Vol.2018-HPC-165 No.40
2018/8/1

# PGASモデルによるマルチGPU対応OpenMPコンパイラ

中尾 昌広[1,a)]　村井 均[1]　佐藤 三久[1]

**概要**：HPC 分野で用いられる計算ノードの多くは，1 台の計算ノードに複数のアクセラレータを搭載している．しかしながら，複数のアクセラレータに対するプログラミングは，単一のアクセラレータを用いるよりも難しい．そこで本稿では，複数のアクセラレータに対するプログラミングを簡易に行える OpenMP 拡張を提案する．物理的に分かれている複数のアクセラレータのメモリで Partitioned Global Address Space（PGAS）を扱えるように OpenMP を拡張することで，高生産と高性能を両立させるプログラミングを可能にする．初期評価として，提案する OpenMP 拡張を用いて STREAM ベンチマークを実装した．2 基の GPU を搭載した計算ノードを用いて性能評価を行った結果，提案する OpenMP 拡張は十分な性能を発揮することを確かめた．

Masahiro Nakao[1,a)]　Hitoshi Murai[1]　Mitsuhisa Sato[1]

## 1. はじめに

優れた電力性能比とメモリバンド幅を持つアクセラレータが計算資源として広く利用されている．本稿執筆時点で最新の 2018 年 6 月の Top500 リスト [1] と Green500 リスト [2] の上位にランクインしているシステムの多くは，NVIDIA 社の GPU，Intel 社の Xeon Phi，PEZY 社の PEZY-SC2 などのアクセラレータを搭載している．そして，HPC 分野で利用されるシステムの多くは，より電力性能比を高めるため，1 台の計算ノードに複数のアクセラレータを搭載している．例えば，上記の Top500 1 位であるオークリッジ国立研究所の Summit は，1 台の計算ノードにつき 6 基の NVIDIA Tesla V100 GPU を搭載している．

アクセラレータ用プログラミング言語には，CUDA [3]，OpenCL [4]，OpenMP [5]，OpenACC [6] などがある．この中でも OpenMP と OpenACC は指示文ベースの言語であるため，ホスト用の逐次コードからアクセラレータ用の並列コードを簡易に開発できる．さらに，OpenMP と OpenACC は特定のアーキテクチャに依存しないため，ポータビリティも高いという利点がある．

しかしながら，既存のアクセラレータ用プログラミング言語の問題点として，複数のアクセラレータを扱うことは難しいことが挙げられる．その原因は，複数のアクセラレータのメモリ空間は物理的かつ論理的に分離しているため，ユーザは手動でデータとタスクを分割し，それらを適切に各アクセラレータにマップする必要があるからである．

同様の問題は PC クラスタなどの分散メモリシステムでも存在している．分散メモリシステムにおけるプログラミング環境には MPI が広く用いられているが，前述したデータの分割およびマップなどを行う必要があるため，そのプログラミングは一般的に難しい．そこで，高生産と高性能の両立を目的とする分散メモリシステムに対するプログラミングモデルとして，Partitioned Global Address Space（PGAS）言語モデルが提案されている [7–10]．PGAS とは，複数のプロセスのメモリで構成されるグローバルメモリ空間のことであり，各プロセスはそのメモリ空間に対して自由にアクセスできる．また，そのメモリ空間においては，ローカルメモリと各プロセスとのアフィニティが提示されているため，ユーザはデータのローカリティを意識したプログラミングを行うことも可能である．

そこで，本稿では複数のアクセラレータを簡易に扱うため，OpenMP に対して PGAS を扱うことができる拡張を提案する．提案する OpenMP 拡張のいくつかは，分散メモリシステム用の指示文ベースの PGAS 言語 XcalableMP（XMP）[11, 12] を参考にしている．本稿では，提案する OpenMP 拡張の初期評価を行う．

本稿の構成は下記の通りである．2 章では関連研究について述べる．3 章では既存の OpenMP を用いたアクセラレータのプログラミングの概要を説明する．4 章では提案

[1] 理化学研究所 計算科学研究センター
RIKEN Center for Computational Science
[a)] masahiro.nakao@riken.jp

する OpenMP 拡張について説明し，5 章では初期評価を行う．6 章ではまとめと今後の課題について述べる．

## 2. 関連研究

既存のアクセラレータ用プログラミング言語を用いて複数のアクセラレータを用いる方法としては，(1)OpenMP などを用いて複数のスレッドを立ち上げ，各スレッドが 1 つのアクセラレータを用いる方法 [13–15]，(2)MPI などを用いて複数のプロセスを立ち上げ，各プロセスが 1 つのアクセラレータを用いる方法 [13,16]，などがある．(1) の方法は 1 台の計算ノードを対象としているのに対し，(2) の方法はアクセラレータを搭載したクラスタシステム（アクセラレータクラスタ）でも動作させることができる．ただし，(2) の方法はホストで用いるデータも，アクセラレータで用いるデータと同様に各プロセスに分散させる必要があるため，(1) の方法と比べてプログラミングの複雑度は高くなる場合がある．また，1 章で述べた通り，どちらの方法もデータの分割およびマップなどを手動で行う必要があるため，1 つのアクセラレータを用いる場合と比べてプログラミングの複雑度は高い．

OpenACC を用いた複数のアクセラレータを利用するための研究がいくつか存在する．Xu ら [17] は，複数のアクセラレータをサポートするための OpenACC 拡張の提案を行っている．提案された OpenACC 拡張はアクセラレータ間の通信をサポートしているが，データの分割およびマップは手動で行う必要がある．Komada ら [18] は，ループ文を複数のアクセラレータに自動的に分割する OpenACC 拡張の提案を行っている．さらに，複数アクセラレータにおけるデータ整合性を自動的に保つ仕組みも提案している．ただし，ユーザが任意に挿入できるアクセラレータ間の集合通信については reduction のみのサポートである．Matsumura ら [19] は，既存の OpenACC の書式を変更せずに，処理系によるフロー解析のみで，ループ文を複数のアクセラレータに分割できる処理系を開発している．ただし，分割できるループ文は書き込みがアフィンアクセスのみで構成されているなど，いくつかの制限が存在する．また，計算に利用するホストのすべてのデータを各アクセラレータに転送する必要があるため，複数のアクセラレータが利用できるメモリ量は，1 つ分のアクセラレータと同じである．

アクセラレータクラスタ用の指示文ベースの並列言語に XcalableACC（XACC）がある [20,21]．XACC は XMP と OpenACC を組合せたモデルであり，ホスト側のプログラミングは XMP を用い，アクセラレータ側のプログラミングには OpenACC を用いる．また，XACC はアクセラレータ間の直接通信を行うための記法も提供している．1 台の計算ノードにおいても，XACC による複数のアクセラレータの利用は可能であるが，OpenACC だけでなく XMP の文法も習得する必要がある．

## 3. OpenMP によるアクセラレータプログラミング

### 3.1 概要

1997 年 10 月に Fortran に対応した仕様である OpenMP 1.0 がリリースされ，1998 年 10 月に C および C++に対応した仕様がリリースされた．OpenMP 1.0 から 3.1 までは共有メモリ型システムを対象とした仕様であったが，2013 年 7 月にリリースされた OpenMP 4.0 においてアクセラレータにも対応した．本稿執筆時点の最新の仕様は 2015 年 11 月にリリースされた OpenMP 4.5 である．本稿では，OpenMP 4.0 および OpenMP 4.5 をまとめて OpenMP 4.x と呼ぶ．

OpenMP に対応したコンパイラの一覧は公式サイト [22] に記載されており，その中でオープンソースかつアクセラレータに対応した OpenMP コンパイラは，Heterogeneous OpenMP（for NVIDIA GPU）[23]，Clang（for NVIDIA GPU）[24]，GNU GCC（for Intel MIC）[25] である．

### 3.2 OpenACC との比較

OpenMP 4.x と同様に，指示文を用いてアクセラレータプログラミングを行える言語に OpenACC がある．OpenACC も OpenMP と同様に，Fortran/C/C++に対応している．ただし，OpenMP は共有メモリ型システムからアクセラレータに対応するように拡張したのに対し，OpenACC は最初からアクセラレータに対応しているという，各言語における発展の経緯は異なる．また，OpenACC の最初の仕様は 2012 年 11 月にリリースされており，OpenMP 4.0 のリリースよりも早い．本稿執筆時点の OpenACC の最新の仕様は，2017 年 11 月にリリースされた OpenACC 2.6 である．OpenMP 4.x の仕様は，従来の共有メモリ型システムのための機能とアクセラレータに対する機能との両方が記載されているため，アクセラレータのみに特化した OpenACC の仕様の方が比較的簡易である．これらの理由から，現時点では 2 章でも挙げた通り，OpenMP よりも OpenACC の方がアクセラレータプログラミングに対する研究が活発に行われている．

アクセラレータに対する OpenMP 4.x と OpenACC 2.6 のプログラミングモデルおよび指示文の記法はよく似ている．OpenMP 4.x と OpenACC 2.6 の両方には，ホストとアクセラレータ間でデータ転送を行うための指示文や，ループ文をアクセラレータで実行するための指示文が提供されている．データ管理に関する細かな違いとして，OpenACC ではデータ転送を処理系が自動生成できるのに対し，OpenMP ではデータ転送はユーザの責任で行う必要がある．また，機能における相違点として，例えば OpenACC 2.6 は構造体メンバの Deep Copy に対応して

```
1  double a[N], b[N], c[N], scalar = 1.0;
2  #pragma omp target map(to: b,c) map(from: a)
3  #pragma omp teams distribute parallel for
4  for(int i=0;i<N;i++)
5   a[i] = b[i] + scalar * c[i];
```

図 1: OpenMP 4.x によるアクセラレータを用いた STREAM ベンチマーク

いるのに対し，OpenMP 4.x では対応していない点などがある．しかし，機能については，各仕様はお互いに影響し合っているため，その差は今後小さくなる，もしくは 1 つに統一されていくと考えられる．

本稿において OpenACC ではなく OpenMP を対象とした理由は，OpenACC よりも OpenMP の方がホスト（CPU スレッド）に対して多様な機能を持っているからである．アクセラレータを搭載した計算ノードのすべての計算資源を引き出すには，アクセラレータだけでなくホストの利用も不可欠である．例えば OpenACC をアクセラレータプログラミングに用いる場合，ホスト側の計算は OpenMP などを用いることになるため，1 つのコードに 2 種類のプログラミング言語が混在することになる．それに対し，OpenMP を用いると，1 つの言語でヘテロジーニアスな計算資源に対するプログラミングを行うことができるため，学習コストが小さく抑えられると考えられる．

### 3.3 アクセラレータプログラミング例

OpenMP によるアクセラレータに対するプログラミングの例を**図 1** に示す．この例は，HPC Challenge Benchmark Suite [26] の中の STREAM ベンチマークを簡略化したものである．

図 1 の 2 行目の **target** 指示文は，その指示文の範囲内（2〜5 行目）はアクセラレータで動作するカーネルであることを示し，そのカーネルの実行を指示する．同じ行にある **map** 節は，ホストとアクセラレータ間のデータ転送を指示しており，カーネル実行前に配列 *b[]* と *c[]* をホストからアクセラレータに転送し，カーネル実行後に配列 *a[]* をアクセラレータからホストに転送することを指示する．3 行目の **teams distribute parallel for** 指示文は，以下に述べる複数の動作を行うための shortcut である．(1) **teams** は，複数のチームを含むリーグを作成する．チームとは複数のスレッドを内包した粗粒度の並列レベルの単位である．(2) **distribute** は，後に続くループ文を各チームのマスタースレッドで分割することを指示する．(3) **parallel for** は，後に続くループ文を各スレッドが並列に実行する．

### 3.4 複数のアクセラレータを利用したプログラミング例

図 1 に対して複数のアクセラレータを用いて並列化した例を**図 2** に示す．

```
1  double a[N], b[N], c[N], scalar = 1.0;
2  int ndevs = omp_get_num_devices();
3  int chunk = N / ndevs;
4  assert((N % ndevs) == 0);
5  #pragma omp parallel num_threads(ndevs)
6  {
7   int dev = omp_get_thread_num();
8   int lb = dev * chunk;
9   int ub = lb + chunk;
10 #pragma omp target map(to:b[lb:chunk],c[lb:chunk]) map(
       out:a[lb:chunk]) device(dev)
11 #pragma omp teams distribute parallel for
12  for (int i=lb;i<ub;i++)
13   a[i] = b[i] + scalar * c[i];
14 }
```

図 2: OpenMP 4.x によるマルチアクセラレータを用いた STREAM ベンチマーク

2 行目の関数 **omp_get_num_devices()** は，ホストに接続されているアクセラレータの数を取得している．3 行目は，1 つのアクセラレータが実行するイテレーション数を決定している．この例では総イテレーション数 $N$ はアクセラレータの数 *ndevs* で割り切れると仮定しているが，一般的には割り切れないケースも考慮する必要がある．5 行目は，各ホストスレッドに 1 つのアクセラレータを操作させるため，**parallel** 指示文を用いてアクセラレータの数のスレッドを生成し，ホスト上で並列実行を行っている．7 行目の関数 **omp_get_thread_num()** は，ホストスレッドの番号を取得している．8〜9 行目は，各アクセラレータが処理するイテレーションの範囲を計算している．10 行目の **target** 指示文は，**device** 節で指定したアクセラレータがその指示文の範囲（10〜13 行目）を実行することを指示している．**map** 節で指定した配列では，配列のある範囲だけをマップするための array section が用いられている．array section では，配列の角括弧内のコロンの左の数値は開始転送要素であり，コロンの右の数字は転送要素数である．11〜13 行目は，各アクセラレータが自分の担当範囲の計算を行っている．

この例からわかるとおり，複数のアクセラレータを用いる場合，各アクセラレータに対するタスクとデータの分割とマップを明示的に行う必要がある．また，本例では登場しないが，アクセラレータ間の通信を必要とするアプリケーションの場合，各アクセラレータ上の分割されたデータに対して，OpenMP が提供する関数 **omp_target_memcpy()** などを用いて通信を発生させる必要がある．これらの作業は，分散メモリシステムにおける MPI を用いたプログラミングに似ており，非常に煩雑である．

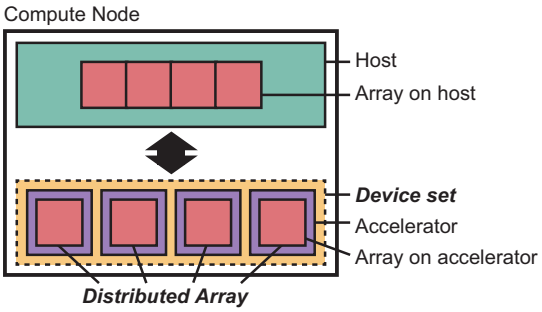


図 3: OpenMP 拡張の概念図

## 4. マルチアクセラレータのための OpenMP 拡張

### 4.1 概要

複数のアクセラレータを簡易に利用するための OpenMP 拡張を提案する．具体的には，3.4 節で行ったような，タスクとデータの分割およびマップを複数のアクセラレータに対して行うための指示文を提案する．分散メモリシステム用の PGAS 言語である XMP が提供する指示文は，タスクとデータの分割およびマップを複数のプロセスに対して行うことができる．このことは複数のアクセラレータを用いる場合とよく似ているため，XMP の指示文をベースに OpenMP 拡張を提案する．

本稿で提案する OpenMP 拡張の概念図を**図 3** に示す．OpenMP 拡張では「デバイスセット」と「分散配列」という概念を用いる．デバイスセットとは 1 つのホストに接続されている複数のアクセラレータの組である．OpenMP 拡張はデバイスセットに対して処理を記述することで，複数のアクセラレータを 1 つの計算資源のように扱うことができる．分散配列はデバイスセット上にマップされた配列である．ホストメモリにある 1 つの配列は，複数のアクセラレータメモリに分散してマップされる．そして，分散配列は PGAS を用いて操作することができる．

OpenMP 拡張は，下記を行うための機能を提供する．

- デバイスセットと分散配列の定義
- ホストとデバイスセット間におけるデータ転送
- デバイスセット内のデバイス間におけるデータ転送

### 4.2 仕様

本節では OpenMP 拡張について説明する．具体的な拡張内容は下記の通りである．

- デバイスセットを扱うために，既存の **device** 節で array section を扱えるように拡張する．
- タスクとデータの分割とマップを行うために，**layout** 節を新設する．**layout** 節は，既存の **map** 節，**declare target** 指示文，**distribute** 指示文と共に現れる．
- ステンシル計算を簡易に行うために，**shadow** 節と **target reflect** 指示文を新設する．**shadow** 節は **layout** 節と共に現れる．
- アクセラレータ間で通信を行うために，**target gmove** 構文，**target bcast** 指示文，**target reduction** 指示文を新設する．

```
1  double a[N], b[N], c[N], scalar = 1.0;
2  #pragma omp target map(to: b,c) map(from: a) device(0:4)
       layout(block)
3  #pragma omp teams distribute parallel for device(0:4)
       layout(block)
4  for(int i=0;i<N;i++)
5   a[i] = b[i] + scalar * c[i];
```

図 4: OpenMP 拡張によるマルチアクセラレータを用いた STREAM ベンチマーク

```
1  double u[N][N], v[N][N];
2  initialize_on_host(u);
3  initialize_on_host(v);
4  #pragma omp target enter data map(to:u) device(0:4)
       layout(block,*)
5  #pragma omp target enter data map(to:v) device(0:4)
       layout(block,*) shadow(1)
6      :
7  for(int k=0;k<TIMES;k++){
8  #pragma omp target teams distribute parallel for layout(v)
9   for(int j=1;j<N-1;j++)
10   for(int i=1;i<N-1;i++)
11    v[j][i] = u[j][i];
12
13 #pragma omp target reflect (v)
14 #pragma omp target teams distribute parallel for layout(v)
15  for(int j=1;j<N-1;j++)
16   for(int i=1;i<N-1;i++)
17    u[j][i]=(v[j-1][i]+v[j+1][i]+v[j][i-1]+v[j][i+1])/4.0;
18 }
19 #pragma omp target exit data map(from:v) layout(v)
```

図 5: OpenMP 拡張による 2 次元ラプラス方程式

図 1 に対して OpenMP 拡張を用いて並列化した例を**図 4** に示す．2 行目と 3 行目の **device** 節は，*device(0)* から *device(3)* の 4 つのデバイスで構成されたデバイスセットを用いることを示している．なお，*device(*)* のようにアスタリスクを指定すると，そのホストが持っているすべてのデバイスという意味になる．2 行目の **layout** 節は，配列 *a[]*，*b[]*，*c[]* をブロック分割し，それらを各デバイスにマップすることを示している．例えば，*device(0)* は *a[0:ceiling(N/4)]* の要素を持つ（*ceiling()* は天井関数である）．3 行目の **layout** 節は，後に続く for 文のイテレーションをブロック分割することを示している．配列のサイズと for 文のイテレーションが一致していないことにも対応するために，3 行目の **layout** 節は *layout(a)* のように配列名も指定できる．この場合，指定された配列の分割に従って for 文のイテレーションが分割される．なお，配列名を指定した場合，処理系は配列の情報からデバイスセットの情報を取得できるため，**device** 節は省略可能である．

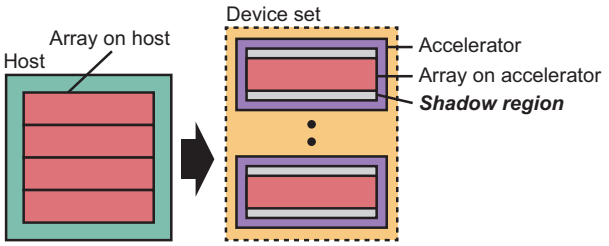


図 6: **shadow** 節の概念図

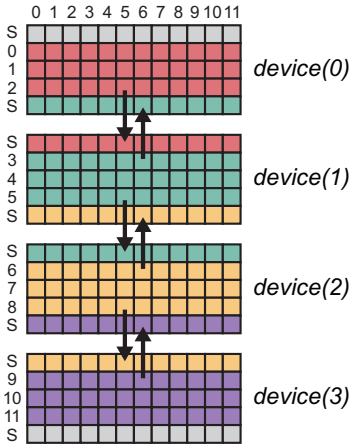


図 7: **target reflect** 指示文の概念図（図中の S は袖領域を指す）

次に，多次元配列に対するデータのマップの例を**図 5** の 2 次元ラプラス方程式を用いて説明する．2 行目と 3 行目はホスト上で配列 *u[][]* と *v[][]* の初期化を行っている．4 行目と 5 行目の **target enter data** 指示文は，配列 *u[][]* の 1 次元目をブロック分割し，それをデバイスセットにマップしている．**layout** 節中においてアスタリスクがある次元は，その次元は分散しないことを意味する．なお，多次元配列において，複数の次元を分散させたい場合は，**#pragma omp target enter data map(to:u) device(0:2,0:2) layout(block,block)** のように，**device** 節において多次元のデバイスセットを定義し，そのデバイスセットに対してデータの分散を行う．5 行目の **shadow** 節は，ステンシル計算のために必要な幅 1 の袖領域を分散配列の上限と下限に追加する．本例における **shadow** 節の概念図を**図 6** に示す．8〜11 行目と 14〜17 行目は，最外ループのみがブロック分割され，4 つのデバイスが並列に実行する．13 行目の **target reflect** 指示文は，袖領域の更新を行うためにデバイス間の隣接通信を行う．本例における **target reflect** 指示文の概念図を**図 7** に示す．この図では配列サイズを *N=12* としており，例えば *device(0)* が持つ *v[2][0:12]* の要素を隣接デバイスである *device(1)* の上の袖領域にコピーすることを示している．

最後に，アクセラレータ間通信について述べる．**図 8** に，**target gmove**，**target bcast** と **target reduction** のコード例を示す．この例では，要素数 16 の配列 *a[]* が

```
1  double a[16], b, c;
2    :
3  #pragma omp target enter data map(to:a) device(0:4)
       layout(block)
4  #pragma omp target enter data map(to:b,c) device(0:4)
5    :
6  #pragma omp target gmove
7    a[1:4] = a[10:4];
8
9  #pragma omp target bcast (b) from_device(0) device(0:4)
10 #pragma omp target reduction (+:c) device(0:4)
```

図 8: **target gmove**, **target bcast** と **target reduction** 指示文のプログラミング例

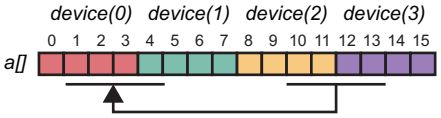


図 9: **target gmove** 指示文の概念図

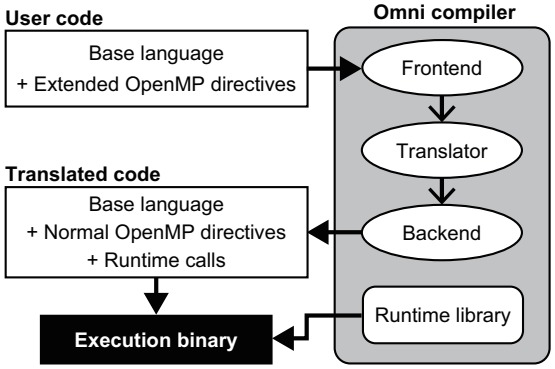


図 10: Omni Compiler のコンパイルフロー

4 つのデバイスに分散されている．また，スカラ変数 *b* と *c* は，4 つのデバイスに重複して保存されている．重複保存する場合は，**layout** 節は必要ない．6〜7 行目の **target gmove** 指示文は，array section と代入文の形式で分散配列に対する通信を発生させる．この通信の概念図を**図 9** に示す．*device(2:2)* 上の *a[10:4]* の値は，*device(0:2)* 上の *a[1:4]* にコピーされる．このように **target gmove** 指示文を用いると，分散配列がどのように分散されているかをユーザは意識せずに，分散配列に対する通信を発生させることができる．9 行目の **target bcast** 指示文は，*device(0)* が持つスカラ変数 *b* を他のデバイスに放送する．10 行目の **target reduction** 指示文は各デバイスが持つスカラ変数 *c* に対して集約通信を発生させ，その合計値を計算している．

### 4.3 実装

本機能の実装には，source-to-source のためのコンパイラ基盤である Omni Compiler [27, 28] を用いる．Omni Compiler の動作の流れを**図 10** に示す．まずフロントエ

表 1: 評価に用いる計算ノードの仕様

| | |
|---|---|
| CPU | Intel Xeon CPU E5-2680 2.7GHz x 2 Sockets |
| Memory | DDR3-1600, 64GB, 51.2 GB/s |
| GPU | NVIDIA Tesla K20Xm x 2 GPUs, GDDR5 6GB, 249.6 GB/s |
| Software | Clang-ytk (hash:49d8020e03), CUDA 9.1 |

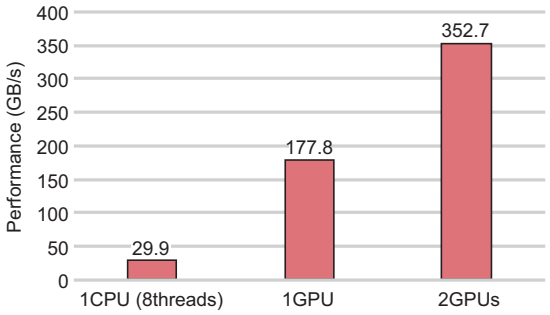


図 11: STREAM ベンチマークの結果

ンドは，本稿で提案した OpenMP 拡張を含むベース言語（C/C++/Fortran）を解析し，その結果を中間ファイルに変換する．次にトランスレータは，その中間ファイルに対してコード変換を行う．最後にバックエンドは，変換された中間ファイルを元のベース言語に変換し，その変換されたコードはバックエンドコンパイラによって実行バイナリに変換される．

本稿で開発する箇所は，フロントエンド，トランスレータ，ランタイムライブラリである．フロントエンドの開発では，OpenMP 拡張を中間ファイルに変換できるようにする．トランスレータの開発では，中間ファイルから複数のアクセラレータを扱える新しい中間ファイルを生成する．トランスレータが生成する中間ファイルは，既存の OpenMP 指示文とランタイムの呼び出しに変換することを考えているため，バックエンドは新しく開発する必要はない．ランタイムライブラリでは，アクセラレータ上に分散配列を確保する関数やデータ転送を行うための関数を作成する．アクセラレータとして NVIDIA 社の GPU を対象とするため，ランタイムライブラリは既存の OpenMP の API や NVIDIA Collective Communications Library（NCCL）[29] を用いて作成する予定である．

## 5. 初期評価

### 5.1 測定環境

**表 1** に示す 2 基の GPU を持つ計算ノードを用いて，提案する OpenMP 拡張の初期評価を行う．Omni Compiler で用いるバックエンドコンパイラには Clang-ytk の最新版（ハッシュは 49d8020e03）を用いた．

### 5.2 結果

図 4 に示した STREAM ベンチマークを用いて，OpenMP 拡張の初期評価を行った．ただし，Clang-ytk のバグを回避するため，Omni Compiler ではなく手動で並列化を行っている．

配列サイズを *N=100,000,000* とし，CPU 1 基（8 スレッド），GPU 1 基，GPU 2 基のそれぞれを使った性能評価を行った．CPU の性能評価に用いたコンパイルオプションは**-O3 -fopenmp** であり，GPU の性能評価に用いたコンパイルオプションは**-O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda** である．結果を**図 11** に示す．この結果より，GPU の結果は CPU の結果よりも高く，また GPU 2 基を使った結果は GPU 1 基の結果のほぼ 2 倍の性能を発揮できることがわかった．

## 6. まとめと今後の課題

本稿では，複数のアクセラレータを簡易に利用するために，タスクとデータの分割およびマップを簡易に行える OpenMP 拡張を提案した．提案内容の初期評価を行うため，OpenMP 拡張を用いて STREAM ベンチマークを実装した．2 基の GPU を持つ計算ノードを用いて性能を測定した結果，提案する OpenMP 拡張は十分な性能を発揮することを確かめた．

今後の課題として，OpenMP 拡張の仕様書の作成，Omni Compiler による実装，他のベンチマークの作成が挙げられる．また，アクセラレータクラスタに対応するために，分散メモリシステム用の指示文ベースの PGAS 言語 XMP と本提案内容とを組合せることも考えている．

## Acknowledgements

本研究は JSPS 科研費 18K11331 の助成を受けたものである．

### 参考文献

[1] TOP500 Supercomputer Sites. `http://www.top500.org`.
[2] The Green500 List. `http://www.green500.org`.
[3] NVIDIA Developer. CUDA Toolkit. `https://developer.nvidia.com/cuda-toolkit`.
[4] OpenCL : The Open Standard for Parallel Programming of Heterogeneous Systems. `https://www.khronos.org/opencl/`.
[5] OpenMP. `https://www.openmp.org`.
[6] OpenACC. `http://www.openacc-standard.org`.
[7] Chamberlain B.L. and Callahan D. and Zima H.P. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, Vol. 21, No. 3, pp. 291–312, August 2007.
[8] Charles Philippe and Grothoff Christian and Saraswat Vijay and Donawa Christopher and Kielstra Allan and Ebcioglu Kemal and von Praun Christoph and Sarkar Vivek. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, Vol. 40, No. 10, pp. 519–538, October 2005.
[9] R. Numwich and J. Reid. Co-Array Fortran for paral-

lel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[10] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 1–26, 2002.

[11] XcalableMP Specification. `http://xcalablemp.org/specification`.

[12] Masahiro Nakao and Hitoshi Murai and Hidetoshi Iwashita and Taisuke Boku and Mitsuhisa Sato. Implementation and evaluation of the HPC challenge benchmark in the XcalableMP PGAS language. *The International Journal of High Performance Computing Applications*, pp. 1–14, 2017.

[13] Michael Wolfe. SCALING OPENACC APPLICATIONS ACROSS MULTIPLE GPUS. `http://on-demand.gputechconf.com/gtc/2014/presentations/S4474-scaling-openacc-across-multiple-gpus.pdf`.

[14] J. Guan and S. Yan and J. M. Jin. An OpenMP-CUDA Implementation of Multilevel Fast Multipole Algorithm for Electromagnetic Simulation on Multi-GPU Computing Systems. *IEEE Transactions on Antennas and Propagation*, Vol. 61, No. 7, pp. 3607–3616, July 2013.

[15] James Beyer. IWOMP 2016 Tutorial: OpenMP Accelerator Model. `http://iwomp2016.riken.jp/wp-content/uploads/2016/10/tutorial-accelerator.pdf`.

[16] S. Potluri and H. Wang and D. Bureddy and A. K. Singh and C. Rosales and D. K. Panda. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pp. 1848–1857, May 2012.

[17] Xu, Rengan and Tian, Xiaonan and Chandrasekaran, Sunita and Chapman, Barbara. Multi-GPU Support on Single Node Using Directive-based Programming Model. *Sci. Program.*, Vol. 2015, pp. 3:3–3:3, January 2016.

[18] T. Komoda and S. Miwa and H. Nakamura and N. Maruyama. Integrating Multi-GPU Execution in an OpenACC Compiler. In *2013 42nd International Conference on Parallel Processing*, pp. 260–269, Oct 2013.

[19] Matsumura, Kazuaki and Sato, Mitsuhisa and Boku, Taisuke and Podobas, Artur and Matsuoka, Satoshi. MACC: An OpenACC Transpiler for Automatic Multi-GPU Use. In *Supercomputing Frontiers*, pp. 109–127, Cham, 2018. Springer International Publishing.

[20] XcalableACC Specification. `http://xcalablemp.org/XACC.html`.

[21] Masahiro Nakao and Hitoshi Murai and Hidetoshi Iwashita and Akihiro Tabuchi and Taisuke Boku and Mitsuhisa Sato. Implementing Lattice QCD Application with XcalableACC Language on Accelerated Cluster. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 429–438, Sept 2017.

[22] OpenMP Compilers and Tools. `https://www.openmp.org/resources/openmp-compilers-tools/`.

[23] Liao, Chunhua and Yan, Yonghong and de Supinski, Bronis R. and Quinlan, Daniel J. and Chapman, Barbara. Early Experiences with the OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pp. 84–98, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[24] clang-ykt/clang. `https://github.com/clang-ykt/clang/`.

[25] Offloading Support in GCC. `https://gcc.gnu.org/wiki/Offloading`.

[26] HPC Challenge Benchmarks. `http://icl.cs.utk.edu/hpcc/`.

[27] Omni Compiler. `http://omni-compiler.org`.

[28] Mitsuhisa Sato, Hitoshi Murai, Masahiro Nakao, Hidetoshi Iwashita, Jinpil Lee, Akihiro Tabuchi. Omni Compiler and XcodeML: An Infrastructure for Source-to-Source Transformation. Platform for Advanced Scientific Computing Conference (PASC16), 2016. `http://omni-compiler.org/download/papers/PASC2016.pdf`.

[29] Nvidia collective communications library (nccl). `https://developer.nvidia.com/nccl`.