Vol.2014-HPC-145 No.39
2014/7/30

XcalableMPとOpenACCの統合によるGPUクラスタ向け並列プログラミングモデル

田渕 晶大[1]　村井 均[2]　朴 泰祐[1,3]　佐藤 三久[1,3]

概要：GPU クラスタ用のプログラミングでは MPI と CUDA を用いるのが一般的であるが，学習コストやプログラミングコストの増加の点で問題がある．そこで指示文により記述できる生産性の高い GPU プログラミングモデルとして OpenACC が注目されている．また同様に指示文を用いた並列プログラミング言語 XcalableMP(XMP) の開発が進められている．現在，XMP と OpenACC を統合した GPU クラスタ向けのプログラミングモデルである XcalableACC(XACC) が検討されている．XACC では XMP と OpenACC を組み合わせて使う際に問題であったデバイス間の通信を行う指示節を導入することにより，生産性と性能を向上させる．さらに仮想デバイス型を導入することでマルチデバイスへのオフロードを容易にする．XACC によるデバイス間の袖通信を NVIDIA GPU を対象に実装して 2 次元ラプラス方程式と Himeno ベンチマークを用いて性能を評価した結果，MPI と CUDA による実装の 91%以上の性能が得られた．また MPI と CUDA を用いるより少ないコードで実装できることから生産性の高さも示された．今後は MPI の CUDA 支援を利用した袖通信の実装や袖通信以外の通信への対応が必要である．

1. はじめに

HPC の分野では演算加速器 (アクセラレータ) として GPU を用いたクラスタが普及している．GPU は数百から数千個の演算器を用いることで CPU よりも高い浮動小数点演算性能を有している．またその電力性能比の高さは，スーパーコンピューターの電力性能比のランキングである Green500 の上位を GPU クラスタが占めていることから明らかである．

通常 GPU クラスタ上で動作するプログラムを開発するには，CUDA[1] や OpenCL[2] を用いて GPU のプログラミングを行い，さらに MPI を用いてノード間の通信を記述する必要がある．しかしながら GPU での計算と MPI による通信を組み合わせたコードは複雑で，学習コストやプログラミングコストが高いという問題がある．

GPU のプログラミングでは，CUDA や OpenCL に替わるプログラミングモデルとして OpenACC[3] が注目されている．OpenACC では指示文により C・C++・Fortran のコードの一部をアクセラレータにオフロードすることが可能である．また分散メモリ環境でのプログラミング言語として PGAS 言語 XcalableMP(以後，XMP)[4] が開発されている．XMP では指示文により C・Fortran のコードをクラスタ上で並列に動作させ，ノード間の通信も行うことが可能である．また XMP を GPGPU に対応させた XMP-dev[5], [6] も開発され，GPU への計算のオフロードと GPU 上のデータの袖領域を通信する機能が実装されている．しかしながら GPU へのオフロードを制御する指示文が OpenACC に比べて少なく，マルチ GPU に対応していないという問題があった．

そこで XMP と OpenACC を統合した言語仕様 XcalableACC(以後，XACC) が理研 AICS プログラミング環境研究チームと筑波大学 HPCS 研究室により検討されている．XACC では通常の XMP と OpenACC 指示文に加えてデバイス間通信を行うための指示節を XMP 指示文に追加することで，アクセラレータを搭載したクラスタ上で高速に動作するプログラムを容易に記述することが可能である．さらに OpenACC に仮想デバイス型を導入することにより，マルチデバイスへのオフロードも容易に記述が可能となる．本研究では XACC の GPU 用の袖領域通信の実装を行い，その性能と生産性を評価する．

本稿では，本稿を含め 8 章で構成される．2 章で関連研究を紹介する．3 章で XcalableMP，4 章で OpenACC を概説し，5 章で XcalableACC の仕様を示す．6 章で袖領域

[1] 筑波大学大学院 システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 独立行政法人理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science
[3] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba

通信の実装を説明し，7 章でその評価と考察を行う．8 章で結論と今後の課題を述べる．

2. 関連研究

XMP-dev は XMP をデバイス向けに拡張した言語である [5], [6]．OpenACC と同様にオフロードモデルであり，デバイス上にホスト上のデータの複製を作成し，ホストとデバイス間でデータをコピーをすることが可能である．またデバイスでの計算は XMP の `loop` 指示文と同じように記述できるので，XMP との親和性が高い．デバイス間での通信も可能で，袖領域の通信を行う `reflect` が実装されている．実装にはデバイス用のコードを CUDA で出力する実装 [5] と OpenCL で出力する実装 [6] の 2 種類が存在する．

また Tightly Coupled Accelerators アーキテクチャ [7] に向けた XMP の拡張の提案がなされている [8]．XMP に TCA 用の `reflect_tca` という指示文を追加することで，TCA による通信を記述できるようにしている．

3. XcalableMP

XcalableMP[4] は PC クラスタコンソーシアムの XcalableMP 規格部会によって策定されている PGAS 言語の並列プログラミングモデルである．分散メモリ上での並列プログラミングのために C および Fortran に対して言語拡張を行っており，その多くは OpenMP に似た指示文である．XMP ではノード間の処理分割や通信をユーザが明示する必要がある．それによりプログラミングコストは増加するが，ユーザがプログラムの最適化を行いやすいという利点がある．

3.1 実行モデルとメモリモデル

XMP の実行モデルは MPI と同じく Single Program Multiple Data(SPMD) である．すなわち通常は各ノードで重複して処理が実行され，指示文で指定されたときのみ異なるデータに対してそれぞれ処理が行われる．メモリに関しても MPI と同様に通常は各ノードで重複して確保される．そして指示文で指定されたときのみメモリを分割して各ノードで保持する．

3.2 プログラミングモデル

XMP ではグローバルビューモデルとローカルビューモデルという 2 つのプログラミングモデルを提供している．ここではグローバルビューモデルの説明のみ行う．グローバルビューモデルは逐次プログラムに指示文でデータや処理の分散を指定することで並列プログラミングを行うプログラミングモデルである．グローバルビューモデルによるプログラミングの例を**図 1** に示す．この例を用いてプログラミング方法を説明する．

```
1  #pragma xmp template t(0:N-1)
2  #pragma xmp nodes p(4)
3  #pragma xmp distribute t(block) onto p
4  double a[N];
5  #pragma xmp align a[i] with t(i)
6
7  #pragma xmp loop (i) on t(i)
8  for(int i = 0; i < N; i++){
9        a[i] = i;
10 }
```

図 1 グローバルビューモデルによるプログラミングの例

```
1  double a[N];
2  #pragma xmp align a[i] with t(i)
3  #pragma xmp shadow a[1:1]
4
5  #pragma xmp reflect (a)
```

図 2 shadow・reflect の例

3.2.1 データ分散

`template` 指示文はテンプレートを定義する．テンプレートとは仮想インデックスの集合であり，XMP ではテンプレートを用いてデータや処理の分散を記述する．例では 0 から N-1 までのインデックスをもつ 1 次元のテンプレートを定義している．`nodes` 指示文はノード集合を定義する．XMP における 1 ノードは MPI の 1 プロセスと等価である．`distribute` 指示文はテンプレートをノード集合に分散する．この際テンプレートの各次元の分散方法を指定でき，例では block 分散を指定している．`align` 指示文は配列をテンプレートに従ってノードに分散する．`loop` 指示文は for ループをテンプレートに従ってノードで分散して実行する．

3.2.2 データ通信

XMP には定型的な通信を行うための指示文がいくつかあるが，ここでは XACC にて実装を行った `reflect` 指示文に関連する指示文のみ解説する．`shadow` 指示文は分散配列の袖領域を定義する．袖領域は隣接するノードにある上端・下端の要素を保持する領域である．分散配列の各次元の上端・下端に任意幅の袖領域を作成することが可能である．`reflect` 指示文は袖領域の更新を行う指示文である．隣接するノードと上端・下端を交換し，袖領域が最新の値になるようにする．

4. OpenACC

OpenACC[3] はホストコードの一部をアクセラレータにオフロードするための指示文ベースプログラミングモデルである．2013 年 6 月には仕様の Version2.0 が公開された．（現在は改訂版の 2.0a が公開されている．）C・C++・Fortran のコードに指示文を挿入することで，負荷の高い計算をアクセラレータ上で実行させることが可能で，高い生産性と可搬性が特徴である．現在，PGI・Cray・HMPP コン

```
1 int a[N];
2 #pragma acc data copyout(a)
3 #pragma acc parallel loop
4 for(int i = 0; i < N; i++){
5   a[i] = i;
6 }
```

図 3 OpenACC の記述例

パイラが OpenACC に対応している．またオープンソースの実装として accULL[9]，OpenUH[10]，Omni OpenACC Compiler[11] が開発されている．

4.1 実行モデル

OpenACC の実行モデルはオフロードモデルである．すなわち，指示文によって指定されたコード領域は GPU のようなアクセラレータで実行され，それ以外の部分はホストで実行される．`parallel` または `kernels` 指示文によりアクセラレータで実行するコード領域を指定できる．OpenACC では `gang`，`worker`，`vector` の 3 段階の並列性が存在する．`gang` は粗粒度，`worker` は細粒度の並列性である．各 `gang` は 1 つ以上の `worker` を持ち，各 `worker` では `vector` を用いてベクトル演算を行うことができる．これら 3 つの並列性が実際のアクセラレータとどのように対応付けられるかはコンパイラに依存する．通常，プログラムは 1 つの `worker` の 1 つのベクトルレーンを用いて各 `gang` で重複して実行される．並列化可能なループに対して `loop` 指示文によりワークシェアリングが指定されたときのみ，並列に実行される．

4.2 メモリモデル

OpenACC のメモリモデルではホストメモリとデバイスメモリはそれぞれ独立している．したがって，アクセラレータにオフロードするにはホストメモリとデバイスメモリ間で計算に用いるデータや計算結果を転送する必要がある．OpenACC ではホストメモリとデバイスメモリ間のデータ転送は基本的にコンパイラにより暗黙に行われるが，ユーザが指示文により明示し不要なデータ転送を削減することも可能である．

4.3 プログラム例

OpenACC の記述例を**図 3** に示す．*data* 指示文はデバイスメモリでデータを確保し，さらに領域の入口・出口でのホストとのコピーを指定する．例では配列 a をデバイスメモリに確保し，領域の終わりでデバイスからホストへコピーするよう指定している．*parallel loop* 指示文は直後の for ループをアクセラレータで並列に実行するよう指示する．使用する並列性は指定されていないのでコンパイラが自動で選択する．

5. XcalableACC

XcalableACC は GPU クラスタのようなアクセラレータを搭載したクラスタで動作するプログラムを開発するための言語である．既存の XMP と OpenACC 指示文に加えて独自の拡張を導入することにより，XMP と OpenACC のハイブリッド並列化よりもより簡潔で高性能なプログラムの生成が可能になる．

5.1 XcalableMP の拡張

XMP では `reflect` 等の通信の指示文の対象はホスト上のデータのみである．XACC ではホスト上のデータだけでなく OpenACC で確保されたデバイス上のデータに対しても通信を実行できるようにするため，通信を行う指示文に以下の指示節を導入する．

host
　ホストメモリ上の分散配列を対象とする．
device
　デバイスメモリ上の分散配列を対象とする．
host_and_device
　ホストメモリ・デバイスメモリ上の両方の分散配列を対象とする．

`host`，`host_and_device`，`device` のいずれも指定されない場合は `host` が指定されたものとし，通常の XMP 指示文と同様の動作をする．

5.2 OpenACC の拡張

現在の OpenACC の問題点として，ホストに接続されたマルチデバイスへのオフロードの記述が容易でない点が挙げられる．ユーザは並列ループをデバイス数で分割し，`acc_set_device` ライブラリの呼び出しによりデバイスを切り替えて，それぞれのデバイスで分割したループを実行しなければならない．XACC ではマルチデバイスを 1 つの仮想デバイスと見なす機能を提供することで，マルチデバイスへのオフロードを簡潔に記述可能にする．以下は仮想デバイスを定義するためのライブラリコールである．

```
acc_deveice_t
xacc_create_device(acc_device_t base_t
                   ,int num_array[], int size);
```

`base_t` は実デバイスの型である．例えば NVIDIA の GPU であれば `acc_device_nvidia` である．`num_array` は仮想デバイスに含まれる実デバイス番号のリストである．`size` は仮想デバイスに含まれる実デバイスの数である．また指示文の拡張として以下の指示節を追加する．

on_device_type(*acc_device_t*)
　指定されたデバイス型で実行する．
on_device_num(*int-expr*)
　指定された番号のデバイスで実行する．

これらの指示節を OpenACC の `parallel`, `kerenels`, `update` 指示文に付加することにより，その OpenACC 指示文を特定のデバイスのみに限定できる．

6. XcalableACC の実装

XACC のコンパイラを XMP のリファレンス実装である Omni XMP Compiler をベースにして実装する．この XACC コンパイラでは XMP の指示文は XMP のライブラリコールに置き換えるが，OpenACC の指示文は XMP によって変更された変数名の書き換えのみ行う．また拡張した OpenACC 指示節はその指示に従ってコードを変更した後に取り除く．これにより出力されるコードには通常の OpenACC 指示文のみが残り，既存の OpenACC コンパイラを利用してコンパイルすることが可能である．今回は Omni XMP C Compiler に NVIDIA GPU を対象とした XACC の reflect device 機能を実装した．

6.1 reflect device の実装

GPU 上のデータの reflect を実装するには幾つかの方法がある．

(1) MPI と CUDA を用いる方法．CUDA のランタイム API を用いてホストと GPU 間のメモリコピーを行い，MPI を用いてホスト間の通信を行う．袖通信のようにストライドのデータを送る場合には，GPU 内で送信前にパックし，受信後にアンパックすることで効率よく通信できる．

(2) CUDA 支援機能のある MPI を用いる方法．Open-MPI，MVAPICH2 など CUDA 支援機能がある MPI では，デバイスのアドレスを直接 MPI に渡して通信を行うことが可能である．記述が (1) よりも簡単であるうえ，GPU Direct 機能による性能向上の可能性がある．

(3) XMP `reflect` と OpenACC `update` を用いる方法．OpenACC の `update` 指示文によりホストとデバイス間のメモリコピーを行い，ホスト間の袖通信は XMP の `reflect` を用いる方法である．CUDA を用いる場合と比べて性能は低下するが，GPU 以外のデバイスでも OpenACC が対応しているならば動作するため可搬性が高い．

この他にも TCA を用いることが考えられる．今回は (1) と (3) の手法で実装を行った．また XMP における reflect の効率的な実装の研究が村井らによって行われており，実装の参考とした [12].

```
1 #pragma acc host_data use_device(_XMP_ADDR_a)
2 {
3   _XMP_gpu_reflect__(_XMP_DESC_a, _XMP_ADDR_a);
4 }
```

図 4　MPI と CUDA による XACC 実装における袖通信の変換例

6.1.1 MPI と CUDA による `reflect device` の実装

実装した reflect の処理の流れを解説する．まず袖領域を GPU メモリからホストメモリへコピーする．この際，袖領域が GPU メモリ上で不連続であるならパックした後にコピーする．文献 [12] によると袖領域は MPI でのベクトル型で表すことができる．ベクトル型はブロックの数 *count*，各ブロックに含まれる要素数 *blocklength*，ブロックの間隔 *stride* から成る．このベクトル型をパックする CUDA カーネルを用意しておき，パックの際に呼び出すようにした．また GPU との同期を減らすために，CUDA の非同期メモリコピーと Stream を利用した．Stream は GPU のコマンドキューで，1 つの Stream に割り当てた処理は割当順に実行されるが，他の Stream に割り当てた処理とは順不同に実行される．実装では 1 つの袖ごとに Stream を作成し，その Stream を用いてパックとメモリコピーを実行するようにした．ホストメモリへのコピー後，ノード間で袖を送受信する．その後，ホストメモリから GPU メモリへコピーし，データがパックされている場合はアンパックする．

袖が上端と下端に 1 ずつある分散配列 *a* に対して `device reflect` を行うコードの変換を図 **4** に示す．*_XMP_gpu_reflect__*関数を呼び出すことで袖通信を実行する．その際に `acc host_data` 指示文を用いてホスト上の配列 a に対応する GPU メモリ上の配列のポインタを渡す．

6.1.2 XMP reflect と OpenACC update による `reflect device` 実装

XMP の `reflect` の前後に OpenACC の `update` 指示文を追加する．reflect の前には隣接ノードへ送る領域のみデバイスからホストへコピーし，reflect の後には隣接ノードから受け取った領域のみをデバイスへコピーする．OpenACC の update 指示文には部分配列を記述できる．部分配列は配列の各次元に下限と長さを指定したものである ([*lower* : *length*])．部分配列を用いることで袖部分のみをコピーすることが可能である．

袖が上端と下端に 1 ずつある分散配列 *a* に対して `device reflect` を行うコードの変換を図 **5** に示す．3 行目では分散配列のローカルでの大きさを XMP のランタイムライブラリを用いて得る．XMP では分散配列はすべて 1 次元配列として保持しているが，それを OpenACC で直接多次元配列として扱うことができない．そこで 5 行目でポインタを元の次元の配列へのポインタへキャストし，さらに 6 行目で `acc present` によりデバイス上に存在することを明示する．8,9,11,12 行目では `acc update` によりホスト一

表 1　Cray XK6m の構成

CPU	AMD Opteron 6272 2.1GHz (16Core)
Memory	DDR3-1600 32GB
GPU	Tesla K20 (GDDR5 5GB, ECC on)
OS	CNL (Compute Node Linux)
Compiler	CCE8.2.0, CUDA5.5

GPU 間のメモリコピーを行う．その際，隣接するノードが存在するかを判定する関数*_XMP_reflect_need* を用いて，隣接ノードがある場合に限りコピーを行う．

7. 評価

評価に用いた計算機は Cray XK6m である．その構成を**表 1** に示す．ベンチマークには 2 次元ラプラス方程式と Himeno ベンチマーク [13] を用いた．評価では，袖通信の記述方法・実装方法が異なる以下の 4 種類の実装の性能を測定した．

- MPI+CUDA
- XACC(MPI+CUDA)
- XACC(reflect+update)
- XMP-dev

MPI+CUDA は袖通信を MPI と CUDA で直接記述したものである．XACC(MPI+CUDA)，XACC(reflect+update) は XACC により袖通信を記述し，実装した XACC コンパイラでコンパイルを行ったものである．MPI+CUDA，XACC(MPI+CUDA)，XACC(reflect+update) は計算部分は OpenACC を用い，そのコンパイルには Omni OpenACC Compiler を用いた．XMP-dev は袖通信及び計算部分を XMP-dev で記述したものである．XMP-dev にはデバイス上のデータの reduction が実装されていないため 2 次元ラプラス方程式のみ評価した．

7.1 2 次元ラプラス方程式

これは 2 次元ラプラス方程式をヤコビ法を用いて解くプログラムである．ソースコードの一部を**図 6** に示す．XMP を用いて問題を 2 次元分割し，さらに各ノードで計算を GPU にオフロードしている．また 23 行目では，XACC で導入した `reflect device` を使用して GPU 間の袖の交換を実行している．問題の大きさは 1K,2K,4K,8K とし，X 方向と Y 方向で 2 次元分割を行った．1 ノードにつき 1 プロセスとし，1 から 16 ノードまでノード数を変化させた．各袖通信の実装における性能を**図 7** に示す．袖通信に MPI+CUDA を使用した場合と比べ，XACC(MPI+CUDA) を使用した時は 91.9%〜102%の性能であった．また XACC(reflect+update) では 64.1%〜97.7%の性能であった．しかし XMP-dev は最大で 28.3%の性能しか得られなかった．

```
1  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
2  #pragma xmp nodes p(NUM_NODES_X, NUM_NODES_Y)
3  #pragma xmp template t(0:YSIZE-1, 0:XSIZE-1)
4  #pragma xmp distribute t(block, block) onto p
5  #pragma xmp align u[j][i] with t(i, j)
6  #pragma xmp align uu[j][i] with t(i, j)
7  #pragma xmp shadow uu[1:1][1:1]
8  /* ... */
9  void lap_main(void){
10   int x, y, k;
11 #pragma acc data copy(u, uu)
12   for(k = 0; k < niter; k++){
13     /* old <- new */
14 #pragma xmp loop (y, x) on t(y, x)
15 #pragma acc parallel loop gang
16     for(x = 1; x < XSIZE-1; x++){
17 #pragma acc loop vector
18       for(y = 1; y < YSIZE-1; y++){
19         uu[x][y] = u[x][y];
20       }
21     }
22
23 #pragma xmp reflect (uu) device
24
25     /* update */
26 #pragma xmp loop (y, x) on t(y, x)
27 #pragma acc parallel loop gang
28     for(x = 1; x < XSIZE-1; x++){
29 #pragma acc loop vector
30       for(y = 1; y < YSIZE-1; y++){
31         u[x][y] = (uu[x-1][y] + uu[x+1][y]
32                 + uu[x][y-1] + uu[x][y+1])/4.0;
33       }
34     }
35   } // end of niter
36 }
```

図 6　XACC による 2 次元ラプラス方程式の記述

7.2 Himeno ベンチマーク

Himeno ベンチマークはポアッソン方程式解法をヤコビの反復法で解くベンチマークである [13]．問題は i x j x k の 3 次元だが，k 次元の大きさが高々 256 であるため分割すると GPU の性能が出しにくく，さらに袖領域がストライドになるため i と j 次元のみで 2 次元分割をした．問題の大きさは Small, Medium, Large の 3 種類を用いた．各袖通信の実装における性能を**図 8** に示す．袖通信に MPI+CUDA を使用した場合と比べ，XACC(MPI+CUDA) を使用した時は 91.0%〜102.9%，XACC(reflect+update) では 65.6%〜97.4%の性能であった．

7.3 考察

MPI+CUDA と XACC(MPI+CUDA) の性能差の原因は主に 2 つある．1 つは reflect 関数のオーバーヘッドである．2 つ目は OpenACC により生成されたカーネルである．値のコピーと更新部分はどちらも OpenACC を用いているが，XACC では XMP によって変換されたループを OpenACC で変換するため，同じカーネルにはならない．問題サイズが小さいときには MPI+CUDA で生成されるカーネルの方が高速で，問題サイズが大きいときには

```
{
  int _XMP_REF_a_d1s;
  xmp_a_lsize(_XMP_DESC_a, 1, &(_XMP_REF_a_d1s));
  {
    double * _XMP_REF_a = (double * )(_XMP_ADDR_a);
#pragma acc data present (_XMP_REF_a[0:(_XMP_REF_a_d1s+1)+1])
    {
#pragma acc update host (_XMP_REF_a[1:1]) if(_XMP_reflect_need(_XMP_DESC_a, 0, 1, 0))
#pragma acc update host (_XMP_REF_a[(1+_XMP_REF_a_d1s)-1:1]) if(_XMP_reflect_need(_XMP_DESC_a, 0, 1, 1))
      _XMP_reflect__(_XMP_DESC_a);
#pragma acc update device (_XMP_REF_a [0:1]) if(_XMP_reflect_need(_XMP_DESC_a, 0, 1, 0))
#pragma acc update device (_XMP_REF_a [1+_XMP_REF_a_d1s:1]) if(_XMP_reflect_need(_XMP_DESC_a, 0, 1, 1))
    }
  }
}
```

図 5　XMP reflect と OpenACC update による XACC 実装における袖通信の変換例

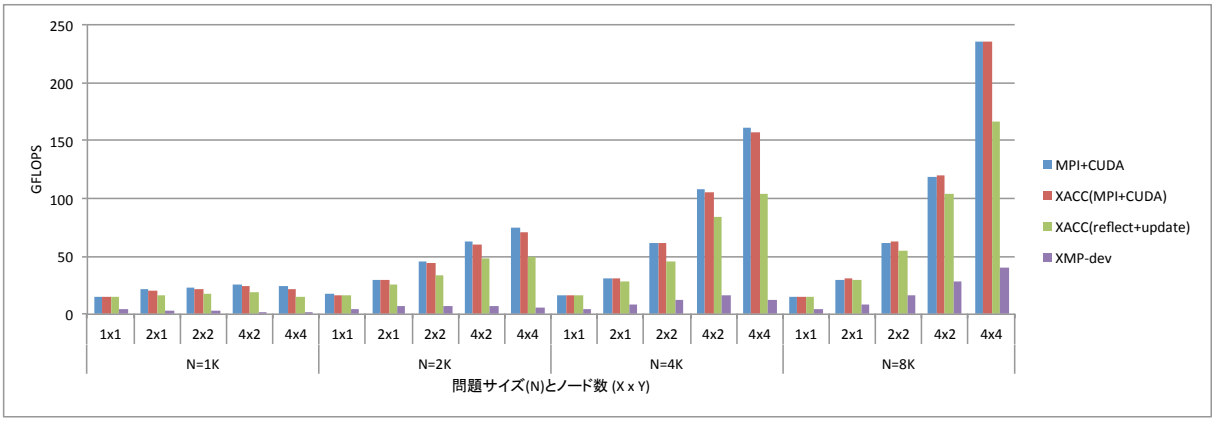

図 7　2 次元ラプラス方程式の性能

XACC(MPI+CUDA) で生成されるカーネルの方が高速であった．この原因については今後詳しく調査する．

また XACC(reflect+update) 実装では MPI+CUDA 実装と比べて大きく性能が低下した．原因の 1 つは余分なパック・アンパックである．MPI+CUDA 実装ではパックとアンパックは送信側と受信側の GPU 上でそれぞれ 1 度ずつ行われる．一方，XACC(reflecct+update) では OpenACC の update でパックとアンパックがそれぞれ 2 度行われ，さらに XMP の reflect が用いる MPI の内部でパック・アンパックが行われる．すなわちホスト側でパック・アンパックがそれぞれ 2 度余計に行われることになる．メモリ帯域幅の小さいホストメモリ上でのパック・アンパックが大きく性能を低下させている．

XMP-dev は他の実装と比べて非常に性能が低かった．性能低下の大きな要因は袖をパック・アンパックするたびに行っている GPU メモリの確保と解放である．cudaMalloc による GPU メモリの確保は評価に用いた Tesla K20 でおよそ 100us ほどかかり，それを毎回行うため非常に時間がかかる．また Kepler 世代の GPU 用にコードが生成されないため，計算部分での性能低下も起こっていた．

表 2　コードの行数．() 内は XMP 指示文の数

実装方法	ラプラス	Himeno
逐次	69	138
MPI+CUDA	270(6)	496(6)
XACC	99(22)	183(35)

7.4　生産性

XACC のコードと MPI+CUDA のコードで実装に要した行数を調べた．コメント行や空行を除いたコードの行数を表 2 に示す．逐次コードに比べ，MPI+CUDA では百行単位でコードが増加しているのに対し，XACC では数十行の指示文と範囲指定のための'{', '}' の追加のみである．少ないコード量で記述可能なことから，XACC は高い生産性があると言える．

8.　結論と今後の課題

本稿では GPU クラスタ向けに検討中の XcalableMP と OpenACC を統合したプログラミングモデルである XcalableACC の解説とその一部の GPU 間の袖通信を実装した．XACC では XMP と OpenACC を組み合わせて使う際に問題であったデバイス間の通信を行う指示節を導入することにより，生産性と性能を向上させる．さらに仮想デバイス型を導入することでマルチデバイスへのオフ

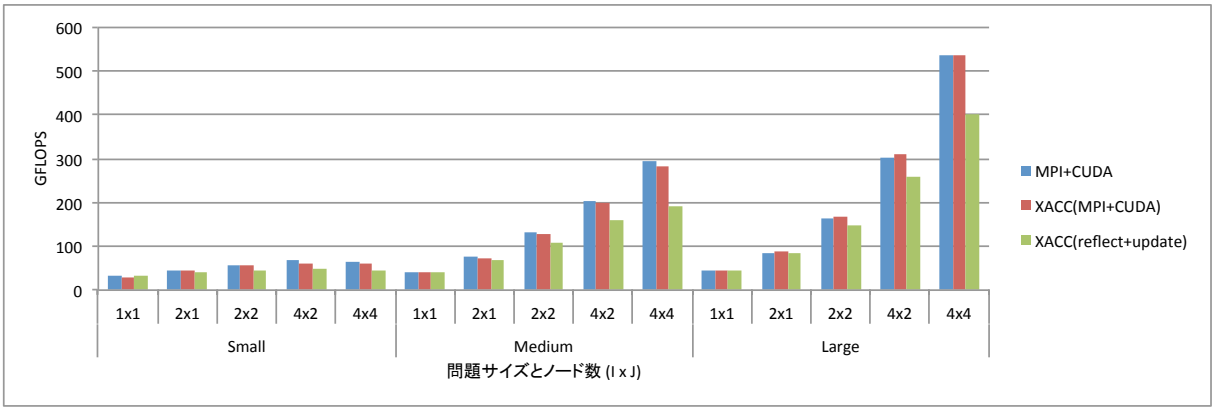

図 8 Himeno ベンチマークの性能

ロードを容易にする．XACC のうち GPU 間の袖通信を行う `reflect device` 指示文を MPI と CUDA を用いる方法と，XMP reflect と OpenACC update を用いる方法の 2 種類で実装した．2 次元ラプラス方程式と Himeno ベンチマークを用いて性能を評価した結果，MPI と CUDA による XACC 実装は MPI と CUDA を用いた場合の 91%以上の性能を達成した．また XMP reflect と OpenACC update による XACC 実装はパック・アンパックが余計に必要になることから性能が低下するものの，既存の XMP-dev による実装を上回る性能が得られた．さらにコードの行数が MPI+CUDA よりも大幅に少ないことから生産性が高いと言える．

今後の課題として，本稿で実装しなかった MPI の CUDA 支援を使用した `reflect device` の実装を考えている．また袖通信以外の通信 (`gmove`，`coarray` 等) への対応も必要である．

謝辞 本研究に際しご協力頂いた理化学研究所計算科学研究機構プログラミング環境研究チームの皆様に深く感謝する．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．

参考文献

[1] NVIDIA: CUDA 並列プログラミングおよびコンピューティングプラットフォーム, `http://www.nvidia.co.jp/object/cuda-jp.html`.
[2] Khronos Group: *OpenCL*, `http://www.khronos.org/`.
[3] OpenACC-Standard.org: *The OpenACC Application Programming Interface Version 2.0*, `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf` (2013).
[4] XcalableMP Specification Working Group: *Xcalablemp specification version 1.2*, `http://www.xcalablemp.org/download/spec/xmp-spec-1.2.pdf` (2013).
[5] 李 珍泌，チャントゥアンミン，小田嶋哲哉，朴 泰祐，佐藤三久：PGAS 並列プログラミング言語 XcalableMP における演算加速装置を持つクラスタ向け拡張仕様の提案と試作，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 5, No. 2, pp. 33–50 (2012).
[6] 野水拓馬，高橋大介，李 珍泌，朴 泰祐，佐藤三久：並列言語 XcalableMP のアクセラレータ向け言語拡張の OpenCL 実装，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]，Vol. 2012, No. 9, pp. 1–8 (2012).
[7] Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: Interconnection Network for Tightly Coupled Accelerators Architecture, *High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on*, pp. 79–82 (2013).
[8] 中尾昌広， 村井均，下坂健則，田渕晶大， 塙敏博，児玉祐悦， 朴泰祐，佐藤三久：Tightly Coupled Accelerators アーキテクチャに向けた XcalableMP 拡張，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]，Vol. 2014, No. 34, pp. 1–8 (2014).
[9] Reyes, R., López-Rodríguez, I., Fumero, J. and de Sande, F.: accULL: An OpenACC Implementation with CUDA and OpenCL Support, *Euro-Par 2012 Parallel Processing*, Lecture Notes in Computer Science, Vol. 7484, Springer Berlin Heidelberg, pp. 871–882 (2012).
[10] Tian, X., Xu, R., Yan, Y., Yun, Z., Chandrasekaran, S. and Chapman, B.: Compiling a High-level Directive-Based Programming Model for GPGPUs (2013).
[11] Tabuchi, A., Nakao, M. and Sato, M.: A Source-to-Source OpenACC Compiler for CUDA, *Euro-Par Workshops*, pp. 178–187 (2013).
[12] 村井 均，佐藤三久：並列プログラミング言語 XcalableMP におけるステンシル通信の効率的な実装，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， Vol. 2013, No. 8, pp. 1–9 (2013).
[13] 理化学研究所情報基盤センター：姫野ベンチマーク，`http://accc.riken.jp/2145.htm`.