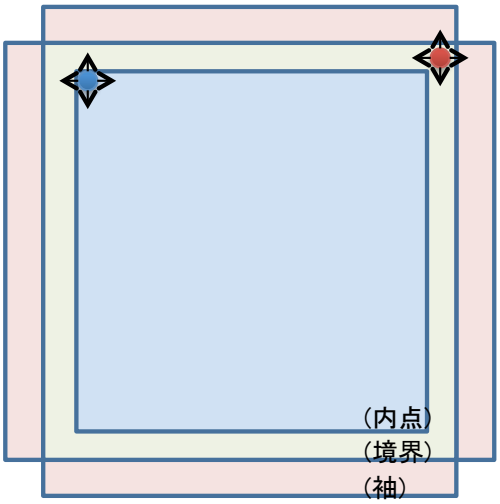

図 3.11　袖領域の概念図．青点は袖領域のデータを必要としない内点を表わし，赤点は袖領域のデータを必要とする境界を表す．

袖領域の処理と前後の計算の 2 つのオーバーラップをまとめた図が図 3.12 である．2 の計算カーネル k_1, k_2 の間で `bcdf` 関数で袖領域を交換している場合を表している．GPU カーネルの起動，CPU〜GPU 間通信，MPI 通信をそれぞれ非同期に実行する．GPU の非同期操作は CUDA Stream を用いて行い，MPI の非同期通信には `MPI_Isend` と `MPI_Irecv` を用いる．

CUDA Stream は計算用と通信用の 2 つの CUDA Stream を用意し，それぞれに計算命令と転送命令を発行する．複数の CUDA Stream を同時に利用すると，それぞれの CUDA Stream 内に発行された命令は，発行された順番に実行されるが，異なる CUDA Stream 間は並列に実行される．したがって，2 つの CUDA Stream を用いると，通信と計算を同時に行える．

3.3.2　`lfp` 関数における最適化

`lfp` 関数の中には，MPI から受信データしたデータを GPU に書き戻す処理が存在する．細切れな MPI 転送を行うと性能が劣化するため，GT5D はいくつかの小さいデータの塊を 1 つの大きな塊にパッキングし，それを送信する最適化を行っている．加えて，受信したデータを次の計算に用いる際にデータの転置を行う必要がある．したがって，GPU で転送結果を利用して計算を行う前に，パッキングされたデータを解体し，さらに転置する処理が必要になる．

データアンパックおよび転置作業を CUDA にある CPU〜GPU 間の転送用 API (`cudaMemcpy`) を利用するナイーブな方式では，十分な性能が得られず `lfp` 関数が全体のボトルネックとなることが判明したため，この関数におけるデータ移動の最適化について検討を行う．本節では，次に示す 3 つの実装方式について比較検討を行う．また，それぞれの方式の転送方法の違いについて図 3.13 に示す．

Method 1

各チャンク毎に `cudaMemcpy` 関数でデータをコピーする．

Method 2

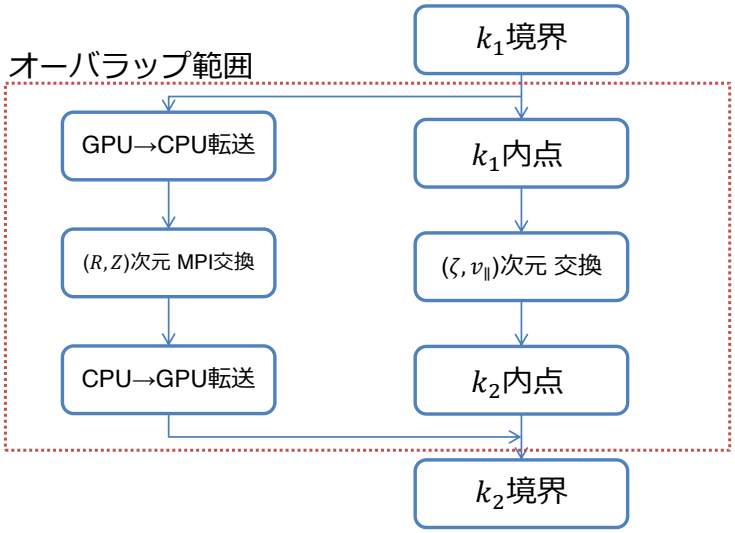

図 3.12 通信と計算のオーバーラップの概念図.

表 3.2 `lfp` 関数における各データ転送方法の性能比較.

	Time [ms]
Method 1 (`cudaMemcpy`)	438.5
Method 2 (アンパック→ GPU 転送)	79.8
Method 3 (GPU 転送→アンパック)	28.9

CPU 上でデータをアンパックし，そのデータを GPU に送る.

Method 3

パック済みデータを GPU に送り，GPU 上でアンパック用のカーネルを実行する.

表 3.2 に性能比較の結果を示す．3 つの方式の中で Method 1 が最も性能が悪いことがわかる．Method 1 では，163,840 回の `cudaMemcpyAsync` および `cudaMemcpyAsync2D` 関数を呼び出す必要がある．それぞれの CUDA API 関数の呼び出しには，1 回あたりおよそ $3\mu s$ のオーバーヘッドがあり，多回数の CUDA API 呼び出しは性能を悪化させる．次に，Method 2 と Method 3 で比較すると，Method 2 の方が性能が悪い．Method 2 の方が性能が悪い理由は CPU〜GPU 間のデータ転送量の差にあり，Method 2 の方がおよそ 2 倍のデータサイズを GPU に転送しなければならない．以上の結果より，GT5D の GPU 化では最も高速な Method 3 の方式を採用する.

3.3.3 `fld_sfls` 関数における最適化

本章の基本的な GPU 化のポリシーは，全ての計算を GPU にオフロードすることである．しかしながら，`fld_sfls` 関数には，GPU では効率的に実装することが難しい計算が存在する．この計算は，4 次

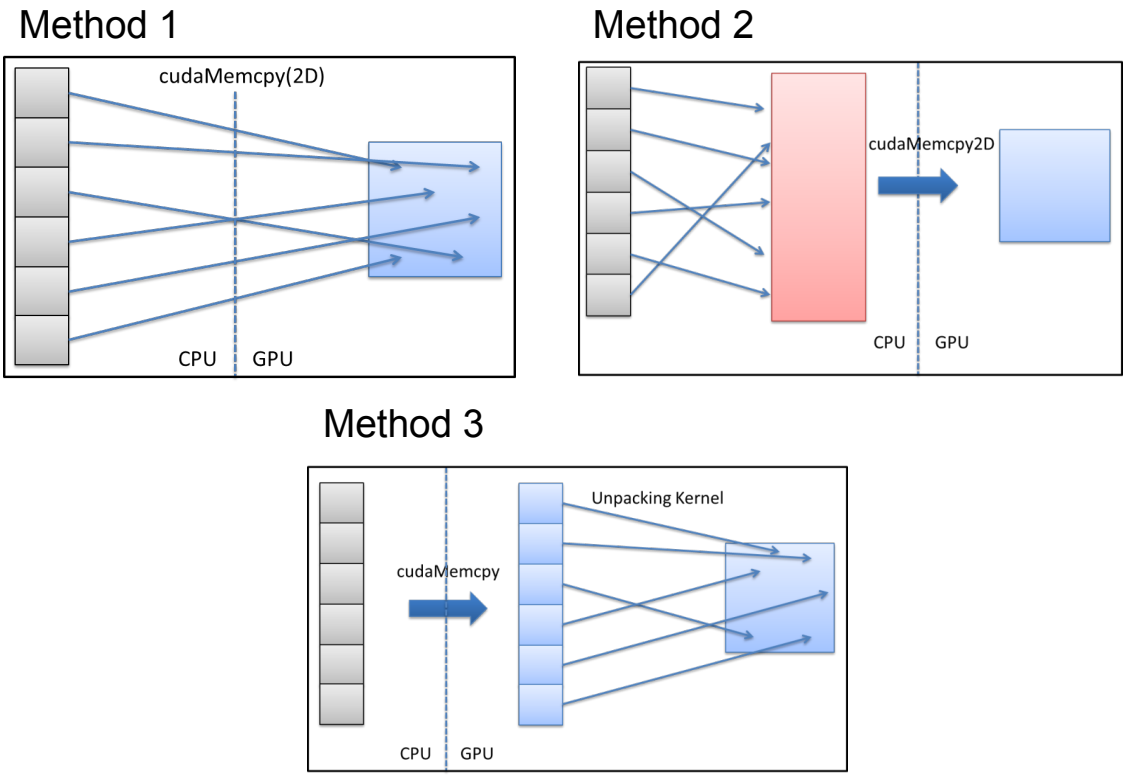

図 3.13　`lfp` 関数における各データ転送方法の違い.

元配列 *fmat* と *dfem* の値を 2 次元配列 *phw* に集約するものである．配列 *phw* への加算がスレッド間で衝突するため，単純な加算では結果が正常に得られない．このような reduction 演算をどのように GPU 上で扱うかを決定する事が難しいことは，GPU コンピューティングでは良く知られている問題の 1 つである [27]．性能評価を行なった結果，データ移動のコストを含めてもこの計算は CPU 上で行う方が効率的であると判断した．性能評価の詳細については以下で述べる．

GT5D のオリジナルの CPU における実装では，OpenMP によるスレッド並列を利用しており，データレースを避けるために，OpenMP の parallel ディレクティブに対して reduction 節を付与し，*phw* への書き込みを制御している．OpenMP のランタイムは，それぞれの CPU スレッド毎に *phw* 配列のローカルコピーを作成し，スレッドローカルな計算を行なった後に，スレッド間の計算を行っている．しかしながら，GPU で CPU と同様の仕組みを用いてデータレースを回避することは困難である．一般的に，GPU 上では数千のスレッドが実行されるため，配列のローカルコピーを大量に用意しなければならず，大量のメモリを消費してしまう．したがって，GPU 向けの最適化された実装が必要となる．

表 3.3 に 2 つの GPU の向け実装と CPU 実装の性能を示す．GPU (A) は少ないスレッド数 (1 ブロック 34 スレッド) で GPU カーネルを起動し，全スレッドで部分和を計算した後に代表の 1 スレッドが配列 *phw* を更新するものである．GPU (B) は CUDA の atomic 命令を利用したものである．ただし，Fermi アーキテクチャの GPU は，整数や単精度不動小数点数に対する `atomicAdd` 関数は実装されているが，倍精度浮動小数点数の `atomicAdd` 関数はサポートしていないため，加算の後に 64bit 幅の CAS

表 3.3 `fld_slfs` 関数の性能比較.

	Time [ms]
CPU	0.5
GPU (A; 低並列)	29.3
GPU (B; atomic)	204.3

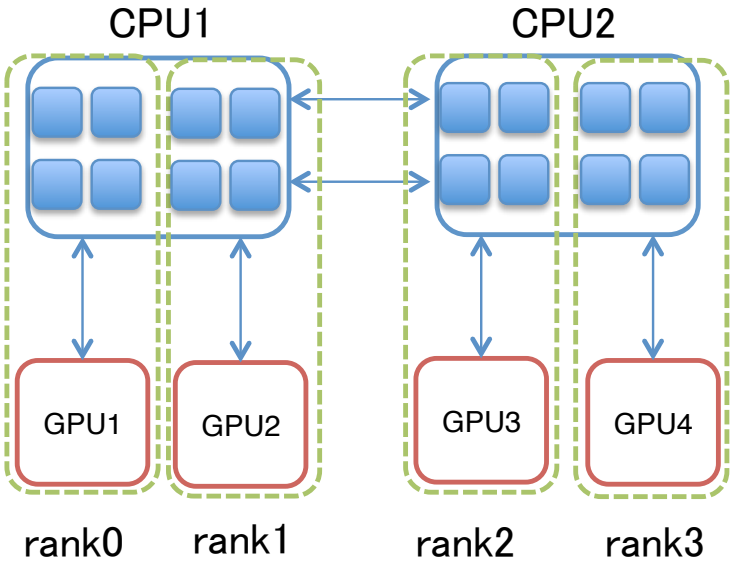

図 3.14 HA-PACS ベースクラスタのプロセス割り当てを示す図．青の四角形は CPU コアを表す．

(Compare-and-Swap) 命令を用いて更新することで atomic な加算を実装する．GPU (B) は GPU (A) よりも遅く，Fermi アーキテクチャにおける atomic 命令の性能が低いことがわかる．

GPU 上での 2 実装 (A), (B) および CPU における実装 (データ転送時間込み) を比較した結果，本計算は CPU 上で実行するのが最も良いという結果となった．加えて，この計算の後に MPI 通信があるため，どちらにせよ GPU からデータを転送しなければならず，CPU 上で計算する事による性能への影響は小さいと考えている．

3.4 性能評価

性能評価には HA-PACS ベースクラスタ [28] を用いる．HA-PACS ベースクラスタの性能諸元を表 3.4 に示す．HA-PACS ベースクラスタは 1 つのノードに Intel Xeon CPU (Sandybridge) が 2 つ，NVIDIA Tesla GPU (Fermi) が 4 つ搭載されている高密度 GPU クラスタである．1 ノードあたり 4 つの GPU が搭載されているため，図 3.14 にあるように 1 ノードあたり 4 つの MPI プロセスを立ち上げる．

GT5D は LAPACK (Linera Algebra PACKage) および FFT (Fast Fourier Transform) ライブラリを計算に用いる．CPU 向けには LAPACK および FFT の実装として Intel MKL [29] を利用する．NVIDIA は

表 3.4　HA-PACS ベースクラスタの性能諸元.

CPU	Intel Xeon E5-2670 × 2 (2.6 GHz)
	CPU (8 cores/CPU) × 2 = 16 core
CPU Memory	128GB, DDR3 1600 MHz
GPU	NVIDIA Tesla M2090 × 4
GPU Memory	6 GB/GPU
OS	CentOS 6.4
CUDA Toolkit	ver. 5.5
PGI Compiler	ver. 13.9
PGI Compiler	`-Mcuda=cc20,5.5,flushz`
Options	`-fastsse -Mipa=fast,inline`
MPI	MVAPICH2 1.8.1
Interconnect	Infiniband QDR 4x, 2 Rails

GPU 向けの LAPACK 実装を提供していないため，BLAS ライブラリである cuBLAS [30] を利用して同等の計算を行うものを作成し，FFT の実装としては NVIDIA の cuFFT [31] を利用する.

3.4.1　通信を含まない関数の性能評価

本節では，通信を含まない計算のみの関数の性能評価を行う．本評価では，各種パラメータを以下に示すように設定し測定を行う．測定対象の CPU 版 GPU 版それぞれの関数を呼び出し計算結果が一致しているかどうかと，処理時間を計測するテストプログラムを作成し，本測定で使用する．ただし，測定用のテストプログラムは MPI 並列を使用せず 1 ノードのみで動作する．実行時のメッシュ分割数は $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (128, 128, 128, 128, 4)$，CPU 側の OpenMP スレッド数は 4，GPU 使用数は 1 とする.

`timedev1`〜`timedev9` 関数を GPU 化し，CPU(4 コア) と性能を比較した結果を表 3.5 と図 3.15 に示す．最も性能が改善した関数は `timedev1` のケースで，CPU と比べ 3.35 倍高速になった．また，`timedev1`〜`timedev9` 関数の平均では，CPU と比べ 2.58 倍高速になった.

`l4dx_r`, `l4dx_s`, `l4dx_l`, `l4dx_nl` 関数を GPU 化し，CPU(4 コア) と性能を比較した結果を表 3.6 と図 3.16 に示す．最も性能が改善した関数は `l4dx_r` 関数のケースで，CPU と比べ 1.43 倍高速になった．`l4dx_s`, `l4dx_nl` 関数でも速度向上がみられるものの，`l4dx_l` 関数は CPU と比べて 0.71 倍高速と，GPU で実行する方が遅くなってしまった.

`dn3d`， `drift_nl` 関数を GPU 化し，CPU(4 コア) と性能を比較した結果を表 3.7 と図 3.17 に示す．最も差が小さい `drift_nl` 関数で 0.90 倍高速と，これらの関数はどちらも GPU の方が実行が遅くなってしまった.

表 3.5　`timedev1`〜`timedev9` 関数の性能評価.

関数名	CPU(4 コア)[ms]	GPU[ms]	Speedup
`timedev1`	18.2	5.4	3.35
`timedev2`	17.5	8.6	2.03
`timedev3`	21.2	9.6	2.21
`timedev4`	21.5	11.2	1.92
`timedev5`	21.3	11.2	1.90
`timedev6`	23.3	7.2	3.26
`timedev7`	18.1	5.4	3.34
`timedev8`	28.1	8.8	3.17
`timedev9`	20.9	10.2	2.05

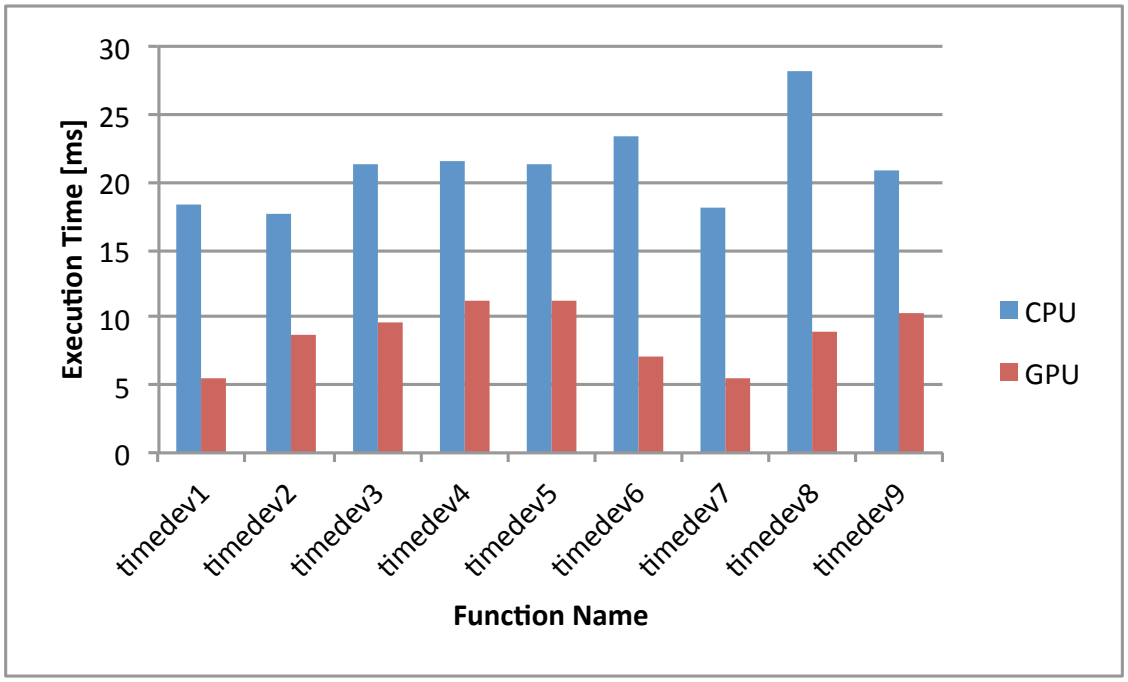

図 3.15　`timedev1`〜`timedev9` 関数の性能評価のグラフ.

表 3.6　`l4dx_r`，`l4dx_s`，`l4dx_l`，`l4dx_nl` 関数の性能評価.

関数名	CPU(4 コア)[ms]	GPU[ms]	Speedup
`l4dx_r`	34.4	24.0	1.43
`l4dx_s`	41.5	40.8	1.02
`l4dx_l`	75.7	106.4	0.71
`l4dx_nl`	132.0	127.6	1.03

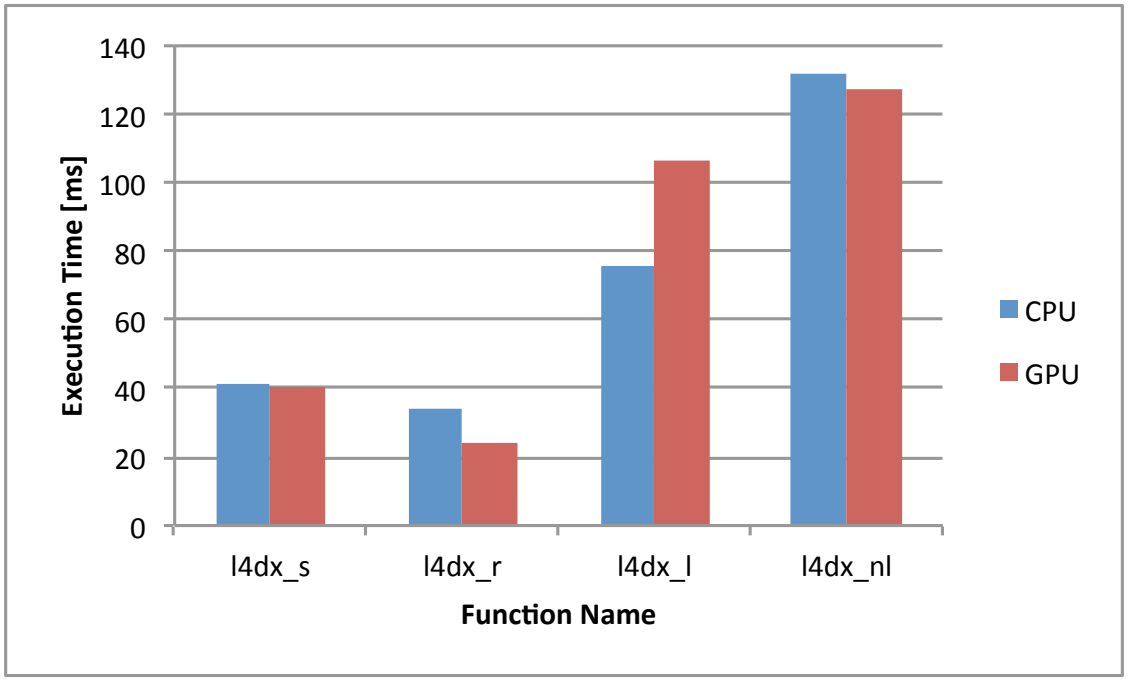

図 3.16　`l4dx_r`, `l4dx_s`, `l4dx_l`, `l4dx_nl` 関数の性能評価のグラフ.

表 3.7　`dn3d`, `drift_nl` 関数の性能評価.

関数名	CPU(4 コア)[ms]	GPU[ms]	Speedup
`dn3d`	0.6	0.7	0.82
`drift_nl`	1.1	1.2	0.90

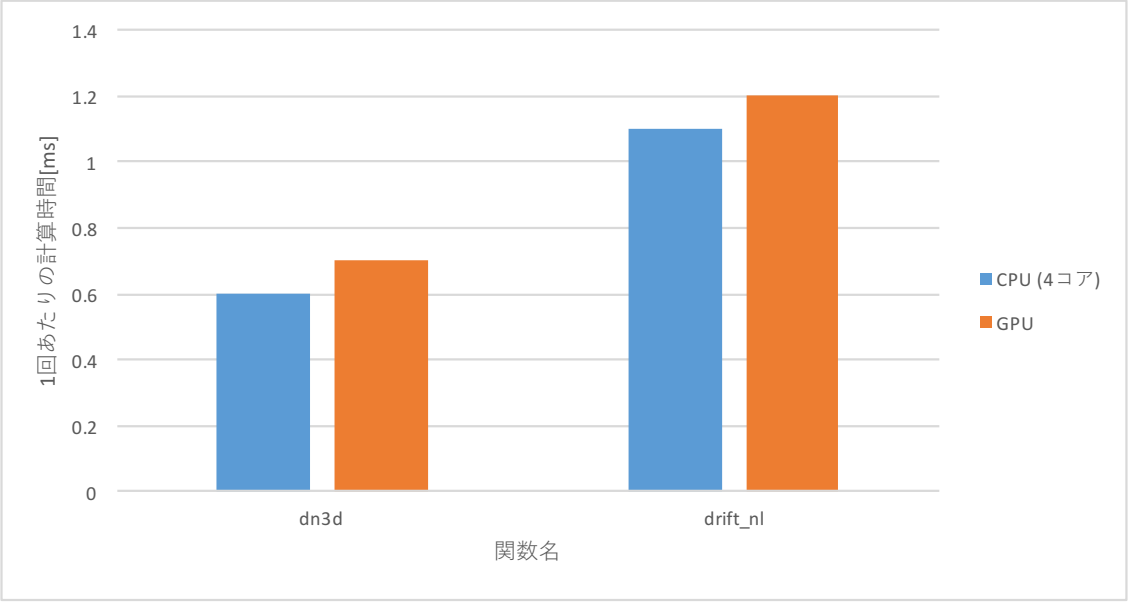

図 3.17　`dn3d`, `drift_nl` 関数の性能評価のグラフ.

3.4.2 `bcdf` 関数の通信と計算のオーバーラップ評価

`bcdf` 関数の通信と計算のオーバーラップによる通信時間の隠蔽評価を行う．通信と計算のオーバーラップは前後の計算を行う関数によって以下の 5 パターンが存在する．

1. `timedev_2` → `bcdf` → `l4dx_s`
2. `timedev_3` → `bcdf` → `l4dx_s`
3. `timedev_4` → `bcdf` → `l4dx_s`
4. `timedev_4` → `bcdf` → `timedev_6`
5. `timedev_4` → `bcdf` → `timedev_8`

以上 5 つのパターンの内，本評価では 3 番のパターンについて評価を行う．3 番のパターンは，図 3.6 の波線部で示す内部ループで用いられているパターンで，実行回数が他のパターンより多いため評価対象とする．

なお，CPU 側の処理時間の測定は `MPI_Wtime` 関数を使用し，GPU 側の処理時間の測定は cudaEvent を使用する．オーバーラップを行う際は，CUDA カーネルは非同期に実行されるため，`MPI_Wtime` などの CPU 側で時間を計測する手段では処理時間を求められない．cudaEvent は GPU の処理の開始や終了を検出するために用いる機構であるが，`cudaEvent` は 2 つのイベントの間の時間を求める `cudaEventElapsedTime` 関数があり，それを用いて処理時間を計測する．また，ジッタなどの影響を軽減するために，関数の入口で `MPI_Barrier` 関数を用いて全ての MPI プロセスの待合せを行う．

関数全体の計算時間を表 3.8 に示す．GT5D のパラメータは $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (128, 128, 128, 128, 4)$，MPI プロセス数は $(n_R, n_Z, n_\mu) = (4, 4, 4)$ で性能評価を行う．オーバーラップなしの場合は 1 回あたり 143.1ms かかっていたが，オーバーラップを行うことで 1 回あたり 71.9ms と，71.2ms (1.99 倍) の短縮の効果が得られる．また，オーバーラップありの場合の詳細な処理時間を表 3.8 に示す．Calc, Transfer の 2 つの縦線はオーバーラップに用いる 2 つの CUDA Stream を表し，CPU の縦線は MPI の通信状況を表す．また，線の上に書かれている箱は各処理の時間を表す．ただし，図の構成上，箱の高さと実際の処理時間の比率は一定ではない．それぞれの処理の内容は以下の通りである．

timedev4 bonudary
: timedev4 カーネルの境界部の計算．

timedev4 inner
: timedev4 カーネルの内点部の計算．

bcdf pack
: bcdf MPI exchange で MPI 通信を行うために timedev4 boundary で計算した境界部の計算結果を CPU 側に転送する．

bcdf MPI exchange
: MPI を用いて隣接プロセスと袖領域のデータを交換する．

表 3.8　bcdf 関数の計算時間.

オーバーラップ	時間 [ms]	Speedup
Disabled	143.1	-
Enabled	71.9	1.99

表 3.9　時間発展部の計算時間.　“Threads” は OpenMP のスレッド数を示す.

Time [s]	Threads = 1	Threads = 2	Threads = 4
CPU	26.96	17.67	15.12
GPU (overlap なし)	13.51	13.05	12.89
GPU (overlap あり)	7.90	7.92	8.02

bcdf exch. inner

周期境界条件となっている次元の袖領域の交換処理を行う．ノード内でデータ移動が完結するため MPI 通信は発生しない.

bcdf exch. boundary

bcdf MPI exchange で更新されたデータを GPU 側に書き戻す.

l4dx_s boundary

l4dx_s カーネルの境界部の計算.

l4dx_s inner

l4dx_s カーネルの内点部の計算.

図 3.18 の赤線は l4dx_s inner の処理の終りを示し，緑線は bcdf exch. boundary の処理の終りを示す．計算 (緑) が通信 (赤) よりも早く完了しているため，GPU が約 3.7ms 遊んでいることがわかる．l4dx_s boundary は bcdf exch. boundary の処理が終わらなければ計算できないためである.

3.4.3　時間発展全体の性能評価

時間発展全体の性能評価を行う．strong scaling の評価を行うために，メッシュ数のパラメータは固定で MPI プロセス数を変化させる．メッシュ数は $(N_R, N_\zeta, N_Z, N_{v_\parallel}, N_\mu) = (128, 128, 128, 128, 4)$ で固定するが，MPI プロセス数は $n_\mu = 16$ で固定し $(n_R, n_Z) = (4, 4, 4)$ と変化させる．1 ノードあたり 4 プロセス起動するため，ノード数換算では 64 ノードとなる.

時間発展 1 回あたりの計算時間を表 3.9 に示す．オーバーラップのありなしは，bcdf 関数におけるオーバーラップのありなしを示す．CPU と比較して，GPU を用いる場合，オーバーラップなしの場合で 1.17 倍，オーバーラップありで 1.91 倍の高速化が得られていることがわかる．また，オーバーラップのありとなしで比較すると，オーバーラップありの方が 1.63 倍高速であることがわかる.

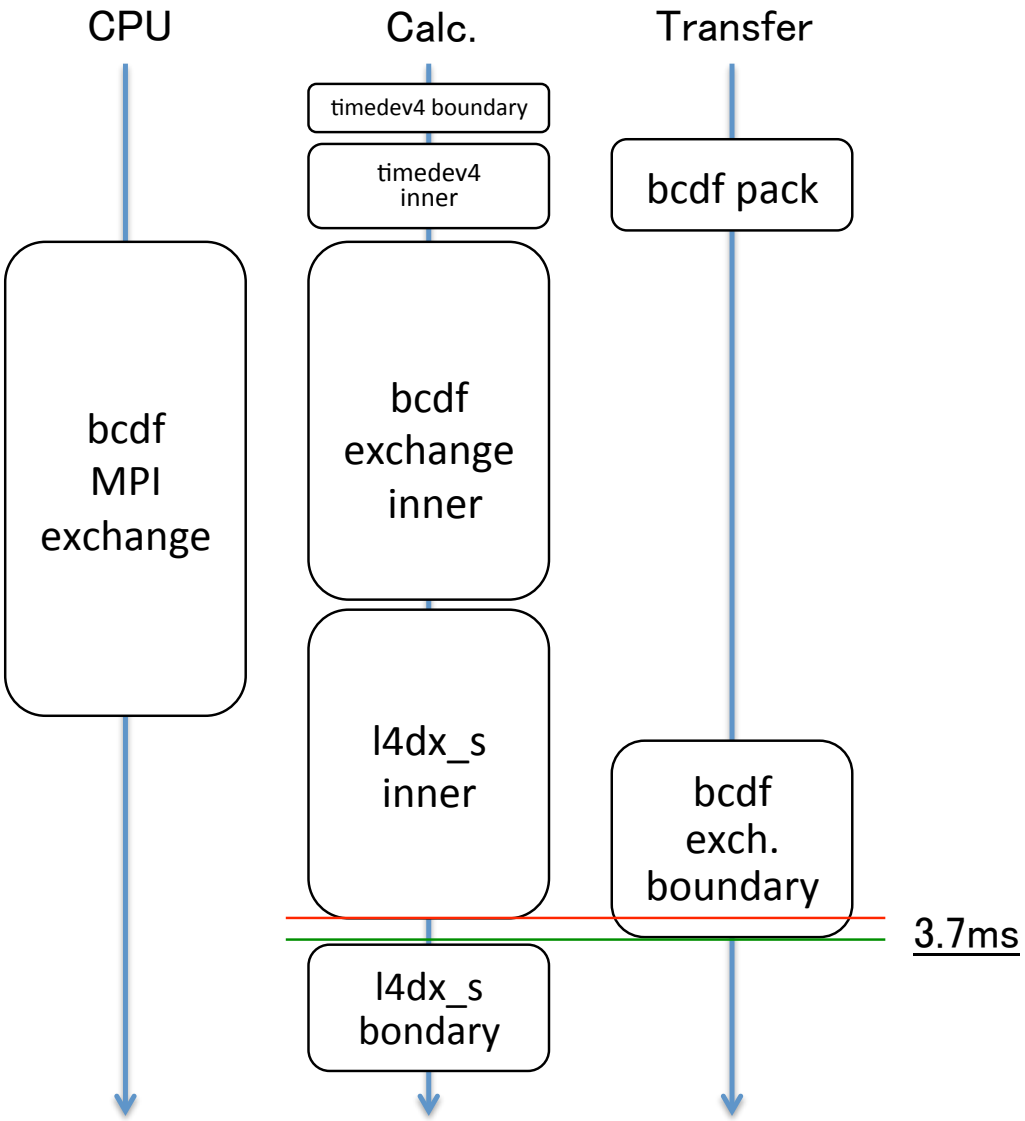

図 3.18 オーバーラップありの場合の bcdf 関数の処理時間の詳細.

3.5 考察

本章では GT5D のいくつかの関数を GPU 化を行なったが，一部の関数について高速化が達成できていない．`timedev` 系の関数は 2 倍以上，`l4dx_r` 関数については GPU の方が 1.43 倍高速であるが，`l4dx_s`, `l4dx_nl` 関数については 2 倍未満の高速化しか得られず，`l4dx_l`, `dn3d`, `drift_nl` 関数については GPU の方が遅いという結果が得られた．それぞれのカーネルについて，実行時の情報を表 3.10 に示す．これらの情報は Compute Profiler を用いて取得している．また，パラメータについては性能評価の際に使用したものと同じであり，1 つの関数名の項目に対して複数のカーネル名の項目がある関数は，

複数のカーネルから関数が構成されていることを表す．

dn3d の様な例外はあるものの，GPU の方が遅い 2 つの関数 l4dx_l, drift_nl は Occupancy が低いことがわかる．CUDA のアーキテクチャは，メモリアクセスやその他の要因によってスレッドの実行ができない状態になると，他のワープの実行に切替えて，演算器が遊ばないようする制御が行われている．なお，切替え単位がワープなのは，SM の最小制御単位がワープなためである．

Occupancy が低いということは，SM が実行ワープを切替える際の切替え先の候補が少ないということであり，特にメモリアクセスの比率が高いカーネルにおいて，全てのワープが実行不能になり，結果として性能が低下する可能性が高くなる．一方で Occupancy が低いが性能が高いカーネルは，演算比率が高いカーネルであると考えられ，そのようなカーネルでは実行が停止するワープが少なく，SM が保持するワープの数が少なくとも実行効率が高い状態になっていると考えられる．

l4dx_l と l4dx_nl 関数に含まれる reduce カーネルは，計算カーネル本体で求めた結果の総和を取る補助カーネルである．現在の CUDA では，ブロックをまたいでの同期命令が存在しないため，ブロックをまたいでの総和などを求める場合には，総和を求めるだけのカーネルを作らなければならない．総和を求めるカーネルが実行される時には，前に実行されている計算カーネルが終了していることが保証されるが，総和計算は並列度が高くないため，1 ブロック ×12 スレッドといった少ないスレッド数のパラメータでカーネルを起動しなければならず，Occupancy が低くなってしまう問題が避けられない．

CUDA では 1 スレッドが使えるスレッド数の上限が 63 であるが，l4dx_nl カーネルはレジスタの使用数が 63 となっており，制限に抵触している．そのようなカーネルでは，プログラムを実行するにあたって要求されるレジスタの数が 63 よりも多いがレジスタを使えないため，ローカルメモリにデータを退避させることでプログラムを実行している．ローカルメモリはレジスタよりもアクセスに時間がかかるため，レジスタが溢れている状態は性能に悪影響を及ぼす．l4dx_nl カーネルでは，8 バイトのデータがローカルメモリに置かれる状態になっており，プログラムの修正等でレジスタの使用量を減らせれば，性能が改善されると考えられる．

3.6　GT5D の GPU 化に関する結論

GT5D の時間発展部分を GPU 化するにあたり，CPU 版でプロファイルを取り，どの箇所の計算に時間がかかっているのかを測定した．測定の結果，l4dx_s 関数がおよそ 30% の時間を消費していることがわかった．また，関数として分離されていないため，プロファイルの結果には表われなかったが，main に直接記述されているいくつかのループが合計でおよそ 40% の時間を消費していることがわかった．

いくつかの関数を GPU 化し，計算時間のおよそ 80% の計算を GPU で行えるようになった．timedev1 〜timedev9 関数の平均では，CPU と比べ 2.58 倍高速になり，l4dx_r, l4dx_s, l4dx_l, l4dx_nl 関数では，最も性能が改善した関数は l4dx_r 関数のケースで，CPU と比べ 1.43 高速になった．l4dx_s, l4dx_nl 関数でも速度向上がみられるものの，l4dx_l 関数は CPU と比べて 0.71 倍高速と，GPU で実行する方が遅くなってしまった．dn3d， drift_nl 関数をでは dn3d 関数で 0.82 倍，drift_nl 関数で 0.90 倍と，どちらの関数も GPU の方が遅いという結果になった．

bcdf 関数の計算と通信のオーバーラップでは，MPI 通信と計算のオーバーラップだけでなく，CPU〜GPU 間の通信と計算のオーバーラップも実装を行なった．bcdf 関数の計算と通信のオーバーラップに

表 3.10 各カーネルの実行時情報．ただし smem はシェアードメモリの使用量を示す．

関数名	カーネル名	レジスタ数	smem[byte]	Occupancy	speedup
timedev1	timedev1	15	0	1.000	3.35
timedev2	timedev2	18	0	1.000	2.03
timedev3	timedev3	21	2048	0.833	2.21
timedev4	timedev4	20	0	1.000	1.92
timedev5	timedev5	32	6144	0.667	1.90
timedev6	timedev6	19	0	1.000	3.26
timedev7	timedev7	16	0	1.000	3.34
timedev8	timedev8	23	0	0.833	3.17
timedev9	timedev9_1	22	0	0.083	2.05
	timedev9_2	21	0	0.833	
	timedev9_3	21	0	0.833	
	timedev9_4	21	0	0.833	
	timedev9_5	21	0	0.833	
	timedev9_6	21	0	0.833	
	timedev9_7	21	0	0.833	
	timedev9_8	21	0	0.833	
	timedev9_9	21	0	0.833	
l4dx_l	l4dx_l	60	27264	0.167	0.71
	reduce	22	0	0.021	
l4dx_nl	l4dx_nl	63	14336	0.333	1.03
	reduce	23	0	0.021	
l4dx_r	l4dx_r	33	4352	0.333	1.43
l4dx_s	l4dx_s	43	3200	0.333	1.02
dn3d	dn3d	20	4360	1.000	0.82
drift_nl	k1	8	0	0.833	0.90
	k2	45	0	0.417	
	k3	57	0	0.312	

よる性能改善では，計算の方が早くおわってしまうため，通信の完了を 3.7ms 待ってしまうものの，通信隠蔽による性能向上は大きく，関数一回あたり 71.2ms の性能改善が達成できた．

時間発展全体の性能評価では，GPU 版のコードは CPU 版のコードよりも，オーバーラップなしの場合で 1.17 倍，オーバーラップありで 1.91 倍の高速化が得られ，また，オーバーラップのありとなしで比較すると，オーバーラップありの方が 1.63 倍高速化が達成された．

したがって，GPU アプリケーションで良い性能を得るためには，通信と計算のオーバーラップによる

通信隠蔽が重要であるとわかった．しかしながら，既に計算時間と通信時間がほぼ同じであり，通信隠蔽の限界に達しつつある．今後，アクセラレータの性能が向上し計算時間が短縮されたとしても，PCIe バスの性能がボトルネックとなり，通信隠蔽が十分にできないため，より細粒度のオーバーラップや，アクセラレータのメモリに対して直接アクセスしデータ転送を行うなど，通信レイテンシを削減するための工夫が必要である．

第 4 章

GPU 間通信のハードウェアによる高速化

4.1　GPU 間通信におけるボトルネック

HPC 分野では複数のノードにあるアクセラレータを利用して並列計算を行うことが一般的であり，そのような環境下では計算を行う上でノードをまたぐアクセラレータ間での通信は避けられない．一般的な PC クラスタ環境では，アクセラレータは PCIe バスを通じてシステムと接続されており，CPU とのデータのやり取りや GPU への制御指令といった情報は PCIe バスを通じて送られる．また，ノード間の通信を行うインターフェイスも PCIe バスに接続されていることが一般的である．

PCIe バスの帯域は GPU のメインメモリや CPU のメインメモリの帯域と比べて狭く，GPU でアプリケーションを実行する際のボトルネックとなりやすい．また，一般的な環境では，ノードをまたぐ GPU 間でデータを通信する場合は GPU メモリのデータを一旦 CPU メモリにコピーし，それを相手ノードの CPU メモリにコピーし，最後に GPU に書き戻すという 3 回のコピーが必要になる．CPU 間の通信であれば，余分なコピーは発生しないところであるが，GPU 間の場合は 3 回のコピーが必要となるため，GPU 間の通信は CPU 間に比べてレイテンシが大きくなる．一般的に，強スケーリングの設定でノード (GPU) 数を増やすと，1 GPU あたりが計算する問題サイズが小さくなり，同時に通信のメッセージ長も小さくなる．短メッセージの通信では低レイテンシな通信が重要であり，GPU を用いる環境では良い強スケーリング性能達成することが難しい．

この問題を解決するために，TCA (Tightly Coupled Accelarators) アーキテクチャが提唱されている．また，TCA アーキテクチャの実証実装として PEACH2 (PCI Express Adaptive Communication Hub Ver.2) が開発されており，ノードをまたぐ GPU 間の低レイテンシ通信を可能としている．

TCA に関する研究は筑波大学 計算科学研究センターとの共同研究であり，本論文の成果の範囲ではないが，第 5 章において TCA を利用しているため，第 5 章に必要な TCA と PEACH2 に関する情報を本章で述べる．さらに詳細な TCA および PEACH2 については，論文 [32, 33, 34, 35] が詳しい．

4.2　TCA アーキテクチャの概要

TCA アーキテクチャでは，アクセラレータは同一ノード内だけでなく異なるノードにあるアクセラレータとも直接通信を行えるものとしており，アクセラレータ間の通信レイテンシの削減および，通信レ

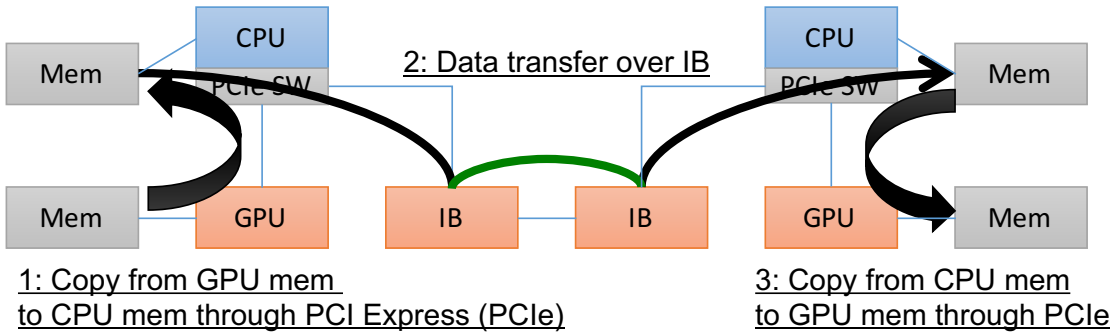

図 4.1 MPI および InfiniBand を利用する一般的な環境で GPU 間通信を行う場合のデータの流れ.

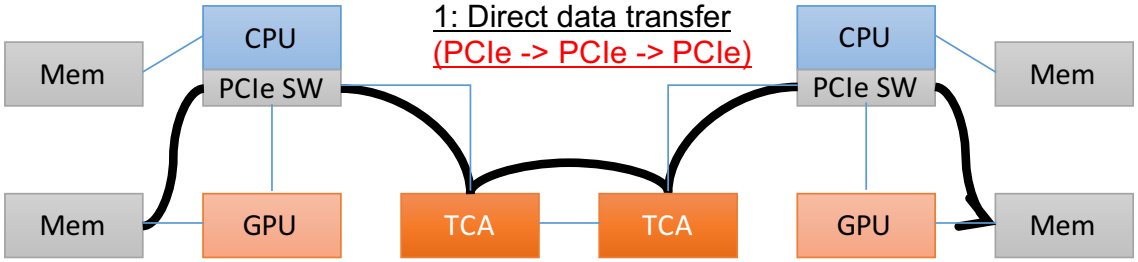

図 4.2 TCA を利用する環境で GPU 間通信を行う場合のデータの流れ.

イテンシが減ることにより TCA を用いない環境とよりも良い強スケーリングの性能が得られる.

図 4.1 に，通信方式として MPI および InfiniBand を利用しアクセラレータに GPU を用いる環境において，GPU 間通信を行う場合のデータの流れを示す．このような環境では，GPU→CPU (図 4.1 の 1 番)，ノード間通信 (図 4.1 の 2 番)，GPU→CPU (図 4.1 の 3 番) の合計 3 回のデータコピーが必要となる.

図 4.2 に，通信方式として TCA アーキテクチャをアクセラレータとして GPU を利用する場合における GPU 間通信のデータの流れを示す．TCA アーキテクチャを用いる環境では，TCA の通信機構が直接 GPU のメモリへアクセスできるため，図 4.2 の様な余分なデータコピーをすることなく，1 回のデータ移動で通信が行え，通信レイテンシが削減される.

通信を隠蔽するために，通信と計算を同時に行うといった手法が広く利用されているが，アクセラレータは世代を増すごとに性能を向上しており，性能の向上に伴い計算時間が短くなるため，通信隠蔽が困難になりつつある．したがって，アクセラレータを持つシステムで強スケーリングで良い性能を発揮するためには，ノードをまたぐアクセラレータの間で低レイテンシな通信をしなければならない.

4.3 FPGA による TCA アーキテクチャの実装

4.3.1 PEACH2 について

PEACH2 は Altera 社の FPGA (Field-Programmable Gate Array) を用いる TCA アーキテクチャの実装の 1 つであり，PEACH2 は GDR テクノロジを利用して，PCIe プロトコルに基いてノード間通信および

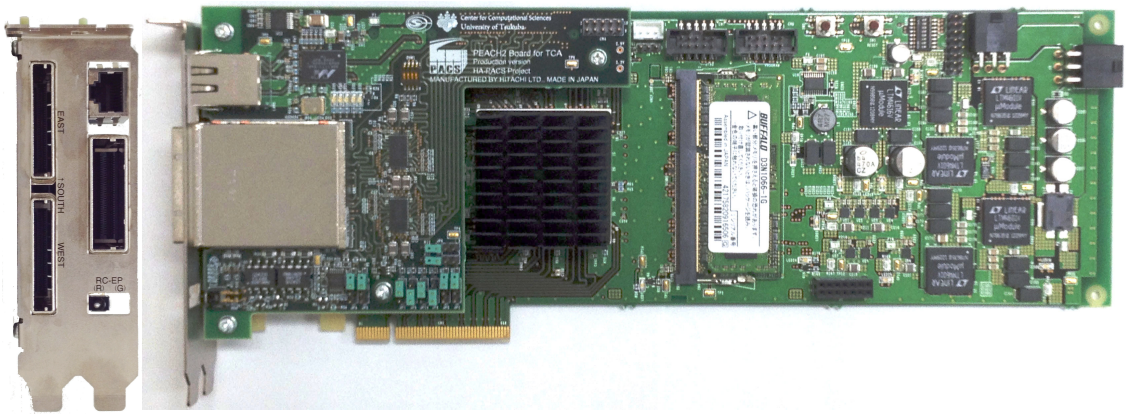

図 4.3　PEACH2 ボードの写真.

アクセラレータへのメモリアクセスを行う．PEACH2 は NVIDIA 社の GPU をターゲットとして開発されており，ノードをまたぐ GPU 間の直接通信を実現する．PEACH2 はノード内およびノード間の通信プロトコルとして PCIe を利用しており，PEACH2 は PCIe バスを通じて GPU 内のメモリにあるデータを他のノードの GPU へ直接転送できる．

PEACH2 ボードを図 4.3 に示す．図 4.3 の中央にあるヒートシンクの下に PEACH2 が実装された FPGA チップがあり，PEACH2 ボードをマザーボード上の PCIe スロットに挿入し，ホストと接続して利用する．

TCA はノード間の通信チャネルとして PCIe を用いる．PCIe バスは周辺機器を接続するためのバスとして一般的に使われているシリアルバスであり，GPU だけでなく，イーサネットボードや InfiniBand ボードの接続に用いられている．PCIe は一般的にノード内といった短距離通信に用いられているが，PCIe ケーブル [36] を用いると，ノード間のような長距離通信に PCIe プロトコルを利用できる．PEACH2 チップは PCIe gen.2 x8 レーンの接続を 4 つ持っており，1 つをホストとの接続，残り 3 つを他の PEACH2 との接続に利用する．多数のノードを 1 つのネットワークで接続する場合は，3 つある外部 PCIe リンクと PCIe ケーブルを用いて，リングネットワークを形成する．

PCIe はパケットを用いる通信プロトコルであり，ネットワーク内に 1 つの RC (root complex) といくつかの EP (end point) から構成されている．RC と EP の間で 1 つの PCIe アドレス空間を共有している．一般的なコンピュータでは，CPU が RC となり拡張ボードが EP となる．PCIe ネットワーク内に存在できる RC の数は 1 つだけという制限があるため，特殊なパケットルーティング機構なしに複数のノードを 1 つのネットワークとして扱えない．PEACH2 チップは，イーサネットネットワークにおけるルーターのようにノードをまたぐ通信を扱い，PCIe ネットワーク内の RC 数の制限を解決する．

PEACH2 では，次のような CPU/GPU 環境を想定して設計されている．ホスト CPU には Intel Xeon E5 (SandyBridge-EP もしくは IvyBridge-EP アーキテクチャ) を，アクセラレータとして GPU には NVIDIA Kepler アーキテクチャを想定して設計が行われている．これらの CPU は CPU 内に PCIe スイッチを内蔵しており，CPU に直接接続されたデバイスに関する PCIe パケットを配送できる．そして，全ての CPU と GPU は同じ PCIe アドレス空間を共有しているため，PEACH2 はそれらのデバイスの全てのメモリに

アクセスできる．

Intel CPU を用いてマルチソケット環境を構築する場合，それぞれの CPU に独立したメモリが搭載され，NUMA (Non Uniform Memory Architecture) と呼ばれる環境が構築される．CPU 間の接続には，QPI (Intel QuickPath Interconnect) と呼ばれる Intel のプロプライエタリなバスが用いられる．QPI は異なる CPU に接続されているメモリへのアクセスや，キャッシュコヒーレンシを維持するために用いられる．

GPU は前述した通り，それぞれの CPU が内蔵する PCIe スイッチに接続されるため，異なる CPU のスイッチに接続されている GPU にアクセスする場合は，QPI を通じて GPU と通信を行う．しかしながら，QPI を通じて PCIe デバイス間で通信を行うと性能が劣化することが知られており，PEACH2 を用いて GPU 間通信を行う場合も，この制限の影響を受け，性能が劣化する．したがって，技術的には可能であるが性能上の問題から，PEACH2 は QPI を経由する GPU に対しては利用しないものとする．

QPI 経由の PCIe 通信の問題はよく知られている問題であり，この問題によって TCA のコンセプトが制限されることはない．例えば，PLX といった PCIe スイッチを Intel Xeon E5 CPU 内蔵の PCIe スイッチのかわりに利用し，QPI を経由することを回避すれば，PEACH2 は性能の劣化なしに 4 台の GPU やそれ以上の GPU にアクセスできる．

4.3.2 PEACH2 の DMA Controller の機能について

PEACH2 が通信を行う際は PEACH2 に内蔵されている DMAC (DMA Controller) を用いる．DMAC は PEACH2 の核となる機能であり，PEACH2 の中に 4 チャンネルの DMAC が実装され，それぞれの DMAC が独立して動作できる．

PEACH2 の DMAC を操作する際には，DMA Descriptor と呼ばれるデータ構造を用いる．DMA Descriptor には，送り元のメモリ領域の情報や送り先のメモリ領域の情報といった通信に必要な情報が格納されている．

DMA Descriptor の構造を図 4.4 に示す．ただし，図 4.4 の構造は説明のために必要な部分のみを抽出したものであり，実際の DMA Descriptor のデータ構造と同一ではない．“source address” に送信元のメモリアドレスを，“destination address” に送信先のメモリアドレスを指定する．ただし，PEACH2 は物理アドレスベースのメモリアクセスを行うため，2 つのフィールドには仮想アドレスではなく物理アドレスを指定する．“transfer size” はデータ転送の長さを指定するフィールドであり，“flags” フィールドは転送方法などを指定する汎用的なフィールドである．“next” フィールドは次に述べる Chaining DMA 機能のためのフィールドである．

PEACH2 の DMAC は Chaining DMA という特徴的な機能を持っており，複数の DMA Descriptor を接続して，1 つの DMA Descriptor では表現できないような複雑な通信に対応できる．Chaining DMA を表現するために DMA Descriptor はリスト構造となっており，DMA Descriptor の “next” フィールドに次のディスクリプタの物理アドレスを指定する．もし，終端の DMA Descriptor で，次の DMA Descriptor がない場合は “next” フィールドには 0 を指定する．Chaining されている DMA Descriptor リストの先頭の DMA Descriptor の転送開始を DMAC に指示すると，先頭の DMA Descriptor に繋がっている DMA Descriptor も自動的に DMAC が認識し，連続して転送される．例えば 3 回の通信を連続して行う場合について考えると，Chaining DMA 機能を使わない場合は 3 回の DMAC 操作が必要になる．一方で，

source address	送信元アドレス
destination address	送信先アドレス
transfer size	転送長
next	次のディスクリプタのアドレス (0=なし)
flags	各種フラグ

図 4.4　PEACH2 の DMA Descriptor のデータ構造.

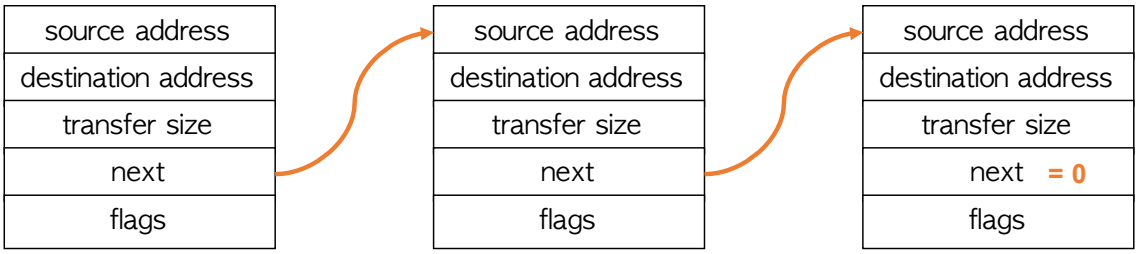

図 4.5　DMA Chainig の例．3 つの DMA Descriptor を接続する例．終端の DMA Descriptor の “next” フィールドには 0 を指定する.

Chaining DMA 機能を用いる場合，あらかじめ 3 つの DMA Descriptor を接続すれば，図 4.5 の様に 1 回の DMAC 操作で 3 つの転送を行える．PEACH2 の DMAC に対して指示を行う場合は，PCIe を経由して PEACH2 内のレジスタを操作するためオーバーヘッドが存在するが，Chaining DMA 機能を利用すれば，DMAC の操作回数を減らせるためオーバーヘッドの削減に繋る.

科学技術計算では，メモリアドレスや転送長が同じパターンの通信を繰り返して行うことが一般的であり，DMA Descriptor の再利用や PEACH2 の DMA Chaining の機能を効果的に利用できる．2 次元ステンシル計算における袖領域の通信を模して PEACH2 の DMA Chaining 機能を用いて通信する場合の例を図 4.6 に示す．図 4.6 は，4 つの袖領域 NORTH, EAST, SOUTH, WEST の通信に対して，それぞれ DMA Descriptor を作成し，4 つの DMA Descriptor を NORTH, EAST, SOUTH, WEST の順番で接続している状態を示す．この状態で，NORTH 通信用の DMA Descriptor の通信開始を DMAC に指示すると，NORTH, EAST, SOUTH, WEST の 4 つの通信が 1 度の指示で順々に行われる.

4.4　TCA 実証開発環境: HA-PACS/TCA

本節では，TCA アーキテクチャの実証環境として筑波大学計算科学研究センターに設置されている，HA-PACS/TCA システムについて述べる [37]．HA-PACS/TCA のノードの写真を図 4.7 に，ノードの仕様を表 4.1 に示す.

HA-PACS/TCA は全章で述べた HA-PACS ベースクラスタと一体として運用されており，ユーザからは 1 つのクラスタとして見える．しかしながら，ノードの構成はベースクラスタと異なっており，CPU と GPU のどちらのプロセッサもベースクラスタより 1 世代進んだものが利用されており，1 ノードあた

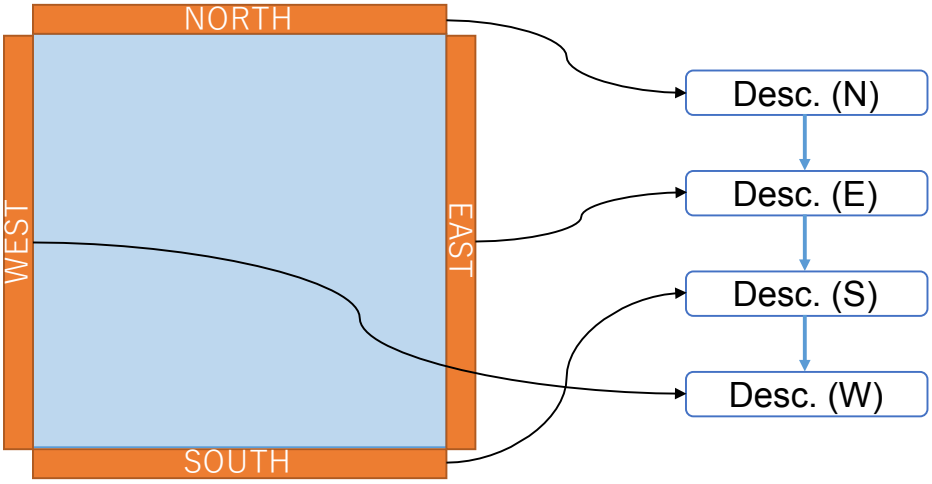

図 4.6 2 次元ステンシル計算における袖領域の通信で DMA Chaining を利用する例.

表 4.1 HA-PACS/TCA のノードの仕様.

CPU	Intel(R) Xeon(R) CPU E5-2680 v2 2.80 GHz × 2
CPU Memory	64 GB / CPU
GPU	NVIDIA Tesla K20X × 4
GPU Memory	6 GB / GPU
IB HBA	Mellanox Connect-X3 Dual-port QDR
PEACH2	1 board / node
OS	CentOS 6.4
CUDA Toolkit	version 6.5
GPU Driver	version 340.32
MPI	MVAPICH2-GDR 2.0 built for CUDA 6.5

り 2 つの Intel Xeon CPU (Ivybridge) と，4 つの NVIDIA Tesla GPU (Kepler) が搭載されている．また，HA-PACS/TCA は TCA アーキテクチャの開発および性能評価に用いることを想定し，通信デバイスとして，1 ノードあたり 1 枚の PEACH2 ボードと 1 枚 InfiniBand HCA ボードが搭載されている．PEACH2 だけでなく InfiniBand でも通信を行えることから，PEACH2 と InfiniBand の性能比較に適する環境である．

GPU は 1 ノードあたり 4 つ搭載されているものの，前節で述べた QPI を経由する PCIe 通信の性能劣化問題があるため，GPU の利用には制限が加えられる．QPI を経由すると十分な性能が得られないため，PEACH2 の通信の対象とする GPU は GPU0, GPU1 の 2 つのみである．また，InfiniBand も GDR を用いて利用する場合は PEACH2 と同様の QPI の制限を受けるため，GDR 利用時の InfiniBand の通信の対象

図 4.7　HA-PACS/TCA クラスタの写真.

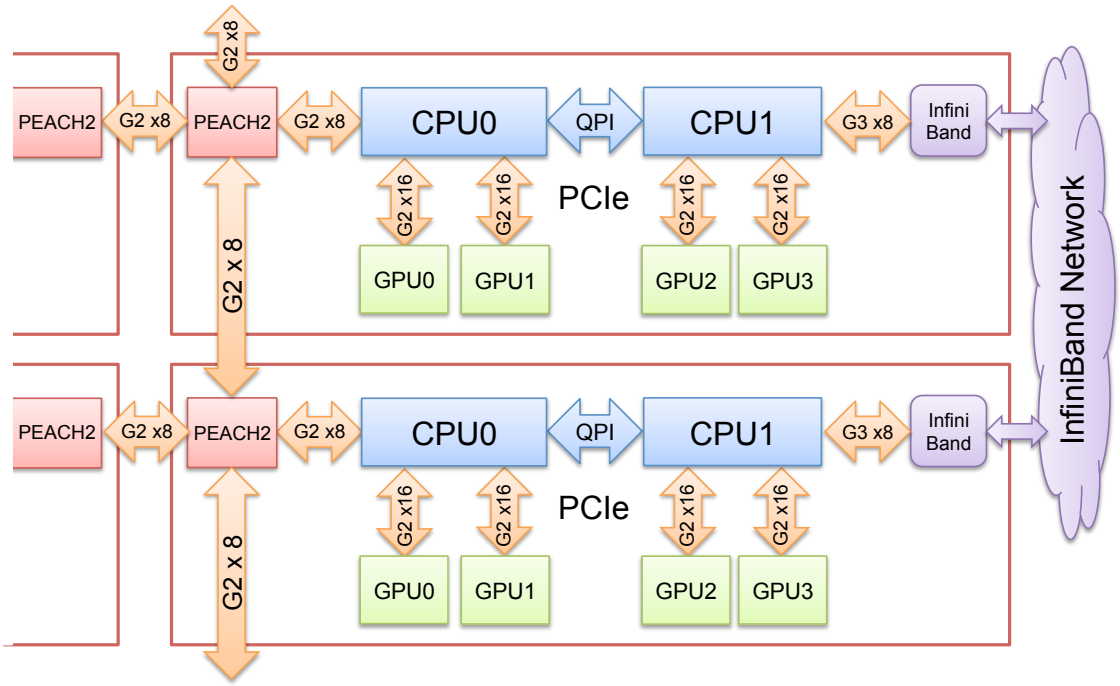

図 4.8　HA-PACS/TCA クラスタにおける各コンポーネント間の接続関係図.

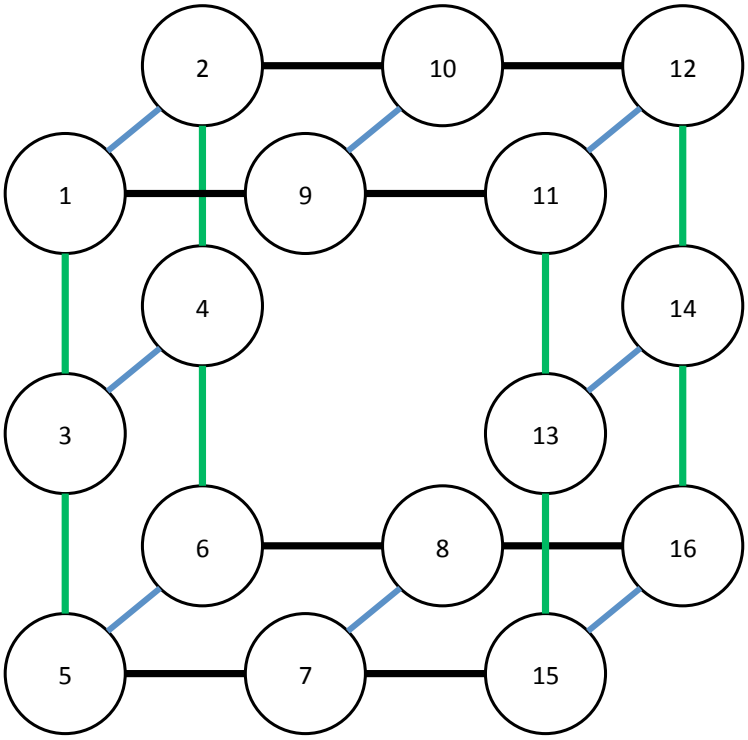

図 4.9 HA-PACS/TCA における PEACH2 ネットワークの構成図．8 ノードのリングが 2 つあり，合計で 16 ノードが 1 つのネットワークを構成している．図中の数字は PEACH2 におけるノード番号を示す．

とする GPU は GPU2, GPU3 の 2 つのみである．ただし，QPI による性能低下は PCIe デバイス同士の通信にのみ影響され，CPU と GPU 間のアクセスに関しては問題はなく，例えばホストメモリを経由して通信する場合は性能の低下はない．

HA-PACS/TCA は全体で 64 ノードから構成されているが，PEACH2 は 1 ネットワークに含められるノード数が最大で 16 ノードであるため，16 ノードから構成される PEACH2 ネットワークが 4 つ存在し，それぞれが独立している．一方で，InfiniBand ネットワークは fat-tree 構造となっており，全 64 ノードが 1 つのネットワークに含まれており相互に通信できる．HA-PACS/TCA における PEACH2 ネットワークの構成を図 4.9 に示す．PEACH2 ネットワークは PEACH2 が持つ 3 つの外部リンクを利用して構築され，8 ノードの一本のリングが 2 つあり，それら 2 つのリングを向い合せに接続し，合計で 16 ノードが 1 つのネットワークを構成する．なお，図 4.9 における数字は PEACH2 ネットワークにおけるノード番号を意味し，HA-PACS/TCA 全体で割り当てられているノード番号とは異なる．

第 5 章

GPU 向け Lattice QCD ライブラリ QUDA の TCA による高速化

5.1　Lattice QCD ライブラリ QUDA について

QUDA はオープンソースの Lattice QCD (Quantum Chrono-Dynamics) フレームワークライブラリであり，NVIDIA GPU アクセラレータの利用をメインターゲットとして開発されている．本章の目的は，QUDA のステンシル計算に含まれている通信について，TCA および RMA に対応する QUDA の通信コードを開発することで QUDA の計算の高速化を行うことである．

QUDA のプログラムの構造は以下の通りである．QUDA における計算は大きく分けて 2 つの種類があり，1 つは Dirac operator におけるステンシル計算，もう 1 つはクリロフ部分空間法による連立一次方程式の求解である．QUDA は CG (Conjugate Gradient) 法や BiCGSTAB (BiConjugate Gradient Stabilized) 法といった複数のクリロフ部分空間法を実装しており，解く対象の行列によってアルゴリズムを切り替える．

QUDA はシングル GPU の環境に加えて，1 つのノード内に複数の GPU があるマルチ GPU 環境もサポートしている．また，複数のノードを利用するノード間分散もサポートしており [38]，x, y, z, t の 4 つの次元に対して領域分割を行える．ノード間の通信には，MPI [39] と QMP (Lattice QCD Message Passing) [40] を利用できる．本章では，MPI 通信の拡張として TCA のサポートを QUDA に追加する．現時点では TCA ネットワークは単体で利用できないため，TCA 通信の準備のために MPI を利用するが，計算中の通信は MPI は利用せず，TCA のみによって行われる．

QUDA のステンシル計算の流れを図 5.1 に示す．水色の箱は CPU 側コードにおける CUDA カーネルの起動，紫色の箱は GPU 上でのカーネルコードの実行，緑色の箱は MPI 通信を，赤色は CPU と GPU の間の同期のための `cudaEventSynchronize` の呼び出しを表す．

オリジナルの QUDA の実装では，MPI と QMP の両方の実装において，通信に point-to-point プロトコルを用いている (MPI においては `MPI_Send` 関数および `MPI_Recv` 関数)．しかしながら，TCA は RMA Write と RMA Read のみしか行えないため，QUDA の通信部を point-to-point プロトコルを利用するオリジナルの構造から RMA を利用するように変更しなければならない．また，図 5.1 内の “Halo data exchange” と “Calc. of inner points” の計算の通信については，通信時間の隠蔽のためにパイプライニング

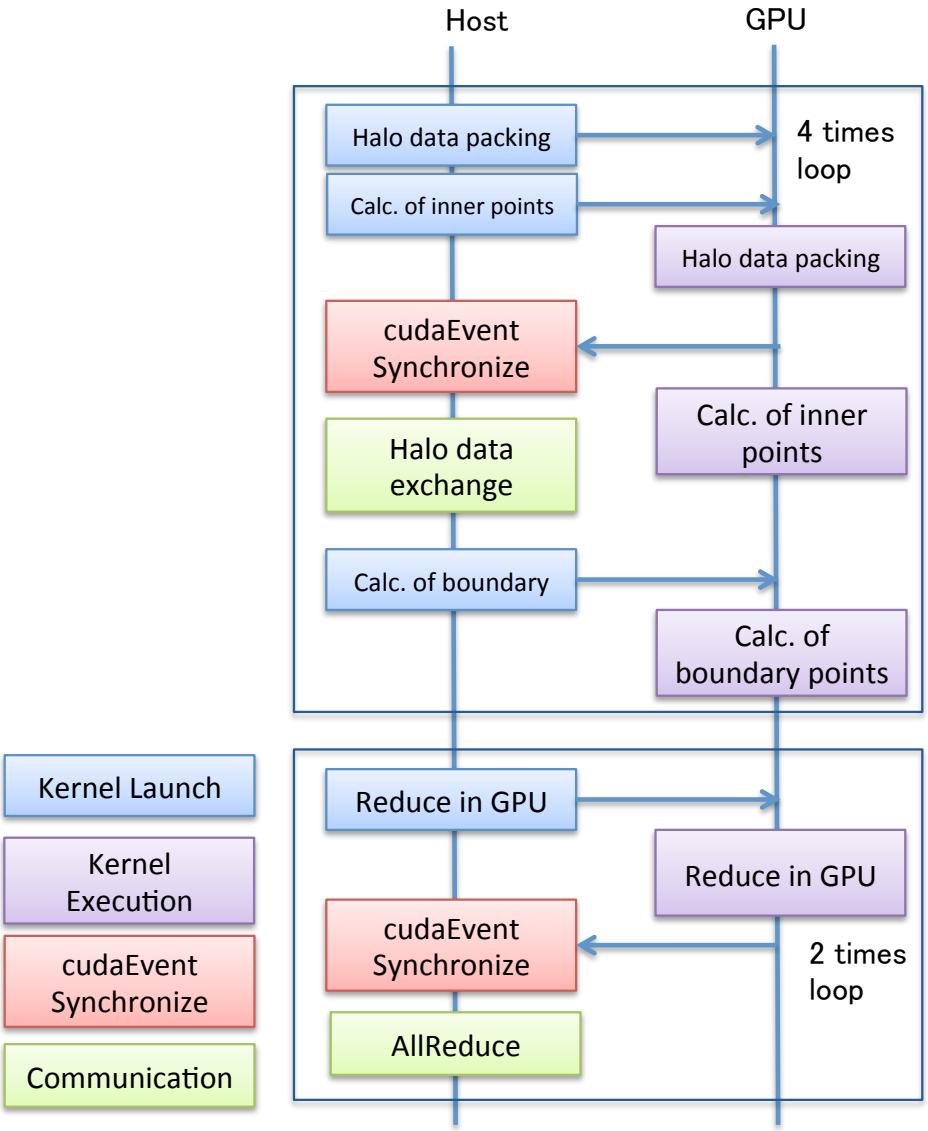

図 5.1 QUDA ステンシル計算の流れ.

を用いてオーバーラップするように実装されている.

QUDA を TCA によって加速する研究は NVIDIA および NVIDIA Japan との共同研究として行われており，ベースとなる QUDA は Mike Clark らによって開発が行われている [41, 42]. なお，QUDA の実装はオープンソースとして公開されており，github のプロジェクトページより入手できる [43].

5.2 QUDA の Remote Memory Access への対応

5.2.1 Remote Memory Access への対応

TCA は MPI の片方向通信における `MPI_Put`, `MPI_Get` 関数のような片方向の Remote Write, Remote Read 命令のみサポートしている．しかしながら，PEACH2 の RMA Read は，PEACH2 のドライバによるソフトウェア処理によってリモートノードが返答するプロトコルになっており，Read のオーバーヘッドが Write よりも大きいため，Read 通信の利用は避けるべきである．QUDA の TCA による実装では，Write 通信のみで必要な通信を実装できており，Read 通信は使用していない．

TCA は MPI といった他の通信手段を併用することを前提としており，分散プログラムを書くために必要な一部機能は提供していない．たとえば，MPI の実装では `mpirun` や `mpiexec` といった名前で提供されている複数のノードでプログラムを起動するための機能や，`MPI_Comm_rank` 関数のようなプロセスマッピングのための機能は提供されていない．また，TCA で通信を行うためには，通信に必要なアドレスなどの情報を事前にやり取りしておく必要があるが，TCA 通信に必要な情報を TCA で送ることはできないため，他の通信手段を用いて行う．QUDA の TCA による実装では，既に QUDA は MPI による並列化プログラミングが行われているため，QUDA の MPI 実装をベースにして TCA の実装を行う．

QUDA の RMA 実装では，TCA による実装だけではなく，MPI-3 RMA を用いる実装も行う．MPI と TCA の性能比較を行う際は，MPI peer-to-peer による実装と比較するよりも，MPI-3 RMA による実装と比較した方がより公平であり，加えて，QUDA の通信を peer-to-peer から RMA に変更することによる性能変化についても調査が必要だからである．また，GPU のメモリへのアクセスは CPU のメモリにアクセスする場合よりも時間が必要であり，バッファリング等の複雑なメモリ操作が必要な peer-to-peer プロトコルよりも，ただメモリにデータを書き込むだけのシンプルな RMA プロトコルの方が性能が良いと考えられる．また，RMA の実装に TCA, MPI-3 RMA2 つの方式を持つことで，プログラムにバグが発生した場合に，TCA のシステムの問題なのか，RMA プログラムのバグなのかの切り分けが行いやすいという利点もある．

QUDA は通信を抽象化する層を持っており，オリジナルの QUDA の実装では MPI と QMP の実装を抽象化レイヤーの下に実装しており，2 つの通信手法を切り替えて利用できる．本章では，QUDA の通信抽象化レイヤーを拡張し，RMA 通信用の API を実装する．以下，RMA 用の抽象化レイヤーを RMA API と呼ぶ．そして，RMA API に対する RMA 通信の実装として TCA および MPI-3 RMA を実装する．

QUDA の point-to-point 通信を RMA API を用いて RMA 通信に変更する際は，TCA の仕様にあわせて RMA Write のみを用いる．したがって，point-to-point 通信の Send 側を RMA Write に置き換え，point-to-point 通信の Recv 側を削除する．本来，point-to-point 通信では，送信側は受信側のどこのメモリに書き込まれるかわからず，受信側も同様に送信側のどこのメモリから書き込まれるのかわからないため，単純に RMA 通信に置き換えることはできないが，QUDA のステンシル計算の袖領域の通信では，予めどこのデータがどこに転送されるかが明かであり，また，その通信パターンは計算中に変化せず固定であるため，peer-to-peer Send を RMA Write に置き換えることで RMA 通信に対応できる．

5.2.2 RMA 通信用 Message Handle 拡張

QUDA の通信抽象化レイヤーは，通信を MsgHandle オブジェクトとして表現する．これは，一回の通信を表現するもので，例えば MPI_Send 一回の呼び出しに相当する．また，MsgHandle は固定された通信のパターンを表現し，アプリケーション実行中に使い回して利用することを想定している．MsgHandle は使い捨てではなく，同じ通信パターン (メモリアドレスおよび通信長) であれば何度でも通信に利用できる．

1 つの MsgHandle は複数回利用されるため，MsgHandle の実装においては永続通信関数を利用して実装できる．MPI peer-to-peer 実装では， MsgHandle の実装に MPI_Send_init 関数や MPI_Recv_init 関数によって作られる MPI_Request を用いている．MPI_Request は，通信データのメモリアドレスや型情報，データ長といった， MPI_Send や MPI_Recv 関数に必要とされる通信に必要な情報を全て含んでおり，MPI_Start 関数で実際の通信を開始できる．

QUDA RMA API を実装する上で MsgHandle の概念はそのまま利用するが，TCA，MPI-3 RMA それぞれの実装で MsgHandle の中身は異なり，それぞれの通信ライブラリに適合する MsgHandle を作成する．TCA，MPI-3 RMA それぞれの実装における MsgHandle の拡張の詳細は次節で述べる．

5.2.3 Memory Window Object

QUDA の RMA 実装に RmaWindow オブジェクトを導入する．RmaWindow オブジェクトは RMA 通信で読み書きするメモリ領域を表すオブジェクトである．RmaWindow はメモリ確保・開放の方法も抽象化する役割を持ち， RmaWindow を確保する際は RmaWindow のサイズも指定し，実装にあわせた最適な方法でメモリを確保する．TCA では，RMA 通信に用いるメモリ領域は専用の API で確保しなければならないため，このような設計とする．

MPI-3 RMA では，RMA 通信先は RMA 用として登録されたメモリ領域しか使えないが，RMA 通信元のメモリ領域は RMA 用として登録されていない領域でも利用できる．しかしながら，TCA での通信では，MPI-3 RMA とは異なり，通信先と通信元どちらのメモリ領域も API によって管理されていなければならない．よって，QUDA RMA API では，RMA 通信の対象とする送信元と送信先のメモリ領域は，RmaWindow によって管理されているメモリ領域でなければならないとする．なお，QUDA は RMA 通信に GPU メモリ領域しか利用しないため， RmaWindow は GPU 上のメモリを指すポインタのみを含む．

5.2.4 RMA Operation Queue

QUDA RMA API に RMA 通信の制御を行うためのオブジェクトとして RmaQueue オブジェクトを導入する．RmaQueue は RMA 通信の開始と完了を管理するオブジェクトであり，どの MsgHandle を通信するかの情報と，どのランクからの RMA 通信を待機するのかの 2 つの情報を持つ．CUDA プログラミングにおける CUDA Stream (cudaStream_t 型および操作用 API) のように，どのような RMA 通信処理を行うのかをキュー登録し利用する．

RmaQueue は以下に示す 8 つの操作を行える．

Alloc

```
RmaQueue *comm_queue_alloc(RmaWindow *window);
```

RmaWindow *window* に紐付けられた RmaQueue を作成する.

Free

```
void comm_queue_free(RmaQueue *queue);
```

RmaQueue *queue* を破棄する.

Push

```
void comm_queue_push(RmaQueue *queue, MsgHandle *mh);
```

RmaQueue *queue* に MsgHandle *mh* を追加する.

Commit

```
void comm_queue_commit(RmaQueue *queue);
```

RmaQueue *queue* の設定が完了し， RmaQueue の準備が完了したことを宣言する.

Add Origin

```
void comm_queue_add_origin(RmaQueue *queue, int rank);
```

RmaQueue *queue* に対してプロセス *rank* からの RMA 通信を待機するかを指定する.

Start

```
void comm_queue_start(RmaQueue *queue);
```

RmaQueue *queue* に格納された MsgHandle の通信を開始する.

Wait

```
void comm_queue_wait(RmaQueue *queue);
```

Start 操作によって開始された RMA 通信の完了を待機する.

Clear

```
void comm_queue_clear(RmaQueue *queue);
```

RmaQueue *queue* に登録された全ての情報を消去する.

Alloc 操作は，ある RmaWindow の通信を行うための RmaQueue を作成するためのものである．現状の仕様では，1 つの RmaQueue は 1 つの RmaWindow しか通信の対象にできない．しかしながら，QUDA を RMA で実装する上では 1 つの RmaWindow のみ対象にできれば十分であるため，問題にはならない.

Free 操作は *Alloc* 操作で作成した RmaQueue を開放するためのものである．なお， *Free* 操作は RmaQueue のみを開放するだけであり，RmaQueue に登録された MsgHandle や RmaWindow は開放されない.

Push 操作は RmaQueue に通信用の MsgHandle を登録するものであり，RmaQueue で通信を開始すると， RmaQueue に登録されている MsgHandle に登録されている通信を順次起動していく.

Commit 操作では， RmaQueue に *Push* 操作による MsgHandle の登録が完了し，通信準備が完了したことを RmaQueue に通知する．RmaQueue 内部では， *Commit* 操作のタイミングで通信の準備や最適化を行う.

Add Origin 操作は他のプロセスによる RMA Write の完了を待機するために使われる．RmaQueue の通信同期はバリア同期のように全プロセスを一斉に同期するものではなく，実際に通信を行う相手とだけ同期を行い同期にかかるオーバーヘッドを削減する．自プロセスからどのプロセスに書き込みに行くかは，*Push* 操作で登録された MsgHandle を参照すれば判明するが，どのプロセスから書き込まれるのを待機しなければならないかは MsgHandle を参照しただけでは判断できない．その問題を解決するために， *Add Origin* 操作を用いてどのプロセスから書き込まれるかの情報の登録が必要となる．

Start 操作は RmaQueue に登録された全ての MsgHandle の通信を開始する．*Start* 操作は RmaQueue の内容を変更しないため，同じ通信をくりかえす場合は，同じ RmaQueue に対して複数回 *Start* 操作を行っても良い．

Wait 操作は，自プロセスが開始した書き込み RMA 通信と，他プロセスから書き込まれる RMA 通信の両方が完了するまで待機する．どのプロセスから書き込まれるのを待機するかは，*Add Origin* の説明で述べたように， *Add Origin* 操作によって登録された情報に基いて決定する．

Clear 操作は RmaQueue に登録された情報を消去するためのものである．例えば，通信パターンが変化したため RmaQueue を再構築する場合などに利用する．

5.2.5 QUDA RMA API

本 API を用いてプログラムを開発する場合の典型的な処理の流れを図 5.2 に示す．まず，計算の初期化フェイズ (図 5.2 の Loop 前部) で RMA 通信のための準備を行う．この段階で RmaWindow や RmaQueue の作成を行う．また，通信パターンにあわせた MsgHandle を作成し， RmaQueue に登録する．同時に RmaQueue の *Add Origin* 操作で，RMA 通信で書き込まれる通信に関する情報も登録する．そして，最後に *Commit* 操作を行い通信の準備を完了する．

計算ループ内では，*Start* および *Wait* 操作を使い通信を行う．この図では，計算中に通信パターンは変化しないと想定しているため，RmaQueue を変更せずに，計算のループイテレーションが進んでも同じ RmaQueue を再利用する．

最後に，後処理フェイズ (図 5.2 の Loop 後部) で後処理を行う．MsgHandle や RmaWindow や RmaQueue の開放を行い，各種リソースを開放する．

5.3 RMA 通信の TCA および MPI-3 RMA による QUDA RMA API 実装

5.3.1 MPI-3 RMA による実装

QUDA オリジナルの MPI point-to-point 通信では， MPI の永続通信 API（MPI_Send_init や MPI_Recv_init など）を用いており MsgHandle はそれらの API で作成される MPI_Request を保持しており，通信時には MPI_Request を MPI_Start で開始する実装となっている．しかしながら，MPI-3 RMA の仕様では，RMA 通信に対して永続通信の API を定義しておらず，peer-to-peer 通信と同様の実装はできない．QUDA RMA API の MPI-3 RMA 実装では，永続通信 API が存在しないため，

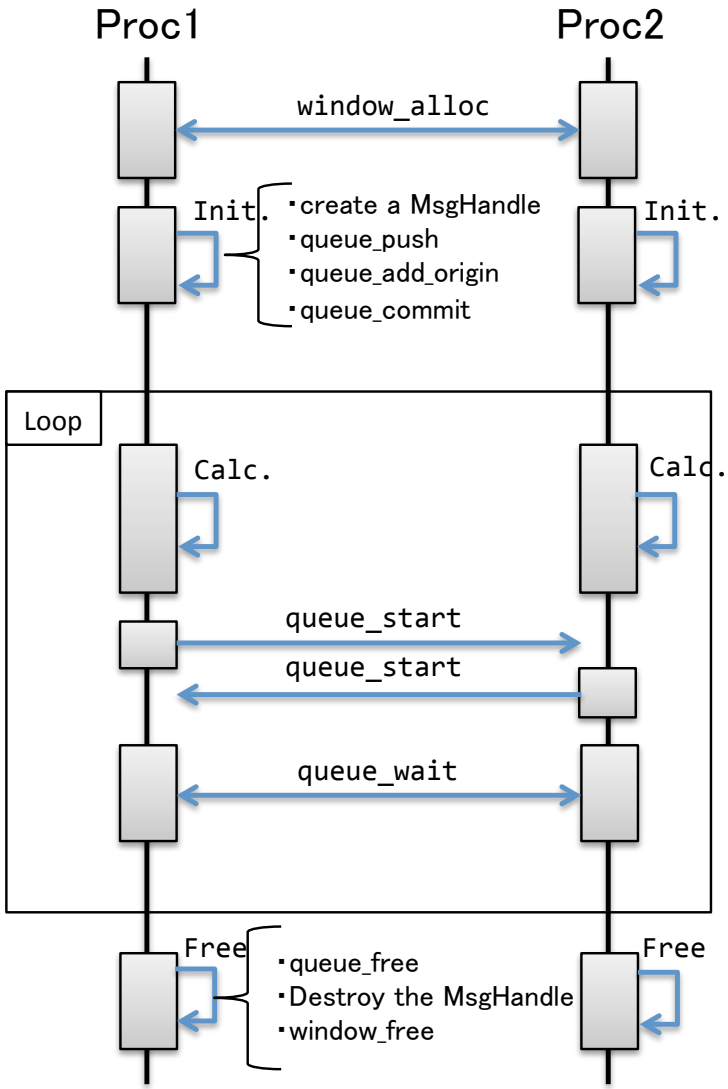

図 5.2 RMA 通信の典型的な流れ.

MsgHandle には MPI_Put 関数に渡される引数を内部に保持しており，通信時には MsgHandle 内に保持されているパラメータを MPI_Put 関数にそのまま渡すことで通信を行う.

MPI の GDR 実装では， MPI_Win_create 関数に GPU のポインタを渡すと GPU メモリ上の MPI_Win を作成できるため，RmaWindow は MPI_Win 型の変数を保持するだけであり， RmaWindow の作成は cudaMalloc 関数で GPU 上のメモリ領域を確保し，その領域に対して MPI_Win_create 関数で MPI_Win を作成する事と等価である．また，QUDA での実装では MPI_Get による RMA Read 通信は行わないが，受信用のメモリ領域にも RmaWindow を作成している.

RmaQueue の *Start* 操作は既に述べた通り MsgHandle に格納されている引数情報を用いて MPI_Put 関数を呼び出す動作となる．MPI-3 RMA の実装では *Commit* 操作では特別な動作は行なっておらず，空の関数が呼ばれる.

RmaQueue の *Wait* 操作では，MPI_Win_post，MPI_Win_start，MPI_Win_complete，MPI_Win_wait の 4 つの関数を用いて通信の同期を行う．プロセス全体でのバリア同期によって同期を行う関数もあるが，これら 4 つの関数は全体ではなく特定のプロセスとのみ同期を行う関数であり，どのプロセスと同期を取るかは MPI_Group を引数に与えることで指定する．RmaQueue では RmaQueue に登録されている MsgHandle の情報と，*Add Origin* 操作によって登録された書き込まれるプロセスに関する情報から MPI_Group を作成し利用する．

MPI の仕様では，MPI_Win に対して RMA 通信を行える期間を RMA Epoch として定義しており，RMA Epoch を開始する API と終了する API がいくつか定義されており，MPI_Put や MPI_Get といった通信用関数は RMA Epoch 内でしか呼び出してはならないと定められている．先程述べた MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait の 4 つの関数は，それぞれ RMA Epoch を開始または終了する関数であり，RmaQueue の実装ではこれらの関数を用いて RMA Epoch を操作する．

MPI_Win_post, MPI_Win_start の 2 つの関数は RMA Epoch を開始する関数である．2 つの関数は対となっており，MPI_Win_post は書き込まれる側の MPI_Win で RMA Epoch を開始し，MPI_Win_start は書き込む側の MPI_Win で RMA Epoch を開始する関数である．一方で，MPI_Win_complete, MPI_Win_wait の 2 つの関数は RMA Epoch を終了する関数である．これらの関数も対となっており，MPI_Win_post, MPI_Win_start の場合と同様に，書き込まれる側か書き込む側かで関数が分かれており，MPI_Win_post が書き込まれる側の関数，MPI_Win_complete が書き込む側の関数である．

5.3.2 TCA による実装

TCA の開発環境は，NVIDIA によって提供されている GPU 向け開発環境の CUDA Toolkit[6] の上に構築されており，TCA を利用している場合でも，GPU メモリ管理や GPU カーネル起動といった CUDA プログラミングの部分については変更を加える必要なく TCA を利用できる．

まず，TCA プログラミングにおけるメモリ管理および RmaWindow の実装方法について述べる．TCA 通信に用いるメモリは TCA 専用のメモリ確保関数 tcaMalloc を通じて確保する．CPU メモリの場合は，ハードウェアの制限から PEACH2 がアクセス可能な領域にメモリを確保するため，tcaMalloc はカーネルドライバを通じてその領域にあるメモリを確保する．また，通信性能をより良くするため，tcaMalloc で確保される CPU メモリは物理メモリ上で連続したアドレスに割り当てられる．GPU メモリの場合特殊なメモリ確保方法は必要なく，tcaMalloc は cudaMalloc を呼ぶだけであるが，TCA 通信の性能を良くするために，メモリ領域の先頭を GPU ページ境界 (64KB) に配置する処理が含まれている．

TCA による GPU メモリの通信を行う際には，まず通信したいメモリ領域を指す tcaHandle と呼ばれるメモリハンドルを作成する．TCA では転送を行う際は物理メモリアドレスを基して転送を行うため，転送する CPU メモリ・GPU メモリの物理アドレスを取得する手段が必要である．物理アドレスに関する操作はカーネル中でしかできないため，TCA プログラムは TCA のランタイムライブラリを通じてドライバを操作し，物理アドレスを得る．TCA ライブラリでは，RMA 通信のためのメモリ領域を

tcaHandle と呼ばれる構造体で表す．tcaHandle には，メモリ領域の物理アドレスだけでなく，対応する仮想アドレス，メモリ領域の大きさや tcaHandle が所属するノード番号格納されており，MPI における MPI_Win に相当するデータ構造である．ただし， tcaHandle に渡すポインタは tcaMalloc で確保したメモリ領域でなければならないため，既存の CUDA+MPI プログラムを TCA に書き換える際は，通信に用いるメモリの確保する箇所を変更する必要がある．

アプリケーションが TCA による通信を行う際には，送信元のプロセスは送信先のプロセスの物理アドレスが必要であり，何らかの手段で宛先の tcaHandle を取得していなければならない．しかしながら，TCA による通信の準備のために TCA を利用することはできないため，何らかの他の手段で tcaHandle を取得する必要がある．QUDA RMA API では，MPI との併用を前提とし， tcaHandle の交換作業は MPI を用いて行う．tcaHandle の交換に MPI を用いるとしても，それは通信の準備段階に MPI を利用するだけであり，計算中の重要な通信は TCA によって加速されている．

QUDA RMA API では， RmaWindow の作成時に内部でメモリも確保するという仕様のため，RmaWindow を作成する際に tcaMalloc によるメモリ確保および tcaHandle の作成と交換を行う．この段階では，どのプロセスと通信するかわからないため，MPI_Alltoall によって全対全交換を行い，全てのプロセスが他の全てのプロセスの tcaHandle を所有する状態となる．MPI_Win は 1 つの変数で全てのプロセスとの通信に利用できるが， tcaHandle はノード毎に 1 変数であるため， tcaHandle 用の配列を用意し格納する．

MPI-3 RMA 実装とは違い，TCA 実装においては RmaQueue の *Commit* 操作は最適化の上で重要である．TCA の性能を引き出すためには，DMAC への指示のオーバヘッドを削減するために，出来る限りの通信を DMA Chaining で繋げることが重要であるため，*Commit* 操作の中で，通信用の DMA Descriptor および DMA Chaining を作成する．RmaQueue の中に登録されている全ての MsgHandle を順々に繋げて，1 つの DMA Chaining を作成する．TCA 実装における MsgHandle は，MPI-3 RMA 実装と同様に，DMA Descriptor を作成するための TCA API に渡される全ての引数を保持しており，*Commit* 操作で DMA Descriptor を作成する際は，その情報を用いて DMA Descriptor を作成する．

Start 操作では，TCA 通信を開始する関数である tcaStartDMADesc 関数で通信を開始する．*Commit* 操作で通信内容は 1 つの DMA Chaining に接続されているため, DMA Chaining の先頭の DMA Descriptor の通信開始を DMAC に指示すると， RmaQueue に含まれている全ての通信が行われる．Wait 操作では RmaQueue に追加されている全ての RMA 通信が完了するのを待機する．自プロセスから行う RMA 通信だけでなく，Add Origin 操作で設定されたプロセスからの書き込みが完了するのも併せて待機する．

5.4 性能評価

本節では，MPI point-to-point (オリジナル版による実装), MPI-3 RMA, TCA の 3 つの実装の性能を比較評価する．

5.4.1 計算機環境

性能評価には，前章で述べた筑波大学 計算科学研究センターに設置されている HA-PACS/TCA GPU クラスタを用いる．QUDA は 1 プロセスあたり 1 GPU を利用できる．QUDA でノード内の複数の GPU を利用する場合は，MPI のプロセス数を増やす必要があるが，PEACH2 はマルチプロセスから同時に利用すると，排他制御やキューといった保護機構がないため，正常に動作できない．したがって，1 ノードあたり 1 プロセス，1 GPU を上限として性能を測定する．

QPI をまたぐ PCIe デバイス間通信は性能が劣化するため，PEACH2 と InfiniBand HCA はそれぞれのデバイスが接続された CPU と同じ CPU に接続されている GPU のみを利用する．したがって，図 4.8 において，PEACH2 は GPU0 と GPU1 のメモリにダイレクトにアクセスでき，InfiniBand は GPU2 と GPU3 のメモリにダイレクトにアクセスできる．QUDA は 1 プロセスあたり 1 つの GPU を利用するため，TCA の性能評価を行う際は GPU0 を，InfiniBand の性能評価を行う場合は GPU2 を利用する．

HA-PACS/TCA に搭載されている InfiniBand HCA は 2 つのポートを持つモデルであり，MPI 等の通信ライブラリが複数ポート利用に対応していれば，2 つのポートを同時に利用して通信を行えるため，通信帯域を 2 倍にできる．MVAPICH2 は複数ポート利用に対応しており，環境変数 `MV2_NUM_PORTS` を設定することで何ポート利用するかを制御でき，例えば，`MV2_NUM_PORTS` を 2 に設定すると 2 つのポートを利用できる．

本章の性能評価では，TCA 1 リンクと InfiniBand 1 リンクの間での性能比較を意図しており，`MV2_NUM_PORTS` を 1 に設定し性能を測定する．InfiniBand の使用ポート数を 1 に制限すれば，HA-PACS/TCA に搭載されている InfiniBand HCA の 1 ポートの帯域は，PEACH2 の 1 リンクの帯域と同じ 4GB/s の帯域を持つことになる．

5.4.2 性能測定に用いる各種設定

性能評価には QUDA のテスト用プログラムである `invert_test` を用いる．`invert_test` プログラムは，MPI-3 RMA および TCA による加速のターゲットとしたステンシル計算によって生成される連立一次方程式を CG 法で解き，CG 法の反復回数と演算性能を GFLOPS で表示するものである．

X, Y, Z, T はそれぞれ x, y, z, t 次元のメッシュサイズを表し，n_X, n_Y, n_Z, n_T はそれぞれ x, y, z, t 次元の分割プロセス数を表す．また，N_X, N_Y, N_Z, N_T はそれぞれの次元における合計メッシュサイズを表し，$(N_X,\ N_Y,\ N_Z,\ N_T) = (X \times n_X,\ Y \times n_y,\ Z \times n_Z,\ T \times n_T)$ で求められる．

性能測定には Small Model と Large Model の 2 つの問題サイズを用いる．Small Model のサイズは $(N_X,\ N_Y,\ N_Z,\ N_T) = (8,\ 8,\ 8,\ 8)$ であり，Large Model のサイズは $(N_X,\ N_Y,\ N_Z,\ N_T) = (16,\ 16,\ 16,\ 16)$ である．4 つの次元で問題を分割しているため，同じノード (プロセス) 数の設定で比較すると，Small Model の 1 ノードに割り当てられる問題の粒度は Large Model の $\frac{1}{16}$ である．

PEACH2 では，1 つのネットワーク内のノード数の上限が 16 ノードであるため，性能測定では最大 16 ノードを使用する．本評価では x, y 次元について分割を行い，z, t 次元については分割を行わない．MPI peer-to-peer 実装と MPI-3 RMA 実装では，環境変数 `MV2_USE_CUDA` を 1 に，`MV2_USE_GPUDIRECT` を

表 5.1 2 つの問題サイズにおける QUDA の通信データサイズ (L_x, L_y). ただし単精度浮動小数点数を利用する場合.

プロセス数	L_x, L_y (Small Model)	L_x, L_y (Large Model)
$n_x, n_y = 1$	24KB	384KB
$n_x, n_y = 2$	12KB	192KB
$n_x, n_y = 4$	6KB	96KB

1 に指定し，GDR を有効にして測定を行う.

MVAPICH2 にはチューニング用パラメータとして `MV2_GPUDIRECT_LIMIT` 環境変数がある. `MV2_GPUDIRECT_LIMIT` 環境変数は，何バイトまでの通信に GDR による通信を利用するかを指定するためのパラメータであり，`MV2_GPUDIRECT_LIMIT` 以下までの通信は GDR を利用して行い，`MV2_GPUDIRECT_LIMIT` より大きい通信については GDR を用いない通信手法に切り替わる. また，`MV2_GPUDIRECT_LIMIT` を明示的に指定しない場合のデフォルト値は 8KB である.

`MV2_GPUDIRECT_LIMIT` の最適な値は環境依存であり，利用する GPU デバイスの種類，InfiniBand アダプタの種類や MVAPICH2 のバージョンによって異なると考えられる. HA-PACS/TCA では，事前評価の結果， `MV2_GPUDIRECT_LIMIT` は 512KB に設定するのが良いとわかっており，QUDA の評価でも `MV2_GPUDIRECT_LIMIT` を無指定 (=8KB) の場合と，512KB に指定する場合の 2 パターンを測定する.

今回の測定に利用する条件では，通信量は 512KB をこえないため， `MV2_GPUDIRECT_LIMIT` を 512KB に指定する場合は全ての通信に GDR が利用されることになる. PEACH2 はホスト経由の通信を行わず，全て GDR 経由で直接 GPU メモリにアクセスするため，`MV2_GPUDIRECT_LIMIT` を 512KB に指定する場合は InfiniBand と PEACH2 で GPU へのメモリアクセスの条件は同じになる.

5.4.3 通信データサイズについて

QUDA のステンシル計算中に発生する通信のデータサイズは，x 次元方向のメッセージ長を L_x，y 次元方向のメッセージ長 (バイト) を L_y とすると，L_x, L_y は次の 2 つの式で示せる. $L_x = 12 \times s \times Y \times Z \times T$, $L_y = 12 \times s \times X \times Z \times T$ ただし，s は計算に用いる浮動小数点数のバイト数であり，例えば単精度だと $s = 4$，倍精度だと $s = 8$ となるが，本節では単精度を用いて性能評価を行うため $s = 4$ である. また，それぞれの分割された次元で 2 つの隣接プロセスがあるため，1 つの次元にあたり 2 つの通信メッセージが発生する.

Small Model の評価では，$X = Y = Z = T = 8$ であり，z, t の次元には分割しない ($n_z = 1, n_t = 1$) ため，$L_x = 3072 \times Y, L_y = 3072 \times X$ である. 同様に，Large Model の評価では，$X = Y = Z = T = 16$ であり，z, t の次元には分割しない ($n_z = 1, n_t = 1$) ため，$L_x = 24576 \times Y, L_y = 24576 \times X$ である.

評価に用いる各分割数における，1 通信あたりの通信データサイズを表 5.1 に示す. n_x, n_y はそれぞれ $1, 2, 4$ の値に設定し性能評価を行う. また，$X = Y$ であるため，問題分割数が同じならば，X 次元も Y 次元も通信データサイズは同じとなる.

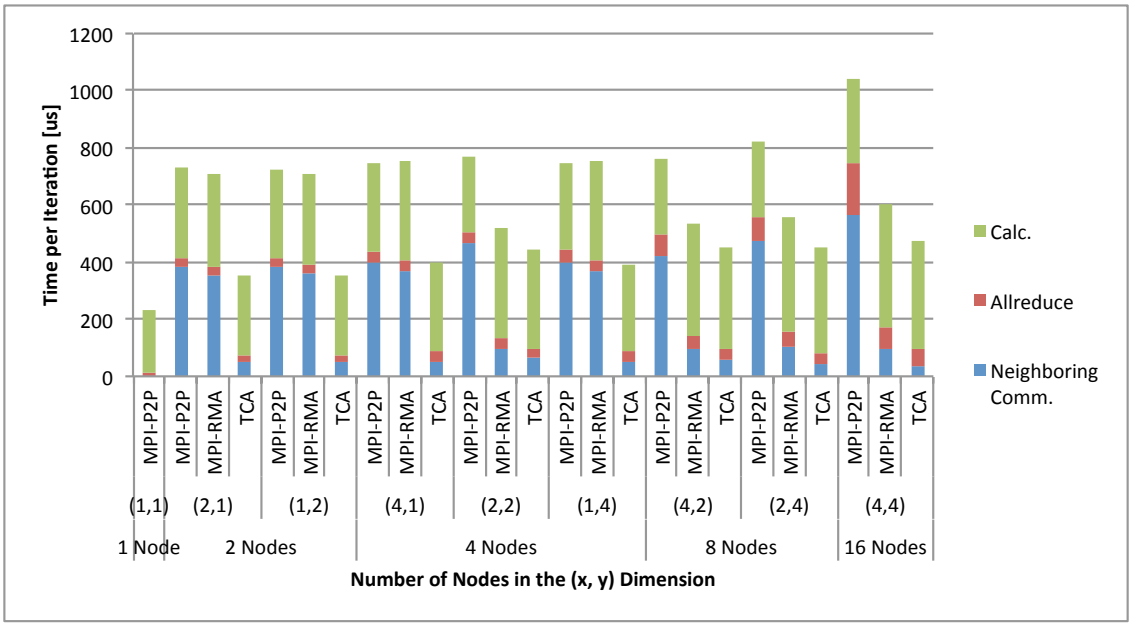

図 5.3 Small Model の計算時間と通信時間の内訳．(MV2_GPUDIRECT_LIMIT = 8KB)

5.4.4 測定結果

Small, Large の 2 つの問題サイズと，GDR パラメータ MV2_GPUDIRECT_LIMIT 環境変数を無指定 (8KB) と 512KB の 2 パターンを合せて 4 パターンの性能測定を行う．Small Model (MV2_GPUDIRECT_LIMIT = 8KB) における演算性能の比較を図 5.5 に，計算時間の内訳を図 5.3 に示す．Small Model (MV2_GPUDIRECT_LIMIT = 512KB) における演算性能の比較を図 5.6 に，計算時間の内訳を図 5.4 に示す．Large Model (MV2_GPUDIRECT_LIMIT = 8KB) における演算性能の比較を図 5.9 に，計算時間の内訳を図 5.7 に示す．Large Model (MV2_GPUDIRECT_LIMIT = 512KB) における演算性能の比較を図 5.10 に，計算時間の内訳を図 5.8 に示す．なお，GFLOPS の値は invert_test プログラムによって報告されるものであり，問題サイズやノード数の設定が異なると CG 法の反復回数が異なる場合があるため，計算時間の内訳のグラフは反復回数で正規化している．また，これらの結果は Rank 0 上で測定したものである．

Small Model では，どのノード数設定においても TCA による実装は 2 つの MPI による実装よりも高速である．MV2_GPUDIRECT_LIMIT = 8KB の場合では，$(n_x, n_y) = (4, 4)$ の場合に TCA 実装は MPI point-to-point 実装よりも 2.19 倍高速であり $(n_x, n_y) = (2, 1)$ の場合に TCA 実装は MPI3-RMA 実装よりも 2.02 倍高速である．MV2_GPUDIRECT_LIMIT = 512KB の場合では，$(n_x, n_y) = (4, 4)$ の場合に TCA 実装は MPI point-to-point 実装よりも 2.19 倍高速であり $(n_x, n_y) = (4, 4)$ の場合に TCA 実装は MPI3-RMA 実装よりも 1.23 倍高速である．しかしながら，ノードあたりのメッシュサイズが小さく CUDA kernel の起動コストがボトルネックとなり，ノード数を増やしても計算時間がスケールしていない．MPI 同士の実装で比較すると，MPI-3 RMA 実装は全てのノード数において MPI point-to-point より

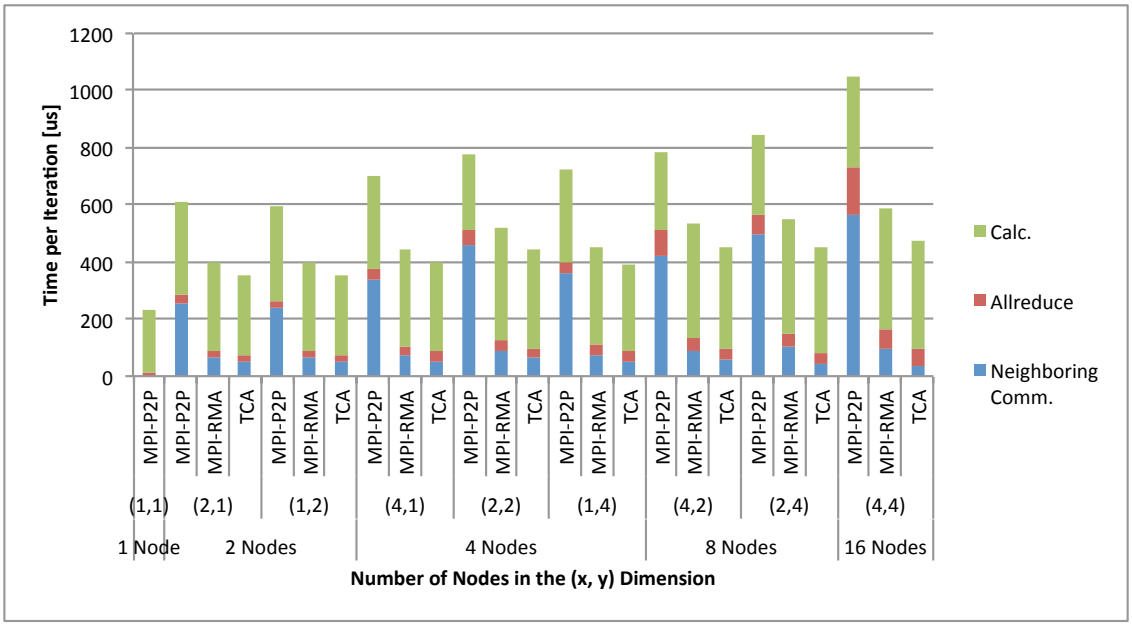

図 5.4 Small Model の計算時間と通信時間の内訳. (MV2_GPUDIRECT_LIMIT = 512KB)

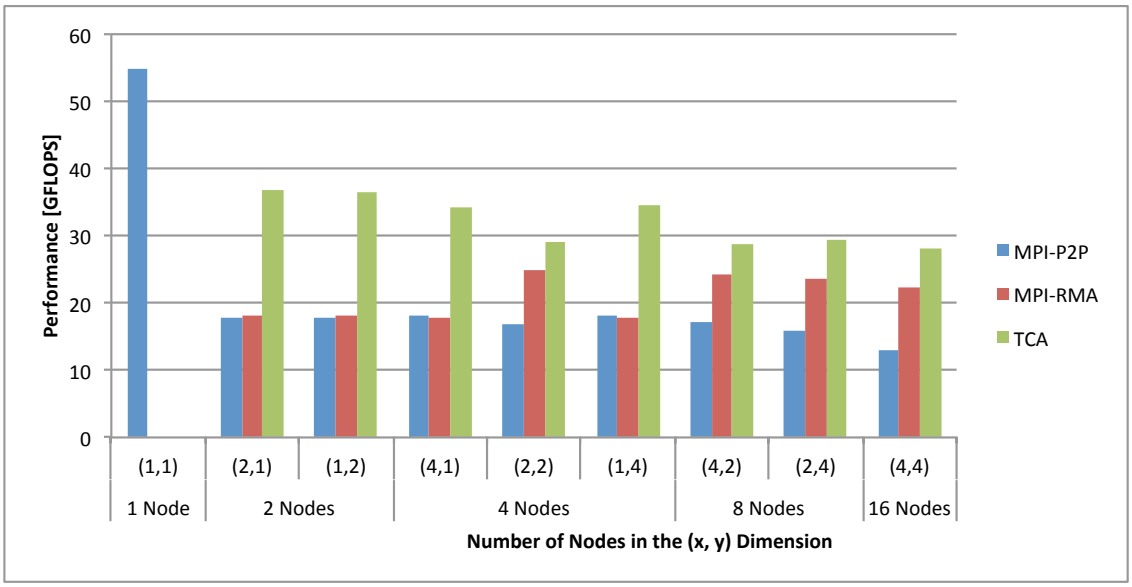

図 5.5 Small Model の FLOPS グラフ. (MV2_GPUDIRECT_LIMIT = 8KB)

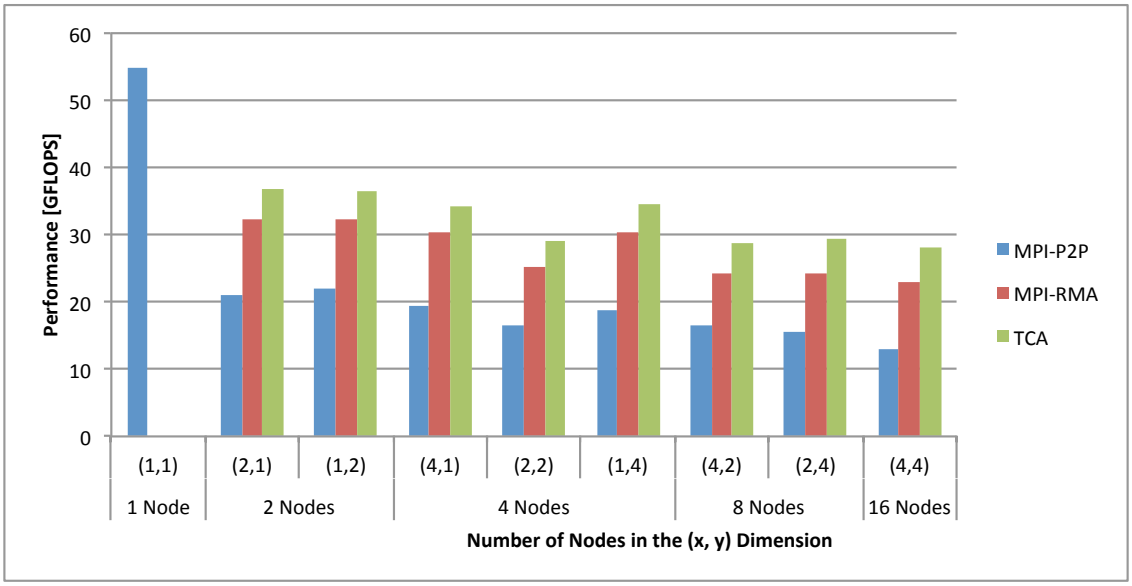

図 5.6 Small Model の FLOPS グラフ．(MV2_GPUDIRECT_LIMIT = 512KB)

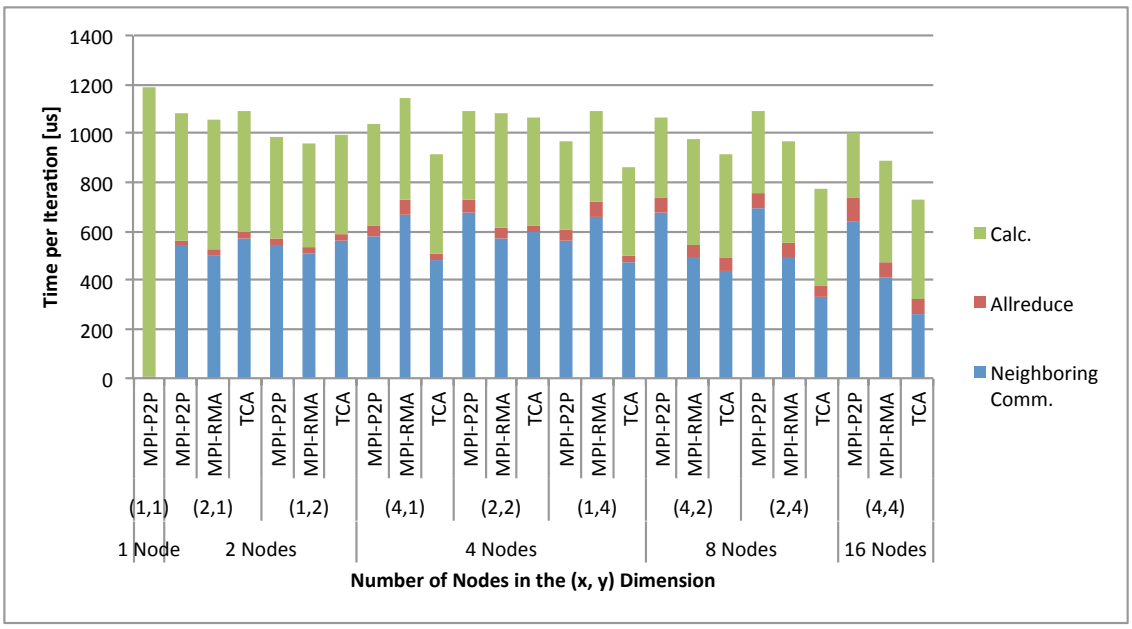

図 5.7 Large Model の計算時間と通信時間の内訳．(MV2_GPUDIRECT_LIMIT = 8KB)

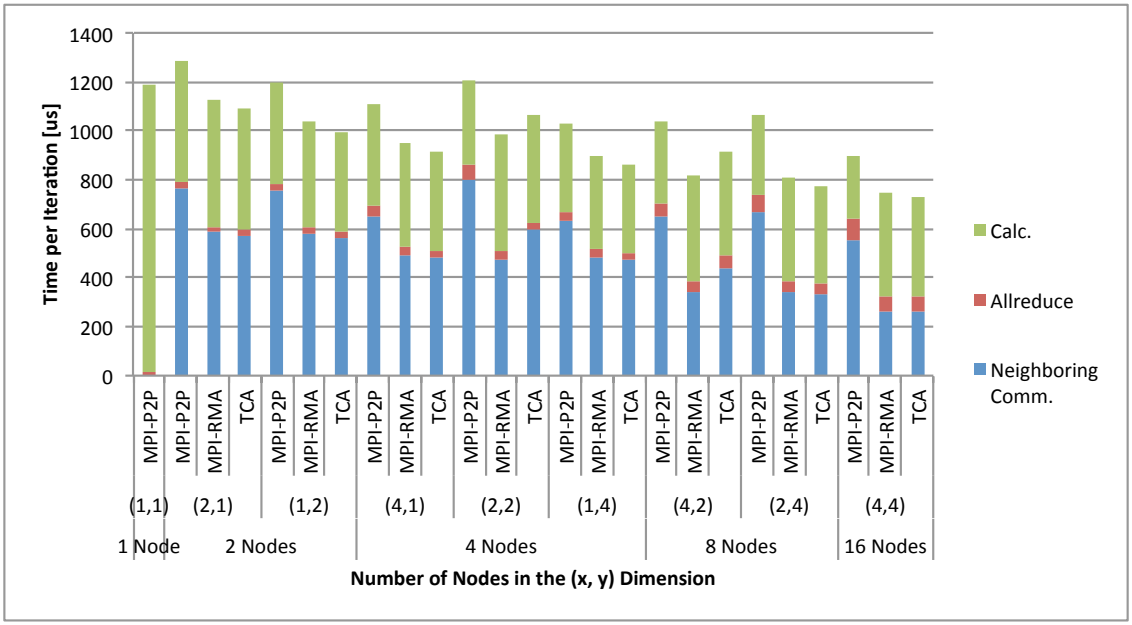

図 5.8 Large Model の計算時間と通信時間の内訳. (MV2_GPUDIRECT_LIMIT = 512KB)

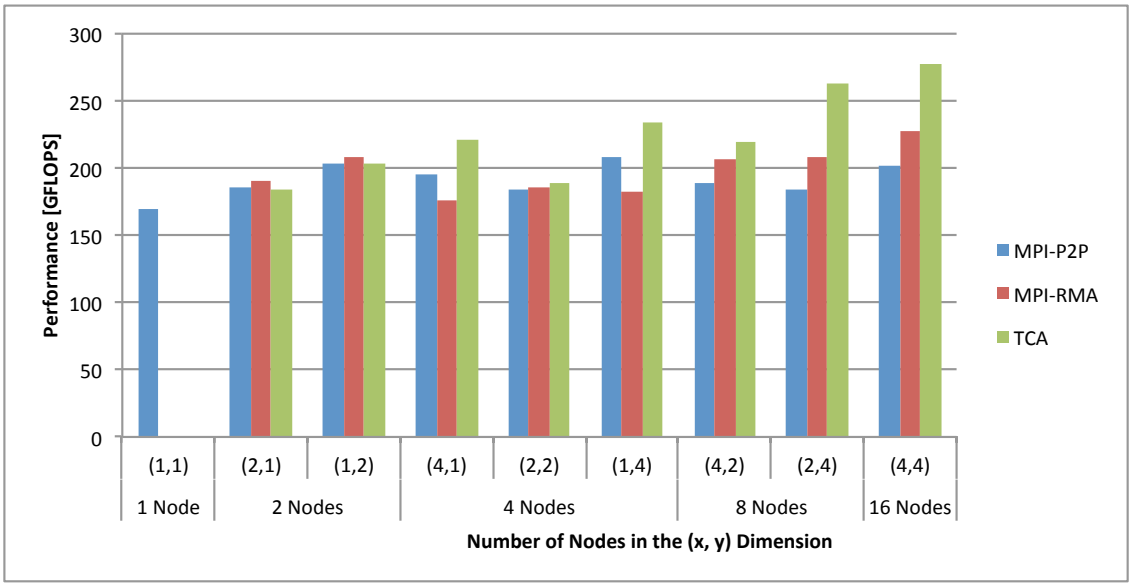

図 5.9 Large Model の FLOPS グラフ. (MV2_GPUDIRECT_LIMIT = 8KB)

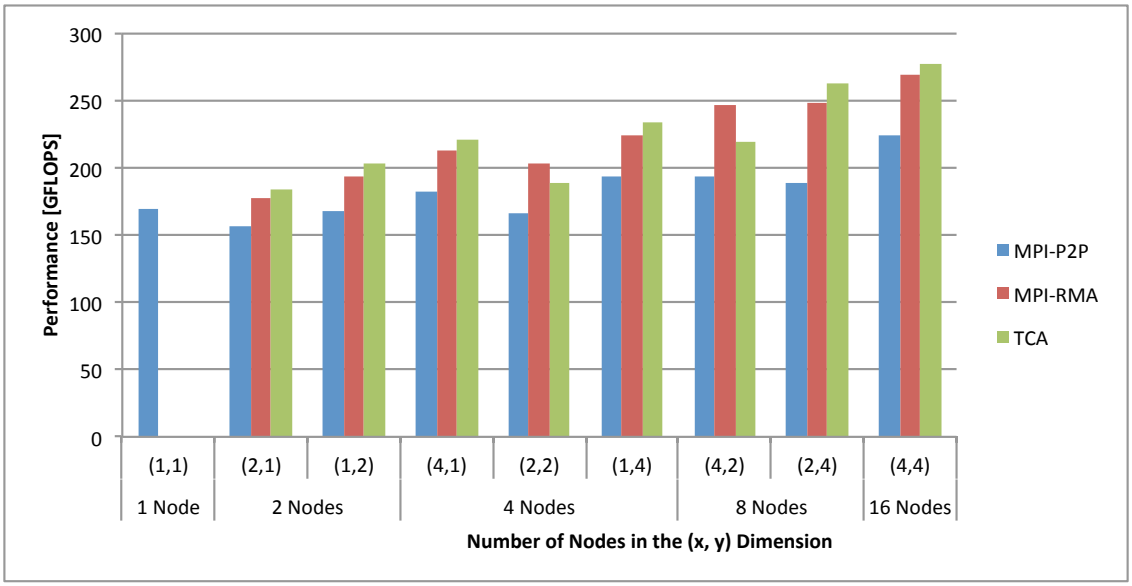

図 5.10 Large Model の FLOPS グラフ．(MV2_GPUDIRECT_LIMIT = 512KB)

も高速であり，MPI-P2P よりも MPI3-RMA の方が GDR による加速効果が大きいことがわかる．

一方で，Large Model では，TCA 実装が MPI の実装よりも性能が良い場合，遅い場合どちらのケースも存在する．2 ノード時では TCA 実装は 2 つの MPI 実装よりも遅く，8 ノード時では TCA 実装と MPI 実装は同程度の性能となり，16 ノード時は TCA 実装は MPI 実装よりも高速になる傾向がある．MV2_GPUDIRECT_LIMIT = 8KB の場合では，$(n_x,\ n_y) = (2,\ 4)$ の場合に TCA 実装は MPI point-to-point 実装よりも 1.42 倍高速であり $(n_x,\ n_y) = (1,\ 4)$ の場合に TCA 実装は倍 MPI3-RMA 実装よりも 1.27 高速である．MV2_GPUDIRECT_LIMIT = 512KB の場合では，$(n_x,\ n_y) = (2,\ 4)$ の場合に TCA 実装は MPI point-to-point 実装よりも 1.39 倍高速であり $(n_x,\ n_y) = (2,\ 4)$ の場合に TCA 実装は MPI3-RMA 実装よりも 1.06 倍高速である．Large Model の場合でも，Small Model と同様に MPI-3 RMA 実装は GDR による性能改善が大きい．

5.5 考察

まず，QUDA の RMA 通信対応による性能の変化を見るため，MPI の 2 つの実装で性能を比較する．MPI の peer-to-peer と RMA による実装で比較すると，RMA による通信は GDR 利用時および短メッセージ時の通信性能が改善するとがわかる．Small Model, MV2_GPUDIRECT_LIMIT = 8KB (図 5.5) と Small Model, MV2_GPUDIRECT_LIMIT = 512KB (図 5.6) で比較すると，2 ノードおよび 4 ノード時に MPI-3 RMA 実装の性能が改善していることがわかる．一方で，peer-to-peer 実装は性能が良くなっているものの，MPI-3 RMA 実装より改善率が小さい．MPI-3 RMA 実装の性能が良くなるのは，大きなメッセージまで GDR による通信が用いられるようになるからだと考えられる．8 ノード，16 ノード時に性能が同じなのは，MV2_GPUDIRECT_LIMIT = 8KB の設定の段階で既に GDR が用いられているからである．

同様に Large Model での性能変化について考える．Large Model, MV2_GPUDIRECT_LIMIT = 8KB (図

5.9) と Large Model, MV2_GPUDIRECT_LIMIT = 512KB (図 5.10) で比較すると，Small Model の場合と同様の傾向が見られ，MPI-3 RMA 実装は長いメッセージまで GDR を利用すると性能が改善される．一方で，peer-to-peer 実装は MV2_GPUDIRECT_LIMIT = 512KB の 2，4 ノードの場合に性能が劣化しており，peer-to-peer 通信に最適化する場合は MV2_GPUDIRECT_LIMIT = 128KB など，より小さい値が最適だと考えられる．

以上の結果から，ステンシル計算では，peer-to-peer よりも RMA 通信の方が性能が良くなり，また，GDR による GPU メモリアクセスを用いることで，小さいデータサイズの RMA 通信の性能が向上することがわかる．

PEACH2 は FPGA によって実装されており，FPGA が持つ PCIe 能力の不足のため，全ての PCIe ポートは PCIe gen.2 x8 レーンで動作している．一方で，GPU は PCIe gen.2 x16 レーンで接続されているため，gen.2 x8 レーンでは GPU の PCIe 接続に対して帯域が十分ではない．InfiniBand は PCIe gen.3 x8 レーンによって接続されており，PCIe レーン数は半分であるものの，1 レーンあたりのが帯域 gen.3 は Gen.2 と比べてほぼ倍[*1]の帯域であるため，GPU と同等の帯域でシステムと接続されている．したがって，TCA は InfiniBand よりも短いレイテンシを達成できるが，一定よりも大きなメッセージ長では帯域が低くなり，TCA 実装が MPI 実装よりも低速である結果となっていると考えられる．

また，MVAPICH2-GDR では，メッセージ長に応じて，通信の手法を切り替えている．例えば，本章でもパラメータを切り替えて性能評価を行なっているが，環境変数 MV2_GPUDIRECT_LIMIT の値によって，通信時に GDR の利用の有無を調整できる．小さいメッセージ長の通信を行う場合は，ホストメモリを経由せず，GDR を用い直接 GPU メモリへアクセスし，大きいメッセージ長の通信を行う場合は，ホストメモリを経由した通信を行う．

CPU が PCIe 経由で GPU メモリにアクセスする場合と，GDR によって GPU のメモリにアクセスする場合で性能が異なっており，ホストメモリを経由することで通信レイテンシは増加するものの，通信メッセージ長が長いためレイテンシが増加しても性能への影響は少なく，GDR のみを利用するよりも高速な結果が得られるため，MVAPICH2-GDR は以上の様な通信手法を採用している．しかしながら，TCA は全ての通信を GDR を経由して行い，MVAPICH2-GDR の様なホストメモリ利用の最適化はしておらず，長メッセージの通信では，ネットワークのピーク帯域の低さに加えて，GDR のみしか利用しない点で InfiniBand よりも性能面で不利であると考えられる．

例えば，MPI-3 RMA 実装の性能を Large Model の $(n_x,\ n_y) = (2,\ 1)$ のノード数設定で，MV2_GPUDIRECT_LIMIT パラメータの大小 (図 5.9 および 図 5.10 を参照) で比較すると，通信メッセージ長が長くなり，GDR を使わない方が性能が良いメッセージ長の範囲に入っており，MV2_GPUDIRECT_LIMIT が 8KB の設定の方が 512KB の設定の方よりも性能が良いことがわかり，TCA でも同様にホストメモリを経由する最適化を行えば性能が向上すると考えられる．ただし，ホストメモリを経由したとしても，PEACH2 の理論ピーク帯域は変らないため，長メッセージでは InfiniBand よりも性能が悪い事に変りはないが，TCA の有利なメッセージ長の範囲を広げられる．

[*1] PCIe gen.2 は 1 レーンあたり物理層では 5Gb/s の速度で信号が流れているが，その上で 10b/8b エンコーディングを用いているため，ハードウェアが実際にデータ用に使える帯域は 4Gb/s = 500MB/s である．PCIe gen.3 は物理層の速度が 8Gb/s であるものの，10b/8b エンコーディングではなく 130b/128b エンコーディングを用いている．したがって，PCIe gen.3 のデータ帯域は 7.877Gb/s = 984MB/s となり，gen.3 は gen.2 の厳密な 2 倍の帯域を持つのではない．

TCA は小さいメッセージの通信が発生する場合に有益であるといえる．Small Model のメッセージ長は最も長いケースで 24KB であり，Large Model で最も TCA で性能が改善した $(n_x,\ n_y) = (4,\ 4)$ の設定の場合，メッセージ長は x および y 次元の方向に 48KB である．しかしながら，大きいメッセージに対する TCA の帯域は InfiniBand よりも狭く [32]，Large Model の 2 ノード時の性能は十分なものではない．メッセージ長が大きくなることによる TCA の性能低下は，ホストメモリを経由する最適化や，メッセージ長と通信の RMA 書き込み先の GPU の位置によって通信チャンネル (TCA，InfiniBand 間の切り替え，もしくは両者併用) を変更するハイブリッドな手法によって解決できると考えられる．

Small Model による性能評価では，TCA は MPI peer-to-peer の実装に対して 2 倍以上の性能向上が得られている．しかしながら，1 ノードで実行した方が，MPI peer-to-peer，MPI-3 RMA，TCA の 3 つの並列化実装よりも高速であり，Small Model では並列計算による加速が得られていないため，TCA は有用であるが並列計算が有用であるとはいえない．Large Model では，ノードあたりの効率が悪いものの，TCA 16 ノード利用時に 1 ノードと比べて約 1.63 倍の高速化が得られており，Large Model は TCA も並列計算も効率が悪いものの有用であるといえる．Large Model よりもさらに問題サイズを増やすと計算量が増えるため，並列化効率は良くなると考えられるが，同時に通信メッセージ長も伸びるため TCA の優位性が失われる．一般的に強スケーリングの問題設定では，アプリケーションの分割数を増やすにつれてメッセージ長が短くなり，TCA の InfiniBand に対する性能の優位性が表れてくると考えられる．

また，QUDA の計算部分にも最適化を行える点は存在する．QUDA は MPI による通信を前提に GPU カーネルが設計されており，データのパッキングを行った上で通信を行うが，TCA を用いる場合，短いメッセージ長でも効率よく通信できるため，そのような最適化をしない方が TCA では性能が良くなると考えられる．また，現在の QUDA の計算カーネルは小さい計算に分割されており，複数のカーネルを呼び出すことで一連の計算を行なっている．GPU あたりの問題サイズが大きい場合は，そのようなカーネル分割の方法を取っても計算時間が十分に長くオーバーヘッドが性能に与える影響は少ないが，本論文で評価したような小さい問題サイズの計算では，カーネル起動のオーバーヘッドが無視できない．カーネルが細切れであることによる性能低下は QUDA 開発チームでも認識されており，計算カーネルを 1 つに融合したバージョンの開発が行われており，カーネルを融合したバージョンに対しても TCA を適用すれば，小さい問題サイズで複数ノードを利用しても性能がスケールしない問題が改善されると考えられる．

5.6 QUDA の TCA 実装における結論

本章では，オープンソースの Lattice QCD フレームワークライブラリであり，NVIDIA GPU による演算加速をサポートしている QUDA を，我々が開発している TCA ネットワークで通信が行えるように実装を行なった．TCA は PCIe のネットワークをノード間の通信に拡張するネットワーク機構であり，ノードをまたいだ GPU 間の直接通信を可能とするものである．

TCA は MPI における `MPI_Put` 関数のような，片方向の RDMA 書き込みのみしかサポートしていないため，QUDA の通信抽象部を RMA 通信に対応するように拡張を行い，その上に TCA と MPI-3 RMA 実装を行なった．

まず，QUDA の RMA 通信対応による性能の変化を見るため，MPI の 2 つの実装で性能を比較する．MPI の peer-to-peer および RMA 実装を比較すると，ステンシル計算では，peer-to-peer よりも RMA 通

信の方が性能が良いという結果が得られた．RMA の方が仕組みが単純であり，通信やメモリアクセスの最適化を行いやすいからだと考えられる．また，GDR を用いることで，peer-to-peer および RMA の両方のプロトコルで性能が向上するが，特に小さいデータサイズの RMA 通信の性能が向上することがわかった．

MPI point-to-point 実装と MPI-3 RMA 実装と TCA 実装の間での性能評価では，CG 法の反復時間が TCA を使うことで，Small Model と `MV2_GPUDIRECT_LIMIT`= 512KB の場合で，$(n_x,\, n_y) = (4,\, 4)$ の場合に TCA 実装は MPI point-to-point 実装よりも 2.19 倍の高速化が，$(n_x,\, n_y) = (4,\, 4)$ の場合に TCA 実装は MPI3-RMA 実装よりも 1.23 倍の高速化が達成された．TCA はホストメモリを経由する通信を行うことで長いメッセージへの最適化を行う余地があり，また，MPI-3 RMA および TCA における通信の同期に関してはさらなる最適化を行う余地があり，今後これらの領域を最適化することで，さらなる性能改善が得られると考えられる．

第 6 章

アクセラレータにおける通信に関するまとめ

本論文では，核融合シミュレーションコード GT5D と Lattice QCD フレームワーク QUDA の 2 つの実アプリケーションにおける通信の最適化を行い，アクセラレータを用いて計算を行うアプリケーションにおける通信の最適化について述べた．一般的に，CPU のみを用いて並列計算を行う際に発生する通信はノード間のデータ通信によるもののみであるが，アクセラレータを用いる場合はノード間通信だけでなく，CPU〜GPU 間の通信も発生する．どちらか片方の通信のみを最適化するのではなく，両方の通信を十分に最適化することが計算全体の性能を高める上で重要である．

GT5D において適用した最適化は，特殊なソフトウェアやハードウェアなどを使わない最適化であり，通信ライブラリとして MPI とアクセラレータとして NVIDIA GPU を用いる環境ならば，どのような環境でも利用できる最適化である．一方で，QUDA の通信最適化には GPU 間直接通信 (GDR) を利用しており，GDR を利用できるハードウェアおよびソフトウェアの利用を前提としており，MPI の実装として MVAPICH2-GDR を利用し，ネットワーク機構として InfiniBand を利用する．また，筑波大学で開発しているアクセラレータ間を直接かつ密に結合する TCA アーキテクチャおよび，TCA アーキテクチャの FPGA による実装である PEACH2 を用いる最適化も適用した．

GT5D の GPU 化に関する研究では，2 つの種類の通信最適化を行なった．1 つは `lfp` 関数内にある CPU〜GPU 間通信の最適化で，もう 1 つは `bcdf` 関数内にあるステンシル計算における袖領域の交換における通信である．`lfp` 関数では MPI 通信によって受け取ったデータを後の計算で利用するために，受信したデータの転置を行う必要がある．データを転置する際のメモリアクセスパターンはランダムアクセスであり，CPU で転置を実行する方が高速になると考えられるため，CPU で転置を行う実装と GPU で転置を行う実装の比較を行なった．2 つの方式を比較した結果，GPU 上で転置を行う方式の方が高速という測定結果が得られた．CPU で転置を行う方式の方が低速である理由は，CPU〜GPU 間の転送データ量がおよそ 2 倍となるためであり，GPU が不得意な処理であっても，CPU〜GPU 間のデータ通信量を削減できるのであれば，GPU 側で処理を行う方が高速であるということがわかった．

`bcdf` 関数では，ステンシル計算の袖領域の交換における通信の最適化を行なった．袖領域の交換では，CPU〜GPU 間の通信だけでなく，ノード間通信も同時に行われるため，両方の通信に対して計算とのオーバーラップを行う．通信用と計算用の 2 つの CUDA Stream を作成することで通信と計算のオー

バーラップを実現する．GT5D の計算全体の性能評価で，bcdf 関数のオーバーラップの有無を切り替えて比較するとオーバーラップありの方が高速であり，アクセラレータを用いてステンシル計算を行う際に，袖領域の交換の通信時間を隠蔽することが重要であることがわかる．

しかしながら，GT5D ではオーバーラップしている通信時間と計算時間の差がおよそ 4ms しかなく，通信隠蔽の限界に達しており，今後アクセラレータがさらに高速になった場合に対応ができない．この問題は GT5D 固有の問題ではなく，一般的な他のアプリケーションでも同様の状態になると考えられる．そこで，QUDA の通信最適化では，InfiniBand と PEACH2 の 2 つのノードをまたぐ GPU 間の直接通信が可能な通信機構を利用することで最適化を行った．また，QUDA における最適化は GPU 間直接通信を行うことだけでなく，peer-to-peer (MPI_Send など) 型の通信から，RMA (MPI_Put など) を利用する通信に変更する最適化も行う．MPI 実装における RMA 通信には MPI-3 規定されている RMA 通信を用いる．QUDA のステンシル計算における通信は通信パターンが計算を通じて固定されており，RMA 通信化が可能である．

QUDA に RMA 通信化および直接通信化を行い，MPI peer-to-peer，MPI-3 RMA，TCA の 3 つの通信手法で性能を測定し比較した．性能測定には Small Model と Large Model の 2 つの問題サイズを用いた．性能測定の結果より，2 つの RMA 実装 (MPI-3 RMA，TCA) はどちらも MPI peer-to-peer 通信よりも高速であることがわかった．MPI peer-to-peer および MPI-3 RMA はどちらも InfiniBand を利用するにもかかわらず，MPI-3 RMA の方が高速であり，したがって，GPU を用いて計算を行うステンシル計算のアプリケーションでは，peer-to-peer 通信ではなく RMA 通信を用いた方が良いといえる．RMA プロトコルを用いる方が高速な理由は，RMA 通信のプロトコルでは peer-to-peer のプロトコルにおけるバッファリングや送受信の待合せなどがなく，単純なものであるからと考えられる．

また，TCA と MPI-3 RMA による実装を比較すると，一定より長いメッセージ長で TCA は InfiniBand よりも帯域が狭いため，メッセージ長が長くなる Large Model の少ノード数実行では MPI-3 と同程度という結果となるが，Large Model の 16 ノード実行時や，メッセージ長が Large Model より短かくなる Small Model では TCA は MPI-3 よりも高速であるという結果が得られた．以上の結果より，ノードをまたぐアクセラレータ間を TCA アーキテクチャによって密に結合することで，ステンシル計算を行う GPU アプリケーションの強スケーリング性能を向上できることがわかった．

第 7 章

まとめと今後の課題

HPC の分野では，アクセラレータの利用が一般的になってきており，そういったシステムで高い実効性能を得るためには，CPU だけでなくアクセラレータもうまく利用しなければならない．一般的にアクセラレータはホスト CPU と組み合わせて利用され，CPU とアクセラレータ間は PCIe バスによって接続されている．アクセラレータのメモリは CPU と比べると広帯域のバスで接続されているものの，一方で，PCIe バスの帯域は CPU メモリやアクセラレータのメモリ帯域と比べて低く，CPU〜アクセラレータ間のデータ通信がソフトウェアのボトルネックとなりやすい．

GT5D は核融合炉におけるプラズマ乱流をシミュレーションするアプリケーションであり，本論文では GT5D の時間発展部分のフル GPU 化を行い，計算時間のおよそ 80% の計算を GPU で行えるようになった．関数単位での性能評価では `timedev1`〜`timedev9` 関数の平均では，CPU と比べ 2.66 倍高速になり，`l4dx_r`, `l4dx_s`, `l4dx_l`, `l4dx_nl` 関数では，最も性能が改善した関数は `l4dx_r` 関数のケースで，CPU と比べ 2.16 倍高速になった．`l4dx_s`, `l4dx_nl` 関数でも速度向上がみられるものの，`l4dx_l` 関数は CPU と比べて 0.77 倍高速と，GPU で実行する方が遅くなってしまった．`dn3d`， `drift_nl` 関数をでは `dn3d` 関数で 0.82 倍，`drift_nl` 関数で 0.79 倍と，どちらの関数も GPU の方が遅いという結果になった．`bcdf` 関数の計算と通信のオーバーラップでは，MPI 通信と計算のオーバーラップだけでなく，CPU〜GPU 間の通信と計算のオーバーラップも実装を行なった．通信隠蔽による性能向上は大きく，関数一回あたり 21ms の性能改善が達成できた．時間発展部全体の性能比較では，GPU 版のコードの方がオーバーラップなしの場合で 1.17 倍，オーバーラップありで 1.91 倍の高速化が得られ，また，オーバーラップのありとなしで比較すると，オーバーラップありの方が 1.63 倍高速化が達成された．

GT5D の GPU 化より，GPU アプリケーションにおいては，各カーネル関数の最適化による性能も重要であるが，ノードをまたぐ GPU 間通信を行う際は，ノード間通信に加えて CPU〜GPU 間通信も必要なため，通信にかかるレイテンシが CPU アプリケーションよりも大きく，GPU アプリケーションでは通信隠蔽が重要であるとわかった．

しかしながら，GT5D では，`bcdf` 関数における計算時間と通信時間の差が 3.7ms とほぼ差がなく，既に通信隠蔽の限界に達しつつある．今後，アクセラレータの性能が向上し計算時間が短縮されたとしても，PCIe バスの性能がボトルネックとなり，通信隠蔽が十分にできないため，より細粒度のオーバーラップや，アクセラレータのメモリに対して直接アクセスしデータ転送を行うなど，通信レイテンシを削減するための工夫が必要である．また，問題サイズを変化させずにノード数を増やす強スケーリングの場合で

は，一般的に，ノード数を増やすにつれてメッセージサイズが小さくなり，オーバーヘッドによる通信影響への影響が大きくなり，GPU アプリケーションでの強スケーリングの達成は困難を伴う．

アクセラレータを持つクラスタ環境での通信レイテンシを改善するために，筑波大学計算科学研究センターでは TCA アーキテクチャを提唱している．TCA アーキテクチャでは，アクセラレータは同一ノード内だけでなく異なるノードにあるアクセラレータとも直接通信を行えるものとしており，アクセラレータ間の通信レイテンシの削減および，強スケーリング時の性能改善が得られる．また，TCA アーキテクチャの実装の 1 つととして PEACH2 が開発されている．PEACH2 は Altera 社の FPGA を用いて実装されており，NVIDIA GPU のメモリに対して PCIe バスを通じてアクセスし，データを転送できる．

実アプリケーションでの TCA の性能評価を行うために，GPU による演算加速に対応している Lattice QCD フレームワークである QUDA に対して TCA を適用した．Small Model と `MV2_GPUDIRECT_LIMIT`= 512KB の場合で，$(n_x,\ n_y) = (4,\ 4)$ の場合に TCA 実装は MPI point-to-point 実装よりも 2.19 倍の高速化が，$(n_x,\ n_y) = (4,\ 4)$ の場合に TCA 実装は MPI3-RMA 実装よりも 1.23 倍の高速化が達成された．

性能測定の結果，小さい問題サイズにおいては TCA によって強スケーリング性能が改善されることを確認したものの，そのような問題設定サイズでは，1 ノードで実行した方が性能が良く，並列計算による性能向上が得られない．QUDA では GPU における実効性能が良い問題サイズの範囲と，TCA によって性能改善が見込まれる問題サイズの範囲がかさなる部分が小さいことが判明した．QUDA では，GPU 上で性能を得るために問題サイズを大きくすると，TCA による通信レイテンシの削減効果が弱まり，一方で，TCA で性能が良くなるような小さい問題サイズを設定すると，GPU へ計算をオフロードするオーバーヘッドが大きくなり，十分な性能が得られない．

MPI peer-to-peer と MPI-3 RMA による 2 つの実装の比較では，MPI-3 RMA の方が高速であり，また，環境変数 `MV2_GPUDIRECT_LIMIT` を大きくした場合の性能向上が MPI-3 RMA 実装の方が大きいことがわかった．したがって，GPU を用いるステンシル計算では RMA 通信を用いる方が良く，環境変数 `MV2_GPUDIRECT_LIMIT` は HA-PACS/TCA ではデフォルト値 (8KB) よりも 512KB の方が性能が良いことがわかった．

QUDA は MPI による通信を前提に GPU カーネルが設計されており，データのパッキングを行った上で通信を行うが，PEACH2 では短いメッセージ長でも効率よく通信できるため，そのような最適化をしない方が PEACH2 では性能が良くなると考えられる．また，現在の QUDA の計算カーネルは小さい計算に分割されており，複数のカーネルを呼び出すことで一連の計算を行なっている．GPU あたりの問題サイズが大きい場合は，そのような計算手法を取っても計算時間が十分に長くオーバーヘッドが性能に与える影響は少ないが，本論文で評価したような小さい問題サイズの計算では，カーネル起動のオーバーヘッドが無視できない．カーネルが細切れであることによる性能低下は QUDA 開発チームでも認識されており，計算カーネルを 1 つに融合したバージョンの開発が行われており，カーネルを融合したバージョンに対しても TCA を適用すれば，小さい問題サイズで複数ノードを利用しても性能がスケールしない問題が改善されると考えられる．また，GDR による GPU メモリアクセスは，通信レイテンシは短いが，大きいデータを読み出す際に時間がかかることがわかっており，MVAPICH2-GDR ではメッセージサイズに応じてホストを経由した通信に切り替えている．同様の仕組みを PEACH2 で実装すれば，長メッセージにおいて性能が改善すると考えられる．

謝辞

本研究を行うにあたり，日頃から厳しくも温かいご指導を頂いた筑波大学 システム情報工学研究科 教授・朴泰祐先生に心より感謝申し上げます．また，お忙しい中，本論文の副査を引き受けていただいた同教授・高橋大介先生, 同教授・櫻井鉄也先生，同教授・安永守利先生，東京大学 情報基盤センター 准教授・塙敏博先生に深謝いたします．

日頃よりお世話になりました筑波大学 システム情報工学研究科 教授・佐藤三久先生，同教授・建部修見先生，同准教授・山口佳樹先生，同助教・多田野寛人先生，同講師・川島英之先生，理化学研究所・児玉祐悦博士に感謝の意を表します．

GT5D のオリジナルの開発者であり，本研究の遂行にあたり数々の助言を頂きました日本原子力研究開発機構・井戸村泰宏博士，GT5D 関連，特にプラズマ物理の分野について助言を頂きました同機構・奴賀秀男博士に感謝の意を表します．

GT5D の GPU 化に関する研究の一部は JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」，および日本学術振興会・多国間国際研究協力事業（G8 Research Councils Initiative）プログラム研究課題「エクサスケール規模の核融合シミュレーション」によるものです．

QUDA のオリジナルの開発者であり，本研究の遂行にあたり数々の助言を頂きました NVIDIA Corporation・Mike Clark 博士，同・Davide Rossetti 氏，同・Dale Southard 氏を始めとする NVIDIA 社，および NVIDIA JAPAN 諸氏に感謝の意を表します．

QUDA の TCA 化に関する研究を行うにあたり，TCA の利用方法やプログラミング手法についてアドバイスを頂き，また，HA-PACS/TCA 上での性能測定の一部を手伝って頂きました HPCS 研究室の後輩である富士通ソフトウェアテクノロジーズ・藤井久史氏に感謝致します．

日頃より御議論いただいている筑波大学計算科学研究センター次世代計算システム開発室および先端計算科学推進室の各メンバに感謝致します．GPU アプリケーションの性能測定に際しましては，筑波大学計算科学研究センターに設置されている HA-PACS ベースクラスタおよび HA-PACS/TCA を利用させて頂きました．両システムの運営に関わっておられるクレイジャパン社および計算科学研究センターの方々に感謝致します．QUDA の TCA 化に関する研究の一部は，JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」によるものです．

最後に，日頃より研究生活における様々な面においてお世話になりましたネットチームの皆様，HPCS 研究室の皆様にこの場を借りて感謝を述べさせて頂きます．

参考文献

[1] Top500 Supercomputer Sites. http://top500.org/.

[2] Intel® Xeon Phi™ Product Family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[3] Tesla K20X GPU Accelerator Board Specification. http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf.

[4] Whtepaper NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™. http://www.nvidia.com/attach/3183943.html?type=support&primitive=0.

[5] Whitepaper NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[6] CUDA Toolkit. http://www.iter.org.

[7] MVAPICH2. http://mvapich.cse.ohio-state.edu/overview/mvapich2/.

[8] Mellanox Technologies. https://www.mellanox.com/.

[9] Mellanox Products: Mellanox OFED GPUDirect RDMA Beta. http://www.mellanox.com/page/products_dyn?product_family=116.

[10] Paulius Micikevicius. 3D Finite Difference Computation on GPUs Using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pp. 79–84, New York, NY, USA, 2009. ACM.

[11] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, Nov 2011.

[12] TSUBAME 計算サービス — TSUBAME 計算サービス. http://tsubame.gsic.titech.ac.jp.

[13] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, Nov 2010.

[14] Max Rietmann, Peter Messmer, Tarje Nissen-Meyer, Daniel Peter, Piero Basini, Dimitri Komatitsch,

Olaf Schenk, Jeroen Tromp, Lapo Boschi, and Domenico Giardini. Forward and Adjoint Simulations of Seismic Wave Propagation on Emerging Large-scale GPU Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 38:1–38:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[15] Introducing Titan — The World's #1 Open Science Supercomputer. https://www.olcf.ornl.gov/titan/.

[16] Cray XK Series Supercomputers — Cray. http://www.cray.com/products/computing/xk-series.

[17] OpenMPI. http://www.open-mpi.org/.

[18] R Ammendola, A Biagioni, O Frezza, F Lo Cicero, A Lonardo, P S Paolucci, D Rossetti, A Salamon, G Salina, F Simula, L Tosoratto, and P Vicini. APEnet+: high bandwidth 3D torus direct network for petaflops scale commodity clusters. *Journal of Physics: Conference Series*, Vol. 331, No. 5, p. 052029, 2011.

[19] R Ammendola, A Biagioni, O Frezza, F Lo Cicero, A Lonardo, P S Paolucci, D Rossetti, F Simula, L Tosoratto, and P Vicini. APEnet+: a 3D Torus network optimized for GPU-based HPC Systems. *Journal of Physics: Conference Series*, Vol. 396, No. 4, p. 042059, 2012.

[20] Damian Alvarez Mallon, Norbert Eicker, Maria Elena Innocenti, Giovanni Lapenta, Thomas Lippert, and Estela Suarez. On the Scalability of the Clusters-booster Concept: A Critical Assessment of the DEEP Architecture. In *Proceedings of the Future HPC Systems: The Challenges of Power-Constrained Performance*, FutureHPC '12, pp. 3:1–3:10, New York, NY, USA, 2012. ACM.

[21] DEEP Project - Homepage. http://www.deep-project.eu/deep-project/EN/Home/home_node.html.

[22] Y.Idomura, M.Ida, T.Kano, N.Aiba, and S.Tokuda. Conservative global gyrokinetic toroidal full-f five-dimensional Vlasov simulation. *Computer Physics Communications*, Vol. 179, pp. 391–403, 2008.

[23] ITER. http://www.iter.org.

[24] Xiaolin Zhong. Additive semi-implicit rungekutta methods for computing high-speed nonequilibrium reactive flows. *Journal of Computational Physics*, Vol. 128, No. 1, pp. 19 – 31, 1996.

[25] S.Jolliet and Y.Idomura. Simulating Plasma Turbulence with the Global Eulerian Gyrokinetic Code GT5D. *Progress in NUCLEAR SCIENCE and TECHNOLOGY*, Vol. 2, pp. 85–89, 2011.

[26] PGI: CUDA Fortran. http://www.pgroup.com/resources/cudafortran.htm.

[27] L.Victor W., K.Changkyu, C.Jatin, D.Michael, K.Daehyun, N.Anthony D., S.Nadathur, S.Mikhail, C.Srinivas, H.Per, S.Ronak, and D.Pradeep. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, Vol. 38, No. 3, pp. 451–460, June 2010.

[28] HA-PACS Base Cluster — Center for Computational Science, University of Tsukuba. http://www.ccs.tsukuba.ac.jp/CCS/eng/research-activities/projects/ha-pacs/base-cluster.

[29] Intel® Math Kernel Library (Intel MKL) 11.0. http://software.intel.com/en-us/

intel-mkl.

[30] CUFFT. https://developer.nvidia.com/cufft.

[31] CUBLAS. https://developer.nvidia.com/cublas.

[32] T.Hanawa, Y.Kodama, T.Boku, and M.Sato. Interconnect for Tightly Coupled Accelerators Architecture. *IEEE 21st Annual Symposium on High-Performance Interconnects (HOT Interconnects 21)*, pp. 79–82, 2013.

[33] T. Hanawa, Y. Kodama, T. Boku, and M. Sato. Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators. In *Processings of IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pp. 1030–1039, May 2013.

[34] 塙敏博, 児玉祐悦, 朴泰祐, 佐藤三久. Tightly Coupled Accelerators アーキテクチャに基づく GPU クラスタの構築と性能予備評価. 情報処理学会論文誌（コンピューティングシステム）, Vol. 6, No. 4, pp. 14–25, October 2013.

[35] Yuetsu Kodama, Toshihiro Hanawa, Taisuke Boku, and Mitsuhisa Sato. PEACH2: An FPGA-based PCIe Network Device for Tightly Coupled Accelerators. *SIGARCH Comput. Archit. News*, Vol. 42, No. 4, pp. 3–8, December 2014.

[36] PGI-SIG. PCI Express External Cabling Specification, Rev. 1.0. 2007.

[37] 計算機設備紹介 — 筑波大学 計算科学研究センター. http://www.ccs.tsukuba.ac.jp/research/computer.

[38] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling Lattice QCD Beyond 100 GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 70:1–70:11, New York, NY, USA, 2011. ACM.

[39] Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/.

[40] Lattice QCD Message Passing (QMP). http://usqcd.jlab.org/usqcd-docs/qmp/.

[41] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Comput. Phys. Commun. 181*, pp. 1517–1528, 2010.

[42] R. Babich, M.A. Clark, and B. Joo. Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, Nov 2010.

[43] QUDA - A Library for QCD on GPUs. http://lattice.github.io/quda/.

付録 A

公表論文リスト

査読付き雑誌論文

1. 藤田 典久, 藤井 久史, 塙 敏博, 児玉 祐悦, 朴 泰祐, 藏増 嘉伸, Mike Clark, “GPU 向け QCD ライブラリ QUDA への TCA アーキテクチャの適用”, ACS 論文誌, 情報処理学会, 第 50 号, pp.25-35, 2015 年 6 月.

査読付き国際会議論文

1. Norihisa Fujita, Husafuji Fujii, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, Yoshinobu Kuramashi, M Clark, “QCD Library for GPU Cluster with Proprietary Interconnect for GPU Direct Communication”, Proc. of International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar) 2014, pp.251-262, Porto, Portugal, August 2014.
2. Norihisa Fujita, Hideo Nuga, Taisuke Boku, Yasuhiro Idomura, “Nuclear Fusion Simulation Code Optimization and Performance Evaluation on GPU Cluster”, Proc. of International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) 2014, pp.1266-1274, Phoenix, USA, May 2014.

査読付き国際会議ポスター論文

1. Norihisa Fujita, Hideo Nuga, Taisuke Boku, Yasuhiro Idomura, “Nuclear Fusion Simulation Code Optimization on GPU Clusters”, Proc. of International Conference on Parallel and Distributed Systems (ICPADS) 2013, pp.420-421, Seoul, Korea, December 2013.

査読付き国内学会ポスター論文

1. 藤田 典久, 奴賀 秀男, 朴 泰祐, 井戸村 泰宏, “GPU クラスタにおける核融合シミュレーションコードの実装”, 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 (HPCS2013), p.82, 東京, 2013 年 1 月.

査読なし論文 (研究会)

1. 藤田 典久, 奴賀 秀男, 朴 泰祐, 井戸村 泰宏, “GPU クラスタ HA-PACS における核融合シミュレーションコードの性能評価”, 2013-HPC-140, online 7 pages, 福岡, 2013 年 7 月.
2. 藤田 典久, 奴賀 秀男, 朴 泰祐, 井戸村 泰宏, “GPU クラスタにおける核融合シミュレーションコードの実装”, 2013-HPC-138, online 8 pages, 福井, 2013 年 2 月.
3. 藤田 典久, 奴賀 秀男, 朴 泰祐, 井戸村 泰宏, “核融合シミュレーションコードの GPU クラスタ向け最適化”, 2012-HPC-135, online 6 pages, 鳥取, 2012 年 7 月.

ポスター発表

1. Norihisa Fujita, Testuya Odajima, “Application and System Software on Tightly Coupled Accelerators Architecture”, Asian Technology Information Program (ATIP) 2014, November 2014.
2. Norihisa Fujita, Hideo Nuga, Taisuke Boku, Yasuhiro Idomura, “Performance Evaluation on Nuclear Fusion Simulation Code on HA-PACS”, AICS Symposium 2013, 神戸, December 2013.