Vol.2011-HPC-130 No.53
2011/7/29

PGAS 言語 XcalableMP の multi-node GPU 向け拡張仕様の実装と評価

Tran Minh Tuan[†1]　李　珍泌[†1]　小田嶋　哲哉[†1]
朴　泰祐[†1,†2]　佐藤三久[†1,†2]

GPU アーキテクチャの汎用化と高速化によって，GPU クラスタは高いコストパフォーマンスと演算性能，省電力化を HPC 分野にもたらしている．これまで汎用計算における GPU の高速化効果を目的とするプログラミング言語モデルの拡張やライブラリが数多く提案されてきた．しかし，これらは GPU を搭載するシングルノード環境を対象とする拡張が多く，GPU クラスタなどのメモリ分散システムを対象とするものがまだ少ない．multi-node GPU クラスタにおける高い性能プログラミングは通常の 1 ノード内のホスト− GPU の拡張だけでは不十分，それぞれのノードにまたがる GPU どうしのプログラミングも意識する必要がある．そこで，現在，我々は Partitioned Global Address Space（PGAS）プログラミングモデルをベースとした並列プログラミング言語 XcalableMP を GPU クラスタに適用可能とするための拡張を行っている．本稿では，行列積計算を対象に，GPU クラスタにおける XMP-ACC 拡張のプログラミングコストと性能について調査した．その結果，4 ノードの GPU クラスタにおいて，CPU のみを用いた XcalableMP プログラムよりも，それに数行の XMP-ACC 指示文の追加したプログラムのほうが約 42 倍の速度向上が得られた．

1. はじめに

近年では，GPU(Graphics Processing Unit) の汎用化・高性能化に伴い，GPU の演算性能を利用して汎用計算を加速させる GPGPU（General Purpose computing on GPU）が普及してきている．GPU は数百基のコアを搭載しており，SIMD(Single Instruction Multiple Data) 型の処理に向いているため，CPU と比べて非常に高い並列性と演算性能を持っている．高性能計算（HPC）分野では，GPU を多数搭載したクラスタを利用するケースが増えてきている．GPGPU 開発環境 CUDA[9]（Compute Unified Device Architecture）では，GPU を制御する API を提供しており，API によって GPU のデータ管理または GPU の並列化を可能にする．しかし，GPU クラスタ環境においてデータ分散，並列処理はプログラマが意識して MPI などで記述する必要がある．そのため，GPU クラスタの性能を引き出すためには，高いプログラミングコストとハイブリッド並列プログラミングの知識が必要になる．

一方，PC クラスタなどの分散メモリ型並列システムでのプログラミングにおいて，MPI(Message Passing Interface) と呼ばれる標準化された規格が広く使われている．しかし，MPI によるプログラミングでは，ノード間のデータ交換・配置などの処理を直接記述しなければならないため，大変難しく，プログラミングコストが大きい．その問題を解決するために，XcalableMP[1] という並列プログラミング言語が提案されている．XcalableMP は PGAS プログラミングモデルに基づいた言語モデルで分散メモリの典型的な処理を指示文で記述できるため，プログラミングモデルが分かりやすく，プログラミングコストが小さい．

そこで，本研究はマルチコア CPU・マルチ GPU 搭載の multi-node GPU クラスタのようなヘテロジニアスな環境に適用可能な XcalableMP のプログラミングモデルとコンパイラの拡張を行う．XcalableMP の特徴である分かりやすいプログラミングモデルと簡潔な記述で，データ分散，カーネル呼び出しに関わる煩雑さを隠蔽し，最適化が明示的に記述できる言語拡張 XMP-ACC を提案する．我々が提案する XMP-ACC は multi node GPU クラスタにおけるハイブリッド並列プログラミングを可能にすることを目的とする．XMP-ACC ではノード間の通信が XMP の指示文によって行われ，ノード内におけるデータ分散，カーネル呼び出しが XMP-ACC 指示文によって行われる．また，XMP-ACC は逐次のプログラムコードに指示文を挿入することだけでホスト− GPU 間のデータ転送，GPU 上の並列処理を指示することができるため，オリジナルのプログラムに最小限の変更で GPU クラスタ上での並列化を可能にする．

本稿は，本章を含めて 7 章から構成される．第 2 章では，XcalableMP 言語について紹介する．第 3 章と第 4 章では，本研究で提案する multi-node GPU に向け XMP-ACC 拡張について述べ，実行モデルとプログラム記述とコード変換について述べる．第 5 章では，GPU クラスタにおいて行列積を計算するプログラムを用いて XMP-ACC の性能評価を行った結果について述べる．第 6 章では，関連研究について述べ，既存の研究内容を概観し，本

†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 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba

Vol.2011-HPC-130 No.53
2011/7/29

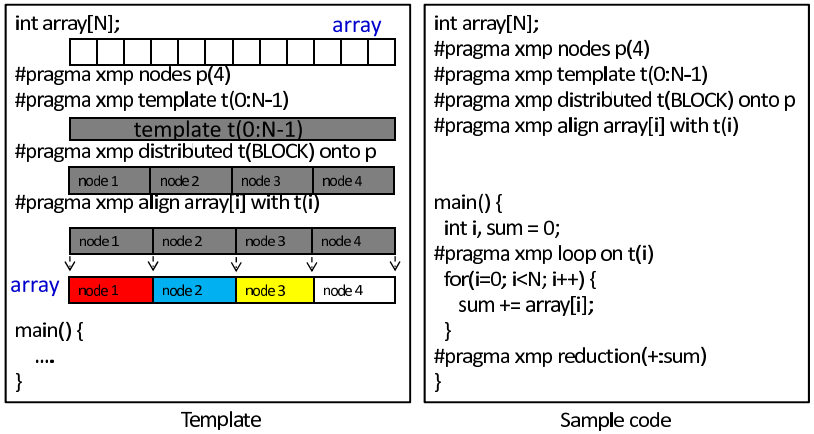

図 1 グローバルビューのデータ分割
Fig. 1 Data Parallelization in Global View Model

研究の立場を明らかにする．第 7 章では，本論文で述べている研究内容とその成果をまとめる．また，残された今後の研究課題についても述べる．

2. XcalableMP の概要

XcalableMP（以降，XMP）は，PC クラスタなどの分散メモリ型並列処理環境をターゲットした並列プログラミングモデルとして提案されている．グローバルビューのデータ並列とワークシェアリングによって，XMP で書かれたプログラムは並列化される．また，XMP は OpenMP-like な指示文を採用し，MPI のように最初から並列化を意識した設計をする必要がなく，OpenMP のように既存のプログラムコードに指示文を追加することで並列化することができる．XMP の言語モデルは，自動的な並列化は行わず，すべての処理はユーザが明示的に指示文で指定するという特徴を持っている．開発する際にプログラミングコストが高くなる傾向がある一方，ユーザがプログラムの並列性を明確にイメージすることができ，外部ライブラリの使用などの性能チューニングが容易であるといったメリットがある．この章では XMP の実行モデルと XMP の指示文について述べる．

2.1 実行モデル

XMP の実行モデルは，MPI と同様にすべてのデータに対して同じプログラムで処理する SPMD（Single Program Multiple Data）を実行モデルである．XMP では実行単位であるプロセスを「ノード」と定義し，それぞれのノードがローカルにデータを持ち，そのデータに対して計算を行うため，計算中にデータ通信などを行っていない．他のノードにあるデータにアクセスをする場合はノード間の通信を行わなければならない．通信，集約や同期が必要な場合，ユーザが XMP の通信記法である指示文で明示的に指定する必要がある．

2.2 プログラミングモデル

プログラミングのコストと記述性の両立を目的とする XMP 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する．グローバルビューモデルは，データ並列を基本として OpenMP-like の指示文を用いることで最小限の変更で既存のコードの並列化を可能とするモデルである．一方，ローカルビューモデルは，各ノードがローカルにあるデータに対してローカル配列のイメージとノード間の通信を意識しながらプログラミングするモデルである．ローカルビューモデルでの通信をより簡単に記述できるよう，XMP では，Co-Array Fortran[3] をベースにした言語拡張 Coarray 機能を提供している．この 2 つのプログラミングモデルを切り替えることによって，同じデータに対して目的に適したプログラミング方法が選択できる．例えば，グローバルビューモデルによって，配列が簡単に並列化される．その配列に対してローカルビューモデルで性能チューニングのためのアルゴリズムが自由に選択できる．

グローバルビュープログラミングモデルを実現するためにデータマッピング，ワークシェアリングとデータ通信・同期の 3 種類の指示文が提供されている．

2.2.1 データマッピング

各ノードにまたがる配列データはテンプレートという機能によって各ノードに分散し，分散された配列が各ノードで並列に計算される．テンプレートとは，ノードにまたがる配列データのインデックスの情報を持つ仮想的な配列であり，**template** 指示文で宣言される．そして，テンプレートは **distribute** 指示文によって各ノードに分割される．割り当ての方法は **distribute** 指示文のブロック分割の *block*，サイクリックの *cyclic*，と任意の分割の *gblock* の 3 つのオプションが指定できる．配列データは **align** 指示文によって宣言されたテンプレートに割り当てされる．**図 1** はグローバルビューにおける配列データの分散の概念を示す．図 1 では $p(4)$ で 4 つノードが宣言されており，配列 *array* が 1 次元のテンプレート $t(0:N-1)$ によって 4 ノードにブロック分割されている．

2.2.2 loop のワークシェアリング

XMP では **loop** 指示文を挿入することで for 文の処理を各ノードに分担させる．ノード毎の通信を意識せず，割り当てされた配列データに対して指定された範囲で処理を行うた

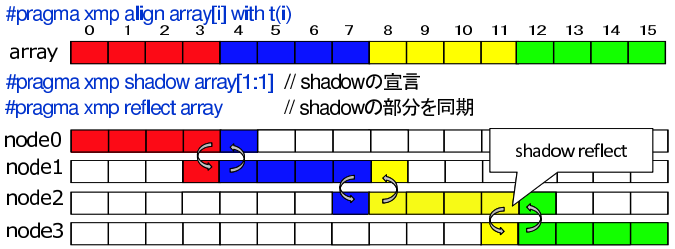

図 2 袖領域の処理
Fig. 2 Shadow reflection

め，効率のよい負荷分散が期待できる．また，**loop** 指示文に，分割したテンプレートを指定した場合，テンプレートに従ってループ文のインデックスに対応するノード集合が決定され，そのノード集合を実行ノード集合とする．図 1 のサンプルコードでは，テンプレート $t(i)$ に従ってループ処理を各ノードに分割されている．今回の例ではノード $p(1)$ はインデックスの 0 から $N/4$ までを処理し，ノード $p(2)$ はインデックスの $N/4+1$ から $N/2$ までを処理する各ノードが分割されたインデックスの部分を担当する．

2.2.3 通信・同期とシャドウ領域

XMP は基本的に通信を自動的に行わず，通信が必要な場合はユーザが明示的に指示文で指定しなければならない．グローバルビューでは，各ノードに分散された配列データの移動やその配列データに対しての計算を同期するために通信の指示文が使われる．計算に必要とする配列のすべてがローカルに存在するとは限らないため，他のノードに分散された配列データにアクセスしたい時，ノード間通信により配列を集める必要がある．XMP では **gmove** 指示文を用いることで，直後の代入文においてノード間通信の必要があることをコンパイラに知らせる．

```
#pragma xmp move
b[:] = array[:];
```

例えば，このように書かれた時，配列 $array$ は，そのデータのあるノードから送信され，b があるローカルのノードで受信され，b に代入される．

差分法を使うアプリケーションなどではノードに分散された配列において 1 つの要素を計算するためにとなりの要素に参照するパターンがよくある．分散された配列の境界付近

```
int a[N], b[N];
#pragma xmp align [i] with t(i) :: a, b
#pragma xmp shadow a[*]
void main(void) { . . .
int i, j;
#pragma xmp reflect a

#pragma xmp acc replicate (a, b)
{
#pragma xmp acc replicate_sync in (a)
```

```
#pragma xmp acc loop on t(i)
for (i = 0; i < N; i++) {
  b[i] = 0;
  for (j = 0; j < N; j++) {
    b[i] += a[j] ;
}
}
#pragma xmp acc replicate_sync out (b)
} // acc replicate
}
```

test.c

図 3 XMP-ACC のサンプルコード
Fig. 3 Sample code of XMP-ACC

の要素にとってこのような参照は隣接のノードからのデータの転送になる．あらかじめに通信に必要な部分を宣言し，隣接のノードからコピーし，保持することで通信の回数が減らす．この拡張された領域をシャドウ領域という．シャドウは **shadow** 指示文で宣言され，各ノードから自由にアクセスできる．**図 2** では，配列 $array$ のシャドウ領域が宣言されており，サイズが 1 要素である．シャドウ領域が常に正しい値であるために **reflect** 指示文によって他のノードと同期する必要がある．その他に XMP はすべてのノードで同期をとる **barrier** 指示文，データの集約演算を行う **reduction** 指示文，データをブロードキャストする **bcast** 指示文などの典型的な通信の記述を提供している．

3. multi-node GPU 向けの言語モデル拡張

前章で述べたように，XMP では逐次処理プログラムのコードに指示文を挿入することで，データマッピング，ワークシェアリングや通信など操作を指定し，プログラムを容易に並列化することができる．この章では，GPGPU に対応するために新しく拡張した XMP の指示文（XMP-ACC）について述べる．これらの指示文によって，GPU コンピューティングの基本的な処理である GPU 上のメモリ領域を確保するデータの宣言，ホスト側と GPU デバイス側の間のデータ転送と GPU によるループの並列化が指定できる．

図 3 は XMP-ACC の拡張指示文によって並列化された配列の要素の和を計算するプログラムを示す．新しく拡張した XMP-ACC はプログラムのコード内で **acc** がふくまれる指示文である．まず，**acc replicate** 指示文によって GPU 上に確保するデータ a，b が宣言さ

れる．GPU の計算に必要のある配列 a は **acc replicate_sync in** 指示文によって GPU のメモリに転送される．**acc loop** 指示文はループの処理が GPU 上で並列化されることを意味する．計算した結果である配列 b が **acc replicate_sync out** 指示文によってホスト側にコピーバックされる．このように XMP-ACC 指示文によって GPU コンピューティングの基本処理であるデータ確保・開放，ホストとデバイスの間のデータ転送とループ処理の GPU 分担を実現できる．XMP-ACC では，GPU に対して **bcast** などのノード間の通信が存在しないため，データがホストメモリに格納し，ホスト側が格納されたデータに対してノード間のコミュニケーションを行う必要がある．モデルを簡略化するために我々は 1 つのノードに 1 つの GPU があると仮定して拡張を行なっている．

3.1 データの宣言

GPU は独自にメモリを持ち，ホストメモリとは完全に独立しているため，処理を行う前に GPU のメモリにデータを確保する必要がある．**acc replicate** 指示文を使用して GPU メモリ上にデータを確保する．**acc replicate** 指示文によって宣言された変数は各ノードのローカルメモリと同じサイズ，データ型で GPU のメモリ上に確保される．シャドウの宣言がある変数はシャドウの部分も含めて確保される．

```
#pragma xmp acc replicate (list)
 compound − statement
```

acc replicate 指示文で宣言された変数の有効スコープは指示文の直後に書かれた複合文 *compound − statement* 内となる．GPU 変数に相当する変数はこのスコープ内で確保または解放される．

3.2 データ転送

XMP-ACC の拡張では，ホストと GPU の間のデータ転送は **acc replicate_sync** 指示文によって指示することが可能になる．**acc replicate_sync** 指示文の構文は下記のようになる．**in** 節または **out** 節を使用することでデータ転送の方向を指定することができる．GPU の計算に必要なデータをホスト側から GPU 側に転送する場合は **in** 節を指定して行う．一方，GPU メモリから計算結果をホスト側にコピーする場合は **out** 節を使う．このように **acc replicate_sync** 指示文によって GPU コンピューティングの基本的なデータ転送が記述できる．

```
#pragma xmp acc replicate_sync clause
 clause ::= in (list) | out (list)
```

3.3 ワークシェアリング

XMP では **loop** 指示文を挿入して，直後にある for 文ループをすべてのノードに分散することを指示する．ループの処理をすべてのノード間での分割だけではなく，各ノードにまたがる GPU にも分割することを目的に **acc loop** 指示文を拡張する．**acc loop** 指示文によって GPU 処理をターゲットした for ループが各ノードの GPU 上で処理される．

```
#pragma xmp acc loop [(list)] on onref [reduction (op:list)] [acc clause]
loop − statement
clause ::= private (list) | firstprivate (list) | shared (list) | num_threads ([x, y[, z]])
```

GPU の処理に必要とするデータは private，firstprivate，shared で宣言される．その中で firstprivate 変数は GPU による並列化するセクションの前に存在している変数の値で初期化される．shared 節で宣言される変数は GPU のメモリ上に確保され，**acc replicate** 指示文によって確保された変数は shared 変数とみなす．**acc loop** 指示文の for 文ループはコンパイル時に GPU のカーネルに変換され，GPU で実行される．また，GPU プログラムでは実行構成を選択することが性能につながるため，GPU を効率に使用するために適切なスレッドブロックサイズを指定することが必要である．ここでは **num_threads** 節を指定することで計算にあった実行構成が選択することができる．特に指定がなかった場合はデフォルトである (16,16) 構成を実行構成とする．

4. XMP-ACC 拡張のコード変換

4.1 コンパイラの流れ

今回の XMP-ACC 拡張は Omni XcalableMP コンパイラ[8] をベースとして実装を行う．コンパイラの処理流れは**図 4** のようになる．コンパイラは入力である XMP-ACC プログラムのファイル *test.c* を CPU で実行される中間コード *test.i* と GPU で実行される *test.cu* の 2 つのファイルに生成する．*test.i* は MPI のコンパイラ mpicc でコンパイルされる．一方，*test.cu* は XMP-GPU コンパイラのバックエンドである NVIDIA 社が提供する CUDA のコンパイラ nvcc によってコンパイルされる．コンパイルされたオブジェクトファイルである *test.o* と *test.xmpgpu.o* をランタイムライブラリとリンクさせることで，実行可能なファイル *test* が生成される．また，コンパイル時にノードにまたがる GPU デバイスの情報も習得することができるため，プログラムの問題サイズと合わせてデバイスに適したプログラムの最適化を行うこともできる．

Vol.2011-HPC-130 No.53
2011/7/29

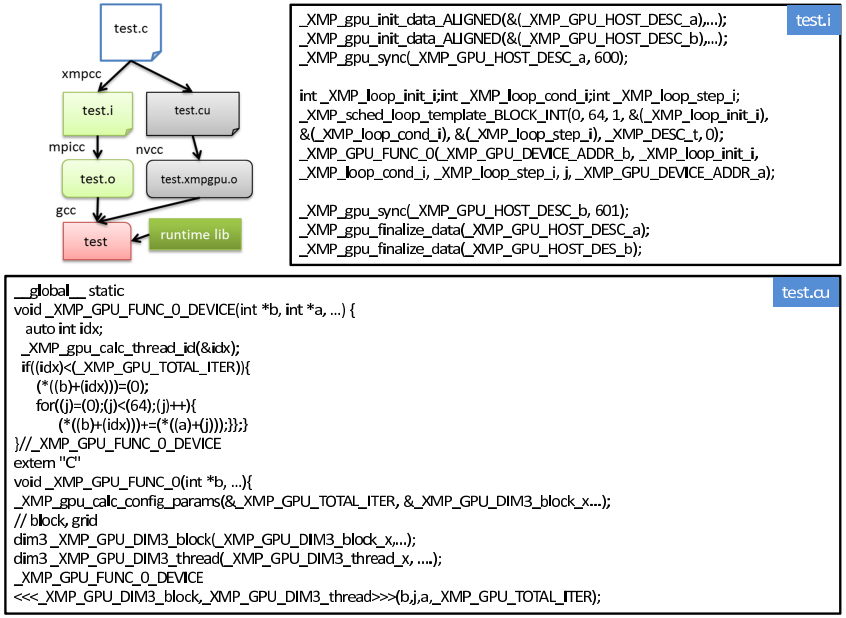

図 4 コンパイルの流れ
Fig. 4 Compile flow

4.2 コードの変換

XMP-ACC 拡張のコンパイラによるコード変換の例として，図 3 の *test.c* ファイルのコードから生成されたコードを図 4 に示す．今回は XMP-ACC 拡張に関連する部分のみを取り上げる．*test.i* は並列化されたホスト側のプログラムコードである．ホスト側のプログラムは基本の GPU コンピューティングの実行モデルに沿ってコードが生成される．まず，**acc replicate** 宣言された変数は_XMP_gpu_init_data_ALIGNED() 関数によって GPU メモリ上にデータを確保する．次に_XMP_gpu_sync() によってホスト側のデータを GPU にデータ転送を行う．GPU 上の並列化と指定された **acc loop** 指示文のループは GPU のカーネル関数に変換され，*test.cu* に生成される．ホスト側のコードではその GPU カーネルを呼び出す関数_XMP_GPU_FUNC_0() が生成される．計算された結果は_XMP_gpu_sync() によって GPU 側からホスト側に転送される．GPU で処理が完了した後，_XMP_gpu_finalize_data() によってメモリ解放が行われる．

GPU で実行されるコードは *test.cu* ファイルに出力される．**acc loop** 指示文の直後にある for ループは GPU カーネルに変換され，*test.cu* に生成される．GPU カーネルの変数は，**acc replicate** 宣言された変数のアドレスと for 文で使用される変数から生成される．コンパイラは **num_threads** 節で指定されたスレッドブロックのサイズから GPU カーネルの呼び出し構成を選択する関数_XMP_calc_config_params() を生成する．特に指定がなかった場合は初期設定である (16,16) 構成を実行構成とする．GPU カーネルを再生する際に分散されたインデックス情報と GPU カーネルの起動構成を基づいて配列のインデックスの計算を行う．

5. 性能評価

ここでは XMP-ACC 拡張によるプログラムの実装の性能評価について述べる．本評価は行列積を計算するプログラム用いて評価した．評価環境には，4 ノードをもつ GPU クラスタを使用した．**表 1** にクラスタのノード構成を示す．

表 1 ノードの構成
Table 1 Node Configuration

CPU	AMD Opteron(tm) Processor 6134 × 2 (8 Cores × 2)
GPU	NVIDIA Tesla C2050 (3GB GDDR5)
Memory	DDR3 SDRAM 4GB (2GB × 2)
Network	InfiniBand (4 × QDR)
OS	Linux version 2.6.18-194.32.1.el5 ×86_64
MPI	OpenMPI 1.4.2
Compiler	NVCC 3.2, GCC 4.1.2

5.1 XMP-ACC による行列積プログラム

本評価は，サイズ $N \times N$ の double 型正方行列 A，B，C について行列積 $C = A \times B$ の計算を XMP と XMP-ACC で実装し，測定した．行列の分割方法は行列 A，B，C については x，y 軸方向にブロック分割した部分を各ノードに分散する．各ノードは A，B，C の 1 部分を持ち，C を計算する時，必要なデータがローカルになければ，通信を行う．**図 5** は XMP と XMP-ACC で実装した行列積プログラムのコードを示す．XMP で実装した *matmul – xmp.c* は C 言語で書かれた逐次プログラム *matmul.c* のコードに **align** 指示文によって配列 a，b，c を x，y 方向で分割し，各ノードに分散する．計算に使う配列 a，b にシャドウ領域を宣言し，今回では a，b の full shadow を使っている．このシャドウ領

```
double a[N][N], b[N][N], c[N][N];
int main(void) {
 for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
     c[i][j] = 0;
     for (k = 0; k < N; k++) {
       c[i][j] += a[i][k] * b[k][j];
     } } }//main
```
matmul.c

```
double a[N][N], b[N][N], c[N][N];
#pragma xmp align [i][j] with t(j,i) :: a,b,c
#pragma xmp shadow a[0:0][*]
#pragma xmp shadow b[*][0:0]
int main(void) {
#pragma xmp reflect a,b
#pragma xmp loop (i, j) on t(j, i)
 for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
     c[i][j] = 0;
     for (k = 0; k < N; k++) {
       c[i][j] += a[i][k] * b[k][j];
     } } }//main
```
matmul-xmp.c

```
double a[N][N], b[N][N], c[N][N];
#pragma xmp align [i][j] with t(j,i) :: a,b,c
#pragma xmp shadow a[0:0][*]
#pragma xmp shadow b[*][0:0]
int main(void) {
#pragma xmp reflect a,b
#pragma xmp acc replicate (a, b, c)
{
#pragma xmp acc replicate_sync in (a, b)
#pragma xmp acc loop (i, j) on t(j, i)
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
     c[i][j] = 0;
     for (k = 0; k < N; k++) {
       c[i][j] += a[i][k] * b[k][j];
     } } }
#pragma xmp acc replicate_sync out (c)
} // acc replicate
}//main
```
matmul-xmp-acc.c

図 5 行列積のプログラム
Fig. 5 Matrix multiplication code

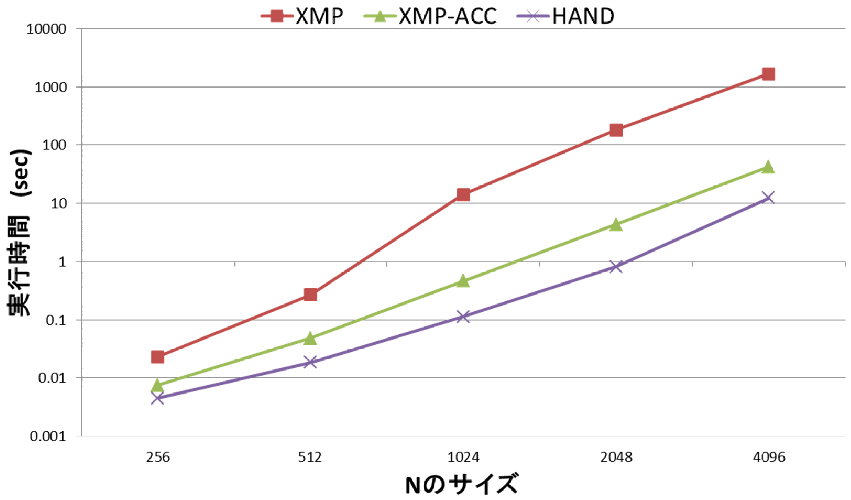

図 6 行列積の実行時間
Fig. 6 Processing Time

域は計算する前に **reflect** 指示文で更新することができる．**loop** 指示文を用いて 4 ノードに for 文の処理を分割する．図 5 の $matmul-xmp-acc.c$ は XMP-ACC の拡張によって実装されたプログラムである．まず，GPU 計算に必要なデータは **acc replicate_sync in** によって配列 a，b がホスト側から GPU 側にコピーされる．**acc loop** 指示文を追加することで GPU で並列化することを指定する．GPU の計算が終了後，**acc replicate_sync out** によって結果である配列 c が GPU 側からホスト側にコピーされる．

5.2 実行結果と考察

図 6 のグラフでは，行列のサイズ N を 256〜4096 まで変化させて 4 ノードを用いてプログラムの実行時間を測定した結果を示す．いずれの N の値においても GPU の拡張である XMP-ACC は CPU のみを用いた XMP よりも大きく速度が向上していることが分かる．しかし，行列サイズの N が 256 の時は問題サイズが小さく，GPU の演算量よりもホストと GPU の間のメモリ転送量が大きかったため，XMP-ACC の速度向上が 3 倍しかできなかったと考えられる．図 6 のグラフのように問題サイズが十分大きい場合において XMP-ACC が常に XMP よりも早いことが見られる．問題サイズを大きくするにつれて XMP と比べて XMP-ACC の速度向上が大きくなる傾向がある．N=2048 である時，速度向上が最大で約 42 倍である．行列積の計算では計算量が $O(N^3)$ に対してデータ転送量が $O(N^2)$ であるため，問題サイズがが大きいほど計算量が転送量より大きくなり，GPU の性能向上効果が見られたからである．また，本評価で用いた GPU NVIDIA Tesla C2050 に搭載される最新の Fermi プロセッサは倍精度演算において非常に高い演算性能（515GFlops）を持っている．この高性能演算が行列積プログラムに適用することができたため，このような速度向上になると考えられる．

しかし，行列のサイズ N が 2048 より大きくなってもその速度向上がほとんど変化が見られない．これは問題サイズがある程度大きくすると，各ノードが計算する問題サイズがキャッシュメモリの容量よりも上回って性能が低下したと考えられる．

現在，XMP-ACC コンパイラ実装では，GPU カーネルにおけるスレッドインデックスの計算が基本的に多次元の配列でも 1 次元に変更して計算を行なっている．それも考慮するために，手動で XMP-ACC コンパイラが生成したカーネル関数のインデック計算を 2 次元配列に対応する x と y 方向でスレッドインデックスを計算しする実装を行った．手動で

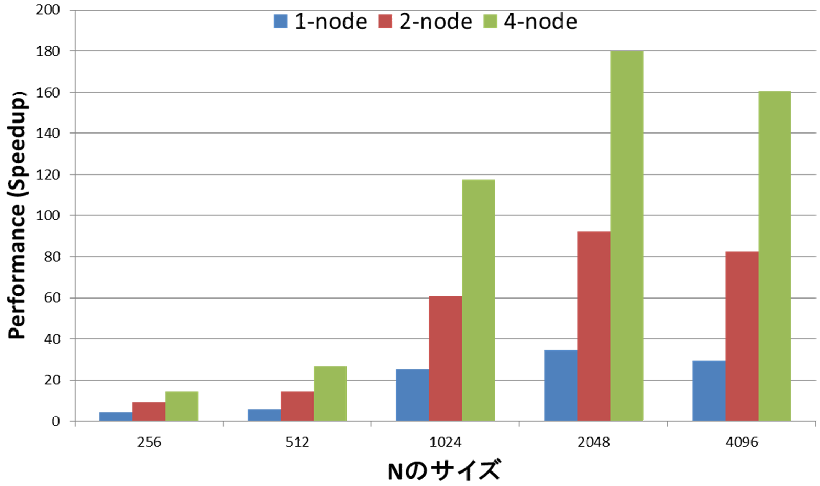

図 7 性能向上
Fig. 7 Performance of Matrix multiplication

実装したものの実行結果が図 6 のグラフの HAND である．グラフが示すように HAND が XMP-ACC よりも計算速度が最大 5 倍高いことが分かる．これは手動で実装したインデックス計算が x，y 方向の 2 次元で計算しているため，インデックス計算によるオーバーヘッドがコンパイラが生成するインデックスの計算より小さいと考えられる．

また，**図 7** のグラフでは，GPU クラスタの 1 ノード，2 ノード，4 ノードを使って測定した結果を示す．この評価では，問題サイズを変化させて 1 ノード，2 ノード，4 ノードで実行する場合，性能向上がどのように得られるかを調査する．性能向上は CPU のみ用いた XMP の逐次処理（1 ノードにおいて 1 スレッドが実行されるとする）の場合と比べたものである．図 7 のグラフのようにいずれの問題サイズにおいてもノード数を増やすにつれて，1 ノードと比べて良いスケーリングが得られている．これは問題サイズが大きいほど，演算量が $O(N^3)$ で増加するに対してノード間のデータ通信が $O(N^2)$ で増加するため，GPU で演算する時間がデータ転送時間とノード間の通信時間よりも大きい割合を占めており，GPU の高速化効果が見られたと考えられる．

このように，逐次の C 言語のコードに 10 行の指示文を追加することだけで XMP-ACC によってプログラムの並列化することができた．実験の結果では multi-node GPU クラスタにおいて XMP-ACC は XMP より速度向上ができることが分かった．しかし，コンパイラが生成したコードにおいても最適化する必要がある．

6. 関連研究

これまで GPU コンピューティングをターゲットとするプログラミング言語モデルの拡張やライブラリなどのアプローチは数多く提案されてきた．そのなかでは OpenMP の指示文を拡張し，指示文を追加するだけでプログラムが GPU 上で実行可能 CUDA プログラムに変換される OpenMPC[5] と OMPCUDA[6] が挙げられる．これらは，OpenMP 仕様をベースとしてデータの分割やループ処理の並列化に関する拡張された．しかし，OpenMPC も OMPCUDA も共有メモリシステムを対象としているため，シングルノード環境のみ対応しており，複数の GPU がまたがるシングルノード環境や分散メモリシステムである GPU クラスタにはまだサポートしていない．

また，本研究と同様に GPU コンピューティングの基本的な機能であるホストと GPU の間のデータ転送や GPU のカーネルの呼び出しを指示文で行うことを可能にする Hybrid Multicore Parallel Programming Workbench[4](HMPP) と Mint プログラミングモデル[7] がある．HMPP コンパイラは指示文で指定されたターゲットによって OpenCL または CUDA コードを生成する．そのため，HMPP はマルチコアの CPU の環境と複数 GPU がまたがるシングルノード環境に対応することができる．しかし，現在の HMPP においても GPU クラスタ環境向けの対応がまだされていない．Mint プログラミンモデルはステンシル計算に特化して設計された C から CUDA へのコンパイラを提供する．Mint コンパイラはステンシル計算専用の演算分析とメモリの最適化機能を提供する．本研究で行った XMP-ACC 拡張はメモリ分散システムの GPU クラスタを対象とし，指示文による簡潔な記述と CPU-GPU のハイブリッド並列プログラミングを可能にするものである．

7. まとめと今後の課題

本研究では，multi-node GPU クラスタなど分散メモリシステムを対象とし，GPU 向けの XcalableMP の言語拡張とコンパイラ設計と実装を試みた．XMP-ACC 拡張は分散メモリ環境におけるプログラミングにおいて指示文で簡潔に記述でき，GPU プログラミングも統一的でユーザに分かりやすい構造となるように設計した．GPU が搭載される GPU-CPU ヘテロジニアスなシステムを分かりやすくモデル化することを意識した．評価実験では，

XMP-ACC 拡張によって逐次プログラムである行列積プログラムのコードに指示文を挿入することで multi-node GPU クラスタに対応した並列化を実現することができ，4 ノードの GPU クラスタにおいて CPU のみ用いた XMP より最大約 42 倍の高速化を得た．

現在では，プログラミングモデルを簡単にするため，1 ノードにおいて 1GPU が搭載されると仮定して言語モデルの拡張とコンパイラの実装を行っている．しかし，近年ではアプリケーションの特徴により 1 ノード複数の GPU を搭載する GPU クラスタまたはシングルノードで複数の GPU を搭載する環境が増えつつある．今後の課題は，このような環境においても XMP-ACC が対応できるための言語モデルの拡張とコンパイラの実装である．また，現在の XMP-ACC コンパイラによって生成された GPU カーネルのコードはまだシェアードメモリの利用，メモリコアレッシングなどの最適化手法を考慮していない．これからは GPU の特性を考慮したコンパイラによる自動最適化機能を実装していきたい．

謝辞 本研究の一部は，JST-ANR 戦略的国際科学技術協力推進事業課題：「ポストペタスケールコンピューティングのためのフレームワークとプログラミングの研究開発」による．XcalableMP の仕様は，次世代並列プログラミング言語検討委員会によるものである．

参 考 文 献

1) XcalableMP Specification Working Group：http://www.xcalablemp.org/.
2) 李　珍泌，朴　泰祐，佐藤三久：分散メモリ向け並列言語 XcalableMP コンパイラの実装と性能評価，情報処理学会研究報告コンピューティングシステム第 31 号 (2010).
3) Coarray Fortran http://www.co-array.org.
4) HMPP Workbench. http://www.caps-entreprise.com/hmpp.html.
5) Seyong Lee and Rudolf Eigenmann: OpenMPC: Extended OpenMP Programming and Tuning for GPUs，In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’10, 2010.
6) Satoshi Ohshima, Shoichi Hirasawa, and Hiroki Honda: OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler，Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th Internationan Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010, Proceedings 2010.
7) D.Unat, X.Cai, and S. Baden：Mint: Realizing CUDA performance in 3D stencil methods with annotated C，Proceedings of the 25th International Conference on Supercomputing (ICS’11).
8) Omni Compiler Project：http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/.
9) NVIDIA Corporation：http://www.nvidia.com./object/cuda_home.html.