Vol.2014-HPC-143 No.35
2014/3/4

GPU向けQCDライブラリQUDAのTCAアーキテクチャによる実装

藤井 久史[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 メモリを介して行う必要があるため，複数回のメモリコピーが発生してしまう．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．そこで，筑波大学計算科学研究センターでは，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 を適用した実装について述べる．

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．

しかし，その高い演算性能に比べて GPU メモリと CPU メモリの間の通信性能には大きなギャップがあるため，アプリケーションのボトルネックになることが多い．さらに，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る．

そこで，筑波大学計算科学研究センターでは，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．

[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. PEACH2 チップと PEACH2 ボード

PEACH2 ボードは，筑波大学計算科学研究センターで開発された，マルチノード GPU 間で低レイテンシを実現するインタコネクトのためのインタフェースボードである．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 [1], [2], [3].

TCA ではノード間の通信に PCI Express[4] (以下, PCIe) を直接用いる．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フェースなどの多くのデバイスで用いられる．PCIe の基本的な機能は CPU 側にあたる RootComplex(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、ノード間の通信を可能にした．

従来の GPU クラスタにおいては，ノード間をまたぐ GPU 間で通信を行うには，少なくとも 3 回のデータコピーを行う必要があった．例えば，ノード A 上の GPU A からノード B 上の GPU B に通信を行うには，以下のコピーが必要となる．

(1) GPU A のメモリから，PCIe 経由でノード A のメモ

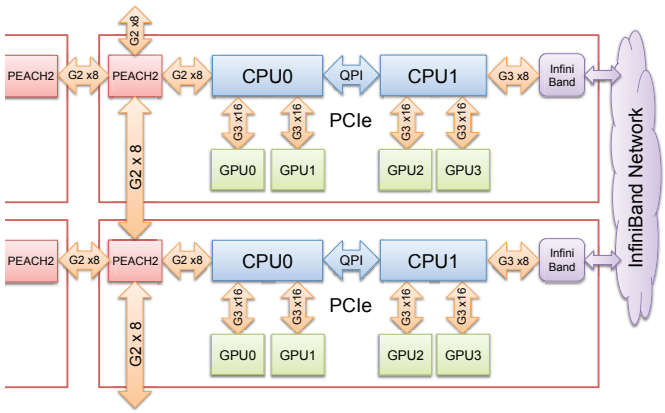

図 1 TCA ノードの構成

リにコピー

(2) ノード A のメモリから，ネットワーク経由でノード B のメモリにコピー

(3) ノード B のメモリから，PCIe 経由で GPU B のメモリにコピー

TCA では，GPU A からのデータを，PCIe のパケットの状態を維持したまま，GPU B へ転送することにより、GPU 間で直接通信することが可能になる．

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 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 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．

2.1 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．

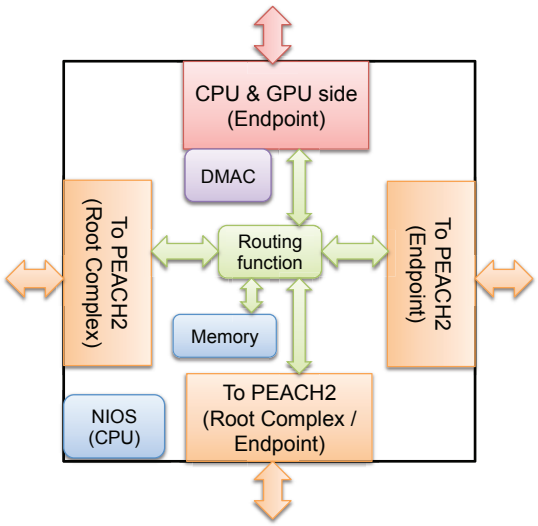

図 2 PEACH2 ボードの写真 (左：パネル面，右：基盤面)

2.2 DMA コントローラ

PEACH2 は DMA コントローラを 4 チャネルもっている．この DMA コントローラ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，ホスト上のメモリに保存しておく．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隠蔽することができる．しかし，chaining DMA では，ホストのメモリにあるディスクリプタを読むオーバーヘッドが存在する．このオーバーヘッドを避けるために，各チャネルごと 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)．また，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．

2.3 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 [6] [7] を用いる．この機能は CUDA 5.0 以降および Kepler 世代以降の GPU 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

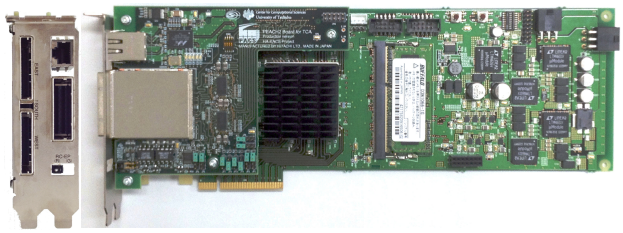

図 3 PEACH2 ボードの写真 (左：パネル面，右：基盤面)

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．

2.4 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 [5]，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 ケーブルポートが個 (E，W，S ポート) が配置されている．E，W ポートは x8 の，S ポートは x16 のポートを使用しているが，S ポートに関しても信号は 8 レーンのみ使用する．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．

2.5 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 などを開発中である．

TCA はアドレスとして，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，TCA ハンドルを作成し，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 Suppoer 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 TCA ハンドルに格納する．

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 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 TCA ハンドルを作成する．
(3) 受信側ノードと送信側ノードで TCA ハンドルの交換を行う．
(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．

3. QUDA の TCA による実装

本節では，GPU 向け QCD ライブラリである QUDA についてと，QUDA 内で行われるノードをまたぐ GPU メモリ間通信を TCA による通信におきかえる実装について述べる．

3.1 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 [9]．QUDA を用いることで，疎行列における連立一次方程式の解を求める計算など格子 QCD で用いられる計算を容易に GPU で行うことができる．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．

3.2 QUDA におけるプロセス間通信

QUDA では，通信に関する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いる (以下，これらの関数を comm 関数と呼ぶ)．comm 関数の内部は MPI などの通信 API を用いて実装されている．

comm 関数は大きく分けて，

(1) プロセスのトポロジーの作成および情報を取得する関数
(2) プロセス間のメッセージパッシング通信を準備する関数 (以下 comm_declare 系関数)
(3) (2) によって準備した通信を開始，および待機する関数 (comm_start() 関数, comm_wait() 関数)
(4) Allreduce などの集団通信関数．

の 4 つに分類できる．このように，QUDA は基本的にメッ

セージパッシング通信 (1 対 1 通信) を行っている．

comm 関数で行われる処理の詳細を知るために，MPI による comm 関数の実装を調べた．すると，(2) の comm_declare 系関数では送信の準備として MPI_Send_init() 関数が，受信の準備として MPI_Recv_init() 関数が使用されており，persistent communication を用いていることがわかった．comm_start() 関数を呼ぶことにより，comm_declare 系の関数における通信設定に基づいて非同期な 1 対 1 通信が開始される．従って，comm_wait() 関数で blocking を行うか，comm_query() 関数を用いて test を行う必要がある．

3.3 片方向通信による非同期 1 対 1 通信の実装

前節で述べたとおり，QUDA は非同期 1 対 1 通信を行っているが，RDMA による片方向通信による書き込みのみをハードウェアで実装している TCA で，そのまま QUDA の通信部分を置き換えることは難しい．1 対 1 通信においては Send 要求の際に送信バッファを指定し，Recv 要求において受信バッファをそれぞれ明示的に指定するため，お互いは相手のバッファのアドレスを必要としないが，片方向通信による書き込みでは，通信を行う時点で送信側ノードが送信バッファのアドレスと受信バッファのアドレスの両方を知っておく必要がある．また，送信側ノードが，受信側ノードへデータを書き込む操作を行うのみで完結するため，基本的に，受信側ノードはデータが届いたかどうかを知ることができない．片方向通信によって，1 対 1 通信を実現するには，送信側ノードに受信バッファアドレスを通知することと，受信バッファに通信の完了を通知することが必要となる．

そこで，本研究では，片方向通信による書き込みのみを用いて，非同期 1 対 1 通信を実装した．受信バッファアドレスの通知と通信完了の通知には，通知のためのバッファを送信側ノードと受信側ノードにそれぞれ設けて，受信側ノードは受信バッファアドレス通知を，送信側ノードは通信完了通知をそのバッファに書き込むことによって通知し，受け取る側はバッファをポーリングすることによって，データが届いたことを確認するようにした．また，非同期通信を実現するために，専用のスレッドを立ち上げて，通知の確認はそのスレッドでポーリングをするようにした．図 4 に送受信の流れを示す．図中の左側が送信側ノード，右側が受信側ノードのプログラムとデータの流れである．

送信側ノードと受信側ノードでそれぞれ発行された送信リクエストと受信リクエストは，まずキュー (Send Queue, Recv Queue) に入れられ，リクエストハンドルを得る．リクエストの発行はすぐに終了し，リクエストハンドルを用いて送信と受信の完了を待つ．キューに入ったリクエストは，それぞれのノードのポーリングスレッドによってバックグラウンドで処理される．

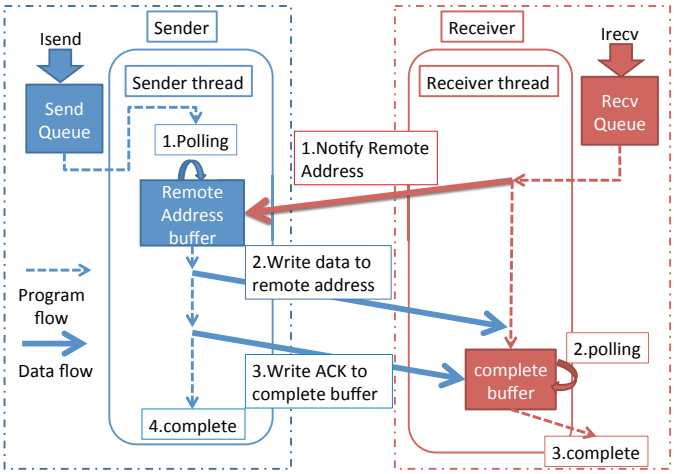

図 4 片方向通信による非同期 1 対 1 通信の実装方法

受信側ポーリングスレッドの処理

キュー (Recv Queue) にリクエストが入っていればそれを取り出し，対応する送信側ノードの受信アドレス通知バッファ (Remote Address Buffer) に，受信バッファのアドレスを書き込む．その後は，受信完了通知バッファ (complete buffer) を確認して，受信が完了するのを待つ．

送信側ポーリングスレッドの処理

受信アドレス通知バッファ (Remote Address buffer) と送信リクエストキュー (Send Queue) を確認して，両方が揃っていれば，届いた受信バッファアドレスに対してデータの書き込みを行う．また，受信側ノードの完了通知バッファ (complete buffer) にフラグを書き込み，受信完了を伝える．

キューに入っているリクエストは順番に 1 つずつ処理していくが，ノード間でのやりとりが増えると，レイテンシが増加してしまう．そこで，受信アドレスは先に送っても問題がないため，送信側ノードは受信アドレス通知バッファを複数持っておき，受信側ノードは送信リクエストが発行されるのを待たずに，受信アドレスを通知する．これにより，もし受信リクエストが複数ある場合は，受信アドレス通知によるレイテンシの増加を隠すことができると考えられる．

今回は，MPI の片方向通信通信関数を用いた実装を作成し，動作することを確認してから，TCA による片方向通信によって非同期 1 対 1 通信の実装を行った．なお，今回は TCA による Allreduce 関数の実装は見送ったが，今後取り組む予定である．

3.4 MPI の片方向通信による実装

MPI における片方向通信は MPI 2.0 から導入された機能である．MPI における片方向通信では，まず “Window” を

作成する．Window は片方向通信を行うためにリモートアクセス可能なメモリの情報を持つハンドルで，MPI 2.0 においては MPI_Win_create() 関数を用いて作成する．ただ，この関数を用いると通信領域は Window の作成時に指定する必要があり，書き込める領域が固定されてしまうが，受信バッファアドレスは Window 作成時には確定しない．そこで，MPI3.0 から導入された MPI_Win_create_dynamic() 関数によって Window を作成することで，実行中に片方向通信の通信領域を動的に変更することができるようにした．

また，MPI における片方向通信ではデータが書き込まれたことを保証するため，同期が必要となる．そのため，MPI_Win_lock()，MPI_Win_unlock() 関数を用いて同期を行った．

3.5 TCA による実装

TCA API による通信では TCA ハンドルが必要となるため，受信バッファの情報を含む TCA ハンドルと TCA ハンドルが指し示す領域からのオフセットを送信側ノードに送る．

TCA の API には，もともと受信側ノードが受信完了を確認する仕組みが備わっており，tcaTestDMARecvDesc() 関数を用いて，通信が完了しているか test を行った．また，逆に TCA の API には，現在自らが開始した DMAC による通信が完了したかどうかを知る仕組みがないため，受信バッファへの書き込みを行うディスクリプタチェインの最後に自身の CPU メモリに書き込みを行うディスクリプタを追加し，これポーリングすることによって完了を待った．

3.6 実装した非同期 1 対 1 通信の QUDA への適用

TCA によって CPU メモリ間の転送を行うことは可能ではあるが，現在，TCA が扱える CPU メモリ領域は，DMA のためにカーネル空間で確保したメモリに限られる．計算データを保持するためにカーネル空間で大きなメモリを確保することは難しいため，そのため，今回実装した非同期 1 対 1 通信では GPU メモリ間通信のみを対象とし，CPU メモリ間の通信には，MPI 関数を用いる．今後，TCA で大きな CPU メモリを確保できるようになれば，CPU メモリ間の通信も TCA で置き換えて行く予定である．

今回実装した，片方向通信による非同期 1 対 1 通信は，persistent communication として実装してはいないので，comm_declare 系関数では，単に渡された引数を保存しておく．そして，comm_start() 関数による通信開始時に，保存しておいた通信の引数を用いて非同期 1 対 1 通信関数を呼び出すことで通信を行う．また，TCA API によって通信を行う場合は，通信前に tcaCreateHandle() 関数によって TCA ハンドルを作成する必要があるので，comm_declare 系関数内で行うこととする．

4. 評価と考察

8 ノードの TCA 実験クラスタ TCAMINI 上で測定を行った．TCAMINI のノード構成を表 1 に示す．TCAMINI 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 によっても接続されている．

表 1 TCAMINI ノード構成

ハードウェア	
CPU	Xeon-E5 2670 2.6GHz ×2
Memory	DDR3 1600 MHx × 4ch，128 Gbytes
Motherboard	SuperMicro X9DRG-QF
GPU	NVIDIA K20 ×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 6.3
	kernel-2.6.32-279.22.1.el6.x86_64
GPU ドライバ	NVIDIA-Linux-x86_64-310.32
GPU プログラム環境	CUDA 5.0
MPI	MVAPICH2 2.0b

ここでは，基本的な pingpong ベンチマークによる GPU メモリ間通信のレイテンシとバンド幅の結果の評価を行う．比較対象として，SandyBridge アーキテクチャでの TCA による片方向通信 (GPU(SB))，IvyBridge アーキテクチャ上での TCA による片方向通信 (GPU(IVB)), MPI 実装の一つであり GPU 間通信をサポートする MVAPICH2 による通信 (MV2(SB)) の結果を共に示す．

現状では，MPI の片方向通信による実装には, 通信サイズが 32KB を超えると動作しない問題があり，性能比較ができていない．また，HA-PACS/TCA においても動作の確認を行っているが，動作が不安定であり原因を調査している．今回の実装を元に QUDA の comm 関数に組み込む作業を進めている．

4.1 非同期 1 対 1 通信の基本性能

図 5 に各通信方法のレイテンシを示す．グラフを見ると，今回実装した TCA による非同期 1 対 1 通信 (図中 TCA-ISR) が最も悪い結果になってしまった．8 Byte の時のレイテンシは 20us と，MVAPICH2 による通信 (図中 MV2(SB)) より悪い結果となっている．

TCA による非同期 1 対 1 通信において最低限必要となる時間は，送信側ノードへの受信アドレス通知と，受信側ノードへの実際のデータの転送である．受信アドレス通知は CPU 間通信であり，通知のサイズは 64Byte 程度なので

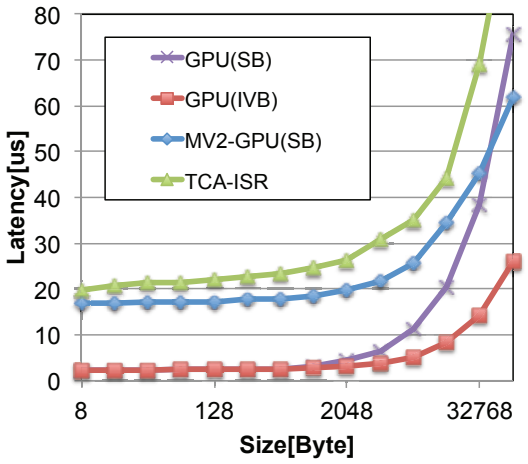

図 5 各通信方法におけるレイテンシ

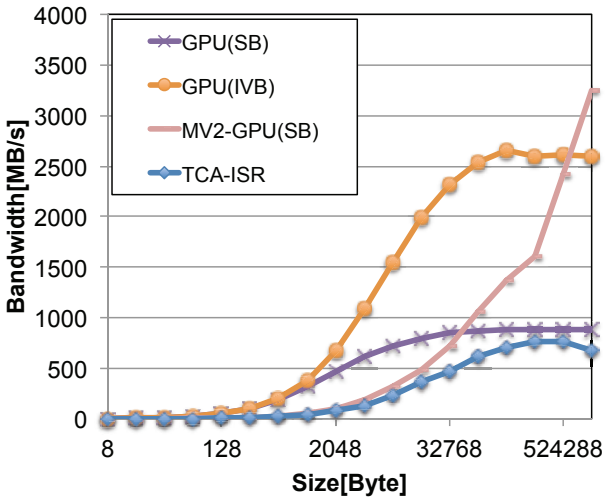

図 6 各通信方法におけるバンド幅

TCA では 2us で転送できる．また，データの転送を GPU 間で行う場合，通信サイズが 8Byte の時のレイテンシは 2.3us である．つまり，理想的な TCA による非同期 1 対 1 通信の通信サイズ 8Byte の時のレイテンシは 4.3us となる．実際には，これにディスクリプタを作成する時間などが加算されるが，現時点で詳細は不明であり，今後の改善が必要である．

図 6 に各通信方法のバンド幅を示す．グラフを見ると，こちらも今回実装した TCA による非同期 1 対 1 通信が最も悪い結果となった．ただ，バンド幅が 750MB/s 付近で頭打ちになるのは，SandyBridge アーキテクチャの CPU に内蔵されている PCIe のスイッチの性能が悪いためであることがわかっており，TCA の片方向通信 (図中 GPU(SB)) によるバンド幅も 800MB/s で頭打ちとなる．このボトルネックは，IvyBridge アーキテクチャの CPU を用いることで解消されることがわかっており，実際，IvyBridge アーキテクチャ上での TCA の片方向通信 (図中 GPU(IVB)) の性能は 2.5GB/s まで達している．HA-PACS/TCA では IvyBridge アーキテクチャの CPU を採用しており，TCA による非同期 1 対 1 通信においても，HA-PACS/TCA で実行することによって，バンド幅の制限は緩和されると考えられる．

5. 関連研究

APEnet+[10] 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 [11]．PEACH2 においても GPUDirect Support for RDMA を用いるが，APEnet+InfiniBand と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．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 [12] 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．

6. おわりに

本研究では，QUDA の通信を TCA によって置き換えることを目的として，TCA による非同期 1 対 1 通信の実装を行った．本実装のレイテンシを pingpong ベンチマークで測定したところ 20us であった．これは，TCA による片方向通信のレイテンシが 2.3us 程度であるのに比べて非常に遅いため，今後の実装の改良が必要である．

今後の課題としては，実装の改良の他に，今回作成した実装を QUDA に適用して性能の測定を行ったり，今回は検討しなかった TCA による Allreduce などの集団通信の実装などを考えている．

謝辞

本研究に際し御協力, 御助言をいただいた Davide Rossetti, Dale Southard 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．本研究の一部は文部科学省特別経費「エクサスケール計算技術開拓による先端学際計算科学教育研究拠点の充実」事業，および 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] 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).
[2] 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).
[3] 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).
[4] PCI-SIG: PCI Express Base Specification, Rev. 3.0 (2010).
[5] PCI-SIG: PCI Express External Cabling Spec- ification, Rev. 1.0 (2007).
[6] NVIDIA Corp.: NVIDIA GPUDirect. http://developer.nvidia.com/gpudirect.
[7] 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.
[8] NVIDIA Corp.: NVIDIA Tesla Kepler GPU Computing Accelerators. http://www.nvidia.co.jp/content/tesla/pdf/Tesla-KSeries-Overview -LR.pdf.
[9] Mike Clark.: QUDA. http://lattice.github.io/quda .
[10] 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.
[11] Mellanox Technologies: Mellanox OFED GPUDirect. http://www.mellanox.com/content/pages.php?pg= products dyn&product family=116&menu section=34.
[12] 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