MPIを埋め込み可能な GPUプログラミングフレームワークの検討

三好健文† 近藤正章† 入江英嗣†
吉永努† 本多弘樹†

GPU を持つ複数の計算ノードで構成されるクラスタ計算機で，所望の計算を効率良く実行するためには，複数の GPU に効率良く処理を割り当てる必要がある．クラスタ計算機内の複数ノードの GPU を使用するプログラムは，GPU 上の処理を記述するコードとデータ通信処理を行う CPU のコードに分断される．分断されたコードは，プログラマにとって見通しが悪く，また，機械的な最適化や形式検証を難しくする．そこで，GPU 同士のデータ授受を見通しよく記述できるようにするために，GPU コード中に **MPI を埋め込み可能なプログラミングフレームワーク**を提案する．埋め込まれた MPI 処理は，コンパイル時に CPU への MPI 処理の要求に変換され，実行時に CPU 上のランタイムルーチンによって適切に処理される．本稿では，まず，提案するプログラミングフレームワークにより，複数の GPU を用いるプログラムの記述が容易になることを示す．次に，このフレームワークを実現するためのコード変換およびランタイムルーチンの設計について述べる．また，提案するフレームワークを用いたプログラムの実行時間をプログラマが CPU コード中に MPI 処理を直接記述したプログラムの実行時間と比較し，実行性能が遜色ないことを示す．

A Study of GPU Programming Framework to Provide Embedded MPI

Takefumi MIYSOHI ,† Masaaki KONDO ,† Hidetsugu IRIE ,†
Tsutomu YOSHINAGA † and Hiroki HONDA †

For leveraging multiple GPUs in a cluster system, it is necessary to assign application tasks to multiple GPUs and execute those tasks with appropriately using communication primitives to handle data transfer among GPUs. In current GPU programming models, communication primitives such as MPI functions cannot be used within GPU. Instead, such functions should be used in the CPU code. Therefore, programmer must handle both GPU and CPU codes for data communications. This makes GPU programming and its optimization very difficult.
In this paper, we propose a programming framework which enables programmers to use MPI functions within GPU kernels. Our framework automatically transforms MPI functions written in a GPU kernel into runtime routines executed on the CPU. With this framework, programmability and readability of programs will much improve. We evaluate the performance and overhead of the framework. The result shows that the proposed framework achieves comparable execution performance with a MPI code written by hand in traditional way.

1. はじめに

GPU(Graphics Processing Unit) は，科学計算の高速化のために広く使われるようになってきている．2010 年 11 月に発表されたスーパコンピュータのランキングである TOP500[1] では上位 5 件のうち 3 件が GPU を備える複数台の計算ノードによる GPU クラスタ計算機である．GPU クラスタ計算機では，タスクを各計算ノードに分割して並列に実行させ，さらに各ノードでプログラムの適切な部分を GPU を活用して高速に実行させることで，高い演算性能を得ることができる．大規模なタスクを複数台の計算ノードに分割し，さらに，それぞれの計算ノードにおいて，分割したタスクの一部を GPU で高速に実行することで，Jacobsen らは CFD を，Komatitsch らは地震波の伝達の解析に適用し，高い計算性能が得られることを示している[2)3)]．また Babich らは，QUDA ライブラリを GPU クラスタ計算機上で並列化している[4)]．GPU はネットワークに直接アクセスできないため，複数の GPU を用いて所望の処理を実行する場合には，同じノードの CPU 上のプログラムを介して GPU 間のデータ共有を実現しなければならない．前述の GPU クラスタ計算機上での実装には，いずれもノード間通信に MPI(Message Passing Interface) による通信が用いられている．

† 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

© 2011 Information Processing Society of Japan

しかし，この GPU クラスタ計算機上で動作するプログラミングでは，プログラマは GPU コード☆，MPI 処理を実行する CPU コード☆☆，GPU と CPU 間のデータ授受のコードの 3 種類のコードおよびデータ構造を管理しなければならない．これはプログラマにとって負担である．そこで，我々は，GPU コード中に GPU 同士のデータ授受を記述できるようにするために，MPI を直接記述できるようにする **MPI を埋め込み可能な GPU プログラミングフレームワーク**を提案する．このフレームワークを利用する場合，GPU 間のデータ通信を行うために CPU 上の MPI プログラムを記述する必要がなくなる．また，MPI 関数の実行に伴う CPU-GPU 間のデータコピーを明示的に記述する必要もない．従って，GPU コードのみに着目してプログラムを記述することができ，見通しがよくなるためプログラミングコストを軽減させることができる．

提案するフレームワークでは，本来通信機能を有さないアクセラレータである GPU の上で，MPI 通信を記述できるようにする．これは，GPU に限らず，複数のアクセラレータに処理を分割して並列に実行するためのプログラミングパラダイムの一つとなりえる．

このプログラミングフレームワークを実現するためには，GPU コード内に記述された MPI 関数の実際の処理を CPU コード上での処理に置き換える仕組みが必要である．そこで，GPU コードからの処理のリクエストを取り扱う実行モデルを設計する．設計する実行モデルは，GPU コード中に記述された MPI 関数の実行を CPU コードで代替する．また，CPU コードが MPI 処理を実行している間，GPU コードは処理を一旦停止ししておき，CPU コードで実行された MPI 関数終了後，処理を再開する．実行モデルの実現に伴うデータの管理や制御をプログラマから隠蔽するため，与えられたプログラムから，静的なコード変換によってこれら一連の処理を行うために必要なコードを生成する．

本稿の構成は次の通りである．2 節で GPU クラスタ計算機上のプログラムを簡潔に記述可能な，我々の提案する，GPU プログラミングフレームワークを提案する．3 節で，このフレームワークを C 言語で実現するための実行モデルおよびコード変換手法を示す．4 節で，提案するフレームワークを利用して記述されたプログラムのオーバヘッドを測定する．

2. MPI を埋め込み可能な GPU プログラミングフレームワーク

図 1 は，GPU をもつ複数の計算ノードによって構成されるクラスタ計算機を図示したものである．一般的な GPU クラスタ計算機では，ノード内の GPU-CPU 間の通信および NIC を介したノード間の CPU-CPU の通信を行うことができるが，GPU から直接 NIC を

☆ 各ノード内の GPU で実行される部分プログラム
☆☆ 各ノード内の CPU で実行される部分プログラム

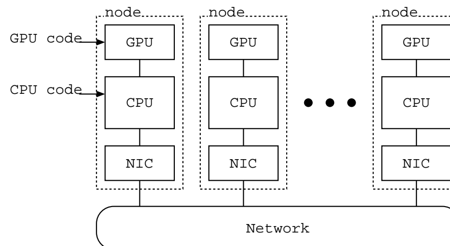

図 1 複数の GPU を持つノードによる GPU クラスタ計算機
Fig. 1 A GPU Cluster Computer that has multiple GPUs

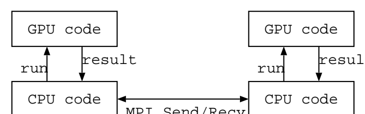

図 2 複数ノード上の GPU を使う場合のプログラミングモデル
Fig. 2 Programming model to use GPUs in multiple node

```
1:  for(i = 0; i < NN; i++){
2:    CC[i] = C[i];
3:    if(i != 0   ){ CC[i] += C[i-1]; }
4:    if(i != NN-1){ CC[i] += C[i+1]; }
5:    CC[i] = CC[i] / 3;
6:  }
```

図 3 データ列の各要素が隣接する要素との平均を求めるプログラム
Fig. 3 Program to average three neighboring elements in a data array

介して他ノードの GPU にアクセスすることはできない． この，GPU クラスタ計算機上の GPU 間でデータ共有したい場合には，**図 2** に示すように，CPU コードの MPI 処理によってデータを共有した後，各 CPU コードが独立にノード内の GPU コードに反映させる．

例として**図 3** に示すプログラムを考える．これは，サイズ `NN` のデータ列において，各要素が隣接する要素との平均を計算するプログラムである． ここで，計算の対象となるデータ `NN` の配列中の要素を，GPU クラスタ計算機を構成する `nCPU` ノードに分割し並列に実行することで高速化することを考える．この時，各ノードに割り当てられた端の要素の値を更新するためには，他ノードに割り当てられたデータが必要であり，その授受に MPI を用いる必要がある．

図 3 に示したプログラムを GPU クラスタ環境を用いて実行するための GPU コード☆☆☆と CPU コードを，それぞれ**図 4** と**図 5** に示す．プログラムは，各ノードが `N=NN/nCPU` ずつの要素の値を計算するとし，GPU で実行するスレッド数は定数 `K` である．CPU コードでは，GPU 上に `N+2` のサイズの配列を確保している．GPU コード上で `C` で指し示されるこの領域の `C[0]` と `C[N+1]` は，それぞれ隣接するノードの `C[N]` および `C[1]` を共有するための領域である．図 5 中の 2 から 6 行目および 7 から 11 行目で MPI 通信によっ

☆☆☆ 本稿では NVIDIA を用いて GPU コードを実行することを想定し，CUDA で記述する．

ⓒ 2011 Information Processing Society of Japan

```
1: __global__ void
2: kernel(float *C,float *CC,int N,int k) {
3:   for(int i = 0; i < k; i++){
4:     int idx = threadIdx.x + i * K + 1;
5:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
6: }}
```

図 4 図 3 の処理を行うための GPU コード
Fig. 4 GPU code for implementation of the program shown in Figure 3

```
 1: cudaMemcpy(&Cd0[1], Ch, sizeof(float) * N,
                                   cudaMemcpyHostToDevice);
 2: if(rank != nCPU-1)
 3:  MPI_Send(&Ch[N],1,MPI_FLOAT,rank+1,0,MPI_COMM_WORLD);
 4: if(rank != 0)
 5:  MPI_Recv(&v,1,MPI_FLOAT,rank-1,0,MPI_COMM_WORLD,&status);
 6: cudaMemcpy(&Cd0[0],&v,sizeof(float),
                                   cudaMemcpyHostToDevice);
 7: if(rank != 0)
 8:  MPI_Send(&Ch[1],1,MPI_FLOAT,rank-1,0,MPI_COMM_WORLD);
 9: if(rank != nCPU-1)
10:  MPI_Recv(&w,1,MPI_FLOAT,rank+1,0,MPI_COMM_WORLD,&status);
11: cudaMemcpy(&Cd0[N+1], &w, sizeof(float),
                                   cudaMemcpyHostToDevice);
12: kernel<<<1, K>>>(Cd0, Cd1, N, N/K);
13: cudaMemcpy(Ch, &Cd1[1], sizeof(float) * N,
                                   cudaMemcpyDeviceToHost);
```

図 5 図 4 の GPU コードで必要な通信を処理する CPU コード
Fig. 5 CPU code to perform data transfer operations required for the GPU code shown in Figure 4

て隣接ノードとデータ授受を行い C[0] と C[N+1] を設定している．これにより GPU コードでは図 4 の 4 から 5 行目の簡潔なコードで演算が実行できる．しかし，そのために，CPU コードでは使用しないデータ授受のための MPI 処理を記述しなければならない．図 5 中の MPI 処理の意図を理解するためには，図 4 と図 5 の両方およびデータの対応関係を把握する必要がある．

GPU コードの実行に必要なデータ共有のための MPI 処理を CPU コードとして記述することには，二点の問題がある．一点目はプログラマが GPU 上で実行するプログラムのデータ転送を CPU 上で実行する MPI 送信/受信に置換する必要がある点である．設計時に考慮していたノード数より多数のノードを利用してプログラムを実行する場合に，GPU コードと CPU コードの両方を記述しなおさなければならない．二点目は，GPU コードと CPU コードがデータコピーによって分離されるため包括的な最適化および形式的検証が困難である点である．機械的に図 5 に示したコードを解析する場合，GPU 上のデータを指し示す Cd と CPU 上のデータを指し示す Ch が同一であることを解析器が知らなければならない．しかし，これは一般的に困難である．機械的な最適化を可能にするためには種々の制約を加えなければならない．

これらの問題を解決するために，MPI を埋め込み可能な GPU プログラミングフレームワークを提案する．提案するプログラミングフレームワークは，**図 6** に示す GPU コード同士が MPI の仕組みを利用してデータを直接共有するプログラムの記述を可能にする． 提案するプログラミングフレームワークを用いて図 3 に示したプログラムを記述することを考える．GPU コードを**図 7** に，CPU コードを**図 8** に示す．図 7 内の cuda_mpi_send および cuda_mpi_recv がそれぞれ MPI 処理の実行に相当する．GPU コード内に MPI 関数呼び出し及びその条件を直接記述することができるため見通しがよい．特に，転送の対象となるデータとして，CPU コード上の変数ではなく，実際に共有したい GPU コード上の変数を指定することができるため，プログラマの意図がコードに示しやすい．また，CPU コードでは MPI 関数呼び出しを取り扱う必要がなく，単に GPU カーネルコード呼び出しのみを記述すればよく，図 5 と比べて，簡潔な記述ができる．

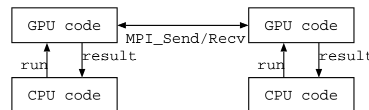

図 6 MPI 埋め込み GPU プログラミングフレームワークでのプログラミングモデル
Fig. 6 Programming model with the proposed programming framework to provide embedded MPI into GPU

```
 1: __global__ void
 2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
 3:   if(rank != nCPU-1)
 4:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
 5:   if(rank != 0)
 6:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
 7:   if(rank != 0)
 8:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
 9:   if(rank != nCPU-1)
10:     cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
11:   for(int i = 0; i < k; i++){
12:     int idx = threadIdx.x + i * K + 1;
13:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
14: }}
```

図 7 提案するフレームワークを用いる場合の GPU コード (MPI コードを埋め込み可能)
Fig. 7 An example of the GPU code with the proposed framework, using embedded MPI code

```
1:   cudaMemcpy(&Cd0[1], Ch, sizeof(float) * N,
                                cudaMemcpyHostToDevice);
2:   kernel<<<1, K>>>(Cd0, Cd1, rank, nCPU, N, N/K);
3:   cudaMemcpy(Ch, &Cd1[1], sizeof(float) * N,
                                cudaMemcpyDeviceToHost);
```

図 8 提案するフレームワークを用いる場合の CPU コード
Fig. 8 An example of the CPU code with the proposed framework

3. 提案するフレームワークの実装

MPI を埋め込み可能な GPU プログラミングフレームワークを実現するためには，GPU コード上に記述された MPI 処理を CPU 上で実行できるように変換する機構が必要になる．本稿ではこの機構を C 言語で実装する手法を示す．他の実装手法として，独自に GPU のドライバと，それに対応するランタイムを作成するという手法が考えられる．しかし，この手法は実装にかかるコストが大きいという問題点がある．C

ⓒ 2011 Information Processing Society of Japan

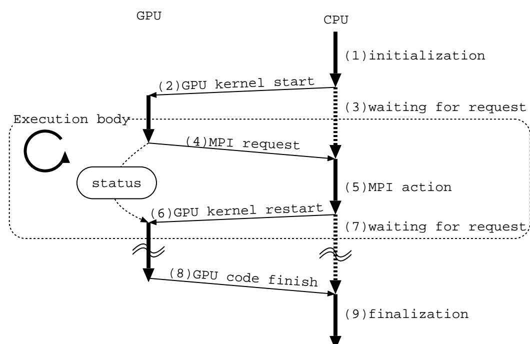

図 9 MPI が埋め込まれた GPU プログラムの実行モデル
Fig. 9 An execution model for the GPU code with embedded MPI

言語による実装であれば，CUDA や OpenCL，その他への移植は容易である．本稿では，GPU コードは CUDA SDK 2.3，MPI 環境には OpenMPI 1.3.3 を用いて，提案するプログラミングフレームワークの実装手法を述べ，実現可能であることを示す．

3.1 実行モデル

提案する MPI を埋め込み可能な GPU プログラミングの実行モデルを説明する．**図 9** に実行モデルを示す．各処理を以下に述べる．

(1) CPU コードは必要な CPU 上，GPU 上のデータを初期化する．
(2) GPU コードを起動する．
(3) CPU コードは GPU コードの開始後 GPU コードからの MPI 処理のリクエスト発行を待機する．
(4) GPU コードから MPI 処理を CPU にリクエストする．GPU コードは再実行のために必要なステータス情報を退避させ，一時実行を終了する．
(5) CPU コードは受理した MPI 処理のリクエストを処理する．
(6) GPU コードを再起動する．GPU コードでは退避させたステータス情報を復帰し，MPI 処理実行後の処理から実行を再開する．
(7) CPU コードは再び MPI 処理のリクエスト発行を待機する．
(8) GPU コードは処理が完了すると，その旨を CPU コードに通知し，実行を終了する．
(9) CPU コードは後処理として確保したメモリ領域を解放する．

GPU でタスクを処理している間は，(4) から (7) の処理が繰り返される．ここで，MPI 処理を行うために CPU で処理している間の中断から再実行までのブロックされた期間は，プログラマによって記述された GPU コードに対しては，隠蔽される．従って，プログラマは図 9 中に点線で囲った GPU 処理および MPI 処理を実行の本体として仮想的に取り扱うことができる．また，MPI 処理のために GPU コードの実行が中断している間に，他の GPU コードを実行することもできる．すなわち通信と GPU 上での計算のオーバラップが自然に実現される．

表 1 MPI 処理を要求するための構造体のメンバ変数
Table 1 Members of structure for MPI request

変数名	型	用途
addr	void *	データ列の先頭アドレス
size	int	データ列の長さ (バイト)
rank	int	MPI 通信の対象ノード
tag	int	MPI 通信のタグ
comm	MPI_Comm	MPI コミュニケータ種別
request	request_t	MPI 処理の種別

3.2 データ構造

ノード間通信を行うために，CPU に処理させるリクエスト情報をまとめた MPI リクエスト・データおよび，ノード間通信終了後に処理を復帰するための GPU ステータス・データの二つがある．

3.2.1 MPI リクエスト・データ

GPU コードは，**表 1** に示すメンバ変数を持つ構造体のデータによって CPU コードに MPI 処理の実行を要求する．型 request_t は，送信や受信など CPU コードに所望の MPI 関数呼び出しを指示するための定数を示す列挙型である．現段階の実装では，MPI 処理をリクエストするための MPI_Send，MPI_Recv，MPI_Barrier の 3 種類に加え，GPU コードの処理が完了したことを CPU コードに通知するための GPU_DONE の計 4 種類の値を取り得る．MPI 処理の対象とするデータはデータ列の先頭アドレス (addr) とサイズ (size) で指定する．MPI リクエスト・データの肥大化を抑止するため，ノード間で共有するデータ列はすべて MPI_CHAR として扱うこととし，データサイズはバイト単位で与えることとする．

MPI リクエストデータは GPU と CPU で共有する必要があるため，CPU コードで CPU 側および GPU 側のデータ領域を確保しなければならない．データ領域を確保するコードおよび，確保した GPU 上の MPI リクエストデータ領域のポインタを GPU コードに引き渡すコードは機械的にコード変換で追加，挿入される．

3.2.2 GPU ステータス・データ

MPI 関数処理のため中断した GPU での計算処理を復帰させるために GPU で保持すべき情報を保持しなければならない．保持すべき情報とは，中断した処理状態の続きから実行を開始するためのコード位置の情報と処理を中断した状態に正常に復帰するために退避されるレジスタ変数群から成る．各関数で退避が必要な変数は，名前が衝突しないように，関数という名前空間を保持したまま退避先を確保する．

中断/再実行ためにコード位置の情報を取り扱う方法として，適宜コード中にラベルを指定しておき，そのラベルをポインタ変数に格納する方法が簡単である．しかし，CUDA SDK 2.3 および OpenMPI 1.3.3 の環境では，ラベル変数を変数で保持し，取り扱うことができない．そのため，復帰すべきコード位置に処理

ⓒ 2011 Information Processing Society of Japan

```
  1: __global__ void
  2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
* 3:  restore();
  4:  switch(*_status) {
* 5:   case 0:
  6:   if(rank != nCPU-1){
  7:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
+ 8:     *_status = 1; store(); goto GPU_CODE_END; }
* 9:   case 1:
 10:   if(rank != 0){
 11:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
+12:     *_status = 2; store(); goto GPU_CODE_END; }
*13:   case 2:
 14:   if(rank != 0){
 15:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
+16:     *_status = 3; store(); goto GPU_CODE_END; }
*17:   case 3:
 18:   if(rank != nCPU-1){
 19:     cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
+20:     *_status = 4; sotre(); goto GPU_CODE_END; }
*21:   case 4:
 22:   for(int i = 0; i < k; i++){
 23:     int idx = threadIdx.x + i * K + 1;
 24:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
 25:   }}
 26:  cuda_mpi_finalize();
+27:  GPU_CODE_END:
 28: }
```

図 10 switch–case 文でステートを分断した GPU コード
Fig. 10 An example of the GPU code divided into multiple states by switch–case statement

を遷移させることができるように GPU コードを文単位で switch–case 文で分割し，その case ラベルの値を実行中の位置を示す値として用いる．

3.3 コード変換

コード変換では，プログラマによって GPU 処理中に埋め込まれた MPI 処理を実行するために，CPU 側および GPU 側で処理しなければならないコードを生成する．提案するコード変換器はソースコードからソースコードへの変換器である．

3.3.1 GPU コードのコード変換

GPU 側でのコード変換では，プログラマによって記述された GPU コードに (1)MPI を処理するために必要な CPU と GPU で共有すべき MPI リクエストのポインタを引き渡すためのスタブの追加，(2)MPI 処理を実行するために GPU コードの実行を停止した後に復帰するための switch–case 文での分割，および，(3)MPI 処理呼び出し後の GPU コード中断と再実行のための準備コードの挿入である．(1) によりユーザは，提案するフレームワークを利用するために必要な変数の初期化などの「おまじない」コードの記述を省くことができる．

本節では，与えられた GPU コードに (2) と (3) を適用する手法を提案するフレームワークに基づいて記述された GPU コード (図 7) が変換される例に基づいて説明する．**図 10** に変換後の GPU コードを示す． 図 10 中の行番号の前に*を付記している行が，switch–case 文でコードを分断するために追加あるいは変更された行である．7，11，15，19 行目の MPI 処理を実行するコードによって GPU コードが中断された後の再開時に，正しく次の処理に遷移できるように，分岐先として case ラベルが 9，13，17，21 行目に挿入される．

```
...
  case 0:
    i = 0;
  case 1:
  case 2:
    while(i < N){
      switch(*_status){
      case 1:
        cuda_mpi_barrier(MPI_COMM_WORLD);
        *_status = 2; goto GPU_CODE_END;
      case 2:
        i++; *_status = 1;
      }
    }
  case 3:
...
```

図 11 ブロック内に階層的に構成される switch–case 文の例
Fig. 11 An example of a hierarchical switch–case statement for block structure

また，+を付記している行は GPU コードの中断と再開のために挿入された行を示す．MPI 処理実行時に GPU コードを停止させるために `goto GPU_CODE_END;` によって強制的に GPU コードを終了する．ただし，MPI 処理終了後に再開できるように `*_status` 変数に実行を再開すべき箇所のラベルの値を保存し，また一時変数の保存を行うための関数呼び出しのコードを挿入している (8，12，16，20 行目)．

switch–case 文は階層的に複文を持つプログラムに対しても適切に付与することができる．**図 11** は，GPU コードにおいて while によるブロック内で繰り返し MPI 処理 (cuda_mpi_barrier) を実行するコードに対して case ラベルを付与する例を示す． 新しいブロック文が出現した場合，その中に新しい switch–case 文を生成し，case ラベルを継続して付与する．そして，ブロック構文内のすべての case ラベルを，そのブロック構文の前に置くことによって，正しく復帰先を保存できる．ただし，図 7 に示したコード中の 6，7，9，10 行目のようにブロック内の文が 1 つだけの場合には，冗長な switch–case による分岐を抑止するために，図 10 のように階層的な switch–case 文の挿入は省略する．また，GPU コード内での関数呼び出しは，case ラベルを付与する時点でインライン展開しておくことで対処できる．CUDA では再帰的な関数呼び出しは禁止されているため，インライン展開することによる制約は何ら追加されない．

以上のように，復帰可能な状態を文の単位で case に分割扱っている．従って，単一の文の中に複数の代入式がある場合には正しく復帰することができない．例えば for ループでは，図 11 の例のようにブロック内を switch–case 文で分割しようとしても，初期化の代入式が複数回実行されて意図した通りにコードが実行できない．そのような単一の文の中に複数の代入式を持つ文は，コード変換時に前もって分解することで対処する．

3.3.2 CPU コードのコード変換

CPU コード側では，(1)GPU とのやり取りに用いるリクエスト構造体の領域の確保と初期化および解放と，(2)3.1 節で述べた実行モデルを実現するための

ⓒ 2011 Information Processing Society of Japan

```
 1:  do{
 2:    kernel_stub<<<1, N>>>
          (Cd, rank, nCPU, info->info_dev, info->status_dev);
 3:    cudaMemcpy((void*)info->info_host,
                  (void*)info->info_dev,
                  sizeof(gpu_info),
                  cudaMemcpyDeviceToHost);
 4:    switch(info->info_host->request){
 5:    case MPI_SEND:
 6:      cuda_mpi_send((gpu_info*)info->info_host); break;
 7:    case MPI_RECV:
 8:      cuda_mpi_recv((gpu_info*)info->info_host); break;
 9:    case MPI_BARRIER:
10:      MPI_Barrier(MPI_COMM_WORLD); break;
11:    }
12:  }while(info->info_host->request != DONE);
```

図 12 CPU コードに挿入される MPI リクエスト処理ループ
Fig. 12 An example of the inserted loop to operate MPI request

表 2 PC クラスタを構成する各ノードの諸元
Table 2 Specification of each node in PC cluster

CPU	Intel(R) Xeon(R) CPU W3520 2.7GHz
メモリ	6GiB
OS	CentOS Release 5.3 (Linux x86_64 2.6.18-128)
ネットワーク IF	Intel(R) PRO/1000 NIC
GPU	NVIDIA Tesla C1060

リクエスト処理ループに相当するコードの生成と挿入，を実行する．GPU とのやり取りに用いるためには CPU-GPU でデータをコピーできる領域を確保しなければならない．CUDA では GPU 上のメモリ領域の確保は CPU コードで行わなければいけない．その確保および初期化は機械的に挿入できる．

3.1 節で述べたように CPU コードは，GPU コードを実行した後，MPI リクエストを待機するループに入る．**図 12** に，この実行モデルを実現するためのコードを示す．このループ内では，GPU コード中で指定された MPI リクエスト・データの `request` フィールドの値によって対応する MPI 処理の実行に分岐する．MPI 処理終了後は再び GPU コードを実行する．

4. 評　　価

提案する MPI 埋め込み可能な GPU コードを用いたプログラムの実行オーバヘッドを評価する．本節では，まず実験に用いた評価環境について説明する．次に，提案するフレームワークを用いた場合の MPI 処理にかかるオーバヘッドと，プログラムの実行時間の評価結果を示す．

4.1 評 価 環 境

実行時間の評価には，Gigabit Ethernet で接続された NIVIDIA の GPU Tesla C1060 を有する 16 台の計算ノードで構成される PC クラスタを用いる．各ノードの諸元は**表 2** の通りである．

コード変換はソースコードからソースコードへの変換器であり，出力は CUDA のコードである．変換後のコードは CUDA 2.3 SDK および OpenMPI-1.3.3-gnu64-4.1.2 によって，ターゲットである CPU コードおよび GPU コードにコンパイルされる．

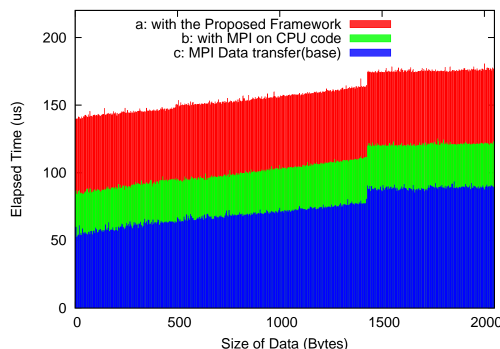

図 13 MPI 処理を行うプログラムの実行時間
Fig. 13 Execution time for perform MPI operation

4.2 MPI 処理にかかるオーバヘッド

GPU コードに埋め込まれた MPI 処理リクエストのプリミティブとしてのオーバヘッドを測定する．測定用のプログラムでは，16 台のノードで，それぞれランク (`rank+1`) `& 0xF` のノードに指定したサイズのデータを送信し，ランク (`rank-1`) `& 0xF` のノードからデータを受信する．プログラムを CPU コード中で GPU-CPU 間のデータコピーと MPI 転送を記述する場合 (“a: with MPI on CPU code”) と提案するフレームワークを利用して GPU コード中に MPI 転送埋め込む場合 (“b: with the Proposed Framework”) の実行時間を測定する．また，MPI でのデータ通信だけを行なうプログラム (“c: MPI Data transfer(base)”) の実行時間も測定した．結果を**図 13** に示す．横軸は通信の対象とするデータのサイズ (バイト数) である．

図 13 の結果は，(a) と (b) にそれぞれデータサイズに依存しない同程度のオーバヘッドが上乗せされていることを示す．ここで GPU コードが起動される回数に着目する．GPU コードが起動される回数は MPI 通信だけを行うプログラム (c) では 0 回，CPU コード中に MPI 処理を記述する場合 (a) では 1 回，そして提案するフレームワークを用いて記述する場合 (b) では 3 回である．(a) の (c) に対する増分と，(b) の (c) に対する増分は，ほぼ 1:3 である．すなわち，GPU コードの起動にかかるオーバヘッドが，直接実行時間の増加に反映され，そのオーバヘッドは約 20μ 秒である．

GPU コードで実行する処理が小さい場合には，この GPU コード起動のオーバヘッドが実行時間の増加に直接つながる．このオーバヘッドを削減する手法としては，複数の MPI 処理リクエストをまとめることで GPU コードの起動回数を削減する最適化が考えられる．

4.3 プログラム実行時間の評価

図 3 に示した配列の要素の平均を取るプログラムを，CPU コードで MPI 関数呼び出しを記述する場合 (“with MPI on CPU code”) と，提案するプログラ

ⓒ 2011 Information Processing Society of Japan

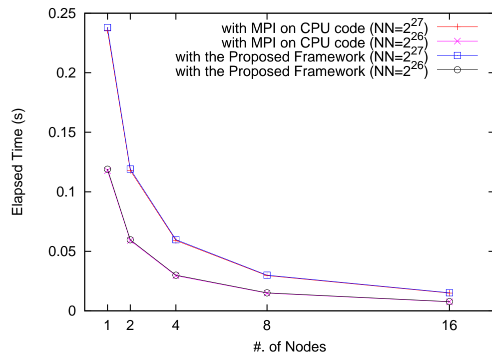

図 14 MPI コードをプログラマが CPU コードとして記述した場合と提案するフレームワークを用いた場合の実行時間
Fig. 14 Execution time of the code in which MPI operations are written as CPU code and the code written with the proposed framework

ミングフレームワークを用いて記述する場合 (“with the Proposed Framework”) の実行時間を比較する．対象とするデータサイズは NN を 2^{26}，2^{27} とし，GPU では 512 スレッドで実行する (K=512)．**図 14** に測定結果を示す．

図 14 の結果は提案するフレームワークを用いて記述したコードの実行性能が期待する通りノード数の増加にスケールすることを示している．また，NN=2^{27} のデータに対し 16 台のノードで処理する場合，提案するフレームワークを用いて記述したプログラムの MPI コードを CPU コードに直接記述するプログラムに対する実行時間の増加は 1.6%である．すなわち，図 14 の結果は，提案するプログラミングフレームワークを用いて記述されるプログラムの実行性能が，MPI 処理コードを明示的に CPU コードに記述したプログラムの実行性能と比較して遜色ないことを示す．

5. 関連研究

GPU クラスタ計算機を活用するために，CUDA と MPI の容易にするプログラミングフレームワークを実現するために，Leung らは，R-Stream から複数 GPU アクセラレータへのソースコードからソースコードへの変換器を提案している[5]．このコンパイラでは，階層的な CPU 間，複数 GPU，GPU 内での並列化と分離を達成している．また，Lowlor は cudaMPI を提案している[6]．cudaMPI では GPU コードのために CPU コード中に記述すべき MPI 通信コードを簡潔に記述することができる．この二つの研究では，あくまで CPU コード中に GPU コードのための MPI 処理を書く必要があり，GPU コードそのものに自然な形でデータ共有を実現するための MPI 処理の記述は実現できない．

GPU クラスタ計算機に限らず，GPU コードと CPU コードを柔軟にあるいは差異をプログラマから隠蔽するための独自言語，コンパイル環境の研究は多い．HPCPE[7] では，CPU と GPU ベースのハイブリッドな並列計算プログラミング環境を提供している．HPCPE では与えられたプログラムを提唱する Two Level Model である計算タスクと制御タスクに分割し，それぞれ CPU と GPU に分割する．既存の並列プログラミングフレームワークである OpenMP や MPI の枠組みで GPU を活用するためのフレームワークとして，OpenMP から GPU コードに変換する手法[8][9] や “軽量な”OpenMP と “重厚な”MPI の両方を使って複数 GPU を利用する[10]，および GPU や FPGA にタスクレベルで分割[11] がある．さらに，既存の言語のフレームワークを利用して独自の言語を実現する DSL を，アクセラレータ用のプログラム記述に活用する研究もある[12][13]．これらは，プログラマにとって馴染み深いプログラミングインターフェイスを提供する．これらの文献では複数アクセラレータの活用については言及されていないが，提案する MPI 埋め込み手法を用いることで，これらに対してデータ授受の仕組みを提供するができる．

GPU クラスタ上でプログラムを実行する際に，処理のスケジューリングにより通信と GPU 計算をオーバラップさせることで高い計算性能が得られる[14][15][16]．提案するフレームワークを用いた場合にも GPU コードの呼び出し順を考慮することで，通信と GPU 計算のオーバラップを実現することができるが，細かな GPU コードと通信処理のスケジューリングは難しい．ただし，提案するフレームワークでは GPU 間の通信が GPU コード内に埋め込まれるため，機械的な最適化の可能性は高まる．

6. まとめ

GPU のコード内に MPI 処理を埋め込み可能なプログラミングフレームワークを提案した．このフレームワークを用いることで，プログラマは CUDA による GPU コードと MPI によるノード間でのデータ授受のコードの二種類のコードを管理する必要がなくなり，プログラミングコストが低減される．またコンパイル時の機械的な解析が容易になり，最適化処理や形式的検証の適用が考えられるようになる．

本稿では，まず，提案するフレームワークを C 言語を用いて実装するための手法を示した．次に，図 3 に示したプログラムを，(1)MPI を CPU コード中に記述する場合と (2) 提案するフレームワークを用いて記述した場合の実行時間を比較した結果を示した．その結果，複数 GPU を活用しうる大規模なデータの計算において，提案するフレームワークを利用する場合の実行時間の増加は 1.6%程度と小さいことが分かった．

今後の課題として，以下の四点を挙げる．一点目は，高性能計算機で一般的とされる Linpack，アクセラレータを含むヘテロジニアス計算環境向けのベンチマーク[17]，および，その他アプリケーションプログラムを提案するフレームワークのもとで記述した場合の記述の容易性および性能を評価することである．二点目は，GPU クラスタ計算機で重要視されている GPU

© 2011 Information Processing Society of Japan

での計算と通信のオーバラップによる最適化を，提案するプログラミングフレームワークに組込むことである．三点目は，一つのノードに複数の GPU を搭載する GPU クラスタ計算機上でのプログラムの実行をサポートすることである．現在はデータ通信に MPI を直接用いているため，ノード内の複数 GPU でのデータ転送はサポートできていない．最後に四点目として，より高水準の抽象化言語のバックエンドとして提案フレームワークを利用することで効率的かつ記述力の高い GPU クラスタ計算機向け開発環境を構築することが考えられる．

参　考　文　献

1) TOP500 Super Computing Sites: TOP500 List - November 2010(1-100), http://www.top500.org/list/2010/11/100.
2) Jacobsen, D. A., Thibault, J. C. and Senocak, I.: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters (2010).
3) Komatitsch, D., Erlebacher, G., Göddeke, D. and Michéa, D.: High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster, *J. Comput. Phys.*, Vol. 229, pp. 7692–7714 (2010).
4) Babich, R., Clark, M. A. and Joó, B.: Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, IEEE Computer Society, pp. 1–11 (2010).
5) Leung, A., Vasilache, N., Meister, B., Baskaran, M., Wohlford, D., Bastoul, C. and Lethin, R.: A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction, *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, New York, NY, USA, ACM, pp. 51–61 (2010).
6) Lawlor, O. S.: Message passing for GPGPU clusters: CudaMPI., *CLUSTER*, IEEE, pp. 1–8 (2009).
7) Chen, Q.-k. and Zhang, J.-k.: A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA, *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, Washington, DC, USA, IEEE Computer Society, pp. 86–89 (2009).
8) Lee, S., Min, S.-J. and Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization, *SIGPLAN Not.*, Vol. 44, pp. 101–110 (2009).
9) 大島聡史, 平澤将一, 本多弘樹: "OMPCUDA：GPU 向け OpenMP の実装", "2009 年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 (HPCS2009)", pp. 131–138 (2009).
10) Noaje, G., Krajecki, M. and Jaillet, C.: MultiGPU computing using MPI or OpenMP, *Proceedings of the Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing*, ICCP '10, Washington, DC, USA, IEEE Computer Society, pp. 347–354 (2010).
11) Tsoi, K. H., Tse, A. H., Pietzuch, P. and Luk, W.: Programming framework for clusters with heterogeneous accelerators, *SIGARCH Comput. Archit. News*, Vol. 38, pp. 53–59 (2011).
12) 中里直人: アクセラレータを活用するためのプログラミング環境, 第 76 回情報処理学会プログラミング研究会, Vol. 3, No. 2, pp. 1–10 (2009).
13) Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A. K., Hanrahan, P., Odersky, M. and Olukotun, K.: Language virtualization for heterogeneous parallel computing, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, New York, NY, USA, ACM, pp. 835–847 (2010).
14) 遠藤敏夫, 額田彰, 松岡聡: 異種アクセラレータを持つ TSUBAME スーパーコンピュータの Linpack 評価, 応用数理, Vol. 20, No. 2, pp. 117–124 (2010-06-25).
15) Phillips, J. C., Stone, J. E. and Schulten, K.: Adapting a message-driven parallel application to GPU-accelerated clusters, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, IEEE Press, pp. 8:1–8:9 (2008).
16) Wang, Q., Ohmura, J., Axida, S., Miyoshi, T., Irie, H. and Yoshinaga, T.: Parallel Matrix-Matrix Multiplication Based on HPL with a GPU-Accelerated PC Cluster, *International Conference on Natural Computation*, Vol.0, pp. 243–248 (2010).
17) Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V. and Vetter, J. S.: The Scalable Heterogeneous Computing (SHOC) benchmark suite, *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, New York, NY, USA, ACM, pp. 63–74 (2010).

ⓒ 2011 Information Processing Society of Japan