

GPU クラスタにおける GPU 間セルフ通信機構に関する提案

桑原 悠太^{1,a)} 埴 敏博² 児玉 祐悦^{3,4} 朴 泰祐^{3,4}

概要: GPU クラスタにおいて, GPU 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) では, ノードを跨ぐ GPU 間通信ではホスト側が通信を行うため, 通信が発生する度に 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る. そのため, 通信が発生する場所で, カーネル関数を切り分ける必要があり, ユーザプログラミングが煩雑になるだけでなく, カーネル関数の起動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. それを防ぐために, 本研究では GPU がカーネル内で直接通信を実行できるような機構を提案・開発する. これを「GPU 間セルフ通信機構」と名付ける. 本機構では, 並列 GPU プログラミングを単純化し, 複数ノードの GPU 間通信におけるカーネル起動のオーバーヘッドの削減による並列処理効率の向上を目指す. 本稿では, GPU 間セルフ通信機構に関する実装方法の調査と予備的な性能評価に関して述べる. その後, 本機構を用いた「MPI on GPU」の実装 (GMPI) と, 簡単なベンチマークとして ping-pong 転送の性能を評価する. GMPI の試験的実装と予備評価の結果, 実装のベースとして用いた MVAPICH2-GDR の性能の約 60%の通信性能が得られた.

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 リスト [1] の中にも GPU を活用したシステムが多数登場する. 2014 年 11 月に発表された TOP500 リストでは, 2 位の Titan, 15 位の Tsubame2.5 や 17 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.

一方, GPU クラスタにおけるプログラミングを考えると, 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] の組み合わせにおいて, ノードを跨ぐ GPU 間通信についてもホストが主体となって通信を行うため, 一旦制御を 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は, カーネル関数を切り分ける必要があり, カーネル関数の起動に伴うオーバーヘッドが生じる. 通信の粒度が細くなると, 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 化す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い. これらの問題を解決するために, GPU カーネル自身が通信を実行できる機構として「GPU 間セルフ通信機構」を提案・開発する. 本研究では, 最も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. 関連研究

電気通信大学の島らによって, カーネル関数に MPI の関数が記述できるようにする 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 はコード変換器であり, コンパイラのプリプロセッサとして実装されている. ソース to ソース変換によって, 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

¹ 筑波大学 情報学群 情報科学類
College of Information Science, University of Tsukuba
² 東京大学 情報基盤センター
Information Technology Center, The University of Tokyo
³ 筑波大学大学院 システム情報工学研究科
Graduate School of System and Information Engineering, University of Tsukuba
⁴ 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

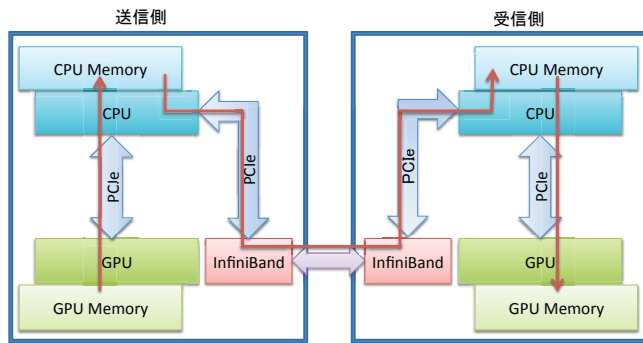


図1 GPU クラスタにおけるノード間通信

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。

本機構に似た実装として、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あり、ソースコードが公開されている。CUQU は CUDA でコーディングされており、GPU カーネルとホスト側とのやりとりを page-locked memory (pinned memory) を介して行う機構である。CUOS は CUDA 環境を離れることなく、GPU カーネルからホストシステムのサービス呼び出すためのフレームワークのプロトタイプである。CUOS では 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機構の開発を目指す。

3. 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。異なるノードの GPU 同士は直接通信できないため、図1のように CPU 側で通信の起動・管理を行う。送信側は、デバイスメモリからホストメモリにデータをコピーし、ホスト側がネットワークを介してデータを送信する。受信側は、ホスト側がホストメモリにデータを受信し、ホストメモリから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、`cudaMemcpy()` 等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必要がある。この場合のコード例を図2に示す。

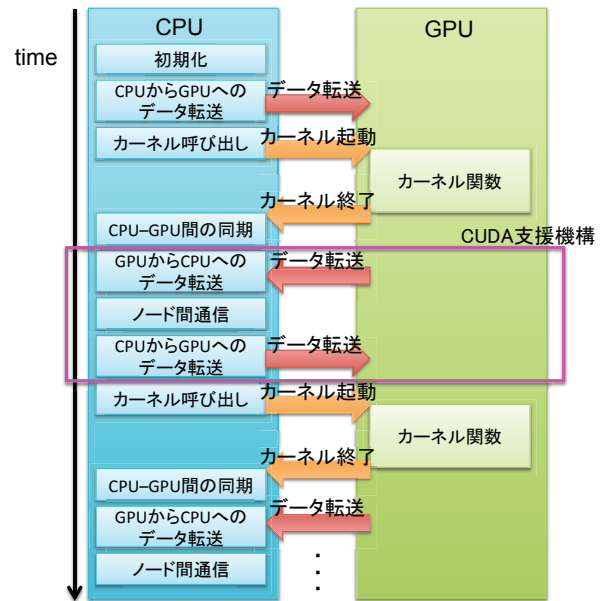


図4 GPU クラスタにおけるノード間通信の流れ

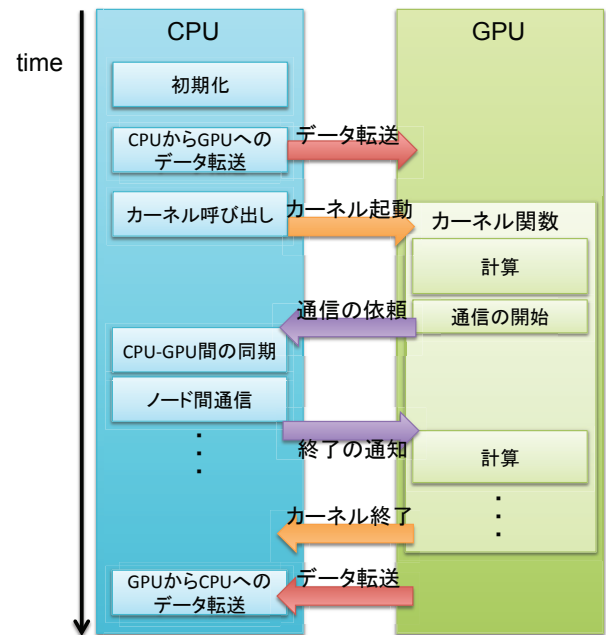


図5 GPU 間セルフ通信機構を適用したノード間通信の流れ

MVAPICH2[8], Open MPI などの MPI 処理系には、MPI の送信及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る。この機能を CUDA 支援機構と呼び、コード例を図3に示す。NVIDIA 社製の GPU では、GPUDirect RDMA 機能 (GDR)[9] 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

```
if(myrank == 0) {
    cudaMemcpy(temp, src, data_size, cudaMemcpyDeviceToHost);
    MPI_Send(temp, data_size, MPI_BYTE, 1, 0, MPI_COMM_WORLD);
} else {
    MPI_Recv(temp, data_size, MPI_BYTE, 0, 0, MPI_COMM_WORLD);
    cudaMemcpy(dest, temp, data_size, cudaMemcpyHostToDevice);
}
```

図 2 異なるノードでの GPU 間通信のコード例

```
if(myrank == 0) {
    MPI_Send(src, data_size, MPI_BYTE, 1, 0, MPI_COMM_WORLD);
} else {
    MPI_Recv(dst, data_size, MPI_BYTE, 0, 0, MPI_COMM_WORLD);
}
```

図 3 CUDA 支援機構を持つ MPI における異なるノードでの GPU 間通信のコード例

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 [10]. 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 4 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が, CUDA 支援機構が有効な場合は 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, 通信が必要になる度にカーネル関数を分割する必要があるが生じ, カーネル関数の起動に伴うオーバーヘッドが生じる.

4. GPU 間セルフ通信機構

通常, カーネル関数が切り分けられるノード間通信が発生する場合において, GPU が独立して通信を開始できる機構として GPU 間セルフ通信機構を提案する. 本機構を利用した場合の GPU クラスタのプログラムの流れを図 5 に示す. 「GPU カーネル内からの通信の起動」はユーザ視点からのモデルであるが, 当然ながら, GPU 本体が単独で InfiniBand 等の通信デバイスに通信を行わせることはできない. したがって, 本機能では, GPU と CPU によって PCI Express を介して共有されるメモリを用い, 通信リクエストを CPU 側に送ることで, カーネル関数からは抜け出さずに GPU 間通信を実現する. このようにして, ユーザプログラム上ではカーネル内で MPI 通信に相当する通信関数を呼び出すだけで通信が記述でき, カーネルを切り分ける必要はなくなる. また, GPU 間通信におけるカーネル起動のオーバーヘッドを削減することにより, レ

イテンシの軽減も目指す.

GPU カーネルとホストプログラム間のやりとりには, CUDA の mapped page-locked memory (mapped メモリと略す) を用いる. ホスト側の mapped メモリは cudaHostAlloc() に cudaHostAllocMapped をフラグとして指定することで確保され, GPU から直接アクセスできるように, GPU のアドレス空間にマッピングされる. cudaHostGetDevicePointer() を用いることでデバイス側のアドレスを取得できる. デバイスにおいて書き込みを行った場合, ホストや他のデバイスにおいて mapped メモリの内容が書き換わるタイミングは未定義であり, 内容の同期をとるためには __threadfence_system() を用いる. この機能を利用するには, GPU が対応している必要があり, deviceQuery プログラムの Support host page-locked memory mapping の項目で確認できる. このメモリを使用する利点として, ホストとデバイス間の転送をより高速に実行できることや, カーネル実行と同時にホストとデバイス間のコピーが可能であることが挙げられる.

現在, GPU 間セルフ通信機構の利用例の 1 つとして, 本機構を MPI に適用したライブラリを開発している. 本稿ではこのライブラリの実装を MPI on GPU と呼ぶ. MPI on GPU の実行イメージを図 6 に示す. GPU 上で通信用のカーネル関数 (CUDA Stream) を常駐させ, ホスト上にも同様に通信管理用のスレッドを常駐させる. 計算用のカーネル関数は通信用のカーネル関数とは独立に走らせる. 本稿では, 通信用カーネルとホスト上の通信管理用のスレッドとがやりとりする通信依頼及び結果に関する情報を Attribute と呼ぶ. 通信を行う手順は次のとおりである.

- 計算用カーネルが, 通信用カーネルに通信の開始を依頼する.
- 通信用カーネルは, mapped メモリを介して Attribute を転送し, ホスト上の通信管理用のスレッドに通信を

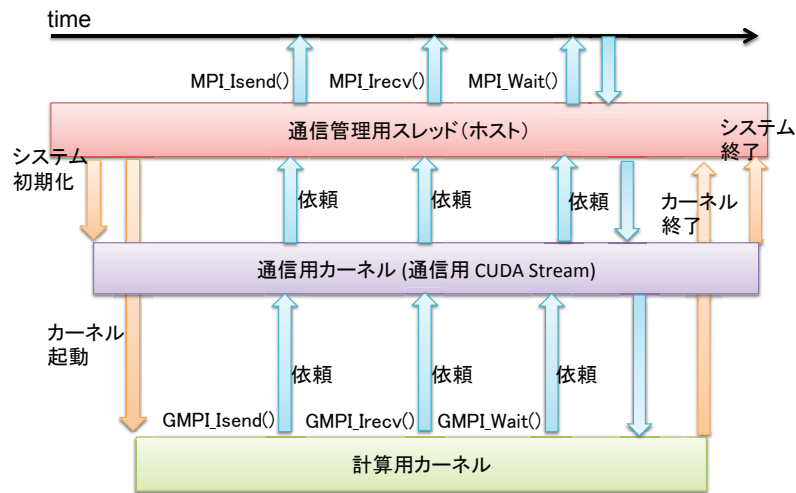


図 6 MPI on GPU の実装イメージ

依頼する。

- 依頼を受けた通信管理用のスレッドは実際の通信を開始する。

このような流れで通信を行うことで、カーネル関数から MPI の通信の開始を実現する。

尚、MVAPICH2-GDR などの GDR に対応した MPI 実装系では、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、共有メモリを用いたデータの読み書き量を最小化する。

5. 予備評価

GPU 間セルフ通信機構を実装するにあたり、カーネル関数の起動オーバーヘッド、Attribute の転送、カーネル関数間の同期に関する予備評価を行う。

5.1 評価環境

本稿における評価は、表 1 及び図 7 に示すノード構成の計算機で行う。CUDA のバージョンに関しては、特に記述がない限りは CUDA6.0 を使用する。MPI に関しては MVAPICH2-GDR 2.0 を使用する。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 を用いる。これにより QPI を越えるデータの転送が発生することを防ぐ。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。

5.2 カーネル関数の起動オーバーヘッド

通常、通信部分で切り分けられた各カーネルにおいて、カーネル関数の実行とそれに伴う起動オーバーヘッドが生じる。GPU 間セルフ通信機構では、カーネル関数の出

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 6.0
MPI	MVAPICH2-GDR 2.0

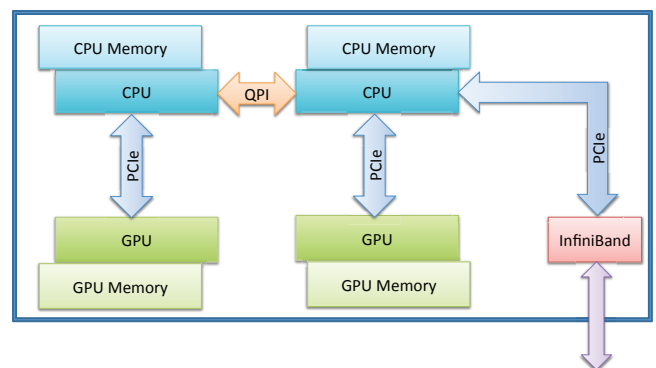


図 7 評価環境のノード構成

入りがなくなる代わりに通信管理用のスレッドに対する通信の依頼が必要になる。したがって、通信管理用のスレッドに対する通信の依頼は、カーネル関数の起動オー

表 2 カーネル関数の起動オーバーヘッド

CUDA version	Processing time [μ s]	
	from host	from kernel
CUDA5.0	3.317067	17.4911
CUDA5.5	3.340967	17.5165
CUDA6.0	3.307859	14.1621
CUDA6.5	3.233323	14.164

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オーバーヘッドにどの程度の時間がかかるかを測定した。Kepler アーキテクチャから Dynamic Parallelism (DP と略す) という機能が導入された。この機能によって、GPU が CUDA カーネルの実行時に、その実行中のカーネル内部から GPU 自身の新たなタスクを生成するという方法で、GPU を自律的に動作させることが可能になった。この機能ではタスクをダイナミックに生成できるが、そこで実行されるカーネル関数の起動オーバーヘッドが、ホストから起動する場合に比べて増加すると予想される。そこで DP を用いたカーネル関数の起動オーバーヘッドにどの程度の時間がかかるかも測定した。カーネル関数の起動オーバーヘッドが 1 回目のみ大きく、カーネル関数の起動回数が少ないと平均の起動時間が大きくなることが観測されたので、16384 回起動を繰り返し、平均時間を測定結果とした。測定結果を表 2 にまとめた。

ホストからの起動オーバーヘッドはいずれのバージョンでも 3.3 μ s 程度であることがわかった。カーネル内部からの起動オーバーヘッドは、CUDA5.0 系では 17.5 μ s 程度、CUDA6.0 系では 14.2 μ s 程度であることがわかった。CUDA6.0 系では DP の実行効率が改善され、実行時間が 3.3 μ s 程度短縮されている。

5.3 Attribute の転送に関する予備評価

通信用カーネルとホスト上の通信管理用スレッド間の Attribute の転送手順は次のとおりである。mapped メモリとして確保されたバッファ及び通信の依頼状況を管理する flag 変数を用意する。まず、通信用カーネルが GPU からバッファにデータの書き込みを行う。GPU からの書き込み後、ホストからデータを参照しても書き込みが完了しているようにするために `_threadfence_system()` で書き込みを完了させ、flag 変数の値を書き換えることで Attribute の書き込み完了をホスト上の通信管理用スレッドに通知する。ホスト上の通信管理用スレッドは flag 変数を busy loop で polling することによって書き込み完了の通知を受け取り、実際の通信を開始する。通信完了後に flag 変数の値を書き換えることで通信の依頼を受け付け可能な状態に戻す。

100 回のダミーループの後に 10000 回のループを実行し、10000 回の平均時間を実行速度として測定を行った。転送するバッファのサイズを 4bytes から 1024 bytes まで増や

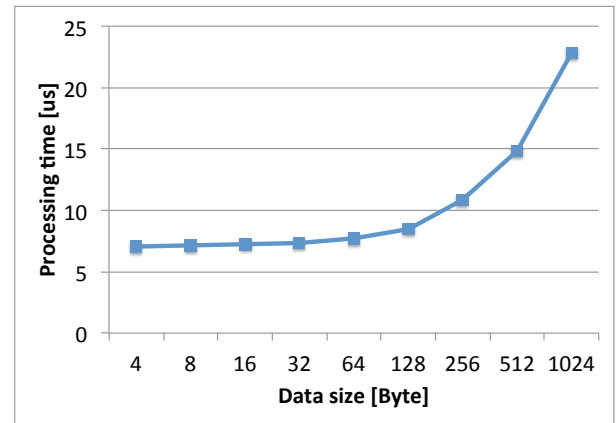


図 8 Attribute の転送に関する予備評価

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。測定結果を図 8 に示す。

ホストからのカーネル関数の起動に対しては、どのデータサイズにおいても Attribute の転送の方が時間がかかるという結果であった。しかしながら、DP を用いたカーネル関数の起動に対しては、4bytes から 256bytes までのデータサイズでは Attribute の転送の方が時間がかからないという結果になった。したがって、Attribute のデータ長は 256 bytes 程度までは許容可能であると結論した。改善の余地があると考えられるため、同期手法や使用するメモリ・関数等について検討していく予定である。

5.4 カーネル関数間の同期に関する予備評価

GPU 間セルフ通信機構では通信用カーネルと計算用カーネルを分ける。1 つの GPU カーネルが通信用カーネルとして、計算用カーネルからの通信依頼の集約とホストへの通信の依頼を担当する。計算用カーネルが通信用カーネルへ依頼し、この依頼を受けて通信用カーネルがホスト上の通信管理用スレッドに通信を依頼する。計算用カーネルと通信用カーネルのやりとりをする上でカーネル関数間の同期が必要になる。同期手法として polling による同期、atomic 関数による同期、CUDA Event による同期の 3 つを検討した。

polling による同期

デバイスメモリ上に通信の依頼状況を管理する flag を用意する。計算用カーネルは通信を行う際に flag に特定の値を代入し、`_syncthreads()` でカーネル間同期を行い、通信用カーネルに通信開始可能な状態を通知する。通信用カーネルはこの flag を busy loop で polling することで通信開始可能な状態を検知し、ホストへの通信の依頼を開始する。

atomic 関数による同期

基本的には polling による同期と同様だが、flag への代入に atomic 関数である `atomicExch()` を用いる。

atomic 関数を用いることで、`__syncthreads()` によるカーネル間同期が不要になる。

CUDA Event による同期

CUDA Event とは GPU でのタスクの状況を追跡できる機構である。CUDA Event を登録する `cudaEventRecord()` 及び CUDA Event による同期を行う `cudaEventSynchronize()` を用いた実装を検討した。

polling による同期手法及び atomic 関数による同期手法の実装は行ったが、CUDA Event による同期の実装は保留にしている。その理由は CUDA C Programming Guide [3] の DP の項目に以下の様な制約があるためである。

CUDA では、CUDA Event を CUDA Stream に登録して同期に利用できる。しかしながら、カーネル関数において、`cudaStreamWaitEvent()` は使用可能だが、`cudaEventSynchronize()`、`cudaEventElapsedTime()` 及び `cudaEventQuery()` は使用できない。`cudaEventElapsedTime()` が使用できないため、CUDA Event は `cudaEventCreateWithFlags()` に `cudaEventDisableTiming` フラグを指定して作成しなければならない。

また、CUDA Stream を生成する `cudaStreamCreate()` はホスト関数であるため、CUDA Stream の実体はホストメモリに置く必要がある。したがって、mapped メモリとして CUDA Stream を Create しなければカーネル関数内で扱うことができないようである。前述のとおり、mapped メモリは atomic でないので、`cudaStreamWaitEvent()` を呼んだ直後に `__threadfence_system()` を呼ぶ必要があると考えられる。

polling による同期手法及び atomic 関数による同期手法に関して、バッファへの書き込み後に flag を変更して同期を取るコードを評価した。2つのカーネル関数を呼び出し、カーネル関数内で 10000 回繰り返し、その平均時間を測定した。Attribute の転送の場合と同様に同期をとるバッファのサイズを 4bytes から 1024 bytes まで増やしていき、測定を行った。測定結果を図 9 に示す。

データサイズが小さい場合には polling による同期手法の方が実行時間が短いが、128 bytes を超えると atomic 関数による同期手法の方が速くなる場合がある。測定したのは往復のデータのやりとりの時間だが、実際には、往路では Attribute 等のデータをやりとりする可能性があるが、復路では戻り値など往路に比べて少ない情報のやりとりになると考えられるため、片道の実行時間に近い時間と仮定すれば良いと考えられる。もし、ポインタ 1 つのやりとりで済めば、片道 $2 \mu\text{s}$ 弱で通信用カーネルへの依頼が行える。

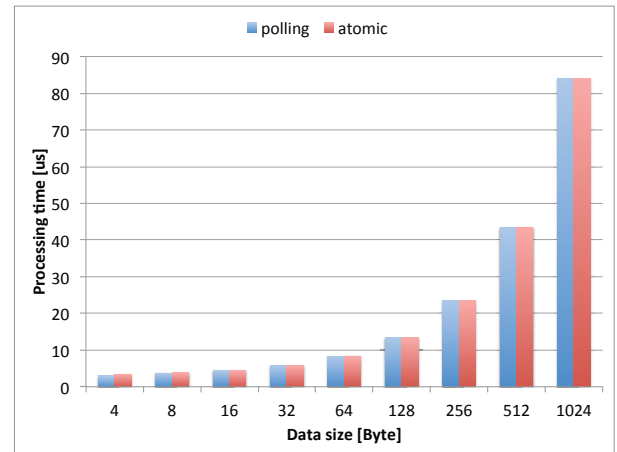


図 9 カーネル関数間の同期に関する測定結果

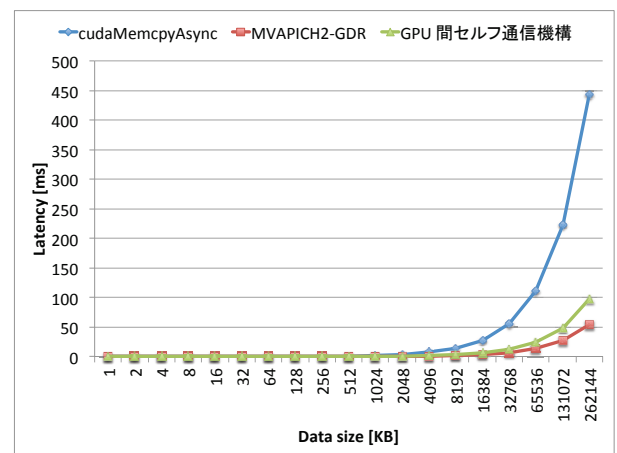


図 10 ノードを跨ぐ GPU 間通信の測定結果

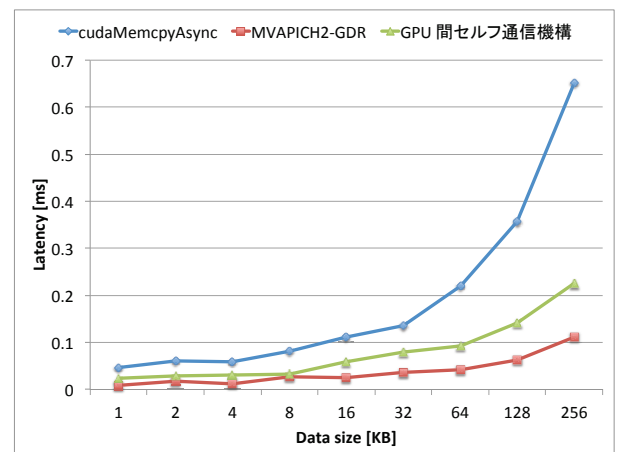


図 11 ノードを跨ぐ GPU 間通信の測定結果 (拡大図)

6. MPI on GPU

本節では GPU 間セルフ通信機構を MPI に適用した MPI on GPU に関する測定と実装について述べる。

6.1 非同期 1 対 1 通信の性能評価

通信用カーネルとホスト上の通信管理用スレッドが At-

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

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

図 12 ping-pong のコード例

tribute をやりとりするために用いるリングバッファを実装した。通信用カーネルがリングバッファに Attribute を enqueue し、ホスト上の通信管理用スレッドがそれを dequeue して通信を開始できることを確認した。

通信用カーネルは Warp の単位である 32 スレッドで動作させ、これに合わせて Attribute の最低単位を汎用ポインタ 32 個 (8bytes × 32threads) とした。threadIdx.x = 0 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リングバッファに enqueue する。このとき、各スレッドがそれぞれの threadIdx に対応する汎用ポインタにアクセスすることで、コアレスシングが起きるようにしている。また、リングバッファの先頭インデックスは Attribute の最低単位である 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、インデックスがバッファサイズを超えるとバッファの先頭に戻るようにしている。

ホストとデバイスの通信に cudaMemcpyAsync() を利用した場合、MVAPICH2-GDR を利用した場合、GPU 間セルフ通信機構を利用した場合、の 3 通りの GPU 間通信の往復の実行速度を測定した。転送するデータのサイズを 1KB から 256MB まで増やしていき、データの転送は、10 回のダミーループの後に 100 回のループを実行し、100 回の平均時間を実行速度として測定した。測定結果を図 10、小メッセージの部分を拡大したものを図 11 に示す。

GPU 間セルフ通信機構を利用した場合の測定結果は全てのデータサイズにおいて cudaMemcpyAsync() を用いた場合より速い結果となった。また、実装のベースである

MVAPICH2-GDR を利用した場合と比較すると約 60% の通信性能となっている。この性能差の要因には、リングバッファへの enqueue / dequeue におけるオーバーヘッドや通信用カーネルがホストに通信の依頼をするときのオーバーヘッド等が挙げられる。MVAPICH2-GDR を利用した場合と比較して 85% 以上の性能に引き上げることを目標とし、チューニングを行っていきたい。

6.2 ライブラリ化

MPI on GPU のプロトタイプ実装として、計算用カーネルが通信用カーネルに通信を依頼するためのリングバッファを構築し、独立した計算用カーネルから実際に通信が開始できるようにライブラリ化を行った。このライブラリを GMPI ライブラリと呼ぶ。GMPI ライブラリを用いた場合の ping-pong のコード例をリスト図 12 に示す。このコードではカーネル関数である send_kernel() 及び recv_kernel() 間で ping-pong を行う。GMPI ライブラリの実装にあたり、通信の依頼に関するデータ等をまとめるための構造体として GMPI_Context 構造体を定義した。現状、このコンテキストに含まれるのはデバイスメモリのリングバッファのみだが、今後必要なパラメータが発生した場合の拡張を考慮している。このライブラリの関数である GMPI_Isend() や GMPI_Irecv(), GMPI_Wait() の第一引数には GMPI コンテキストを指定し、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使用できる。GMPI の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おり、現状の実装は次のようになっている。blockIdx = 0 かつ threadIdx = 0 のスレッドがコンテキ

ストに含まれるリングバッファを介して、計算用カーネルから通信用カーネルに Attribute を転送する。Attribute の内容は MPI の関数を指定するフラグ 及び MPI の関数に指定する引数としている。コード例の src や dst は、cudaMalloc() 等で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 ライブラリの実装には 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。計算用カーネルと通信用カーネルは flag 変数を polling することで同期をとる。依頼を受けた通信用カーネルはホスト上の通信管理用スレッドに依頼するためのリングバッファに Attribute を転送する。通信用カーネルとホスト上の通信管理用スレッドも flag 変数を polling することで同期をとる。ホスト上の通信管理用スレッドは依頼されたフラグが示す MPI の関数に依頼された引数を与えて実際の処理を実行する。

7. おわりに

7.1 まとめ

本研究では GPU 間セルフ通信機構に関する調査及び予備実験を行い、有用性の調査・考察を行った。

Attribute の転送に関する予備評価では、ホストから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依頼することに対し困難が予想されることがわかった。しかしながら、DP を用いたカーネル関数を呼び出す場合のコストよりは小さいオーバーヘッドで呼び出せることが期待できる。このことから、DP との組み合わせに対し提案する通信機構が有効であると考えられる。

カーネル関数間の同期に関する予備評価では、polling による同期、atomic 関数による同期、CUDA Event による同期の3つの手法を提案し、polling による同期及び atomic 関数による同期を実装し、性能評価を行った。どちらの同期手法を用いても、通信依頼状況の flag 及び Attribute ポインタのみで依頼を行えば、片道 $2\mu\text{s}$ 弱で依頼が可能であることが期待できる。

通信用カーネルとホスト上の通信管理用スレッドが Attribute をやりとりするために用いるリングバッファを実装し、ノードを跨ぐ GPU 間通信の動作確認を行った。現状、実装のベースである MVAPICH2-GDR を利用した場合の約 60%の通信性能が得られた。また、プログラミングのしやすさ及び可読性の向上のためにライブラリ化も行った。

7.2 今後の課題

引き続き GPU 間セルフ通信機構を MPI に適用するための予備実験を行い、速度向上を目指していく予定である。この予備実験をベースに、更なるライブラリのリファクタリングや高速化を行っていく予定である。ライブラリの実装後は、姫野ベンチマーク [11] や NAS Parallel

Benchmark[12] といった HPC ベンチマークや、実アプリケーションに対し提案機構を適用することで、実用性の調査を行う。更に、MPI の他に、ノードを跨ぐ GPU 間通信の通信レイテンシの低減を目的として、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)[13] にも本機構を適用し、ライブラリの実装を行っていく予定である。最終的には、GPU 間セルフ通信機構を 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・比較を行いたい。

謝辞

本研究の一部は文部科学省特別経費 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 (online), 入手先 <http://top500.org/>.
- [2] 筑波大学計算科学研究センター : HAPACS ベースクラスター (online), 入手先 <http://www.ccs.tsukuba.ac.jp/research/project/hapacs/cluster>
- [3] CUDA C Programming Guide (online), 入手先 <http://docs.nvidia.com/cuda/cuda-runtime-api/index.html>.
- [4] Message Passing Interface (MPI) Forum Home Page (online), 入手先 <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 $\leftrightarrow$ GPU messaging queue (online), 入手先 <https://code.google.com/p/cuqu/>.
- [7] cuos Offloaded System services for CUDA (online), 入手先 <https://code.google.com/p/cuos/>.
- [8] MVAPICH2 2.0.1 User Guide (online), 入手先 <http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.0.1-userguide.html>.
- [9] NVIDIA Corp. : NVIDIA GPUDirect. (online), 入手先 <http://developer.nvidia.com/gpudirect>.
- [10] Mellanox GPUDirect RDMA User Manual Rev 1.0 (online), 入手先 http://www.mellanox.com/related-docs/prod_software/Mellanox_GPUDirect_User_Manual_v1.0.pdf.
- [11] 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), 入手先 <http://accr.riken.jp/2145.htm>
- [12] NAS Parallel Benchmark (online), 入手先 http://mikelab.doshisha.ac.jp/dia/smpp/01_bench/naspara.html
- [13] 埴 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 クラスターの構築, 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).