情報処理学会論文誌　コンピューティングシステム　Vol.6　No.4　105–116 (Oct. 2013)

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

島 圭吾[1,a)]　吉見 真聡[1]　三好 健文[2]　近藤 正章[1]　入江 英嗣[1]　本多 弘樹[1]　吉永 努[1]

受付日 2013年4月2日, 採録日 2013年7月29日

概要：GPU 搭載 PC クラスタで動作するプログラムは，GPU 上の処理を記述するコードと通信処理を行う CPU のコードで構成される．GPU コードは並列化されたアルゴリズムを高速に実行し，CPU はノード間の通信処理を担当する．ノード間通信には MPI の利用が一般的だが GPU コードには記述できないため，並列化の効果を引き出すためには，プログラマは CPU と GPU のデータの移動を考えつつ CPU コードと GPU コードを並行して実装することになる．そこで，GPU 間のデータ通信に関わるプログラミングコストを低減させるために，MPI を埋め込み可能な GPU プログラミングフレームワーク “FLAT” を提案する．FLAT を用いることで GPU コードに MPI 関数が記述できるようになるため，GPU 間で転送されるデータが明確化される．本論文では，まず，FLAT の実行モデルと実装方法について述べる．その後，Livermore ループ Loop18，オプティカルフロー計算の 2 つの実プログラムを用いて FLAT の有効性と実行性能を示す．実験の結果，GPU コードの計算粒度が粗粒度の場合，FLAT の利用による性能低下率は，3%以下であることが確認された．

キーワード：GPGPU，クラスタシステム，MPI，プログラミングフレームワーク

FLAT: An MPI Friendly GPGPU Programming Framework for GPU Clusters

Keigo Shima[1,a)]　Masato Yoshimi[1]　Takefumi Miyoshi[2]　Masaaki Kondo[1]
Hidetsugu Irie[1]　Hiroki Honda[1]　Tsutomu Yoshinaga[1]

Received: April 2, 2013, Accepted: July 29, 2013

Abstract: A program for a PC cluster which equips GPUs consists of two types of code, for GPUs and for CPUs. The GPU code executes parallelized algorithms to introduce high speed computing supported by a CPU code which performs communication with other nodes. Although MPI library is commonly utilized to transfer data in the CPU code, MPI functions can not be written in the GPU code. Programmers are forced to implement CPU and GPU codes alternately with taking care of data movement among nodes. In order to reduce software development costs, we propose a programming framework called FLAT which enables GPU codes to embed MPI functions. This paper describes execution model and implementation of FLAT, and discusses availability and performance obtained by two case studies, Livermore Loop18 and optical flow programs. Through the experimental results, we confirmed that FLAT increases readability in synthesized GPU codes with maintaining bearable performance degradation, which is less than 3% for a coarse-grained parallel program.

Keywords: GPGPU, cluster system, MPI, programming framework

1. はじめに

GPU（Graphics Processing Unit）は，消費電力あたりの演算性能が高いため，高性能計算に広く使用されている．2012 年 11 月の Green500 [1] では，上位 5 件のうち 3 件が

[1] 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, Chofu, Tokyo 182–8585, Japan
[2] 株式会社イーツリーズ・ジャパン
e-trees.Japan, Inc., Hachioji, Tokyo 192–0045, Japan
[a)] shima@comp.is.uec.ac.jp

GPU 搭載 PC クラスタ（以降，GPU クラスタ）である．GPU クラスタでは，各計算ノードに割り当てたタスクの一部を GPU で実行することで，全体として高い演算性能を得る．Shimokawabe らは天気予報シミュレーション [2] を，Komatitsch らは地震波の伝達解析 [3] が GPU クラスタで高い性能で計算できることを示している．また Babich らは，QUDA ライブラリを GPU クラスタ上で並列化している [4]．

GPU プログラムは，CUDA [5] や APP（旧 ATI Stream）[6]，OpenCL [7] といった特殊な言語で記述する必要がある．また，GPU クラスタのプログラミングでは，計算ノード間およびノード内並列性を考慮する必要があり，多大な労力を要する．具体的な問題として，GPU 間のデータ転送があげられる．一般に，ノード間通信に利用する MPI（Message Passing Interface）は CPU コード[*1]に記述する必要があり，GPU コード[*2]には直接記述できない．したがってプログラマは，(1) GPU コードと (2) MPI 関数を実行する CPU コード，さらに，(3) GPU–CPU 間のデータ転送の 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．

そこで我々は，GPU コード中に GPU 間のデータ転送を記述でき，既存ハードウェア環境にそのまま適用可能な FLAT フレームワーク [8] を提案する．FLAT は GPU コードに MPI 関数を記述可能であるため，3 種類のコードおよびデータ構造を管理する必要はない．GPU 間の通信で管理する必要があるのは，GPU コード内に記述する通信コードと転送対象のデータ構造のみである．

本論文の構成は次のとおりである，2 章で FLAT の概要について述べる．3 章で FLAT を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 章で GPU 間の通信における FLAT の有効性とオーバヘッドを評価する．5 章に関連研究を示し，6 章でまとめる．

2. FLAT

2.1 FLAT の概要

図 1 は，GPU クラスタの構成を図示したものである．

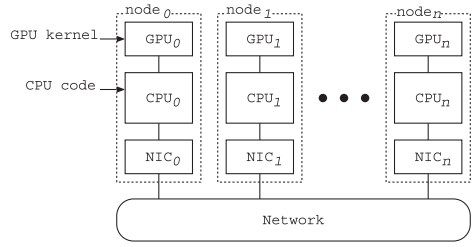


図 1　GPU 搭載ノードで構成される PC クラスタ（GPU クラスタ）
Fig. 1　A GPU equipped PC Cluster, or simply GPU cluster.

*1　各ノード内の CPU で実行されるプログラム．
*2　各ノード内の GPU で実行されるプログラム．

一般的な GPU クラスタでは，ノード内の GPU–CPU 間通信およびノード間の CPU–CPU 通信を行うことができるが，GPU から直接 NIC を介して他ノードの GPU にアクセスすることはできない．この制約は，GPU クラスタ向けのプログラム開発でも同様である．GPU_0 から GPU_n へデータ送信するプログラムを記述する場合，**図 2** に示す手順に相当するコードが必要になる．まず，(1) GPU_0 から CPU_0 へデータをコピーする．次に，(2) CPU_0 から CPU_n へ MPI 通信をする．さらに，(3) CPU_n から GPU_n へデータをコピーすることでデータ送信が完了する．つまり，GPU 間の通信においても，CPU コードを記述する必要がある．

本論文で提案する FLAT は，GPU 間の通信において，CPU コード上での MPI 関数の記述を不要にする．FLAT を利用した場合，GPU 間の通信は GPU_0 から GPU_n への通信に相当する**図 3** に示す関数 `flat_mpi_send()` と `flat_mpi_recv()` を GPU コードに記述すればよい．実際の通信は図 3 の網掛け部分となるが，その実行は FLAT フレームワークによって自動化する．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る．ソース to ソース変換によって，GPU コードに埋め込まれた 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．現在，FLAT がサポートする MPI 関数は，**表 1** のとおりである．

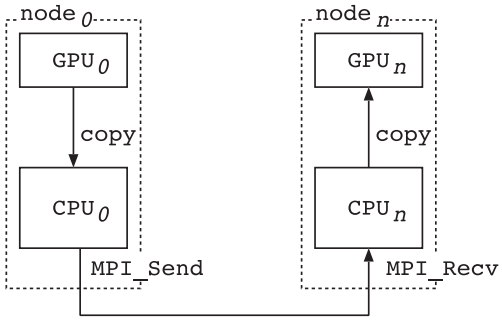


図 2　2 ノードにまたがる GPU 間通信
Fig. 2　Communication between two GPUs on different nodes.

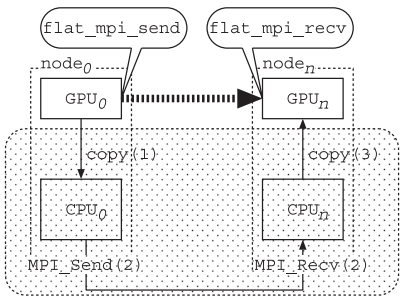


図 3　MPI 埋め込み GPU プログラミングモデル
Fig. 3　A programming model of embedded MPI into GPU codes.

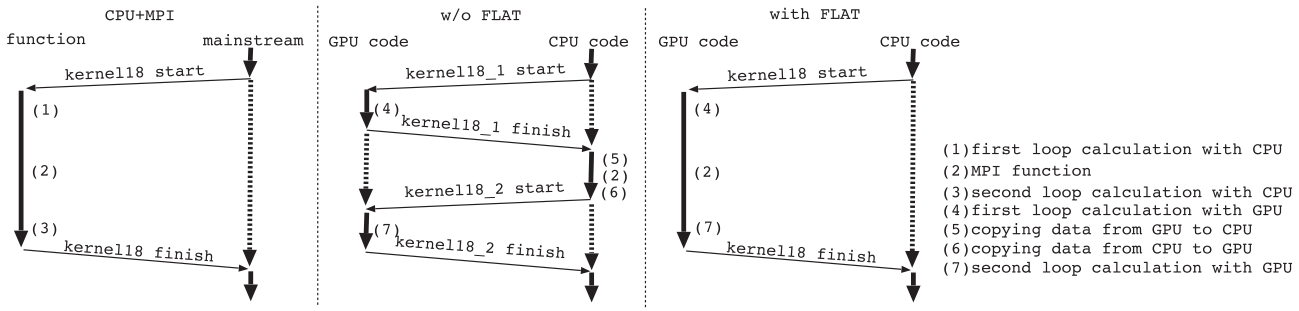


図 5　Livermore ループ Loop18 の実行フロー
Fig. 5　An execution flow of Livermore Loop18.

表 1　FLAT がサポートする MPI 関数
Table 1　Supported MPI functions.

name	corresponding MPI function
flat_mpi_send	MPI_Send
flat_mpi_recv	MPI_Recv
flat_mpi_isend	MPI_Isend
flat_mpi_irecv	MPI_Irecv
flat_mpi_wait	MPI_Wait
flat_mpi_sendrecv	MPI_Sendrecv
flat_mpi_barrier	MPI_Barrier
flat_mpi_scatter	MPI_Scatter
flat_mpi_gather	MPI_Gather
flat_mpi_bcast	MPI_Bcast

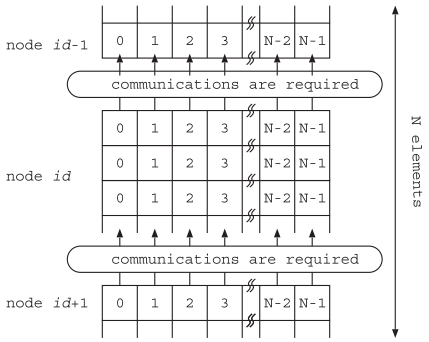


図 4　Livermore ループ Loop18 の GPU 間でデータ転送が必要な例
Fig. 4　An example of communication between GPUs.

2.2　FLAT を用いたベンチマークプログラム

例として，ベンチマークプログラムである Livermore ループ [9] の Loop18 を考える．Livermore ループは，行列演算などの数値計算によく現れる 24 種類のループからなるベンチマークプログラムであり．Loop18 は $N \times N$ 要素の 2 次元配列のデータに対する計算である．Loop18 は，計算途中でデータ転送が必要になる．GPU クラスタを用いて，計算対象となる 2 次元配列を**図 4** に示すように行方向にノード分割して実装する場合，ノード番号 id は MPI を用いて分割した境界部分のデータを $id+1$ から受信し，$id-1$ に送信する必要がある．

図 5 に，CPU のみを使用して MPI で通信する場合（CPU+MPI），FLAT を使用せずに GPU で計算・CPU で通信する場合（w/o FLAT），FLAT を使用する場合（with FLAT）のそれぞれについて，プログラマからみた実行フローを示す．図中の縦実線の矢印は，クラスタの振舞いを制御するためにコードを記述する場所を示す．また，図中の番号はそこに記述すべきコードの内容である．

```
   // function
 1: void kernel18
    (float **za,float **zp,float **zq,float **zr,float **zm,
     float **zb,float **zu,float **zv,float **zz){
 2:  ...(snip)...
 3:  t = 0.0037; s = 0.0041; kn = NUM; jn = n;
 4:  for ( k=1 ; k<kn ; k++ ) {
 5:   for ( j=1 ; j<jn ; j++ ) { /* calculation for za and zb */
 6:    za[k][j]=(zp[k+1][j-1]+zq[k+1][j-1]-zp[k][j-1]-zq[k][j-1])*
               (zr[k][j]+zr[k][j-1])/(zm[k][j-1]+zm[k+1][j-1]);
 7:    zb[k][j]=(zp[k][j-1]+zq[k][j-1]-zp[k][j]-zq[k][j])*
               (zr[k][j]+zr[k-1][j])/(zm[k][j]+zm[k][j-1]);
 8:   }
 9:  }
10:
11: if(id != 0)
     MPI_Send(&zb[1][0],ROW,MPI_FLOAT,id-1,0,MPI_COMM_WORLD);
12: if(id != nCPU-1 )
     MPI_Recv(&zb[COL][0],ROW,MPI_FLOAT,id+1,
                          MPI_ANY_TAG,MPI_COMM_WORLD,&st);
13:
14:  for ( k=1 ; k<kn ; k++ ) {
15:   for ( j=1 ; j<jn ; j++ ) { /* calculation for zu and zv */
16:    zu[k][j]+=s*(za[k][j]*(zz[k][j]-zz[k][j+1])
                   -za[k][j-1]*(zz[k][j]-zz[k][j-1])
                   -zb[k][j]*(zz[k][j]-zz[k-1][j])
                   +zb[k+1][j]*(zz[k][j]-zz[k+1][j]));
17:    zv[k][j]+=s*(za[k][j]*(zr[k][j]-zr[k][j+1])
                   -za[k][j-1]*(zr[k][j]-zr[k][j-1])
                   -zb[k][j]*(zr[k][j]-zr[k-1][j])
                   +zb[k+1][j]*(zr[k][j]-zr[k+1][j]));
18:   }
19:  }
20:  ...(snip)...
21: }
   // mainstream
22: kernel18(za,zp,zq,zr,zm,zb,zu,zv,zz,nCPU,id);
```

図 6　CPU+MPI を用いた Livermore ループ Loop18
Fig. 6　A Livermore Loop18 with CPU+MPI.

まず，CPU+MPI のコードを**図 6** に示す．次に，w/o FLAT のコードを**図 7** に示す．図 7 では GPU コードが，`kernel18_1` と `kernel18_2` の 2 つの関数に分割されている．一般的に，MPI 関数は GPU コード中に記述できないので，プログラムの構造や計算のまとまりとは関係なく，MPI 通信が生じるごとに GPU コードが分割される．一方，FLAT を用いた場合の GPU コードを**図 8** に示す．図 8 の `flat_mpi_send()` および `flat_mpi_recv()` がそれぞれ MPI 関数の `MPI_Send()` と `MPI_Recv()` に相当する．FLAT を用いることで，図 6 と同様の方法で，GPU コード内に MPI 関数呼び出しとその条件を直接記述する

```
    // GPU code
 1: __global__ void kernel18_1
    (float *za,float *zp,float *zq,float *zr,float *zm,float *zb){
 2:  int i = blockIdx.y * blockDim.y + threadIdx.y+1;
 3:  int j = blockIdx.x * blockDim.x + threadIdx.x+1;
 4:  if ( i < COL && j < N ) { /* calculation for za and zb */
 5:   za[i*ROW+j]= ...(snip)...;
 6:   zb[i*ROW+j]= ...(snip)...;
 7:  }
 8: }
 9:
10: __global__ void kernel18_2
    (float *za,float *zr,float *zb,float *zu,float *zv,float *zz){
11:  int i = blockIdx.y * blockDim.y + threadIdx.y+1;
12:  int j = blockIdx.x * blockDim.x + threadIdx.x+1;
13:  if ( i < COL && j < N ) { /* calculation for zu and zv */
14:   zu[i*ROW+j]+= ...(snip)...;
15:   zv[i*ROW+j]+= ...(snip)...;
16:  }
17: }
```

```
    // CPU code
18: kernel18_1<<<dim3(N/BLOCKSIZE, N/NODES/BLOCKSIZE),
           dim3(BLOCKSIZE, BLOCKSIZE)>>>(zad,zpd,zqd,zrd,zmd,zbd);
19: if(id > 0 ){
20:   cudaMemcpy(buf, &zbd[ROW], sizeof(float) * ROW,
                                          cudaMemcpyDeviceToHost);
21:   MPI_Send(buf, ROW, MPI_FLOAT, id-1 , 0, MPI_COMM_WORLD);
22: }
23: if(id < nCPU -1){
24:   MPI_Recv(buf, ROW, MPI_FLOAT, id+1, MPI_ANY_TAG,
                                        MPI_COMM_WORLD, &status);
25:   cudaMemcpy(&zbd[COL*ROW],buf, sizeof(float) * ROW,
                                          cudaMemcpyHostToDevice);
26: }
27: kernel18_2<<<dim3(N/BLOCKSIZE, N/NODES/BLOCKSIZE),
            dim3(BLOCKSIZE, BLOCKSIZE)>>>(zad,zrd,zbd,zud,zvd,zzd);
```

図 7　MPI によって分割された GPU コード

Fig. 7　Two divided GPU codes to transfer data among GPUs by MPI functions.

```
    // GPU code
 1: __global__ void kernel18
    (float *za,float *zp,float *zq,float *zr,float *zm,float *zb,
     float *zu,float *zv,float *zz, int id,int nCPU){
 2:  int i = blockIdx.y * blockDim.y + threadIdx.y+1;
 3:  int j = blockIdx.x * blockDim.x + threadIdx.x+1;
 4:  float s = 0.0041; float t = 0.0037;
 5:  if( i < COL && j < N ){ /* calculation for za and zb */
 6:   za[i*ROW+j] = ...(snip)...;
 7:   zb[i*ROW+j] = ...(snip)...;
 8:  }
 9:
10:  if(id != 0)
11:   flat_mpi_send(&zb[ROW],sizeof(float)*ROW,id-1,
                    0,FLAT_MPI_COMM_WORLD);
12:  if(id!=nCPU-1)
13:   flat_mpi_recv(&zb[COL*ROW],sizeof(float)*ROW,id+1,
                    MPI_ANY_TAG,FLAT_MPI_COMM_WORLD);
14:
15:  if( i < COL && j < N ){ /* calculation for zu and zv */
16:   zu[i*ROW+j]+= ...(snip)...;
17:   zv[i*ROW+j]+= ...(snip)...;
18:  }
19:  ...(snip)...;
20: }
```

```
    // CPU code
21: kernel18<<<dim3(N/BLOCKSIZE, N/NODES/BLOCKSIZE),
              dim3(BLOCKSIZE, BLOCKSIZE)>>>
              (zad,zpd,zqd,zrd,zmd,zbd,zud,zvd,zzd,id,nCPU);
```

図 8　FLAT を用いた Livermore ループ Loop18

Fig. 8　A Livermore Loop18 with FLAT.

ことができる．これにより，図 7 のコードでは必要だった CPU コード上の変数は不要となる．また，実際に転送したい GPU コード上の変数を指定可能になる．さらに，図 7 とは異なり，MPI による通信関数によって GPU コードが分割されることもない．

図 5 に示すように，CPU+MPI と with FLAT の構造は類似している．したがって，これまで GPU を使用していなかった MPI プログラマは，関数を新たに分割し，再構成する必要がなくなる．また，初歩的な GPU プログラミングの知識だけで，GPU クラスタを用いたプログラムが実装可能であり，開発コストを下げることができる．

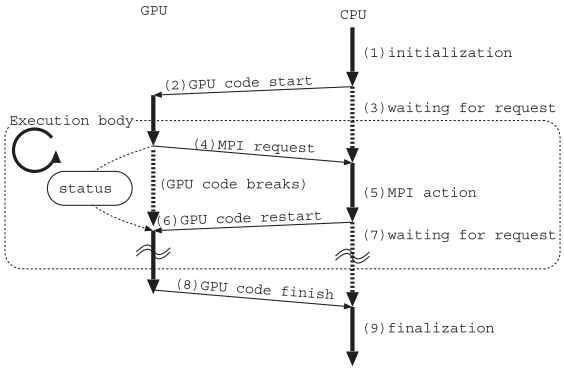


図 9　FLAT の実行モデル

Fig. 9　An execution model of FLAT.

3. FLAT の実装

3.1 FLAT 実装の概要

GPU コード（以後カーネル関数と呼ぶ）に記述された MPI 関数の処理を，CPU コード上の処理に置き換えることによって FLAT を実現する．これにより，CUDA，APP（旧 ATI Stream），OpenCL など，MPI による通信をサポートしない言語にも適用できる．また，置き換えをコード変換で実現し，変換後のコードから既存コンパイラを用いて実行コードを生成することで，容易に FLAT フレームワークを実装することができる．

3.2 実行モデル

カーネル関数に記述した MPI 関数を，CPU コードの処理に置き換える方法は 2 通り考えられる．1 つ目は，カーネル関数に記述された所望の MPI 関数をコード解析によって CPU コードに記述し直すことである．しかし，for 文，if 文など制御構文の内側に MPI 関数が記述された場合には，制御構造ごと CPU コードに記述し直す必要があり，これは非常に困難である．

2 つ目の方法は，MPI 関数のデータのアドレスやサイズ，通信方式など，通信に必要な情報を CPU に通知するコードを，GPU と CPU 双方のコードに挿入するものである．この手法を用いた場合，計算結果を保証するためにカーネル関数に埋め込まれた MPI 関数の直前に，GPU 内スレッドの同期が必要である．FLAT は，2 つ目の方法を用いて実行モデルを設計した．GPU 内スレッドの同期は，カーネル関数を終了することで実現する．

実行モデルを**図 9** に示す．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) カーネル関数を起動する．
(3) CPU コードはカーネル関数からの MPI 処理のリクエスト発行を待機する．

(4) カーネル関数から MPI 処理を CPU にリクエストする．カーネル関数は再実行のために必要なステータス情報を退避させ，実行を中断する．
(5) CPU コードは受理した MPI リクエストを実行する．
(6) カーネル関数を再起動する．カーネル関数では退避させたステータス情報を復帰し，MPI 関数後のコードから実行を再開する．
(7) CPU コードは再び MPI リクエストの発行を待機する．
(8) カーネル関数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を解放する．

GPU はタスクを処理している間，図 9 の (4) から (7) の処理を繰り返す．この間（図 9 で点線で囲った部分）の処理は，プログラマから隠蔽して実行する．

この実行モデルを実装するうえで，重要な仕組みが 2 点ある．1 点目は，GPU から CPU へのリクエスト発行である．FLAT は，CPU で対応する MPI 関数を呼び出すために，GPU から CPU へリクエストを送る．リクエストを送るために，**表 2** に示す MPI リクエスト・データ構造体のメモリを GPU と CPU で確保する．2 点目は，GPU 内スレッドの同期をするためのカーネル関数の中断・再開である．FLAT はカーネル関数の中断・再開を GPU 関数の終了と再起動で実現する．カーネル関数の中断の際に退避すべき情報は，**表 3** に示す GPU ステータス・データ構造体によって保持する．

仕組みの 1 点目を利用することで，FLAT はカーネル関数上の for 文，if 文などの制御構文をコード変換する必要がなくなる．また，仕組みの 2 点目を利用することで，`FLAT_MPI` 関数前後の実行状態を保存する．これら 2 点の仕組みにより，制御構文内であっても FLAT 関数は記述できる．

表 2　MPI リクエスト・データのメンバ
Table 2　Members of structure for MPI request data.

name	type	description
`send_addr`	`void *`	top address of send data
`send_size`	`int`	the length of send data
`dest`	`int`	destination id to send
`send_tag`	`int`	the label of the send data
`recv_addr`	`void *`	top address of receive data
`recv_size`	`int`	the length of receive data
`src`	`int`	source id to receive
`recv_tag`	`int`	the label of the recv data
`root`	`int`	root id
`comm`	`MPI_Comm`	MPI communicator
`mpi_request`	`int`	MPI_Request on GPU
`request`	`request_t`	kind of requesting MPI function

表 3　GPU ステータス・データのメンバ
Table 3　Members of structure for GPU status data.

name	type	description
`status`	`int`	current position of GPU kernel
`next_status`	`int`	next position of GPU kernel

3.3　実行制御

図 8 の 11 行目の `flat_mpi_send()` はコード変換により，**図 10** に変換される．図 10 の 2–3 行目の `STORE_KERNEL_INFO()` と `set_next_status()` は，それぞれデータの退避と再開時のコード開始位置を設定するためのコードである．**図 11** に，図 8 の 2 行目の変数 i に対するレジスタメモリの退避コードを示す．`set_next_status()` の引数は，`LABEL_KERNEL` として設定する．図 10 の 4 行目の `flat_mpi_send()` によって，**図 12** の 3–4 行目で示すとおり，スレッド番号 ((0, 0, 0), (0, 0, 0)) が MPI リクエスト・データにパラメータを格納する．現在の実装では，複数 GPU スレッドが別々の MPI 関数を呼ぶことができないため，MPI 関数は呼び出す数だけ記述する必要がある．また，図 12 の 3–4 行目に示すとおり，if 文の条件式にスレッド ID を使用するので，スレッド番号 ((0, 0, 0), (0, 0, 0)) 以外のスレッドが実行するコードに MPI 関数が記述さ

```
1: {
2:  STORE_KERNEL_INFO();
3:  set_next_status(1);
4:  flat_mpi_send(&zb[ROW],sizeof(float)*ROW,id-1,
                  0,FLAT_MPI_COMM_WORLD);
5:  goto END_OF_KERNEL;
6:  LABEL_KERNEL_1: RESTORE_KERNEL_INFO();
7: }
```

図 10　MPI 処理のためカーネル関数の実行を中断するコード
Fig. 10　A kernel code to break its execution after calling a MPI function.

```
1: #define STORE_KERNEL_INFO()
2:  {
3:   int _thread_id =   threadIdx.x
4:                     + threadIdx.y * blockDim.x
5:                     + threadIdx.z * blockDim.x * blockDim.y
6:                     + blockIdx.x * blockDim.x *
                          blockDim.y * blockDim.z
7:                     + blockIdx.y * gridDim.x *
                          blockDim.x * blockDim.y * blockDim.z;
8:   _status->i[_thread_id] = i;
9:  }
```

図 11　レジスタメモリの退避コード
Fig. 11　A code to store temporal status.

```
1: __device__ void
2: flat_mpi_send(void *addr, int size, int id,
                 int tag, MPI_Comm comm) {
3:   if (threadIdx.x==0 && threadIdx.y==0 && threadIdx.z==0 &&
4:       blockIdx.x==0  && blockIdx.y==0  && blockIdx.z== 0) {
5:    _info->send_addr = addr;
6:    _info->send_size = size;
7:    _info->dest = id;
8:    _info->send_tag = tag;
9:    _info->comm = comm;
10:   _info->request = MPI_SEND;
11:  }
12: }
```

図 12　MPI リクエスト・データにパラメータを格納するカーネル関数
Fig. 12　A GPU code to set parameters for an MPI send function.

れていた場合は，MPI は実行されない．

カーネル関数を中断するために，図 10 の 5 行目の goto END_OF_KERNEL でカーネル関数を終了する．MPI 関数実行後，再び呼び出されたカーネル関数は，goto 文で 6 行目の LABEL_KERNEL_1 まで処理をスキップし，計算を再開する．このとき，RESTORE_KERNEL_INFO() で，退避データを復帰する．シェアードメモリに関しては，FLAT 実行時，配列の要素数が定数になる実装の場合に退避・復帰が可能である．その場合には，コピー直前にブロック内で同期を行い，ブロックごとにコピーする．

一方 CPU では，**図 13** の 3 行目，cudaMemcpy() によってカーネル関数の中断もしくは終了を待っている．リクエストが発行された場合，MPI リクエスト・データを受け取り，図 13 の switch 文で，対応するの MPI 関数が選択される．ここでは，**図 14** に示す host_mpi_send() が呼び出される．この関数内で転送データを GPU から CPU へコピーし，その後 CPU によって MPI_Send() を実行する．CPU による MPI 関数の実行後，図 13 の 11 行目で，次のコード開始位置を特定し，カーネル関数を再起動する．

FLAT の flat_mpi_isend()/flat_mpi_irecv() は，MPI 関数の MPI_Isend()/MPI_Irecv() と同様の方法で使用できる．FLAT は flat_mpi_isend()/flat_mpi_irecv() を，MPI 関数の MPI_Isend()/MPI_Irecv() に変換することで，非同期通信を実現する．FLAT でこれらの関数を呼び出すと，カーネル関数が再起動されて，flat_mpi_wait() 関数の呼び出し位置で同期が行われる．FLAT では，flat_mpi_isend()/flat_mpi_irecv() の発行後に，GPU 間通信と並行して計算のオーバラップができるが，コード変換が行われることで，GPU–CPU 間通信と計算のオーバラップはできない．

FLAT はカーネル関数上に記述された MPI 関数を，カーネル関数と CPU コードに分けて CPU コードで MPI 関数を実行するよう変換するので，MPI 関数が記述されたカーネル関数が CPU コードと衝突することはなく，同時に動作させることができる．

```
 1:  do {
 2:    GPU_FUNCTION_stub<<<N,M>>>(....arguments....,
                                  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:
         host_mpi_send((gpu_info*)info->info_host); break;
 6:    case MPI_RECV:
         host_mpi_recv((gpu_info*)info->info_host); break;
 7:    case MPI_BARRIER:
         host_mpi_barrier((gpu_info*)info->info_host); break;
 8:    case MPI_SENDRECV:
         host_mpi_sendrecv((gpu_info*)info->info_host); break;
 9:    ...(snip)...
10:    }
11:    cudaMemcpy((void*)&(info->status_dev->status),
                  (void*)&(info->status_dev->next_status),
                  sizeof(int), cudaMemcpyDeviceToDevice);
12:  } while (info->info_host->request != GPU_DONE);
```

図 13　MPI 関数の発行を待つ CPU コード
Fig. 13　A waiting loop for MPI invocation in the CPU code.

```
1: void host_mpi_send(volatile gpu_info *info)
2: {
3:   void *send_addr;
4:   send_addr=(void*)malloc(info->send_size);
5:   cudaMemcpy(send_addr, (void*)info->send_addr,
                info->send_size, cudaMemcpyDeviceToHost);
6:   MPI_Send(send_addr, info->send_size, MPI_CHAR, info->dest,
              info->send_tag, info->comm);
7:   free(send_addr);
8: }
```

図 14　カーネル関数が発行した MPI 関数に対応する CPU コード
Fig. 14　A CPU code corresponding to the embedded MPI invocation on GPU.

3.4　コード変換

FLAT のコード変換は，CPU 側および GPU 側双方で行う．コード変換の 1 つに，ラップ関数の挿入がある．このラップ関数は，プログラマが定義したカーネル関数をラップする．生成されたラップ関数の引数は，プログラマが記述した引数に加え，コード変換時に生成する MPI リクエスト・データと GPU ステータス・データのポインタが加えられる．呼び出されたラップ関数は，受け取った 2 つのポインタを保存し，プログラマが定義したカーネル関数を呼び出す．これにより，コード変換後に，GPU–CPU 間で MPI リクエスト・データと GPU ステータス・データの自動的な共有が可能となる．

4.　評価

Livermore ループの Loop18 と，実アプリケーションのオプティカルフロー計算の 2 つを用いて，FLAT の有効性と実行性能に関して議論する．

4.1　オプティカルフロー計算

テストプログラムとして，視覚神経系の数理モデルを GPU 上で処理できるよう実装されたプログラムを用いる [10]．このプログラムは，動画データを 1 フレームごとにノード数で分割し，それぞれに 2 種類の GPU 処理を行う．2 度の GPU 処理の間で，生成された中間データの袖領域交換を行う．16 台の PC ノードを 2 次元メッシュに配置した場合，8 方向にそれぞれ双方向で袖領域の交換が行われるので，GPU–CPU 間，CPU 間合わせて 48 回の通信が行われる．したがって，通信回数とデータ転送量は Loop18 と比べて非常に多くなる．

FLAT を用いた実装には，オリジナルのプログラムと同様に，非同期通信（flat_mpi_isend()/flat_mpi_irecv()）を用いたパイプライン処理を実装している．オリジナルのプログラムでは，GPU 計算，GPU–CPU 間の通信，MPI を用いたノード間通信の 3 カ所がオーバラップして動作する．一方，FLAT は，GPU–CPU 間の非同期通信は不可能であるため，GPU 計算，MPI を用いたノード間通信のみをオーバラップしている．オプティカルフロー計算において FLAT を使用した場合，GPU–CPU 間でオーバヘッド

が生じることが考えられる．

性能比較では，FLAT を用いた実装と，MPI 関数を CPU コードに記述した通常のプログラミング手法による実装を比較する．

4.2　GPU 間の通信における有効性の評価

GPU 間の通信における FLAT の有効性について，プログラマが考慮すべき通信数とメモリ数を指標として評価する．この評価においての通信考慮数とメモリ考慮数を以下に定義する．通信考慮数とは，プログラマが開発時に考慮すべき，GPU 間の通信に関わる通信総数である．MPI 関数を CPU コードに記述した通常のプログラミング手法では，GPU 間の通信における，CPU–GPU 間の通信も考慮する必要がある．メモリ考慮数とは，通信関数にメモリアドレスを指定する数の総数である．オプティカルフロー計算は，画像 1 枚の処理に対する考慮数を示す．

メモリ考慮数と通信考慮数を削減することによって 2 点メリットがある．1 点目は，バグの可能性を減らすことである．メモリ考慮数が減ることによって，送信元，もしくは送信先アドレス指定数が減るため，ミスを減らすことができる．2 点目は，プログラムを容易に構造化できる点である．MPI コミュニケータやループ文を用いて，プログラム全体の通信を簡潔に記述する際，どの通信を同じループ文に入れるのか，また，MPI コミュニケータと GPU–CPU 間の通信をどのように組み合わせるのかを考慮する必要がある．通信考慮数が減ることによって，この作業を容易に進めることができる．加えて通信考慮数は，GPU 間，GPU–CPU 間，CPU 間のどの場所で考慮するかを分けることで，プログラマが管理すべきコードがカーネル関数か CPU コード，あるいはその両方であるかを評価できる．したがって，3 種類のコード[*3]およびデータ構造の削減を評価する指標として用いる．

表 4 に，Loop18，オプティカルフロー計算において，プログラマが考慮すべき，CPU 間の通信に関わるメモリ数と通信数の比較を示す．FLAT により両アプリケーションともにメモリ考慮数と通信考慮数が削減されている．特に，オプティカルフロー計算では，両者とも大きく削減されている．これは，考慮数を指標とした場合に，通信数は 3 分の 1，メモリ数は 2 分の 1 となるためである．このように，データ転送が多い場合は，プログラマの負担を削減できる．

また，with FLAT の考慮数は CPU+MPI と同じである．これは，FLAT によって GPU–CPU 間の通信考慮数が 0 になり，GPU–CPU 間の通信を考慮する必要なくなるためである．つまり，CPU 間の通信を記述することと同様に，GPU 間の通信を記述できることを示す．

さらに，with FLAT が考慮すべき GPU–CPU の通信考慮数は 0 であり，CPU 間の通信考慮数も 0 であるため，CPU コード上で通信関数に指定するメモリ考慮数は 0 になる．したがって，CPU コードの管理は不要となる．つまり，これまで管理する必要があった 3 種類のコードとデータ構造のうち，MPI 処理を実行する CPU コードと，GPU–CPU 間のデータ転送の管理が必要なくなる．これらの理由から，FLAT は GPU–GPU 間の通信におけるプログラミングコストを軽減できると考えられる．

表 4　プログラマが考慮すべき通信数とメモリ数
Table 4　Comparison of the number of communications and memory addresses.

	Livermore Loop18		
	CPU+MPI	w/o FLAT	with FLAT
通信考慮数	1	3	1
メモリ考慮数	2	4	2
	オプティカルフロー計算		
	CPU+MPI	w/o FLAT	with FLAT
通信考慮数	16	48	16
メモリ考慮数	32	64	32

[*3] (1) カーネル関数，(2) MPI 処理を実行する CPU コード，(3) GPU–CPU 間のデータ授受のコード．

4.3　実行時オーバヘッドの評価

4.3.1　オーバヘッドの原因

表 5 に示すノード 16 台を Gigabit Ethernet で接続した GPU クラスタを用いて，実行時間を評価した．

FLAT は MPI 処理のたびにカーネル関数の中断・再開を行うので，カーネル関数内で MPI 関数が連続する場合には実質的な処理のないカーネル関数呼び出しが行われ，まとめて MPI 関数をリクエストできるハンドコーディングに比べ，カーネル関数の呼び出し回数が増える．このオーバヘッドは，予備実験 [11] の結果，カーネル関数の呼び出し 1 回あたり，約 20 μ 秒以上の時間を要することが確認されている．これは，カーネル関数の終了から，GPU から CPU への MPI リクエスト・データの送信時間とカーネル関数の再呼び出し時間を含めて計測したものである．

FLAT を使用する際に生じる実行時のオーバヘッドは，文献 [11] で計測した (1) を含めて，次の 4 つが考えられる．
(1) カーネル関数の呼び出し回数の増加による処理時間
(2) レジスタの退避・復帰に要する時間
(3) GPU から CPU に対するリクエスト（構造体）の送信

表 5　GPU クラスタを構成する各ノードの諸元
Table 5　Specification of a GPU cluster.

CPU	Intel(R) Core(TM) i7-3930K 3.20 GHz
メモリ	64 GB
OS	CentOS release 6.4
	(Linux x86_64 2.6.32-358)
ネットワーク IF	Intel(R) 82579LM NIC
GPU	NVIDIA GeForce GTX 660 (Kepler)

時間

(4) MPI 通信前後で CPU がメモリを確保，解放する時間

本論文で述べた Livermore ループ Loop18 とオプティカルフロー計算の 2 種類のプログラムを用いて，FLAT 使用にともなうプログラム実行時間のオーバヘッドを評価した．これらのプログラムは，上記 (2) の揮発性メモリの退避・復帰のオーバヘッドが存在しないように実装した．レジスタの退避・復帰に関するオーバヘッドについては文献 [12] を参照されたい．

4.3.2　Livermore ループ Loop18

まず，Livermore ループの Loop18 について考察する．プログラムは，1,024 × 1,024 の float 型の 2 次元配列で，データ交換を行う袖領域は 1,024 ワードの設定である．実行結果を**図 15** に示す．縦軸は実行時間であり，横軸はノード数である．with FLAT，w/o FLAT ともに台数効果が得られた．

Loop18 において，カーネル関数の呼び出し回数の差は 1 回であるが，計測結果は 140 μ 秒（4 ノード実行時）から 40 μ 秒（16 ノード実行時）まで，with FLAT のオーバヘッドが減少することが確認された．これは上記 (1) 以外の原因により生じたものである．文献 [11] は，with FLAT，w/o FLAT ともにメモリ確保・解放の時間を含めて計測し，データサイズに依存しない (1) のみのオーバヘッドを計測している．しかし，一般的なプログラムでは，MPI 通信のために確保したメモリは再利用されるため，本論文の w/o FLAT の Loop18 実装は，計測時間にメモリの確保・解放を含めていない．この場合，(4) 通信前後のメモリの確保・解放が，with FLAT のオーバヘッドとして現れたと考えられる．Loop18 の実験では，メモリの確保・解放に最大 80 μ 秒の時間を要することが計測されている．

また，オーバヘッドは 4 ノードよりノード数が大きい場合に小さくなっていくことが確認された．1 ノード時には通信が発生せず，2 ノード時は単方向に 1 度通信が行われるのみなので，FLAT によるオーバヘッドは小さい．双方向の通信が発生する 4 ノード時が最もオーバヘッドが大きくなり，以降は，ノード数が多い場合は，ノードあたりで計算するワーキングセット（データ量）が小さくなるため，オーバヘッドもそれにともなって小さくなる．

w/o FLAT に対する with FLAT の性能低下率は全体で約 3%であり，16 ノード実行では約 2%であることが確認された．

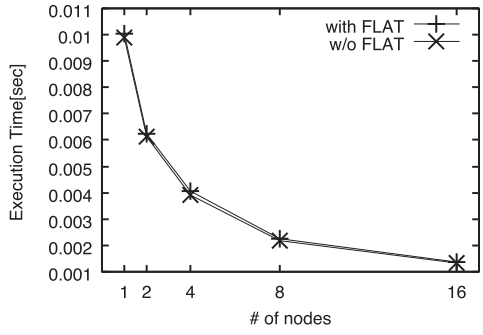


図 15　Livermore ループ Loop18 の実行性能

Fig. 15　Execution time of Livermore Loop18 with and without FLAT.

4.3.3　オプティカルフロー計算

次に，オプティカルフロー計算プログラムについて考察する．**図 16** に，処理する画像サイズ（単位はピクセル）を (a) 320 × 240，(b) 512 × 384，(c) 640 × 480 で実行したときのスループットを示す．横軸はノード数であり，縦軸は 1 秒間に処理したフレーム数（FPS）である．処理データは float 型で，交換袖領域は 16 ノード実行の場合，それぞれ (a) 2,156，(b) 3,332，(c) 4,116 ワードである．

with FLAT は，画像サイズ (a) の 16 ノード実行において 60 FPS を大きく超えており，(b) でも 30 FPS を超えている．したがって，w/o FLAT と同様，リアルタイムでのオプティカルフロー計算が可能である．

画像サイズ (c) の w/o FLAT に対する with FLAT の性能低下率は全体で約 4%であり，16 ノード実行では約 1%である．(a) の 12 ノード実行において with FLAT のパフォーマンスが低下する理由として，ノードを 3 × 4 のメッシュに配置したことがあげられる．このとき，中心の 2 つのノードは周囲の 8 方向のノードと袖領域の交換が必要になるため，カーネル関数の中断・再開の差が w/o FLAT よりも多くなる．このことから，(1) および (4) のオーバヘッドが大きくなったといえる．また，画像サイズによって with

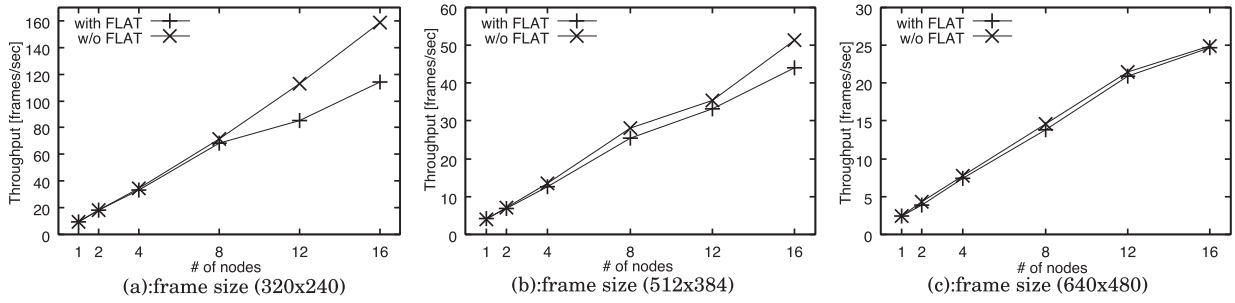


図 16　オプティカルフロー計算の実行性能

Fig. 16　Performance of an optical flow program with and without FLAT.

FLAT と w/o FLAT の性能差が異なる理由は，画像サイズが大きくなると，GPU の計算粒度が粗くなることがあげられる．16 ノード実行において 1 フレームあたりの GPU 計算時間は (a)〜(c) で約 5〜30 ミリ秒と変化するのに対し，FLAT のオーバヘッドは画像サイズに関係なく約 1 ミリ秒である．

FLAT は，GPU 計算時間が長く，通信回数の少ない粗粒度並列処理に対しては性能低下が小さい．GPU 計算時間が長いほど，計算時間に占める FLAT のオーバヘッドの割合が小さくなり，通信回数が少ないほど，カーネル関数の中断・再開の差が少なくなるためである．Loop18 は，GPU 計算時間は短く 16 ノード実行で約 1 ミリ秒であるが，カーネル関数の中断・再開の差は 1 回であるため，FLAT の性能低下率は 2%程度であった．一方で，オプティカルフロー計算では通信回数が増えるので，カーネル関数の中断・再開回数の差が多く，16 ノード実行で 34 回になる．しかし，このような場合であっても，オプティカルフロー計算では，カーネル関数の計算時間が約 30 ミリ秒あるので FLAT の性能低下率は約 1%であった．すなわち，FLAT は粗粒度な並列処理において，通常のプログラミング手法と比べて遜色ない性能が得られることが確認された．

5. 関連研究

GPU 間の通信を容易にするシステムとして，専用の PCIe デバイスを利用し PCIe パケットを Ethernet を介して他ホストに転送する，ExpEther を利用したマルチ GPU システム [13] が提案されている．このシステムにより，ホスト CPU を介さず直接データをやりとりできるため，GPU 間のデータ転送が高速に行えるほか，データ転送のプログラミングが簡潔になる可能性を持つ．しかし，専用ハードウェアを利用する必要があることから導入コストが大きくなり，安価で高性能な計算環境を提供するという GPU クラスタの利点を損なってしまう．

Open MPI [14] は現在 CUDA をサポートしているが，MPI 関数をカーネル関数内に記述することは不可能であり，CPU 側のコードによって GPU 側のメモリ管理が必要である．一方，FLAT を用いると，GPU 間の通信に関するプログラミングコストが大幅に削減できる．

NVIDIA の GPU-Direct™ [15] は，GPU–CPU 間のデータ転送の記述が不要になるという利点を持つ．しかし，データ転送の発行は CPU コードで記述する必要があり，転送対象のデータ構造は CPU コードで管理する必要がある．したがって，煩雑なメモリ管理は依然として存在する．

GPU クラスタを活用し，CUDA と MPI を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 [16]．この実装では，階層的な CPU 間，複数 GPU，GPU 内での並列化と分離を実現している．また，Lowlor は cudaMPI を提案している [17]．cudaMPI ではカーネル関数のために 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 2 つの研究では，あくまで CPU コード中にカーネル関数のための MPI 処理を書く必要があり，自然な形でデータ通信を記述することができない．

Stuart らは GPU クラスタ上で DCGN [18] および GPMR [19] の有用性を示した．DCGN は FLAT と同じくカーネル関数中に MPI 関数を記述できるが，FLAT と異なり，本来の MPI 関数と同じ形式で通信を記述できない．このことから，MPI に親しんだプログラマにとっては，DCGN よりも FLAT のほうが扱いやすい．また，GPMR は MapReduce で書かれたプログラムに対し有用である．一方，FLAT はメッセージパッシングを要するアプリケーションに重点をおいている．

HPCPE [20] 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する．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 からカーネル関数に変換する手法 [21], [22] や “軽量な” OpenMP と “重厚な” MPI の両方を使って複数 GPU を利用する文献 [23]，および GPU や FPGA にタスクレベルで分割する文献 [24] 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 [25], [26]．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フェースを提供する．rCUDA [27] では，CUDA 言語を拡張し，専用ライブラリを用いることによって，ネットワークを介して複数 GPU を透過的に扱うことができる．rCUDA を除くこれらの研究に，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て，これらに対してデータ転送の仕組みを提供することができる．

複数 GPU を簡単に扱いたいという観点から，Kim らは GPU クラスタをあたかも 1 ノードのシステムのように扱うことができるフレームワークを提案している [28]．これは実行時，GPU にタスクを割り当てることで実現している．また，Bueno らは OpenMP のプラグマでタスクやタスク間のデータ入出力を与え，ランタイムシステムでタスク並列を実現する OmpSs [29] を提案している．プラグマは CPU コードに記述する必要があり，カーネル関数そのものに自然な形でデータ転送を実現できない．

6. まとめ

本論文では，MPI を埋め込み可能な GPU プログラミン

グフレームワーク FLAT を提案した．FLAT を用いると，カーネル関数に MPI 関数を記述することが可能になり，CPU コード上で通信用の処理の記述が不要となる．そのため，GPU 間の通信に要するプログラミングコストが軽減される．したがって，我々の提案する FLAT は，安価で高性能な計算環境を提供する GPU クラスタでのプログラミングを促進することができる．

本論文は，まず，FLAT の実装手法を示した．次に，Livermore ループの Loop18，オプティカルフロー計算の 2 つの実プログラムを FLAT で実装し，GPU 間の通信における有効性を実行時オーバヘッドで評価した．FLAT を用いることで，プログラマが GPU クラスタでの通信において考慮すべきメモリ数と通信数が削減できることを示した．

実験を通して粗粒度な並列処理の場合，FLAT を用いない場合と比べて，性能低下率は約 3%以下であることが確認された．

今後の課題として，カーネル関数の中断・再開のためのコンテキストスイッチの軽量化，および細粒度な並列処理に対する最適化が考えられる．

参考文献

[1] Green500 Energy-Efficient Super Computers Sites: The Green500 List - November 2012(1-10), available from ⟨http://www.green500.org/lists/green201211⟩.

[2] Shimokawabe, T., Aoki, T., Muroi, C., Ishida, J., Kawano, K., Endo, T., Nukada, A., Maruyama, N. and Matsuoka, S.: An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code, *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.1–11 (online), DOI: 10.1109/SC.2010.9 (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, *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (*SC'10*), pp.1–11 (2010).

[5] NVIDIA CUDA Compute Unified Device Architecture, available from ⟨http://developer.nvidia.com/category/zone/cuda-zone⟩.

[6] A.M.Devices Accelerated Parallel Processing (APP) SDK (formerly ATI Stream), available from ⟨http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx⟩.

[7] OpenCL, available from ⟨http://www.khronos.org/opencl⟩.

[8] Miyoshi, T., Irie, H., Shima, K., Honda, H., Kondo, M. and Yoshinaga, T.: FLAT: A GPU programming framework to provide embedded MPI, *Proc. 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (*GPGPU-5*), pp.20–29 (online), DOI: 10.1145/2159430.2159433 (2012).

[9] Livermore Loops, available from ⟨http://www.netlib.org/benchmark/livermorec⟩.

[10] Ohmura, J., Egashira, A., Satoh, S., Miyoshi, T., Irie, H. and Yoshinaga, T.: Multi-GPU Acceleration of Optical Flow Computation in Visual Functional Simulation, *Proc. 2011 2nd International Conference on Networking and Computing* (*ICNC'11*), pp.228–234 (online), DOI: 10.1109/ICNC.2011.41 (2011).

[11] 三好健文，近藤正章，入江英嗣，本多弘樹，吉永　努：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，先進的計算基盤システムシンポジウム SACSIS 2011（SACSIS2011），pp.298–305 (2011).

[12] 島　圭吾，三好健文，近藤正章，入江英嗣，本多弘樹，吉永　努：MPI 埋め込み可能 GPU プログラミングフレームワーク適用可能性の評価，電子情報通信学会技術研究報告，CPSY，コンピュータシステム，Vol.111, No.163, pp.49–54 (2011-07-21).

[13] 野村鎮平，中浜徹也，樋口淳一，鈴木　順，吉川隆士，天野英晴：ExpEther を用いたマルチ GPU システムの評価，信学技報，CPSY2012-22，Vol.112, No.173, pp.79–84 (2012).

[14] Open MPI, available from ⟨http://www.open-mpi.org/⟩.

[15] NVIDIA GPUDirect^{TM}, available from ⟨http://developer.nvidia.com/gpudirect⟩.

[16] 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, *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (*GPGPU'10*), pp.51–61 (2010).

[17] Lawlor, O.S.: Message passing for GPGPU clusters: CudaMPI, *CLUSTER*, IEEE, pp.1–8 (2009).

[18] Stuart, J. and Owens, J.: Message passing on data-parallel architectures, *IEEE International Symposium on Parallel Distributed Processing, 2009*, *IPDPS 2009*, pp.1–12 (online), DOI: 10.1109/IPDPS.2009.5161065 (2009).

[19] Stuart, J. and Owens, J.: Multi-GPU MapReduce on GPU Clusters, *Parallel Distributed Processing Symposium* (*IPDPS*), *2011 IEEE International*, pp.1068–1079 (online), DOI: 10.1109/IPDPS.2011.102 (2011).

[20] Chen, Q. and Zhang, J.: A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA, *Proc. 2009 1st IEEE International Conference on Information Science and Engineering* (*ICISE'09*), pp.86–89 (2009).

[21] 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).

[22] 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).

[23] Noaje, G., Krajecki, M. and Jaillet, C.: MultiGPU computing using MPI or OpenMP, *Proc. 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing* (*ICCP'10*), pp.347–354 (2010).

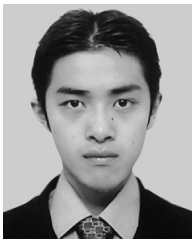
[24] Tsoi, K.H., Tse, A.H., Pietzuch, P. and Luk, W.: Programming framework for clusters with heterogeneous accelerators, *SIGARCH Comput. Archit. News*, Vol.38, pp.53–59 (2011).

[25] 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).
[26] Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M. and Olukotun, K.: Language virtualization for heterogeneous parallel computing, *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications* (*OOPSLA'10*), pp.835–847 (2010).
[27] Duato, J., Pena, A., Silla, F., Fernandez, J., Mayo, R. and Quintana-Orti, E.: Enabling CUDA acceleration within virtual machines using rCUDA, *18th International Conference on High Performance Computing* (*HiPC*), *2011*, pp.1–10 (online), DOI: 10.1109/HiPC.2011.6152718 (2011).
[28] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G. and Lee, J.: SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters, *Proc. 26th ACM International Conference on Supercomputing* (*ICS'12*), pp.341–352 (online), DOI: 10.1145/2304576.2304623 (2012).
[29] Bueno, J., Planas, J., Duran, A., Badia, R., Martorell, X., Ayguade, E. and Labarta, J.: Productive Programming of GPU Clusters with OmpSs, *Parallel Distributed Processing Symposium* (*IPDPS*), *2012*, *IEEE 26th International*, pp.557–568 (online), DOI: 10.1109/IPDPS.2012.58 (2012).

島 圭吾

2011 年京都産業大学理学部卒業．2013 年電気通信大学大学院情報システム学研究科修了．



吉見 真聡　（正会員）

2004 年慶應義塾大学理工学部情報工学科卒業．2009 年同大学大学院理工学研究科開放環境科学専攻後期博士課程修了．博士（工学）．2006 年度より日本学術振興会特別研究員（DC1）．2009 年より同志社大学理工学部助教を経て，現在，電気通信大学大学院情報システム学研究科助教．リコンフィギャラブルシステム，並列処理の研究に従事．電子情報通信学会，人工知能学会各会員．

三好 健文　（正会員）

2007 年東京工業大学大学院物理情報システム専攻博士課程修了．博士（工学）．同年東京大学大学院情報理工学系研究科特任助教．東京工業大学大学院情報理工学研究科産学官連携研究員，電気通信大学大学院情報システム学研究科助教を経て，2012 年株式会社イーツリーズ・ジャパンに入社，現在に至る．FPGA や GPU を活用したアクセラレータ，HW/SW 協調設計に関する研究・開発およびコンパイラ技術に関する研究に従事．IEEE，ACM，電子情報通信学会各会員．

近藤 正章　（正会員）

1998 年筑波大学第三学群情報学類卒業．2000 年同大学大学院工学研究科博士前期課程修了．2003 年東京大学大学院工学系研究科先端学際工学専攻修了．博士（工学）．独立行政法人科学技術振興機構戦略的創造研究推進事業 CREST 研究員，2004 年東京大学先端科学技術研究センター特任助手，2007 年同特任准教授を経て，現在，電気通信大学大学院情報システム学研究科准教授．計算機アーキテクチャ，ハイパフォーマンスコンピューティング，ディペンダブルコンピューティングの研究に従事．電子情報通信学会，IEEE，ACM 各会員．

入江 英嗣　（正会員）

1999 年東京大学工学部電子情報工学科卒業．2004 年同大学大学院情報理工学系研究科電子情報学専攻博士課程修了．博士（情報理工学）．2004 年科学技術振興機構 CREST 研究員，2008 年東京大学大学院情報理工学系研究科助教を経て，2010 年より電気通信大学大学院情報システム学研究科准教授．計算機システムおよびその応用の研究に従事．特に汎用プロセッサアーキテクチャ，ディペンダブルシステム，HCI の研究を進めている．情報処理学会山下記念研究賞（2010 年），同学会論文賞（2010 年）．電子情報通信学会コンピュータシステム研究会幹事（2011 年～）．電子情報通信学会，IEEE，ACM 各会員．

本多 弘樹　（正会員）

1984 年早稲田大学理工学部電気工学科卒業．1991 年同大学大学院理工学研究科博士課程修了．1987 年より同大学情報科学研究教育センター助手．1991 年より山梨大学工学部電子情報工学科専任講師，1992 年より同助教授．1997 年より電気通信大学大学院情報システム学研究科助教授，2007 年より同教授．並列処理方式，並列化コンパイラ，GPU コンピューティング等の研究に従事．工学博士．電子情報通信学会，IEEE-CS，ACM 各会員．

吉永 努　（正会員）

1986 年宇都宮大学工学部情報工学科卒業．1988 年同大学大学院修士課程修了．同年より宇都宮大学工学部助手．1997 年から翌年にかけて電子技術総合研究所客員研究員．2000 年電気通信大学大学院情報システム学研究科助教授．現在，同教授．博士（工学）．計算機アーキテクチャ，並列分散処理等に興味を持つ．ACM，IEEE，電子情報通信学会各会員．