

# GMPI: GPU クラスタにおける GPU セルフ MPI の提案

桑原 悠太<sup>1,a)</sup> 埜 敏博<sup>2</sup> 朴 泰祐<sup>1,3</sup>

**概要:** 近年, GPU クラスタでは, GPU プログラミング環境として CUDA (Compute Unified Device Architecture) が標準的に用いられている. GPU クラスタ上での並列アプリケーションでは, CUDA 環境において, ノードを跨ぐ GPU 間通信が発生し, MPI などによりホスト CPU が処理するのが一般的である. そのため, 通信が発生する毎に GPU 上の CUDA カーネルからホストに一旦制御を戻す必要があり, カーネル関数の起動や同期に伴うオーバーヘッドが生じる. 特に並列処理における通信粒度が細かいほど, カーネル関数の起動回数も増え, オーバーヘッドも増加する. それだけでなく, プログラミングのコストが高く, CPU 向け MPI プログラムを GPU 並列化する場合にソースコードが煩雑になりやすいといった生産性の低下も問題となっている. これらの問題を解決するために, 本研究では GPU カーネル内から MPI 通信の起動を可能とする並列通信システム “GMPI” を提案・開発する. これにより, 並列 GPU プログラミングを単純化し, GPU カーネルの起動や同期に伴うオーバーヘッド削減による並列処理効率の向上を目指す. 本稿では, GMPI の実装と, Ping-Pong 通信および姫野ベンチマークの性能評価を行う. 現状では性能最適化やチューニングが十分でなく, Ping-Pong 通信では従来方式とほぼ同等の性能であるが, 姫野ベンチマークでは従来手法の約半分の性能が得られている.

## 1. はじめに

近年, アクセラレータは高性能かつ低電力な HPC システムにおいて重要な要素となっている. 汎用アクセラレータとして GPU (Graphics Processing Unit) が持つ高い浮動小数点演算能力とメモリバンド幅を利用した GPGPU (General Purpose computing on GPU) が注目されており, GPU を搭載した複数のノードで構成されたクラスタが盛んに開発されている. Top500 List[1]においても, GPU を活用したシステムが多数登場する. 2015年6月に発表された Top500 List では, 2位の Titan, 22位の TSUBAME2.5 や 24位の Tianhe-1A が存在し, 筑波大学でも HA-PACS [2] (初出場順位 41位) が稼働中である.

次に, NVIDIA 社製 GPU におけるプログラミング環境として標準的に用いられる CUDA (Compute Unified Device Architecture)[3] と MPI[4] の組み合わせを用いて GPU プログラミングを行う場合について述べる. ノードを跨ぐ GPU 間通信ではホストが主体となって通信を行う

ため, 制御を一旦 GPU から CPU に戻す必要がある. そのため, 通信が発生する箇所でカーネル関数を切り分けてコーディングする必要があり, カーネル関数の起動や同期に伴うオーバーヘッドも生じる. 通信の粒度が細くなる程, カーネル関数の起動回数も増え, オーバーヘッドも増加する. また何よりも, GPU 版に書き換える前のオリジナルの MPI プログラムを多数のカーネルと MPI 通信の組み合わせに書き換える必要があり, プログラミングのコストが非常に大きく, ソースコードが煩雑になりやすい. これらの問題を解決するために, 本研究では GPU カーネル内からホスト CPU に制御を戻すことなく MPI 通信の起動を可能とする並列通信システム “GMPI” を提案・開発する. 本研究では, 一般的に用いられている NVIDIA 社製 GPU を対象とし, CUDA 環境を前提に開発を行う.

## 2. GPU クラスタにおけるノード間通信

本節では, GPU クラスタにおける一般的なノード間通信とその問題点に関して述べる. 以下, 特に断らない限り, ホスト CPU を単に「ホスト」と呼ぶ. 通常の GPU クラスタでは異なるノードの GPU 同士は直接通信できないため, ホスト側で通信の起動や管理を行う. 送信側では, デバイスメモリのデータをホストメモリにコピーをし, ホストメモリを介してホスト側がデータを送信する. 受信側でも, ホスト側がホストメモリにデータを受信し, ホストメ

<sup>1</sup> 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

<sup>2</sup> 東京大学 情報基盤センター  
Information Technology Center, The University of Tokyo

<sup>3</sup> 筑波大学 計算科学研究センター  
Center for Computational Sciences, University of Tsukuba

a) kuwahara@hpcs.cs.tsukuba.ac.jp

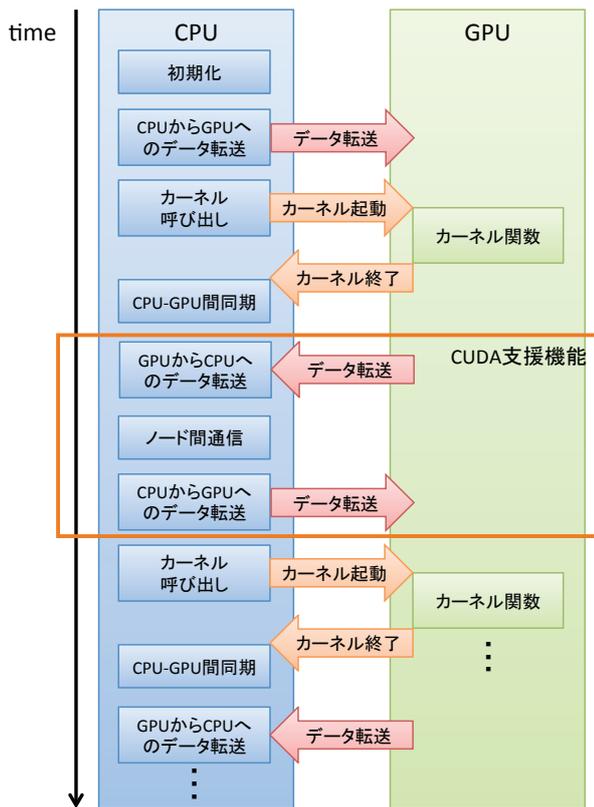


図 1 GPU クラスタにおけるノード間通信の流れ

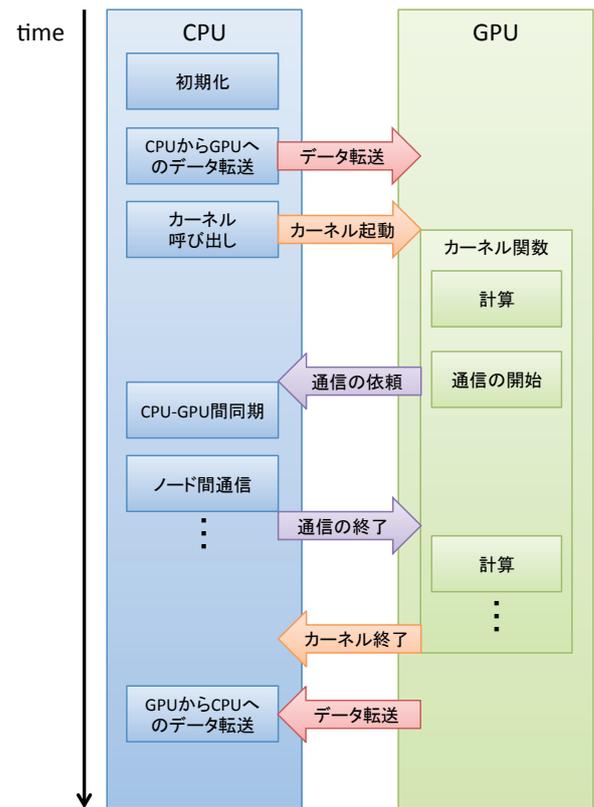


図 2 GMPI システムにおけるノード間通信の流れ

メモリを介してデバイスメモリにデータをコピーする。  
ノード間の通信に MPI を用いる場合、各ノードでホスト側が MPI プロセスを起動し、通信関数を用いて通信する。通常、MPI ではデバイスメモリを扱えないため、CUDA の API である cudaMemcpy() などでデバイスメモリとホストメモリとの間でデータを転送する。しかしながら、これを用いたメモリコピーにはレイテンシが大きいという問題がある。

MVAPICH2[10], Open MPI などの MPI 処理系には、MPI の送信および受信バッファにデバイスメモリを直接指定できる機能を持つものがあり、この機能を CUDA 支援機能と呼ぶ。NVIDIA 社製の GPU では、GPUDirect RDMA 機能 (GDR)[11] を用いることで、GPU 以外の PCIe デバイスが GPU メモリへ直接アクセスできる。GDR を用いた場合、ホスト側で確保した GPU メモリを PCIe アドレス空間にマッピングできる。同じ PCIe 空間の他のデバイスや CPU は、マップされた PCIe アドレスにアクセスすることで、直接 GPU メモリへの読み書きができる。Mellanox 社が提供する InfiniBand のネットワークインタフェースである InfiniBand HCA (Host Channel Adapter) は GDR に対応している [12]。GPU と HCA 間で直接通信を行うことで、CPU メモリを介さずに通信を行える。CUDA 支援機能に対応したプログラムを記述し、GDR 機能を持つ MPI 処理系を用いることで、高い通信性能が得られる。

GPU クラスタのプログラムは図 1 に示すような流れで実行される。通常、図中枠内の処理はユーザがプログラムに記述する必要があるが、CUDA 支援機能が有効な場合は MPI 関数の内部で行われる。異なるノード間での GPU 間通信を行う場合、ホスト側がノード間通信を行うため、カーネル関数からホスト側にプログラムの制御を戻す必要がある。そのため、通信毎にカーネル関数を切り分けてコーディングする必要があり、カーネル関数の起動や同期に伴うオーバーヘッドが生じる。

### 3. GMPI

#### 3.1 概要

ノード間通信が発生する場合において、GPU カーネル内から MPI 通信の起動を可能とする並列通信システムとして“GMPI”を提案する。本システムを利用した場合の GPU クラスタのプログラムの流れを図 2 に示す。GPU カーネル内から通信を起動するのは、あくまでユーザ視点からのモデルであり、GPU 自身が主体となって InfiniBand などの通信デバイスに通信を行わせることはできない。本システムではホスト側が GPU メモリをポーリングして通信リクエストを受け取り、GPU の代わりに MPI 通信を実行することによって、カーネル関数からは抜け出さずに MPI 通信を実現する。ユーザプログラム上では、カーネル内で MPI 関数に相当する通信関数である GMPI 関数を呼び出すだけで MPI 通信が記述でき、カーネルを切り分け

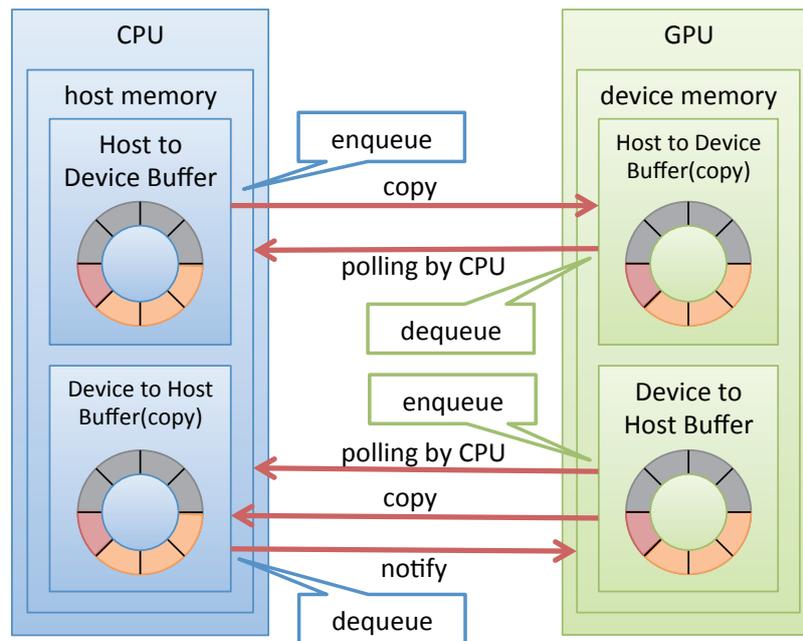


図 3 GMPI の実装イメージ

る必要はなくなる。これにより、カーネル起動や同期に伴うオーバーヘッドの削減やレイテンシの軽減、プログラミングのコストの削減を目指す。

### 3.2 GPU カーネル内から MPI 通信を起動する機構

GMPI の実装イメージを図 3 に示す。ホストメモリ上とデバイスメモリ上には同じ内容のリングバッファがそれぞれ二つずつ存在する。この二つのリングバッファのうち、一つはホストからデバイスにデータを転送するために用いる。もう一つは、デバイスからホストにデータを転送するために用いる。図中の矢印の向きは、実際のデータ転送の方向を示している。まず、ホストからデバイスにデータを転送するためのバッファについて述べる。ホスト側でリングバッファにデータを書き込んだ際、デバイス側のリングバッファもメモリコピーを行って更新する。ホスト側はデバイス側でデータが取り出されたかを確認するために、定期的にデバイス側のリングバッファの先頭位置をポーリングする。デバイス側のリングバッファの先頭位置が更新されていた場合、ホスト側のリングバッファの先頭位置も更新する。次に、デバイスからホストにデータを転送するためのバッファについて述べる。ホスト側はデバイス側でデータが書き込まれたかを検知するために、定期的にデバイス側のリングバッファの末尾の位置をポーリングする。デバイス側でデータの書き込みが行われていた場合のみ、ホスト側のリングバッファの内容を更新する。ホスト側でリングバッファからデータを取り出す際、デバイス側のリングバッファの先頭位置もメモリコピーを行って更新する。以上のようにホスト側でデーモンを動作させることによって、ホスト側とデバイス側で同じバッファの状態

を維持できる。

リングバッファを用いて転送するデータは、MPI 関数の種類を表すパラメータや引数などであり、本稿ではこれを Attribute と呼ぶ。尚、MVAPICH2-GDR などの GDR に対応した MPI 処理系では、MPI 通信における送受信端での実際の通信対象領域として GPU のデバイスメモリを指定できるため、Attribute として CPU と GPU がやりとりするのは実通信データ以外の必要情報のみとし、リングバッファを用いたデータの転送量を最小化してオーバーヘッドを削減する。リングバッファへのデータの書き込みは Warp の単位である 32 スレッドに絞り込んで行う。スレッドの絞り込みには、blockIdx および threadIdx から求められるスレッド毎に固有の ID を用いる。Attribute は最大で汎用ポインタ 32 個（8 bytes × 32 スレッド分）としている。デバイスからホストへ渡す Attribute の詳細として、最初の汎用ポインタには MPI 関数の種類を表すパラメータ、それ以降の汎用ポインタには指定した MPI 関数の引数を指定する。データを書き込む際、代表の一つのスレッドが共有メモリに Attribute を用意し、この共有メモリの内容を 32 スレッドが同時にリングバッファにコピーする。このとき、各スレッドがそれぞれの固有の ID に対応するインデックスのデータをコピーすることで、コアレスリングが起こるようにしている。また、リングバッファの先頭インデックスは Attribute の最大個数である 32 ずつ加算する。バッファサイズ QUEUE.SIZE は 2 の冪乗とし、“QUEUE.SIZE - 1” とのビットごとの論理積を求めることで、バッファがリングバッファとしてサイクリックに動作するように制御する。ホスト側のデーモンは、リングバッファに書き込まれた Attribute に基づいて実際の

表 1 評価環境

ノード構成	
CPU	Intel Xeon CPU E5-2670 v2
コア数	10 コア/ソケット × 2 ソケット
クロック数	2.50GHz
ピーク性能	332.8 GFLOPS / ノード
メモリ	128 GB, DDR3 1866 MHz × 4ch
Motherboard	Supermicro X9DRG-QF
GPU	NVIDIA K20
GPU 数	2GPU / ノード
ピーク性能	1.76TFLOPS / GPU
メモリ	5GB / GPU
InfiniBand	Mellanox Connect-X3 FDR Dual-port ( PCI Express 3.0 x8 )
ソフトウェア	
OS	CentOS 6.5
GPU ドライバ	NVIDIA-Linux-x86_64-340.29
CUDA	CUDA 6.5
MPI	MVAPICH2-GDR 2.0

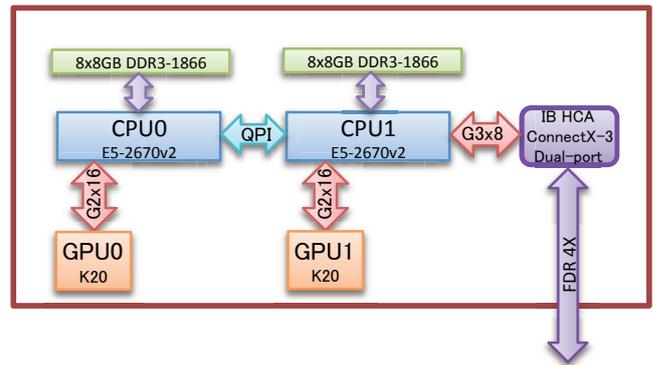


図 4 評価環境のノード構成

MPI 通信を起動する。

### 3.3 ホストとデバイス間のデータ転送

ホスト側がデバイス側のリングバッファを読み書きする場合、ホストメモリとデバイスメモリ間でのメモリコピーが必要となる。一般に、GPU メモリのコピーには CUDA の API である `cudaMemcpy()` を利用するが、これを用いたメモリコピーはレイテンシが大きいという問題がある。我々が以前 [9] で提案した実装では、ホスト側に実データがあり、デバイスからもアクセス可能な mapped page-locked memory を用いていた。しかしながら、このメモリを GPU から読み込むには、数  $\mu$ s のコストを要する。リングバッファでのやりとりには複数回の読み込みが必要となるため、レイテンシが蓄積し、大きくなってしまいう問題がある。そのため、GMPI では GPU メモリコピーを行う際に、`gdrcopy`[13] を用いることによって高速な転送を実現する。`gdrcopy` は NVIDIA 社によって提供されている GDR 機能を用いてホストメモリとデバイスメモリ間のメモリコピーを行うためのライブラリである。`gdrcopy` はレイテンシが極めて小さく、ホストメモリからデバイスメモリへのメモリコピーを高速に行うことができる。しかしながら、`gdrcopy` の デバイスメモリからホストメモリへのメモリコピーは、転送サイズが大きいとレイテンシが大きくなる傾向がある。本システムでは、デバイスからホストへ Attribute を転送する際に、まず MPI 関数の種類を表すパラメータのみを転送し、そのパラメータに基づいて引数の数分だけのデータを転送することで、データの転送量を削減している。

## 4. 予備評価

本節では、GMPI を実装するにあたり、カーネル関数の起動・同期オーバーヘッドおよび Attribute の転送に関する予備評価について述べる。

### 4.1 評価環境

本稿における評価は、表 1 および図 4 に示すノード構成の計算機で行う。プログラムを実行する CPU の指定には `numactl` コマンド、GPU の指定には環境変数 `CUDA_VISIBLE_DEVICES` を用い、QPI を越えるデータの転送が発生することを防ぐ。更に、MVAPICH2-GDR の環境変数として `MV2_USE_CUDA = 1` を設定することにより、CUDA 支援機能と GDR を有効にする。

### 4.2 カーネル関数の起動・同期オーバーヘッド

通常、ノード間通信が発生する毎に CUDA カーネルからホストに一旦制御を戻す必要があるため、通信毎にカーネル関数の起動や同期に伴うオーバーヘッドが生じる。本システムでは、カーネル関数の出入りがなくなる代わりにホスト側への通信リクエストが必要であり、これはカーネル関数の起動や同期に伴うオーバーヘッドより短い時間で行えることが望ましい。そこで、カーネル関数の起動や同期に伴うオーバーヘッドにどの程度の時間がかかるかを調査した。カーネル関数の起動のみのオーバーヘッド、カーネル関数の起動と `cudaDeviceSynchronize()` による同期のオーバーヘッド、カーネル関数の起動と `cudaStreamSynchronize()` による同期のオーバーヘッドの 3 通りについて測定を行った結果を表 2 に示す。ホストからの起動オーバーヘッドは  $3.2 \mu$ s 程度であり、更に同期を含めると  $11 \mu$ s 程度かかることがわかった。

### 4.3 Attribute の転送に関する予備評価

GMPI では、Warp 上の 32 個のスレッド並列を活かすため、各スレッドが 1 つずつ共有メモリからリングバッ

表 2 カーネル関数の起動・同期オーバーヘッド

同期の有無	Processing time [ $\mu$ s]
カーネル起動のみ	3.23
カーネル起動と <code>cudaDeviceSynchronize()</code> による同期	12.5
カーネル起動と <code>cudaStreamSynchronize()</code> による同期	10.9

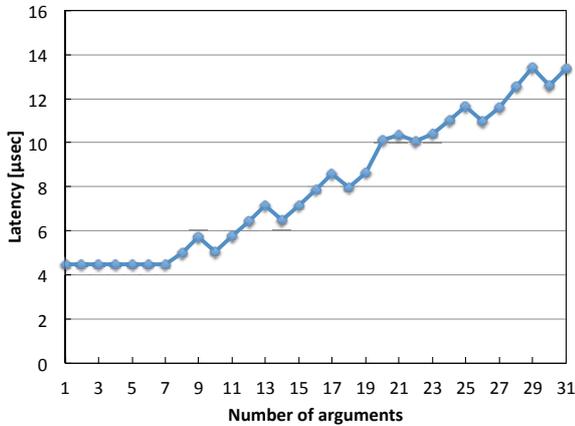


図 5 Attribute の転送に関する予備評価

ファヘータのコピーを行う。この 32 個のデータのうち、1 つを MPI 関数の種類を表すパラメータとするため、引数を最大 31 個までサポートしている。これに基づき、転送する汎用ポインタの数を 1 から 31 まで増やしていき、どの程度のサイズの Attribute の転送であればレイテンシが許容範囲であるかを調査した。カーネル関数を起動したままデバイスメモリに書き込むとメモリの値が即時に反映されないことがある。これを解決するために、デバイス側では `_threadfence()`、ホスト側では `_mm_sfence()` を利用した。実行速度の測定には、GPU コード上で GPU 内部のハードウェアサイクルカウンタを読み取る `clock64()` を利用した。測定結果を図 5 に示す。縦軸がレイテンシ、横軸が引数の個数となっている。

ホストからカーネル関数を起動するのみのレイテンシと比べると、どのデータサイズにおいても Attribute の転送のレイテンシが大きいという結果となった。カーネル関数の起動に加え、同期のオーバーヘッドも含めた場合、11  $\mu$ s 程度までは許容できると考えられる。したがって、引数の数が 23 個以下であればレイテンシを同程度に抑えられる。特に、引数が 7 個以下であれば 4.5  $\mu$ s 程度でホストに引数を渡せることがわかった。この実装を用いることによって、従来どおりカーネル関数の起動・同期を行う場合と比べて、レイテンシを同程度まで抑えられる可能性があると考えられる。

## 5. 性能評価

本節では、GMPI の通信性能を Ping-Pong 通信および姫野ベンチマーク [14] によって評価する。

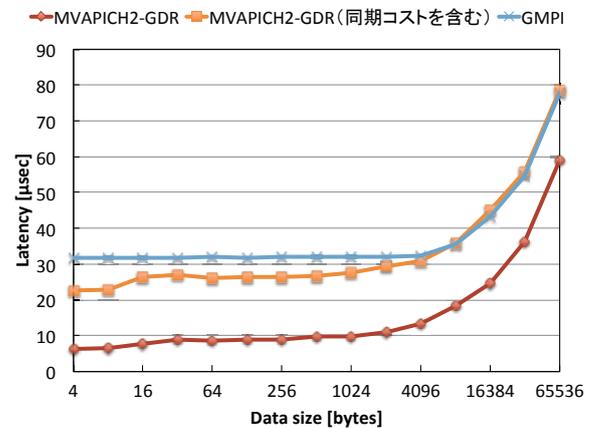


図 7 Ping-Pong 通信の測定結果

### 5.1 Ping-Pong 通信の性能評価

まず、GMPI を用いた場合の Ping-Pong 通信のコードに関して説明する。実際のコード例をリスト 6 に示す。このコードではカーネル関数である `send_kernel()` および `recv_kernel()` 間で Ping-Pong 通信を行う。GMPI で Attribute を扱うためのリングバッファ構造体として `gmapi_buf_t` 構造体を定義した。GMPI 関数はデバイス上で実行される `_device_` 関数として定義されている。GMPI 関数である `GMPI_Isend()` や `GMPI_Irecv()`、`GMPI_Wait()` の第一引数には GMPI のリングバッファ構造体を指定し、それ以降の引数にはそれぞれの MPI の関数の引数を指定して利用できる。コード例の `src` や `dst` は、`cudaMalloc()` など確保したデバイスメモリであり、カーネルの計算結果が格納されることを想定する。GMPI の実装には MVAPICH2-GDR を用いるため、MPI 関数の引数にデバイスメモリを直接指定できる。また、`GMPI_Synchronize()` を利用することにより、これまでホストに依頼した全てのリクエストが完了するように同期できる。これには、GMPI のリングバッファ構造体のメンバ変数であるリクエスト用のカウンタ変数を用いる。デバイスからホスト方向のバッファとホストからデバイス方向のバッファの両方のカウンタの値が同じである場合に、全てのリクエストの処理が完了している状態と定義している。以上のように、GMPI では CPU のみを用いて MPI プログラミングを行う場合とほぼ同様にコーディングできるため、プログラミングの学習コストの軽減や生産性の向上につながると考えられる。

次に、Ping-Pong 通信のレイテンシの測定結果を図 7 に示す。図中の凡例における MVAPICH2-GDR は 1 イテレー

```

__global__ void send_kernel(gmpi_buf_t *buf, int *src, int *dst, MPI_Request *request, MPI_Status
  *status) {
  for(int i = 0; i < DATA_SIZE; i++) {
    GMPI_Isend(buf, src, DATA_SIZE, MPI_INT, 1, i, MPI_COMM_WORLD, request);
    GMPI_Irecv(buf, dst, DATA_SIZE, MPI_INT, 1, i, MPI_COMM_WORLD, request);
    GMPI_Wait(buf, request, status);
    GMPI_Synchronize(buf);
  }
}

__global__ void recv_kernel(gmpi_buf_t *buf, int *src, int *dst, MPI_Request *request, MPI_Status
  *status) {
  for(int i = 0; i < DATA_SIZE; i++) {
    GMPI_Irecv(buf, dst, DATA_SIZE, MPI_INT, 0, i, MPI_COMM_WORLD, request);
    GMPI_Wait(buf, request, status);
    GMPI_Isend(buf, src, DATA_SIZE, MPI_INT, 0, i, MPI_COMM_WORLD, request);
    GMPI_Synchronize(buf);
  }
}

```

図 6 Ping-Pong 通信のコード例

ション毎に同期を行わない場合のレイテンシ, MVAPICH2-GDR (同期コストを含む) は 1 イテレーション毎に `cudaStreamSynchronize()` を利用して同期を行う場合のレイテンシ, GMPI は本システムを利用した場合のレイテンシを示している。通常, 実アプリケーションでは 1 イテレーション毎に同期を行うため, `cudaStreamSynchronize()` を利用して同期を行う場合の速度と比較・考察を行う。尚, 同期を行わない場合のレイテンシは, 実装のベースである MVAPICH2-GDR のみの実行速度の参考とするために記載している。転送するデータのサイズを 4 bytes から 64KB まで増やしていき, 1000 回の平均時間を実行速度として測定した。

測定の結果, データサイズが大きくなるほど, Attribute 転送などのオーバーヘッドが相対的に小さくなり, 従来のホスト上での MPI 実行の場合と比べ, 性能の差が縮まっていくことがわかった。実装のベースである MVAPICH2-GDR と比較すると, 最小サイズである 4 bytes では 7 割程度, 16 bytes では 8 割程度, 2KB では 9 割程度, 8KB 以上では同程度の通信性能が得られた。データサイズが小さい場合の性能差の原因として, GMPI 関数の内部でデバイスメモリの内容を同期する必要があることが挙げられる。これにより, GMPI 関数を実行する毎に, コンスタントに 10  $\mu$ s 程度のオーバーヘッドが生じてしまうことがわかっている。このオーバーヘッドを削減することによる GMPI 関数の高速化が今後の課題として考えられる。また, 現在の GMPI では 128KB 以上のデータ転送を行うと `GMPI_Wait()` で止まってしまいう問題があり, これを解決することも今後の課題として挙げられる。この原因は現在

**表 3** 姫野ベンチマークの問題サイズ ( $i \times j \times k$ )

SMALL	MIDDLE	LARGE
$64 \times 64 \times 128$	$128 \times 128 \times 256$	$256 \times 256 \times 512$

調査中であるが, `cudaMalloc()` によって確保したメモリ領域と `gdrCOPY` が使用するメモリ領域が競合している可能性があると考えている。

## 5.2 姫野ベンチマークの性能評価

本稿では, CUDA に移植した MPI 版の姫野ベンチマークを, NVIDIA Kepler アーキテクチャ向けに最適化し,  $i, j$  次元方向で分割できるようにしたものを利用した [15][16]. ヤコビ法の反復は 1000 回に固定とし, 問題サイズは表 3 に示すものを用いた。姫野ベンチマークにおける通信は, 袖領域交換と収束判定のための Allreduce である。これらの通信をサポートするために, GMPI 関数として, `GMPI_Waitall()` と `GMPI_Allreduce()` を追加した。本稿における GMPI 版の実装では, 元の実装において計算・袖領域交換・収束判定の 3 つを行う関数をカーネル関数化し, その内部で呼び出される関数を `_device_` 関数化し, MPI 関数を GMPI 関数に置き換えた。測定結果を図 8 に示す。

元の実装と比べると, GMPI を用いた場合の結果は, いずれのサイズにおいても 5 割程度の性能にとどまっている。計算・袖領域交換・収束判定のうち, GMPI による実装では計算部分の実行時間がオリジナルの MPI 実装に比べて増大していた。この原因は現在調査中であるが, おそらくカーネル関数から抜け出さず, その中で計算を継続することが何らかの悪影響を与えているのではないかと考えられる。今後, この原因を調査し, 計算部分の実行時間を

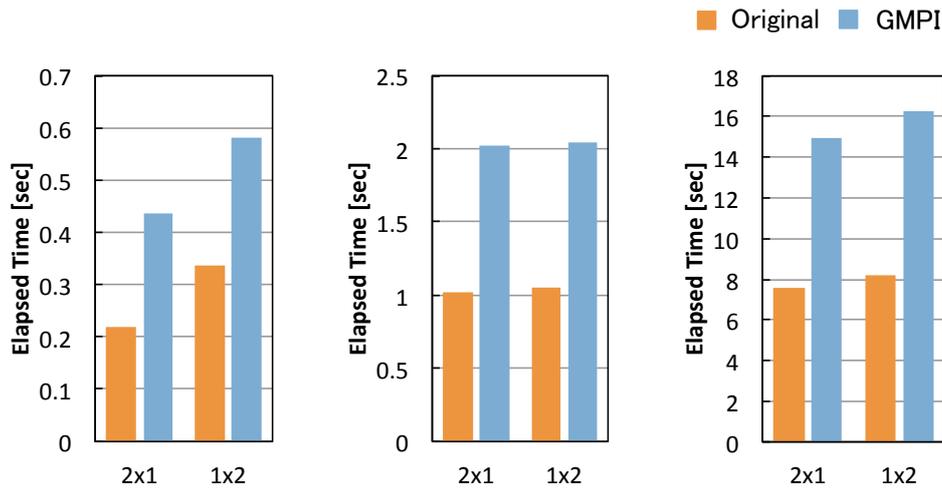


図 8 姫野ベンチマークの測定結果：左から順に SMALL, MIDDLE, LARGE

元の実装と同程度にすることを目標に最適化を行う。そのうえで、通信性能の比較や考察、さらなるチューニングを行う予定である。

## 6. 関連研究

電気通信大学の島らによって、カーネル関数内に MPI の関数を記述可能にする GPU プログラミングフレームワーク FLAT が提案されている [5]。FLAT はコンパイラのプリプロセッサとして実装されており、ソース to ソースのコード変換を行うことにより、カーネル関数に埋め込まれた MPI の関数の実際の処理を CPU コード上での処理に置き換える。それに対し、本研究ではカーネル関数に記述された MPI の関数の処理を、コンパイル時ではなく実行時にホスト側に依頼する機構を実現する。

本システムに似た実装として、Sapienza University of Rome で開発されていた CUQU (CUDA queue)[6], CUOS (CUDA Offloaded System services)[7] というライブラリが存在する。CUQU は CUDA を用いて開発されており、GPU カーネルとホスト側とのやりとりを page-locked memory (pinned memory) のリングバッファを介して行う機構である。CUOS は GPU カーネルからホストシステムのサービスを呼び出すためのフレームワークのプロトタイプであり、CUQU を利用してカーネル関数内から MPI の同期通信を行う例が実装されている。CUDA のバージョンは 4.0 前後を想定しており、特定の CUDA 環境に依存した実装となっている。これらのライブラリの開発は 2011 年 5 月で打ち切られており、その実用例も見当たらない。本研究では最近の CUDA 環境にも適用した、より高速かつ利便性の高いシステムの開発を目指す。

更に、The University of Texas at Austin の Kim らによって開発されている GPUnet[8] も本システムに似た実装として挙げられる。GPUnet は、RDMA over InfiniBand

を用いて、GPU が socket 通信を行えるようにするライブラリである。GPU と CPU との間で共有しているリングバッファを介し、GPU が CPU に socket 通信を依頼する。CPU が GPU メモリをポーリングすることによって通信リクエストを受け取る実装となっている点が CUQU, CUOS とは異なっている。GPUnet では socket 通信を対象としているが、本稿では MPI 通信を対象とし、通信をアプリケーション向けに、より抽象化することによって HPC 向けの性能向上を目指している。

本システムは、[9] で提案した機構に基づいて実装されているが、ホストとデバイス間のやりとりをホストからのポーリングベースに実装し直した点が異なっている。

## 7. おわりに

### 7.1 まとめ

本稿では GPU カーネル内から MPI 通信の起動を可能とする並列通信システム GMPI に関する調査および予備実験を行い、有用性の調査・考察を行った。

テスト環境で Attribute の転送に関する予備評価を行った結果、引数が 23 個以下であれば  $11\mu\text{s}$  以内に Attribute を転送できることがわかった。引数がある程度少なければカーネル関数の呼び出しコストより小さいオーバーヘッドでホストに通信を依頼できると考えられる。

Ping-Pong 通信の性能評価では、実装のベースである MVAPICH2-GDR と比較を行った。測定したデータサイズのうち、最小サイズである 4 bytes では 7 割程度の性能、8KB 以上においては同程度の通信性能が得られた。また、CPU のみを用いて MPI プログラミングを行う場合とほぼ同様にコーディングできるため、プログラミングの学習コスト軽減や生産性の向上につながると考えられる。

姫野ベンチマークの性能評価では、元の実装と比較すると GMPI の性能は 5 割程度にとどまっている。計算・袖

領域交換・収束判定のうち、計算部分の実行時間が最も増加しており、その理由としてカーネル関数を起動し続ける実装に変更したことによる影響が考えられる。

## 7.2 今後の課題

引き続き GMPI に関する調査や予備実験を行い、速度向上やリファクタリングを行う予定である。Ping-Pong 通信の性能評価に関しては、小さいデータを転送する場合のレイテンシも軽減するための最適化を行う予定である。また、128KB 以上のデータを転送できない問題が存在するため、これについても原因を調査し、修正する予定である。姫野ベンチマークに関しては、計算部分の実行時間が増加した原因の調査を行い、元の実装と同程度にすることを目標に最適化を行う。そのうえで、通信性能の比較や考察、さらなるチューニングを行う予定である。また、姫野ベンチマークの他にも、NAS Parallel Benchmark[17] などの実アプリケーションに対し本システムを適用することで、実用性の調査を行う。

更に、MPI の他に、筑波大学計算科学研究センターで開発が行われている密結合並列演算加速機構 TCA (Tightly Coupled Accelerators)[18] に対しても同様の機構を適用し、ライブラリの実装を行っていく予定である。最終的には、TCA に適用したライブラリを用いて、ベンチマークテストやアプリケーションを作成し、評価・比較を行う予定である。

## 謝辞

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。また、本研究における HA-PACS/TCA の利用は、筑波大学計算科学研究センター学際共同利用プログラム・平成 27 年度課題「密結合演算加速機構アーキテクチャに向けたアプリケーションの開発と性能評価」による。

## 参考文献

- [1] TOP500 Supercomputer Sites (online), <http://top500.org/>
- [2] 筑波大学計算科学研究センター : HA-PACS ベースクラスター (online), <http://www.ccs.tsukuba.ac.jp/research/project/hapacs/cluster>
- [3] CUDA C Programming Guide (online), <http://docs.nvidia.com/cuda/cuda-runtime-api/index.html>
- [4] Message Passing Interface (MPI) Forum Home Page (online), <http://www.mpi-forum.org/>
- [5] 島 圭吾, 吉見 真聡, 三好 健文, 近藤 正章, 入江 英嗣, 本多 弘樹, 吉永 努: FLAT: MPI を埋め込み可能な GPU プログラミングフレームワーク, 情報処理学会論文誌コ

- ンピューティングシステム (ACS), Vol. 6, No.4, pp. 105-116 (2013).
- [6] cuqu A CPU ↔ GPU messaging queue (online), <https://code.google.com/p/cuqu/>
- [7] cuos Offloaded System services for CUDA (online), <https://code.google.com/p/cuos/>
- [8] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 201216, Broomfield, CO, October 2014. USENIX Association.
- [9] 桑原 悠太, 埜 敏博, 児玉 祐悦, 朴 泰祐: GPU クラスタにおける GPU 間セルフ通信機構に関する提案, Vol.2015-HPC-148 No.17, 2015.
- [10] MVAPICH2 2.0.1 User Guide (online), <http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.0.1-userguide.html>
- [11] NVIDIA Corp. : NVIDIA GPUDirect. (online), <http://developer.nvidia.com/gpudirect>
- [12] Mellanox GPUDirect RDMA User Manual Rev 1.0 (online), [http://www.mellanox.com/related-docs/prod\\_software/Mellanox\\_GPUDirect\\_User\\_Manual\\_v1.0.pdf](http://www.mellanox.com/related-docs/prod_software/Mellanox_GPUDirect_User_Manual_v1.0.pdf)
- [13] NVIDIA gdrCOPY. <https://github.com/NVIDIA/>
- [14] 姫野ベンチマーク, 独立行政法人 理化学研究所 情報基盤センター (online), <http://acc.riken.jp/2145.htm>
- [15] E. Phillips and M. faticca. Implementing the Himeno benchmark with CUDA on GPU clusters Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium, pp.1-10, Apr. 2010.
- [16] Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, Mitsuhisa Sato. Tightly Coupled Accelerators Architecture for Low-latency Inter-Node Communication Between Accelerators SC14 poster, Nov. 2014.
- [17] NAS Parallel Benchmark (online), [http://mikilab.doshisha.ac.jp/dia/smpp/01\\_bench/naspara.html](http://mikilab.doshisha.ac.jp/dia/smpp/01_bench/naspara.html)
- [18] 埜 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 (SACIS2013) 論文集, pp. 150-157 (2013).