Vol.2014-HPC-145 No.43
2014/7/30

GPU向けQCDライブラリQUDAのTCAアーキテクチャ実装の性能評価

藤井 久史[1]　藤田 典久[1]　塙 敏博[2]　児玉 祐悦[1,3]　朴 泰祐[1,3]　佐藤 三久[1,3]　藏増 嘉伸[3]
Mike Clark[4]

概要：近年，HPC 分野で GPU などの演算加速装置を用いたクラスタの開発が盛んに行われている．このようなクラスタでは，ノード間をまたぐ演算加速装置間の通信を CPU に接続されたネットワークインターフェースを介して行う必要があるため，複数回のメモリコピー等によるオーバヘッドが発生してしまう．このためレイテンシが増加し，アプリケーションの性能を低下させてしまう．この問題に対する解決として，我々は GPU 間通信のレイテンシの改善を目的とした独自開発の密結合並列演算加速機構 TCA (Tightly Coupled Accelerators) の開発を行なっている．2013 年 10 月には TCA 実証システムである HA-PACS/TCA クラスタが筑波大学計算科学研究センターに導入された．本稿では，素粒子物理学のための GPU 向け格子量子色力学 (格子 QCD) ライブラリである “QUDA” に対し，TCA を適用した実装の性能評価を行う．

1. はじめに

近年，HPC の分野で GPU (Graphics Processing Unit) が持つ高い浮動小数点演算能力とメモリバント幅を利用した GPGPU (General Purpose GPU) が注目されている．GPU を搭載したノードから構成された GPU クラスタも盛んに開発されている [1]

GPU クラスタでは複数ノード上にまたがった GPU 間で通信を行う場合，従来の方法では CPU のメモリを介した複数回のメモリコピーを行わなければならない．これによりレイテンシが増加し，比較的サイズの小さなデータの通信では大きなボトルネックとなっていた．近年では，GPU とネットワークインターフェイスの直接通信を実現する GPUDirect support for RDMA[2][3] を用いることにより，CPU のメモリへのコピーを無くすことができるようになっている．しかし，依然としてネットワークを介する必要があるためノードをまたぐ GPU 間通信はレイテンシが大きい．

我々は GPU 間通信のレイテンシの改善を目的とした独自開発の密結合並列演算加速機構 TCA (Tightly Coupled Accelerators) の開発を行なっている．本稿では，GPU を用いた格子 QCD 計算のためのライブラリである QUDA に TCA を適用した実装とその性能評価について述べる．

[1] 筑波大学 大学院 システム情報工学研究科
Graduate School of System and Information Engineering,University of Tsukuba
[2] 東京大学 情報基盤センター
Information Technology Center, The University of Tokyo
[3] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[4] NVIDIA Corporation

2. QUDA

QUDA は NVIDIA 社の Mike Clark らによって開発されている GPU を用いた格子 QCD 計算のためのライブラリである [4][5]．QUDA にはおおまかに分けて，Dirac oparator によるステンシル計算と Krylov ソルバによる線形方程式の求解の 2 つの計算がある．Krylov ソルバとしては CG 法や BiCGStab 法などの複数のアルゴリズムが実装されており，行列のタイプに合わせて選択することができる．

マルチノード/マルチ GPU にも対応しており，計算に用いるノード数，GPU 数を増やすことによって計算速度を向上させることができる [6]．通信方法としては MPI[7] と Lattice QCD Message Passing (QMP) [8] を選択することができる．

3. TCA アーキテクチャと PEACH2

TCA は筑波大学計算科学研究センターを中心として研究開発が行われている通信機構 [9][10][11][12] で，PCI Express [13]（以下，PCIe）を元とした技術である．PCIe は PC と拡張デバイスを接続するためのシリアルインター

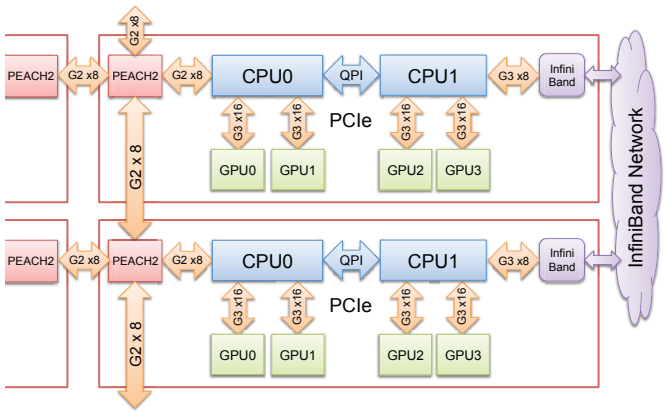

図 1 HA-PACS/TCA のノードの構成

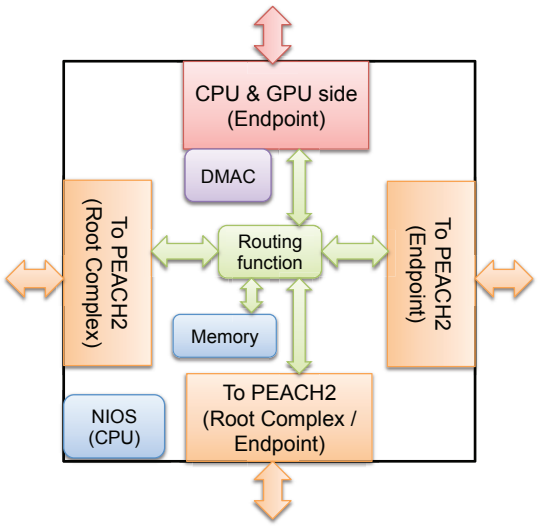

図 2 PEACH2 ボードの写真 (左：パネル面，右：基盤面)

フェース規格であり，GPU や Ethernet，InfiniBand のようなネットワークインターフェースなどの多くのデバイスで用いられる．TCA では，ノード間を PCIe によって接続することにより，ノードをまたぐ GPU 間の PCIe による直接通信を可能とする．

PEACH2 ボードは我々が独自開発したマルチノード GPU 間で低レイテンシを実現するインターコネクトのためのインターフェースボードである．TCA 及び PEACH2 の性能を実計算によって評価するため，筑波大学計算科学研究センターに GPU クラスタ HA-PACS/TCA[10] が設置された．HA-PACS/TCA の各計算ノードには PEACH2 ボードが搭載されており，この PEACH2 ボード間を PCIe ケーブルでつなぐことによって，TCA によるクラスタを構築する．PCIe の基本的な機能は CPU 側にあたる Root-Complex(RC) とデバイスにあたる EndPoint(EP) 間のメモリのリードライト操作である．しかし，実際には双方向でパケットのやり取りをしているに過ぎない．そこで，PEACH2 では PCIe パケットを独自にルーティングすることにより、ノードを跨ぐ PCIe デバイス間の直接通信を可能にした．PEACH2 チップ及びボードについては文献 [9][10][11][12] に詳しいが，ここでは本稿の内容に関係する基本的概要のみを記す．

図 1 に，HA-PACS/TCA のノード構成を示す．HA-PACS/TCA のノードは PCIe Gen3 を 40 レーン持つ CPU (Intel Xeon E5-2680v2) を 2 個搭載している．1 個の CPU ソケットに 2 個の GPU を 32 レーン (1GPU あたり 16 レーン) 用いて接続する．残りのレーンを用いて，一方の CPU ソケットに PEACH2 を，もう一方の CPU ソケットに InfiniBand HCA を接続する．この構成では CPU が RC であり，その他のデバイスが EP である．この場合でも，すべてのデバイスは同じ PCIe アドレス空間を持つため，GPU と PEACH2 間で PCIe プロトコルによる直接通信が可能である．しかし実際には，QPI を超える PCIe デバイス間アクセスは，CPU に内蔵された PCIe スイッチ性能がボトルネックとなり大きく性能が低下することがわかっている．そのため，TCA における通信では，PEACH2 と異なる CPU に接続された GPU を用いることは想定しない．

3.1 PEACH2 チップ

図 2 に，PEACH2 チップの構成を示す．PEACH2 チップは FPGA 上に実装されており，PCIe パケットの中継処理，高度な DMA 転送などをハードワイヤード処理で行う．FPGA を用いることにより，回路を書き換えることができるので，機能の改善や，柔軟な機能の追加が可能となる．FPGA には，PCIe Gen2 x8 のハード IP を 4 ポート内蔵した Altera 社 Stratix IV GX を用いている．

4 ポートある PCIe のハード IP は便宜上それぞれ，N(orth)，E(est)，W(est)，S(outh) ポートと呼ぶ．N ポートはホストとの接続に用い，それ以外のポートは隣接ノードの PEACH2 ボード間の接続に用いる．PCIe は本来ホストとデバイスをつなぐ規格であるため，必ず接続する二点間が RC と EP の対になる必要がある．N ポートはホストと接続するので EP になる．E ポートは EP，W ポートは RC とし，隣接ノード間のこれらを接続することによってリングトポロジを構成する．S ポートは RC と EP を選択できるようにして，S ポート同士を接続する．

3.2 DMA コントローラ

PEACH2 は DMA コントローラ（以下，DMAC）を 4 チャネルを持っている．この DMAC の特徴的な機能として chaining DMA 機能がある．chaining DMA では，あらかじめ送信元と送信先を記述したディスクリプタを複数記述してポインタで連結し，メモリ上に保存しておく．DMA を開始する際には，ディスクリプタの先頭アドレスをセットするだけで，連結された複数のディスクリプタの通信が連続的に行われる．これにより，通信の開始にかかるオーバーヘッドを大幅に短縮することができる．ディスクリプ

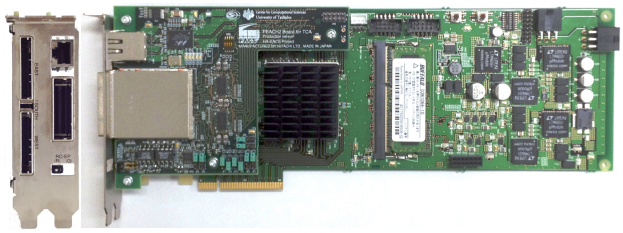

図 3 PEACH2 ボードの写真 (左：パネル面，右：基盤面)

タはホストメモリ上と PEACH2 内蔵メモリ上に作成することができる．ホストメモリ上にディスクリプタを設定する場合，PEACH2 から PCIe を通してホストメモリを読むオーバーヘッドがある．PEACH2 内蔵メモリに設定する場合は，PEACH2 ボード上でアクセスが完結するためホストメモリ上に設定する場合よりもオーバーヘッドが小さい．ただし，PEACH2 内蔵メモリには容量上の都合により 1024 個までしかディスクリプタを設定できないため，これより多くディスクリプタを使用する場合はホストメモリ上にディスクリプタを作成する必要がある．また，各チャネルごとに 16 個まで，PEACH2 上のレジスタにディスクリプタを登録することもできる (レジスタモードと呼ぶ)．レジスタモードでは，PEACH2 内蔵メモリに設定する場合と同様にホストメモリを読む必要が無いため，オーバーヘッドが小さい．

その他に，一定間隔で離れたブロックを転送するブロックストライド転送を指定することもできる．

3.3 GPUDirect Support for RDMA

PEACH2 から GPU への直接通信を行うのには，GPUDirect Support for RDMA 機能 [2] [3] を用いる．この機能は CUDA 5.0 以降および Kepler 世代以降の GPU[14] から使用可能であり，これにより，GPU 上のメモリを PCIe アドレス空間にマッピングすることができる．同じ PCIe 空間に属する他のデバイスは，このマップされたアドレスにアクセスすることによって，CPU へのコピーなしに PCIe プロトコルのみで直接読み書きを行うことができる．

3.4 PEACH2 ボード

図 3 に PEACH2 ボードを示す．このボードは，PCIe 規格に定められたボード仕様を満たしており [15]，ホストと接続するための PCIe Gen2 x8 のエッジコネクタと，左側面に PCIe ケーブルポートが 3 個 (E，W，S ポート) 配置されている．中央部には Altera 社の FPGA Stratix IV GX，DDR3 SO-DIMM 1 枚を搭載する．電源は，右上の PCIe ペリフェラル用コネクタのみから給電され，ボードの右部分には FPGA が使用する各電源電圧を生成するレギュレータ類がある．PEACH2 チップは，PCIe Gen2 IP の動作周波数に合わせ，主要な機能は 250MHz で動作する．

3.5 TCA におけるプログラミング環境

TCA におけるプログラミング環境は，NVIDIA から提供される CUDA 開発環境を基本とする．現在 TCA を使用するための API は PEACH2 のハードウェアを利用するための基本機能と，それら基本機能を用いた allgather や reduce などの集団通信関数 [16] などが用意されている．

TCA は各種制御及び通信対象となるデータの名前空間として，PCIe アドレス空間を用いる．しかし，プログラム中で PCIe アドレスを直接扱うと煩雑なため，TCA API によるプログラミングでは，`tcaHandle` を作成し，そのハンドルに PCIe アドレスと TCA におけるノード番号を格納する．また，GPU メモリを TCA の通信対象とする場合は，GPUDirect Support for RDMA によって GPU メモリを PCIe アドレス空間にマップし，得られた PCIe アドレスを `tcaHandle` に格納する．

TCA がハードウェアによって提供する通信は，送信側ノードのメモリの内容を，受信側ノードのメモリに書き込む RDMA(Remote Direct Memory Accesss) による片方向通信である．片方向通信では，通信を開始する時点で送信側ノードが受信側ノードへの書き込みアドレスを知っておく必要ある．そのため，TCA API による通信では，ノード間で TCA ハンドルの交換を転送前に行っておくことが必要となる．

TCA API では現在のところ，プロセスの起動や TCA ハンドルを交換する機能がないため，それらの不足した機能については MPI と連携する必要がある．

TCA API において，chaining DMA による通信は以下のような流れで行う．

(1) `cudaMalloc` 関数や `tcaMalloc` 関数を用いて，メモリを確保する．CPU メモリに関しては，`tcaMalloc` 関数によって得られたものしか TCA では扱えない．
(2) `tcaCreateHandle` 関数によって `tcaHandle` を作成する．
(3) 受信側ノードと送信側ノードで `tcaHandle` の交換を行う．
(4) 送信側ノードは `tcaCreateDMADesc` 関数を用いてディスクリプタチェインを作成する．
(5) `tcaSetDMADesc_Memcpy` 関数を用いて，ディスクリプタに送信バッファのアドレスと受信バッファのアドレス，サイズなどを設定する．
(6) `tcaSetDMAChain` 関数を用いて，DMAC のチャネルに対してディスクリプタチェインの先頭を設定する．
(7) `tcaStartDMADesc` 関数を用いて，通信を開始する．
(8) TCA の API は受信完了を通知する仕組みを備えているため，受信側ノードは `tcaWaitDMARecvDesc` 関数を用いて受信が完了するのを待つことができる．

4. QUDA の TCA による実装

本稿では，QUDA のステンシル計算に含まれる通信を TCA による通信に置き換えることを検討する．QUDA では基本的に Send と Receive による 1 対 1 通信と，Allreduce などの集団通信を用いている．これらの通信関数はすべて抽象化されており，comm から始まる関数名が付けられており，実装として MPI と QMP を選択することができる．

文献 [17] では，TCA API によって 1 対 1 通信機能を実装し，QUDA の抽象化された通信関数を置き換えることによって QUDA の通信の TCA 化を行った [17]．しかし，TCA のハードウェアでは RDMA による書き込みのみに対応しているため，これを用いて 1 対 1 通信を実現するためには送信側ノードへのアドレス通知が必要なためレイテンシが増加する問題や，TCA の特徴である Chaining DMA 機能を用いた複数の通信の連続処理を活用することができない問題などがあった．

そこで本稿では，QUDA の抽象化された 1 対 1 通信関数の置き換えによって TCA 化をするのではなく，QUDA の通信コードそのものを TCA を使用するのに適した RMA (Remote Memory Access) に書き換えることを検討した．そのために新たに RMA 通信を行う抽象関数を定義し，MPI-3 の RMA 関数を用いた実装と TCA API を用いた実装を作成し，これらの抽象関数を用いて QUDA の通信部分の書き換え，QUDA の通信の TCA 化を行った．今回定義した RMA 通信を行う抽象関数を “QUDA RMA インターフェイス”と呼ぶ．

QUDA RMA インターフェイスの実装はまず，MPI-3 の RMA 関数を用いたバージョンを作成し，QUDA RMA インターフェイスを用いた QUDA で正しい解を得られることを確認してから，TCA API を用いたバージョンを作成した．

この節では，QUDA RMA インターフェイスについて述べる．MPI-3 の RMA を用いた実装については 5.1 で，TCA API を用いた実装については 5.2 で述べる

4.1 RmaWindow

`window_alloc` 関数は，RMA によってデータが書き込まれるメモリ領域を確保し，`RmaWindow` オブジェクトを返す．`RmaWindow` オブジェクトは，RMA 通信の対象となるメモリ領域についての情報を保持している．QUDA RMA インターフェイスを通して行われる RMA 通信では，送信アドレスと受信アドレスの両方の指定を，`RmaWindow` とオフセットを用いて行うため，これらのアドレスは `RmaWindow` の領域内に含まれている必要がある．`RmaWindow` は，全ノードが同じ内容のオブジェクトを持つ必要があるため，`window_alloc` 関数は集団通信関数のように全ノードが同時に行う必要がある．

4.2 MsgHandle

`MsgHandle` オブジェクトは元々 QUDA の 1 対 1 通信の通信パターンを保持するためのハンドルとして定義されていた．MPI による 1 対 1 通信の実装では，`MPI_{Send|Recv}_init` 関数を用いた persistent communication を行っている．persistent communication とは，通信前に通信に関する情報を登録しておき，実際の通信の際には登録された情報を元に通信を行う方法である．登録された情報は何度も使いまわすことができるため，同じ通信パターンを繰り返すアプリケーションにおいては，通信ライブラリ実装による最適化が期待できる．`MsgHandle` は `MPI_{Send|Recv}_init` 関数が返す `MPI_Request` を保持している．`MPI_Request` には送信や受信のためのポインタや，Datatype，データサイズなどの情報が含まれている．実際に通信する際には `MPI_Request` を `MPI_Start` 関数に渡すことより開始することができる．

QUDA RMA インターフェイスでは，`declare_write` 関数を用いて `MsgHandle` を作成する．`declare_write` 関数は引数として，書き込み先 `RmaWindow` とそのオフセット，書き込み元 `RmaWindow` とそのオフセット，そして書き込み先のランクを指定する．作成した `MsgHandle` は RMA 通信に必要な情報を保持しており，実際に通信を行う際は次に述べる `RmaQueue` を通して行う．

4.3 RmaQueue

作成した `MsgHandle` による通信の開始や待機は，`RmaQueue` オブジェクトを通して行う．以下に `RmaQueue` を操作する関数について述べる．

Alloc
: `RmaQueue` を作成する．この時，RMA 通信によってリモートからデータが書き込まれる `RmaWindow` を関連付ける．関連付けられた `RmaWindow` は同期操作などで用いられる．

Free
: 作成した `RmaQueue` の開放を行う．

Start
: `RmaQueue` に登録さている `MsgHandle` に記述された RMA 通信を開始する．

Wait
: Start によって開始された RMA 通信の完了およびリモートノードからの書き込みの完了を待つ．

Push
: 作成した `MsgHandle` を `RmaQueue` に登録する．

Commit
: Push による `MsgHandle` の登録を終了し，`RmaQueue` に登録された RMA 通信を開始できる状態にする．

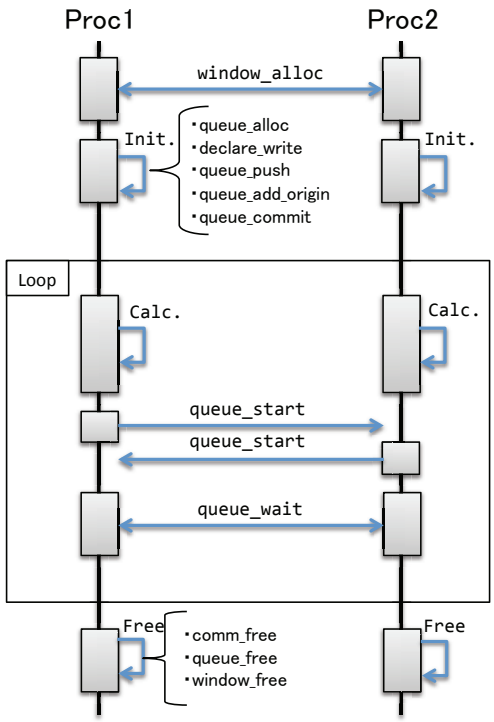

図 4 QUDA RMA インターフェイスによる通信を行うコードの流れ

Add Origin

RmaQueue に関連付けられた RmaWindow に書き込みを行うリモートノードのランク (以下，*Origin* ランクと呼ぶ) を登録する．

Clear

RmaQueue に登録された操作をすべて消去する．

Push 関数と *Commit* 関数では通信の準備などを行うため，オーバーヘッドが発生する可能性がある．特に TCA API を用いた実装の場合，PEACH2 の通信のための DMA ディスクリプタの設定は数 μ 秒のオーバーヘッドがあるが，これらの処理は *Commit* 関数で行われる．QUDA では，同じ通信パターンを繰り返して行うため，初回の通信の時に RmaQueue を作成し *Push* と *Commit* を行えば，それ以降の通信では *Start* と *Wait* を行えばよく，通信の準備のオーバーヘッドを削減できる．*Add Origin* 関数によって登録された *Origin* ランクは *Wait* 関数でリモートノードからの書き込み完了を待機するために用いる．

4.4 QUDA RMA インターフェイスによる通信の流れ

QUDA RMA インターフェイスによる通信を行うコードの流れを図 4 に示す．図 4 では，Proc1 と Proc2 で相互にデータを送り合っている．

通信を行うには，まず RMA によってリモートからデータが書き込み先領域を作成するために全ノードで window_alloc 関数を呼び出す．また，書き込み元領域の作成も同様に window_alloc 関数を用いる．

次に通信を行うための RmaQueue を queue_alloc によって作成する．declare_write 関数によって必要な MsgHandle を作成し，RmaQueue に queue_push 関数によって登録する．また，Origin ランクを queue_add_origin 関数によって RmaQueue に登録する．全ての通信に必要な情報を RmaQueue に登録した後は，queue_commit を呼び出して通信準備を完了させる．同じ通信パターンを繰り返し用いる場合はこれらの処理はプログラム中に 1 度呼び出すのみでよい．

図 4 のプログラムの流れでは計算ループがあることを想定している．このループでは計算の次に通信を行うが，この通信パターンが毎回同じ場合，queue_start 関数による通信の開始と，queue_wait 関数による通信完了の待機のみで通信に関する処理が済むため，通信の準備のオーバーヘッドを削減することができる．

オブジェクトが不要になった場合は，comm_free 関数によって MsgHandle の開放を，queue_free 関数によって RmaQueue の開放を，window_free 関数によって RmaWindow の開放をそれぞれ行う必要がある．

4.5 QUDA RMA インターフェイスによる 1 対 1 通信の置き換え

作成した QUDA RMA インターフェイスによって QUDA の 1 対 1 通信を置き換えた．ただし，PEACH2 は Read 操作に直接は対応していない．TCA を用いて Read 操作を行うには Proxy Write を用いる必要があり，オーバーヘッドが大きい．そのため，Send 操作を Write 操作によって置き換えた．また，RMA の Write 操作は書き込みを行う側のノードのみが行い，書き込まれる側のノードは同期操作のみで良いので，Receive 操作はすべて削除した．

5. QUDA RMA インターフェイスの実装

この節では，QUDA RMA インターフェイスの MPI-3 の RMA 関数を用いた実装と，TCA API を用いた実装について述べる．

5.1 MPI の片方向通信による実装

QUDA の MPI 1 対 1 通信の実装では，MPI_{Send|Recv}_init 関数を用いた persistent communication を行うため，MsgHandle はこの MPI 関数が返す MPI_Request を保持するのみで良かったが，MPI-3 の RMA には，persistent communication と同等の機能がない．そのため，MPI-3 による実装では MsgHandle は書き込み元の RmaWindow とオフセット，書き込み先の RmaWindow とオフセット，その他にも通信サイズや Datatype など通信に必要な情報をすべて保持している．*Start* 関数では，RmaQueue に登録されている MsgHandle に記憶されている通信の情報を元に，MPI_Put 関数によっ

てリモートノードへデータを書き込んでいく．MPI-3 による実装では，*Commit* 関数で特に行うことはない．

window_alloc 関数では MPI_Win_create 関数を用いて MPI_Win を取得し，RmaWindow ではこれを保持している．MPI_Win は MPI-3 の RMA によって操作できる領域を表したハンドルである．MPI_Win は全ノードで共通したオブジェクトを用いるので，MPI_Win_create 関数は全ノードで同時に呼ばれる必要がある．MPI_Put 関数ではリモートへの書き込み先指定は MPI_Win とデータへのオフセットで行う必要がある．ローカルにある書き込み元については通常のアドレスで指定する．

Wait 関数は，MPI_Win_start，MPI_Win_post，MPI_Win_complete，MPI_Win_wait の 4 つの MPI RMA のための同期関数を用いて実装した．MPI_Win_start 関数を呼び出してから，MPI_Win_complete 関数を呼び出すまでの区間を “access RMA epoch” とよび，リモートへデータを書き込む側は access RMA epoch の中で MPI_Put 関数によるリモートへの書き込みを行う．また，MPI_Win_post 関数を呼び出してから，MPI_Win_wait 関数を呼び出すまでの区間を “exposure RMA epoch” と呼ぶ．exposure RMA epoch が終了する MPI_Win_wait 関数から返ってくるときには access RMA epoch で発行された書き込みが完了していることが保証されている．access RMA epoch では，どのノードへ書き込みを行うのかの情報が必要になので，*Push* 時にどのノードへ書き込むかの情報を MsgHandle から取り出し，RmaQueue に記憶しておく．exposure RMA epoch では，どのノードから書き込まれるかの情報が必要であるが，この情報に関しては *Add Origin* 関数によって登録されている *Origin* ランクを使用する．

5.2 TCA による実装

TCA の実装においても *Push* された MsgHandle の内容を覚えておく必要がある．*Push* された MsgHandle は *Commit* 時にディスクリプタの設定が行われ全て chaining によって連結される．*Start* 時には DMAC にディスクリプタテーブルの先頭を指定し，tcaDMADescStart 関数によって DMA を開始する．ディスクリプタの内容の設定は *Commit* 時に完了しているため，ディスクリプタ書き込みのオーバーヘッドを削減することができる．また，chaining を用いて，各方向への通信をすべてつなげることができ，通信の開始時には先頭のディスクリプタの開始のみですべての方向への通信を開始できる．本稿の実装では必要なディスクリプタの個数は 1024 個以下に収まっているため，ディスクリプタは PEACH2 内蔵メモリにディスクリプタを作成した．

window_alloc 関数では，tcaCreateHandle 関数によって GPU メモリの tcaHandle の取得を行う．しかし，tcaCreateHandle 関数で得られる tcaHandle は作成した時点ではそのノードでしか使うことができないので，MPI_Allgater 関数によって全ノードに配布する．これにより，RmaWindow を通して全てのノードへ tcaHandle を用いた RDMA による書き込みが行えるようになる．

TCA において，リモートからの書き込みを待つには tcaDMAWaitDesc 関数を用いることができるが，この関数は特定のノードからの書き込みを待つ関数であるため，事前にどのノードから書き込みがあるかを知っている必要がある．QUDA RMA インターフェイスでは，どのノードから書き込まれるかという情報は *Add Origin* 関数によって *Origin* ランクとして登録されているため，*Wait* では，登録されている *Origin* ランクの全てから書き込み完了通知を受け取るまで待機をすることにより実装した．

6. 評価と考察

6.1 評価環境

TCA を用いた QUDA 実装の評価のため，TCA 実証クラスタ HA-PACS/TCA 上で性能測定を行った．HA-PACS/TCA のノード構成を表 1 に示す．HA-PACS/TCA では，PEACH2 によって隣接ノード間を接続すると同時に，2 系統からなる InfiniBand QDR 4x によってもノード間が接続されている．よって，PEACH2 による実装と MPI/InfiniBand による実装を公平に比較することができる．MPI 実装には MVAPICH2-GDR 2.0b を用いている．

ノードのブロック図は先に示した図 1 のように 2 つの CPU を搭載しており，CPU0 に PEACH2 ボードが接続され，CPU1 に Infiniband HCA が接続されている．しかし，Intel Xeon E5 プロセッサ（SandyBridge アーキテクチャ）及びその改良版の E5-v2 プロセッサ（IvyBridge アーキテクチャ）では，PCIe デバイス同士が QPI を通して通信を行うと大幅にバンド幅が低下することが知られており，そのため，TCA による実装の測定には PEACH2 と同じく CPU0 に接続されている GPU0 もしくは GPU1 を，Infiniband を用いた MPI による実装の測定には Infiniband HCA と同じく CPU1 に接続されている GPU2 もしくは GPU3 を使用する．

6.2 性能測定と考察

性能測定には QUDA で提供されている invert_test プログラムを用いた．invert_test では CG 法によって線形方程式を解き，反復回数と GFLOPS によって性能を出力する．

X，Y，Z，T はそれぞれ x，y，z，t の各次元の各プロセスあたりの格子点数を表し，n_X，n_Y，n_Z，n_T は各次元の分割プロセス数を表す．また，N_X，N_Y，N_Z，N_T はそれぞれ全体の格子点数を表し，$(N_X, N_Y, N_Z, N_T) = (X \times n_X, Y \times n_Y, Z \times n_Z, T \times n_T)$ となる．測定は，$(N_X, N_Y, N_Z, N_T) = (8, 8, 8, 8)$ と $(N_X, N_Y, N_Z, N_T) =$

表 1 HA-PACS/TCA ノード構成

ハードウェア	
CPU	Xeon-E5 2680 2.8GHz ×2
Memory	DDR3 1866 MHx × 4ch，128 Gbytes
Motherboard	SuperMicro X9DRG-QF
GPU	NVIDIA K20x ×1
InfiniBand	Mellanox Connect-X3 Dual-port QDR
PEACH2 ボード	
FPGA	Altera Stratix IV GX 530 1932pin
PEACH2 論理	version 20130222
ソフトウェア	
OS	Linux，CentOS 6.4
GPU ドライバ	NVIDIA-Linux-x86_64-331.75
GPU プログラム環境	CUDA 6.0
MPI	MVAPICH2-GDR 2.0b

$(16, 16, 16, 16)$ の 2 つの問題サイズについて行った．前者を Small Model，後者を Large Modle と呼ぶ．それぞれの問題サイズに対してノード数を変えた強スケーリングについて測定を行った．現在，HA-PACS/TCA では 16 ノードが PEACH2 によって接続されているため，最大 16 ノードを用いて測定を行った．図 5 は Small Model の測定結果を，図 6 では Large Model の測定結果を示している．図中の “MPI-P2P” は従来の MPI による 1 対 1 通信を用いた実装の性能を，“MPI-RMA” は QUDA の通信を QUDA RMA インターフェイスによって書き換え，通信方法として MPI-3 の RMA を用いた実装の性能を，“TCA” は QUDA の通信を QUDA RMA インターフェイスによって書き換え，通信方法として TCA を用いた実装の性能をそれぞれ表す．

CG 法の反復回数は格子点数や分割数によって変化するため，図 5 と図 6 では 1 反復あたりの平均時間を示している．また MPI-P2P では通信時間を載せていないが，これは MPI-P2P では非同期 1 対 1 通信を行っており正確に通信時間を測ることができないためである．

図 5 の Small Model の結果を見ると，どの分割数においても TCA の結果が MPI-P2P と MPI-RMA の結果より二倍以上高速であった．これは，x 次元方向の通信サイズが $\frac{12KB}{n_Y}$，y 次元方向の通信サイズが $\frac{12KB}{n_X}$ と比較的小さく，低レイテンシである TCA を用いるほうが高速に通信できるためである．$(n_X, n_Y) = (4, 2)$ の時には，TCA が MPI-RMA より 2.3 倍，高速である事がわかる．

図 6 の Large Model の結果を見ると，Small Model と比較して性能の差が小さく，ノード数 $(n_X, n_Y) = (2, 1), (1, 2), (4, 1), (2, 2)$ では TCA の方が高速であるが，それ以外では MPI-RMA の方が良い結果となっている．Large Model では，x 次元方向の通信サイズが $\frac{96KB}{n_Y}$，y 次元方向の通信サイズが $\frac{96KB}{n_X}$ と，Small Model と比べて大きくなっている．そのため，TCA を用いた場合と MPI を用いた場合で通信バンド幅の差がほぼ無くなり，性能の差が現れなくなっている．

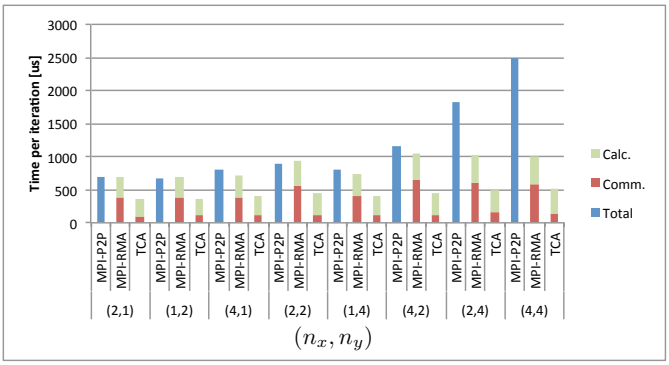

図 5 Smal Model の性能

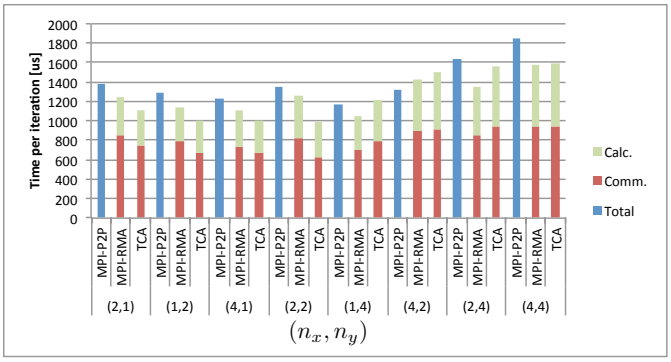

図 6 Large Model の性能

しかし，Small Model，Large Model ともに，ノード数を増やすほど性能が悪くなってしまっている．この原因の一つとして，GPU で計算するには問題サイズが小さすぎるというが考えられる．また，QUDA では通信前に不連続な袖領域を通信に適した連続領域にコピーする packing 処理がある．ノード数が増えると袖領域の数が増え，packing 処理を行う回数も増えるため計算時間が増加していると考えられる．packing 処理に関しては，TCA API の提供するブロックストライド通信機能を用いることにより回避できる可能性があるため，今後これを用いた実装を検討していく予定である．

このように，図 5 と図 6 を比較すると，通信サイズが小さい場合に低レイテンシである TCA が有用であることがわかる．一般的に強スケーリングではノード数が増えるに連れて通信サイズも小さくなるため，TCA による低レイテンシ通信が有効的だと考えられる．

しかし，結果的には Small Model では問題自体の小ささにより，また Large Model では通信時間の影響の相対的な小ささにより，TCA による性能向上は十分示されていない．今後，より多彩な問題サイズへの適用，問題サイズに応じたより詳細な最適化や通信隠蔽手法の改良等，高性能化のための工夫が必要である．

7. 関連研究

APEnet+[18] は，FPGA による独自の 3D トーラスネットワークを開発している．APEnet+においても TCA と同様に GPU とネットワークインターフェース間の直接通信を実現しているが，ネットワーク自体は QSFP+ ケーブルを用いた独自のプロトコルを使用する．

コモディティなネットワークである InfiniBand においても，GPUDirect support fot RDMA を使用して GPU と直接通信ができる [19]．PEACH2 においても GPUDirect Support for RDMA を用いるが，InfiniBand ではネットワークには PCIe と異なるプロトコルを用いるため，ノード間の通信においても PCIe のパケットをそのまま転送できる PEACH2 のほうが，プロトコルの変換なしに通信できるため有利であると考えられる．MPI 実装である MVAPICH2 では，上記の InfiniBand の機能を用いて CPU と GPU 間のコピーを行わずに，ノードをまたぐ GPU 間通信を行うことができる [20] が，TCA では MPI を用いる必要がないのでプロトコルスタックのオーバーヘッドを削減できる．

8. おわりに

本稿では，GPU 向け QCD ライブラリである QUDA の通信部分に対して TCA を適用し，性能評価を行った．QUDA の通信の TCA 化にはまず，QUDA 向けの RMA 抽象通信インターフェイスを定義した後，MPI-3 の RMA 通信を用いた実装を作成してインターフェイスを用いて正しい解が得られることを確認した後，TCA による実装を行った．

その結果，通信サイズが小さくなる Small Model において，TCA を用いた実装が MPI を用いた実装より高速であることが確認できた．特に Small Model において $(n_X, n_Y) = (4, 2)$ の時に，TCA による通信を行う実装が MPI-3 の RMA による通信を行う実装より 2.3 倍，高速することができた．一方，Large Model においては，通信サイズが比較的大きくなるため，TCA を用いた実装と MPI を用いた実装で大きな差は確認できなかった．今後は，QUDA において PEACH2 のブロックストライド転送機能を有効活用する方法について検討を進めていく予定である．

謝辞

本研究に際し御協力, 御助言をいただいた Davide Rossetti 氏, Dale Southard 氏を始めとする NVIDIA 社，および NVIDIA JAPAN 諸氏に深く感謝する．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する．本研究の一部は 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．

参考文献

[1] Top500 Supercomputer Sites. http://top500.org/.
[2] NVIDIA Corp.: NVIDIA GPUDirect. http://developer.nvidia.com/gpudirect.
[3] NVIDIA Corp.: Developing A Linux Kernel Module Using RDMA For GPUDirect. http://developer.download.nvidia.com/compute /cuda/5_0/rc/docs/GPUDirect RDMA.pdf.
[4] 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, pages 15171528, (2010).
[5] Mike Clark.: QUDA - A Library for QCD on GPUs. http://lattice.github.io/quda .
[6] R. Babich, M. A. Clark, B. Joo, G. Shi, R. C. Brower, and S. Gottlieb. Scaling lattice QCD beyond 100 GPUs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC), (2011).
[7] Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/.
[8] Lattice QCD Message Passing (QMP). http://usqcd.jlab.org/usqcd-docs/qmp/.
[9] 塙 敏博, 児玉 祐悦, 朴 泰祐, 佐藤 三久: Tightly Coupled Accelerators アーキテクチャのための通信機構, 情報処理学会研究報告 (アーキテクチャ), Vol. 2012-ARC-201, No. 26, pp. 18 (2012).
[10] 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，2013 年先進的計算基盤システムシンポジウム (SACSIS2013) 論文集，pp. 150–157 (2013).
[11] Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: Interconnect for Tightly Coupled Accelerators Architecture. IEEE 21st Annual Sympsium on High-Performance Interconnects (HOT Interconnects 21), pp. 79-82 (2013).
[12] Kodama, Y.,Hanawa, T., Boku, T. and Sato, M.: PEACH2: FPGA based PCIe network device for Tightly Coupled Accelerators. Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2014).
[13] PCI-SIG: PCI Express Base Specification, Rev. 3.0 (2010).
[14] NVIDIA Corp.: NVIDIA Tesla Kepler GPU Computing Accelerators. http://www.nvidia.co.jp/content/tesla/pdf/Tesla-KSeries-Overview -LR.pdf.
[15] PCI-SIG: PCI Express External Cabling Spec- ification, Rev. 1.0 (2007).
[16] 松本 和也, 塙 敏博, 児玉 祐悦, 藤井 久史, 朴 泰祐. 密結合並列演算加速機構 TCA を用いた GPU 間直接通信による CG 法の実装と予備評価. 情報処理学会研究報告, Vol.2014-HPC-144, (2014).
[17] 藤井 久史, 塙 敏博, 児玉 祐悦, 朴 泰祐, 佐藤 三久, 藏増 嘉伸,Mike Clark. GPU 向け QCD ライブラリ QUDA の TCA アーキテクチャによる実装. 情報処理学会研究報告, Vol.2014-HPC-143, (2014).
[18] Rosetti, D. et al.: Leveraging NVIDIA GPUDirect on APEnet+ 3D Torus Cluster Interconnect (2012). http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0282-GTC2012-GPU-Torus-Cluster.pdf.

[19] Mellanox Technologies: Mellanox OFED GPUDirect. http://www.mellanox.com/content/pages.php?pg=products dyn&product family=116&menu section=34.
[20] Dhabaleswar K (DK) Panda:MVAPICH2: A High Performance MPI Library for NVIDIA GPU Clusters with InfiniBand (2013). http://on-demand.gputechconf.com/gtc/2013/presentations/S3316-MVAPICH2-High-Performance-MPI-Library.pdf