Vol.2017-HPC-160 No.22
2017/7/27

フロー解析によるマルチGPU対応OpenACCコンパイラ

松村 和朗[1,a)]　佐藤 三久[2,3]　朴 泰祐[2,4]

概要：近年の高性能計算ではアクセラレータが活用されている。特に 1 ノードに複数の GPU を搭載した構成のスーパーコンピュータが登場しているが、その計算資源を活用するには単一の GPU を用いる場合よりも更なるプログラミングコストが必要となる。アクセラレータ向けプログラミングモデルである OpenACC もその例外でなく、ノード内に存在する各 GPU への処理の割り当てと、それぞれでのデバイスメモリへの読み書きを考慮した通信の追加が求められる。本論文では単一 GPU 向けに記述された OpenACC プログラムを、ノード内に存在する複数 GPU を用いるよう書き換える Source-to-Source コンパイラを提案する。提案手法では OpenACC プログラムを OpenACC と OpenMP を組み合わせた表現へと変換することにより、マルチ GPU への適用を既存のコンパイラを活用して実現する。出力プログラムでは各 GPU との通信はフロー解析の結果から動的に生成され、OpenACC カーネルの実行はそれぞれの GPU に分散される。性能評価では 4 つの GPU を使用した場合、オリジナルと比較して、Himeno Benchmark のサイズ L では約 3.6 倍、NPB-CG の C クラスでは約 3.1 倍高速となった。

1. 序論

近年の高性能計算機システム、特にクラスタ計算機には性能向上を目的としてアクセラレータを用いるものが多数存在する。アクセラレータのなかでも特に GPU に関しては、そのプログラミングがアクセラレータ向けモデルである CUDA や OpenACC を使用して行え、効率的なプログラムを簡潔な記述によって生成できる環境が実現しつつある。OpenACC は、特定のアーキテクチャに依存しないモデルであるが、GPU などの実際のアクセラレータに対応させるためには個々のデバイス向けの専用コンパイラを用いる必要がある。その一方、近年、1 ノードに複数台の GPU を搭載したスーパーコンピュータが登場しており、こうしたマルチ GPU と呼ばれる環境に関しては一般的に OpenACC だけではコードを記述できず、OpenACC の外での GPU 間通信を何らかの形で記述する必要がある。このため、そのような環境では OpenACC の記述性の高さというメリットが削減されてしまう。

本稿では既存の GPU 向け OpenACC コンパイラを前提に、マルチ GPU をサポートする手法を提案する。提案するコンパイラでは OpenACC プログラムのソースコードを OpenACC と OpenMP を組み合わせたプログラムのソースコードへと変換することにより、ノード内にある全 GPU の並列稼働を容易にし、プログラミングコストを減らしつつ、マルチ GPU 環境での性能向上を実現する。コンパイル時のフロー解析の結果から、複数の GPU を対象としたホスト・GPU 間の通信コードは自動生成され、それらの GPU で個別に実行されるようにカーネルは書き換えられる。各 GPU は OpenMP の並列スレッドによって管理され、それらのスレッド間でのデータ共有と先述のホスト・GPU 間通信コードによって並列 GPU 環境での処理が実現される。性能評価では OpenACC で記述された 2 つのベンチマークを提案コンパイラによって変換し、使用する GPU 数を変化させながら性能を測定する。

本稿は本章を含めた 6 章で構成される。2 章では関連研究を紹介し、3 章にて OpenACC を概説する。4 章では提案コンパイラのアルゴリズムを示す。5 章では Himeno Benchmark と NPB-CG を用いた性能評価を行い、6 章では結論を述べる。

2. 関連研究

マルチ GPU への自動適応の研究として、Komada ら [2] は OpenACC のカーネルを GPU ごとになるべく等しいサイズで分割し実行させるコンパイラを提案した。そのコンパイラでは配列を一定サイズのチャンクとして管理し、GPU のデバイスメモリで書き込みが行われた箇所を他の全 GPU へと共有することによりデータの整合性を保っている。手動での効率化手段は新規指示文により提供されて

[1] 東京工業大学 情報理工学院
[2] 筑波大学大学院 システム情報工学研究科
[3] 国立研究開発法人理化学研究所計算科学研究機構
[4] 筑波大学計算科学研究センター
[a)] matsumura.k.ak@m.titech.ac.jp

いるが、本論文が提案するコンパイラではホスト・GPU 間で通信すべきデータの範囲をフロー解析により自動で特定する。

Thejas ら [3] は Polyhedral モデルを元に C 言語で記述されたプログラムの特定部分をマルチ GPU 上で実行し、必要となる通信は細かな領域の重ね合わせと適切なバッファ管理によって効率的に特定させた。適用可能なプログラムはループカウンタと配列インデックスがアフィンで表現できるループに限られてしまうが、本研究では OpenACC コンパイラのマルチ GPU への対応を目的としているため対象となるプログラムに課す制約は抑えた。ここでインデックスがアフィンであるとは、その式が定数倍した変数と定数の加算であることをいう。

HYDRA[5] は分散環境において各ノードでの単一アクセラレータ利用を自動化するコンパイラシステムであり、元にしている通信生成アルゴリズムは本論文と同一である。この研究では簡易指示文によりノード分散のプログラムを生成するが、本研究では OpenACC を対象としており元のプログラムが動作するノードにて複数 GPU を使用するプログラムを生成させ、その手法はソースコード変換として示す。

最新の NVIDIA GPU では NVLink[6] インターコネクトで GPU・GPU 間の通信を高速に実現できる。NVLink 環境では共有アドレス空間でのデータ共有が各 GPU のデバイスメモリを利用して行うことができ、それに伴って CUDA では複数 GPU にまたがった処理を記述可能になるなど、マルチ GPU をより柔軟に扱うためのシステムが登場している。しかしこうしたシステムはハードウェアサポートに大きく依って実現されている。本研究では OpenACC と OpenMP が利用可能なシステムにおいてマルチ GPU に適応する手法を提案する。この手法はマルチ GPU だけでなく複数台のアクセラレータを搭載した、より一般的な環境にも適用できると考えられる。

また分散環境におけるアクセラレータの利用に対して新たなプログラミングモデルが考案されている [7][8]。これらは分散指示とアクセラレータ使用を明示的に行うモデルである。

3. OpenACC の概要

OpenACC は C 言語などで記述されたソースコードでの指示文を基本とした操作により、アクセラレータを利用することを可能にするプログラミングモデルである。本章では OpenACC が有する実行モデルとメモリモデルの概要と、OpenACC からマルチ GPU を使用する方法を述べる。図 1 に C 言語による OpenACC のソースコード例を示す。

3.1 実行モデル

OpenACC プログラムの実行はホスト (CPU) を中心に

```
#pragma acc data copyout(x[0:N]), present(y)
#pragma acc kernels
for (int i = 0; i < N; i++)
  x[i] = y[i] * y[i];
```

図 1 OpenACC によるアクセラレータ利用

```
numgpus=acc_get_num_devices(acc_device_nvidia);
#pragma omp parallel num_threads(numgpus)
{
  int tnum = omp_get_thread_num();
  int sz = N / numgpus;
  int lb = sz * tnum;
  int ub = lb + sz;
  acc_set_device_num(tnum, acc_device_nvidia);

#pragma acc data copyout(x[lb:sz]), present(y)
#pragma acc kernels
    for (int i = lb; i < ub; i++)
      x[i] = y[i] * y[i];
}
```

図 2 OpenMP による OpenACC のマルチ GPU 使用

行われ、アクセラレータでの実行はホストからのオフロードとして実現される。オフロードされる領域は `parallel` と `kernels` の指示文により構文として明示される。

`parallel` 構文により指定された領域 (並列領域) はアクセラレータ上で実行される。内部のループ文をアクセラレータの構造に合わせて並列化する場合には `loop` 構文によりアクセラレータ非依存の並列性 (粗粒度並列性 gang・細粒度並列性 worker・SIMD 並列性 vector) を指定でき、実行時にはこの並列性が個々のアクセラレータ固有の並列性へとマッピングされる。このようにしてアクセラレータ上で実行されるネストループはカーネルと呼ばれる。また `parallel` 構文への節の付加により並列サイズの指定とリダクション・非同期処理などの実現が可能である。

`kernels` 構文により指定された領域はカーネルの集合とされ、その内部のループ文はカーネルとして実行される。

3.2 メモリモデル

GPU など、内部メモリ (デバイスメモリ) を持ちこれを演算対象とするアクセラレータを使用する場合には、実行するプログラムがアクセスするデータをデバイス内へ送信する必要がある。またアクセラレータ上で更新したデータをホストで使用する場合にはデバイスメモリからホストメモリへと転送する必要がある。

OpenACC では `data` 構文によりアクセラレータ上のデータを定義できる。`data` 構文により指定された領域では対象データがアクセラレータから参照可能になり、その領域の実行前後では節の指定に応じてホスト・アクセラレータ

間でのデータ転送が行われる。data 構文により指示されていないデータに対して OpenACC は暗黙的な自動転送を行うが、適切かつ効率的なデータ移動のために data 構文の使用が推奨される。data 構文もしくは parallel 構文の present 節を用いて指示されたデータは領域内で定義済みであることが明示され、コンパイラによる誤った自動転送を防ぐことができる。

またアクセラレータ上で定義済みのデータをホスト・アクセラレータ間で転送するために update 指示文が用意されている。update 指示文では定義済みデータを指定したサイズ分だけ送受信でき、これは通信量の削減に活用できる。

3.3 マルチ GPU の使用

OpenACC が操作するアクセラレータは基本的に 1 つであり、複数個のアクセラレータを用いる場合は各デバイスに対して明示的に並列化などを行わなければならない。複数デバイスで並行に処理するためには、OpenACC が提供している非同期処理か、マルチスレッドもしくはマルチプロセスによる実行が必要となる。

図 1 と同等の処理を OpenMP のマルチスレッディングによりマルチ GPU で並行実行させるよう変更したソースコードを図 2 に示す。ここではノード内の GPU ごとにスレッドを割り当て、各スレッドは自身が担当する通信領域とループ実行区間を求め、それに応じた処理を OpenACC により実行する。関数 acc_set_device_num ではそのスレッドで使用するアクセラレータを切り替える処理が行われる。

4. ソースコード変換によるマルチ GPU 対応

本章ではマルチ GPU 使用におけるプログラミングコストの削減を目的として OpenACC のマルチ GPU 対応手法を提案する。提案するコンパイラは単一アクセラレータを用いる OpenACC プログラムのソースコードを、OpenACC と OpenMP を組み合わせたコードへと変換することにより、既存の GPU 向け OpenACC コンパイラを活用してマルチ GPU をサポートする。

図 3 に変換フローを示す。提案コンパイラでは、まず OpenACC が提供している省略文法を基本的な文法を用いて書き換える。それに加え kernels 構文の領域内にある tightly nested loop (最内ループ以外は内側に一個のループしか持たない構造) を、イテレーション間で依存が無ければ parallel 構文を用いて明示的に OpenACC のカーネルとする。次に後述のフロー解析によって、各並列領域の配列インデックスを並列領域外にて定義される変数で表現して抽出する。そしてノード内にある全 GPU でカーネル実行を分散するコードを出力する。変換後のソースコードは OpenMP と OpenACC を同時にサポートしているコンパイラによってコンパイル可能であり、生成されたコードで

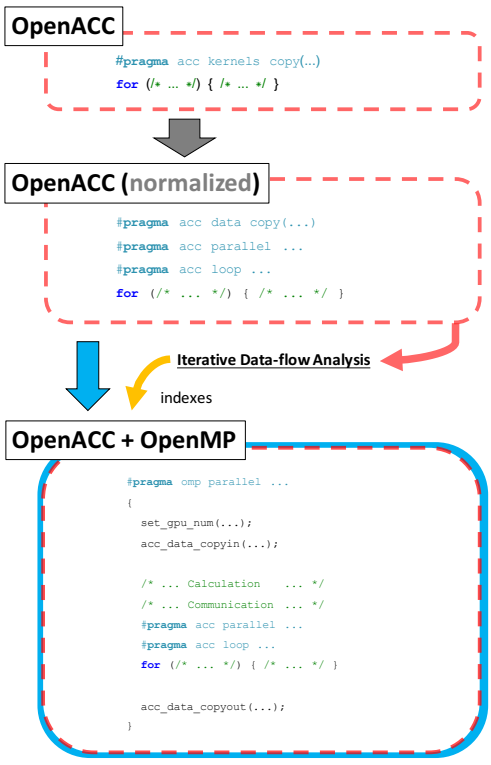

図 3 提案コンパイラの変換フロー

は、解析結果を利用してホスト・GPU 間の通信を動的に生成する。

4.1 実行方法

OpenACC のカーネルは主に最外ループが並列実行されるため、メモリアクセスにおいてプロセッサ間でのデータ依存を抑えるプログラミングが、効率的な実行では求められる。本研究もこれを仮定し、カーネルの最外ループを GPU ごとに等しいサイズで分割して実行を分散する。

提案コンパイラでは書き込みがアフィンアクセスであり、その区間が GPU 間で重複しない場合に限ってマルチ GPU 上での実行を行う。これらの条件を満たさないカーネルでは単一 GPU 上での実行を行う。この仮定は厳しいように見えるが、例えば直交座標上での空間分割手法などでは一般的に守られる制約であり、適用可能な局面は非常に多い。

4.2 通信生成

実行される並列領域は動的に決定されるため、各 GPU が依存するデータを特定することは難しい。そのため data 構文によってホストから GPU へと転送されるデータ (copyin) に関しては全 GPU 上で複製する (アルゴリズム 1)。

単一 GPU 上の実行では data 構文の指示に従うのみでホスト・GPU 間のデータ依存を解決できたが、マルチ GPU 上の実行では各 GPU がデバイスメモリを有するため GPU・

GPU 間のデータ依存も考慮しなければならない。提案コンパイラではこの依存を解決するために必要な通信を Okwan ら [10] のアルゴリズムを元にして特定する。ここでは並列領域・GPU・配列の組み合わせごとに読み込み区間 *USE*・書き込み区間 *DEF* を求める。これらは後述のフロー解析の結果から求められる。各並列領域の実行前にはその値の重ね合わせによって GPU・GPU 間で発生する通信の範囲を算出して、実際に通信を行い、GPU ごとの変更済み区間 *DIRTY* を更新する (アルゴリズム 2)。全ての区間は上限・下限の二値で表される。後述するように GPU・GPU 間通信は OpenACC のランタイムによりホストメモリを経由して行うため、GPU からホストへの通信は内容が重複しないように最適化される。

`data` 構文によって GPU からホストへと転送されるデータ (`copyout`) に関しては、各 GPU からホストへデータ全体の転送を行ってしまうと整合性が取れなくなる。そこで各 GPU はそれぞれの *DIRTY* を活用して最小限の転送を行う (アルゴリズム 3)。

`update` 指示文によってホストからアクセラレータへと転送されるデータは各 GPU で複製し、アクセラレータからホストへの転送は *DIRTY* と重なる区間だけ行う (アルゴリズム 4)。

アルゴリズム 1 `copyin` の生成

```
Create DIRTY
for each gpu i ∈ GPUs do
    Communicate a whole array from Host to GPU i
    DIRTY[i].valid ← false
end for
```

アルゴリズム 2 並列領域実行前の通信生成

```
for each gpu i ∈ GPUs do
    if DIRTY[i] ∩ DEF[i] = ∅ or
            ∃ d ∈ (DEF \ DEF[i]); DIRTY[i] ∩ d ≠ ∅ then
        Communiate DIRTY[i]
                from GPU i to all other GPUs
        DIRTY[i] ← DEF[i]
    else
        for each gpu j ∈ GPUs; j ≠ i do
            COMM[j] ← DIRTY[i] ∩ USE[j]
        end for
        /* GPU からホストへの通信を最適化 */
        GHs ← GPU_TO_HOST_COMM(COMM)
        Communicate for each GHs from GPU i to Host
        if ∃ gh ∈ GHs; DIRTY[i] ⊂ gh then
            DIRTY[i].valid ← false
        end if
        for each gpu j ∈ GPUs; j ≠ i do
            Communiate COMM[j] from Host to GPU j
        end for
        DIRTY[i] ← DIRTY[i] ∪ DEF[i]
    end if
end for
```

アルゴリズム 3 `copyout` の生成

```
for each gpu i ∈ GPUs do
    Communicate DIRTY[i] from GPU i to Host
end for
Delete DIRTY
```

アルゴリズム 4 GPU からホストへの `update` の生成

```
US ← the update section
for each gpu i ∈ GPUs do
    Communicate DIRTY[i] ∩ US from GPU i to Host
end for
```

4.3 フロー解析

提案コンパイラでは、*USE*・*DEF* を求めるためにフロー解析を行う。この解析は各並列領域に対して行い、その内部にて各配列への読み込み・書き込みに用いられるインデックスを取得する。このインデックスは定数とループカウンタ、そして並列領域外で定義される変数によって表現する。出力プログラムでは各並列領域の実行前に *USE*・*DEF* の実際の値が並列領域外の変数と GPU 数などに応じて動的に求められる。

この解析では並列領域内で定義または上書きされている変数を元の式へと展開させながらインデックスを回収する。並列領域は内部にループや制御構造を含むため、その制御フローグラフに対する解析は、取得したインデックスが変化しなくなるまで繰り返し行う (Iterative Data-flow Analysis)。

提案コンパイラではまずソースコードを静的単一代入形式へと変換する。そして Iterative Data-flow Analysis により、ループカウンタ以外の変数を展開させながらインデックスの回収を行う。ここで変数の値が、その変数自身を含む加減乗除で表される場合に、その変数を発散とする。取得したインデックスがアフィンではない、もしくは発散する変数を含む場合は、対象の区間は配列全体を指すものとする。その他のインデックスはアフィンであるため、ループカウンタの上限・下限をその式へと代入することによってインデックスが取りうる区間を特定できる。生成されたコードでは、各配列への書き込みがアフィンアクセスでありその区間が GPU 間で重複しない場合に、GPU ごとにカーネルのトップループを分割してそのループの区間における *USE*・*DEF* を算出する。

4.4 出力形式

提案コンパイラではそれぞれの指示文に対して書き換えを行い、マルチ GPU に対応したプログラムを OpenACC と OpenMP を組み合わせて表現する。生成コードでは、前述の通信生成をサポートするため、OpenACC の通信ランタイムをラップしたルーチンを用いる。このルーチンは、*DIRTY* を GPU・配列の組み合わせ毎に作成・参照・更新

```
#pragma acc data copy(x[0:N])
{
  /* ... */
}
```

```
copyin_routine(x, 0, N);
{
  /* ... */
}
copyout_routine(x);
```

図 4 data 構文の変換前 (上) と変換後 (下)

```
/* ... */
#pragma acc update host(x[a:b])
/* ... */
```

```
/* ... */
update_host_routine(x, a, b);
/* ... */
```

図 5 update 指示文の変換前 (上) と変換後 (下)

```
#pragma acc parallel
/* ... 並列領域 ... */
```

```
if (/* 区間が変化するならば */) {
  /* 区間の再計算 */
}
#pragma omp parallel num_threads(NUMGPUS)
{
  int tnum = omp_get_thread_num();

  // 使用するGPUの設定
  set_gpu_num(tnum);

  // 区間の設定 (通信が発生する)
  set_data_section(/* ... */);

  // 通信の待機
  #pragma omp barrier

  #pragma acc parallel
  /* ... 並列領域 ... */
}
```

図 6 parallel 構文の変換前 (上) と変換後 (下)

```
#pragma acc parallel
#pragma acc loop gang reduction (+ : sum)
for (i = X; i < Y; i++) {
  sum += a[i+p];
}
```

```
{
  static int sections_are_changed = 1;
  sections_are_changed =
    (sections_are_changed
    || last_p != p
    || last_X != X
    || last_Y != Y);
  if (sections_are_changed) {
    section_are_changed = 0;
    last_p = p;
    last_X = X;
    last_Y = Y;

    /* ループ実行範囲の計算 */
    calc_loop_sections(
      loop_sections, X, Y,
      1 /* 増減 */,
      0 /* X == Y のときに実行するか */);

    /* USE, DEFの計算 */
    init_uses(a_uses);
    init_defs(a_defs);
    for (i = 0; i < NUMGPUS; i++) {
      update_section(a_uses[i],
                     loop_sections[i].lb + p);
      update_section(a_uses[i],
                     loop_sections[i].ub + p);
    }
  }
}

#pragma omp parallel num_threads(NUMGPUS) \
          reduction (+ : sum) private (i)
{
  int tnum = omp_get_thread_num();
  set_gpu_num(tnum);

  set_data_section(tnum, a_uses, a_defs);

  #pragma omp barrier

  #pragma acc parallel
  #pragma acc loop gang reduction ( + : sum )
  for(i = loop_sections[i].lb;
      i <= loop_sections[i].ub; i++) {
    sum += a[i + p];
  }
}
```

図 7 parallel 構文の変換例 (変換前:上・変換後:下)

しながら、通信を生成する。data 構文と update 指示文はルーチンの実行として変換され (図 4 及び図 5)、parallel 構文の実行に伴う通信もルーチンの実行によって実現する。data 構文をルーチン実行へと置き換える際には、GPU 上にデータが保持されていることを明示するためにその領域内にある parallel 構文へ present 節の付加を行う。

図 6 に parallel 構文の変換形式を示す。まずループの実行区間と *USE*・*DEF* の算出を行う。そして OpenMP スレッドを作成し、前述の通信を GPU ごとに並行して行う。通信の完了を OpenMP の barrier 指示文によって待機したあとは、元の parallel 構文の実行を行う。ここで対象領域のプログラムがカーネルであり第 4.1 節で述べた条件を満たしていればトップループの分割が行われ、その実行が各 GPU に分散される。図 7 に変換例を示す。区間の算出では、式が含んでいる変数が変化しない限り、前回計算した値が再利用される。並列領域外で定義した変数はスレッド間で共有されるため、マルチ GPU 上の実行でもその書き換えが可能であるが、ループカウンタとして用いている場合はそれを OpenMP によりスレッド毎に複製する。リダクションはそれぞれの GPU ごとの結果を OpenACC により計算した後、全体の結果を OpenMP の reduction 節によりスレッド間で求める。

4.5 実装

提案コンパイラは XcodeML [12] に対するコード変換として実装した。C 言語から XcodeML、XcodeML から C 言語への変換は Omni コンパイラ [13] を用いて行っている。出力したソースコードから実行プログラムへのコンパイルは PGI コンパイラ 16.4 にて行えることを確認している。

現在は C 言語で記述された OpenACC プログラムをサポートしている。提案コンパイラの入力では、配列の次元は 1 次元のみに対応し、OpenACC ランタイムルーチンの呼び出しが含まれることを考慮していないため、それによって確保されるデバイスメモリを複数 GPU 上で複製することはできない。

本研究では二種類の実装を行った。一つは、入力で指定された粗粒度並列性 gang のサイズをカーネル分散時に各 GPU へ均等に分割して実行するものである (gang-div)。もう一つは指示文の変換時に gang サイズの指定を削除して、OpenACC ランタイムによりそのサイズを自動で決定させるものである (gang-rm)。

5. 評価

5.1 ベンチマーク

評価には Himeno Benchmark [14] と NAS Parallel Benchmarks CG (NPB-CG) [15] を用いた。単一アクセラレータ向けに OpenACC で記述されたこれらのプログラムを、本研究で実装したコンパイラによりマルチ GPU 上での実行に適用させ、性能を測定した。

Himeno Benchmark はヤコビ法によりポアソン方程式を解くベンチマークである。オリジナルの OpenACC プログラムでは計算部分と代入部分がアクセラレータ上で実行され、それがタイムステップとして繰り返される。オリジナルにおける gang サイズは 64 である。提案手法によりマルチ GPU 環境への適用を行った場合は、計算カーネルの実行ごとに GPU 間で袖通信が発生する。

NPB-CG は共役勾配法により正値対称な大規模疎行列の最小固有値を求めるベンチマークである。オリジナルの OpenACC プログラムでは疎行列ベクトル積や固有値計算がアクセラレータへオフロードされ、それが繰り返される。オリジナルでの疎行列ベクトル積における gang サイズは行列の行数と等しい。提案手法によりマルチ GPU 環境への適用を行った場合は、疎行列ベクトル積ごとに問題サイズ分のデータが各 GPU から他の全 GPU へ通信される。

5.2 性能

性能評価には筑波大学計算科学研究センターの HA-PACS/TCA [16] の 1 ノードを用いた。ノードの構成を表 1 に示す。出力したソースコードは PGI コンパイラ 16.4 によってコンパイルし実行させた。

表 1 HA-PACS/TCA におけるノードの構成

CPU	Intel Xeon E5-2680v2 10 core	×	2
CPU Memory	DDR3 1866MHz 32GB	×	4
GPU	NVIDIA K20X	×	4
GPU Memory	GDDR5 6GB	/	GPU
OS	CentOS 6.4		

5.2.1 Himeno Benchmark

問題サイズ Large $(i \times j \times k) = (256 \times 256 \times 512)$ に対して、使用する GPU の個数を増加させていった結果を図 8 に示す。gang-div では GPU 数 1 と比べた場合、GPU 数 2, 3, 4 のときにそれぞれ約 1.78 倍, 約 2.54 倍, 約 3.35 倍と、線形に近い速度向上を確認した。一方 gang-rm では GPU 数 3 のときに gang-div よりも性能が低下した。これは OpenACC ランタイムによって自動で決定された並列サイズが非効率なものであったためと推察される。しかし GPU 数 4 のときには gang-div を上回る約 3.62 倍の速度向上が得られていることから、最適な並列サイズは分割された強スケーリングケースでは単純に求められないと思われる。

次に weak scaling の結果を示す。これは各 GPU に $(i \times j \times k) = (128 \times 256 \times 256)$ の処理を行わせ、その 1 タイムステップの処理にかかった平均時間を表している。どちらの実装でも GPU 数を 1 から 2 へ変化させた時に 0.3ms 程度のオーバヘッドが見られた。GPU 数を 2 から

3、3 から 4 へと変化させた場合はオーバヘッドが 0.1ms 程度となった。

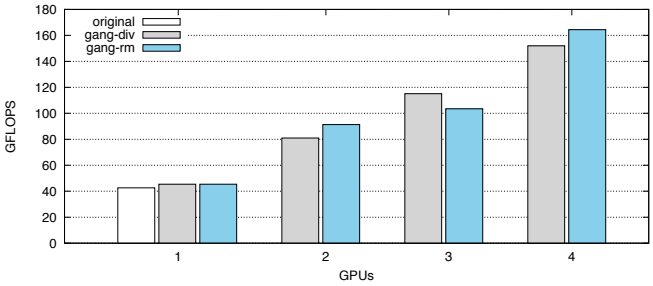

図 8 Himeno Benchmark の性能

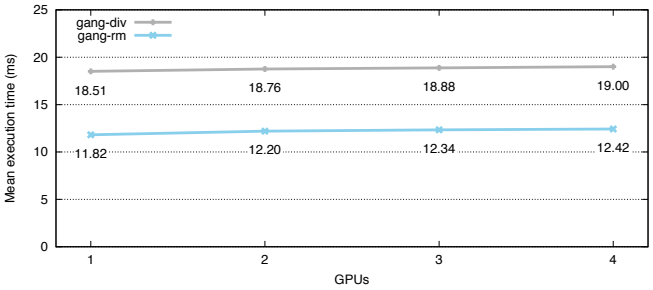

図 9 Himeno Benchmark の weak scaling 性能

5.2.2 NPB-CG

問題クラス A (行数が 14,000), クラス B (75,000), クラス C (150,000) に対して、使用する GPU の個数を変化させていった結果をそれぞれ図 10, 図 11, 図 12 に、また gang-rm での並列化効率を図 13 に示す。gang-div と gang-rm では性能に大きな違いはみられなかった。まずクラス A ではほぼ性能向上せず GPU 数 4 のときには GPU 数 3 よりも性能が低下した。一方クラス B 及びクラス C では問題規模が大きくなった影響で並列化効率が上がり、クラス C では GPU 数 4 のときに約 3.12 倍の速度向上を確認した。

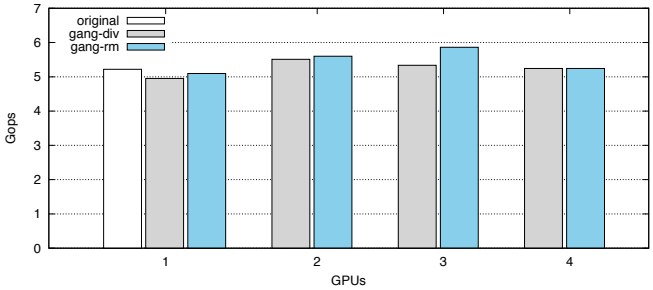

図 10 NPB-CG クラス A の性能

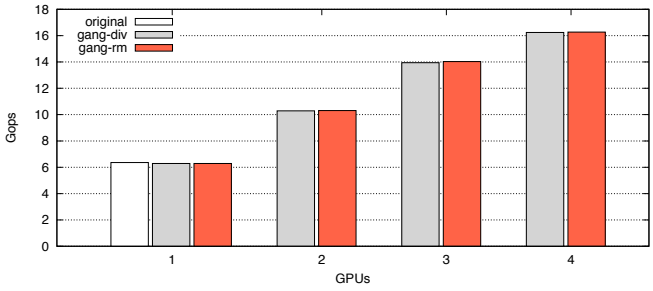

図 11 NPB-CG クラス B の性能

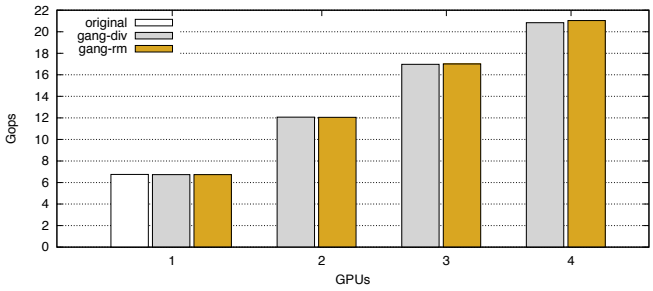

図 12 NPB-CG クラス C の性能

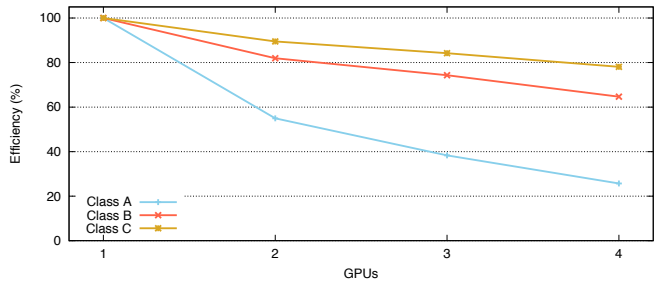

図 13 NPB-CG での並列化効率

6. 結論

本稿では OpenACC のマルチ GPU サポートを既存のコンパイラを用いて行う手法を提案した。大きな仮定として、配列への書き込みがアフィンであることを前提としている。1 ノード上の複数 GPU はノード上の複数コアによって実行される OpenMP コードによって並列化され、各スレッドが 1 つずつの GPU をそれぞれ管理するようにコードが生成される。これによりホスト・GPU 間通信の生成とカーネル実行の分散が自動化され、単一アクセラレータと同様のプログラミングを行うだけで、マルチ GPU 上での実行を可能とすることができた。

課題として完全な OpenACC のサポートと更なる性能調査、そしてカーネルの性質に応じたマルチ GPU 実行の最適化が挙げられる。

謝辞 本論文を執筆する際には筑波大学大学院システム情報工学研究科の田渕晶大氏、津金佳祐氏の協力を得ました。感謝申し上げます。本研究における HA-PACS/TCA の利用は筑波大学計算科学研究センター学際共同利用プロ

グラム・平成 28 年度課題「アクセラレータおよびメニーコアを搭載したクラスタシステムのための高生産並列言語の開発と評価」による。

参考文献

[1] OpenACC-standard.org. OpenACC. https://www.openacc.org/.
[2] Komoda Toshiya, Shinobu Miwa, Hiroshi Nakamura, and Naoya Maruyama. Integrating multi-GPU execution in an OpenACC compiler. In *the 42nd International Conference on Parallel Processing (ICPP)*, 2013.
[3] Thejas Ramashekar, and Uday Bondhugula. Automatic data allocation and buffer management for multi-GPU machines. In *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 10, No. 4, Article 60, 2013.
[4] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *the 16th ACM symposium on Principles and practice of parallel programming (PPoPP)*, 2011.
[5] Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann. HYDRA : Extending Shared Address Programming for Accelerator Clusters. In *the 28th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2015.
[6] NVIDIA NVLink High-Speed Interconnect. NVIDIA. http://www.nvidia.com/object/nvlink.html.
[7] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. XcalableACC: Extension of XcalableMP PGAS Language using OpenACC for Accelerator Clusters. In *2014 First Workshop on Accelerator Programming using Directives (WACCPD)*, 2014.
[8] Jungwon Kim, Seyong Lee, and Jeffrey S. Vetter. An OpenACC-Based Unified Programming Model for Multi-accelerator Systems. In *the 20th ACM symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.
[9] Rengan Xu, Sunita Chandrasekaran, and Barbara Chapman. Exploring Programming Multi-GPUs Using OpenMP and OpenACC-Based Hybrid Model. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2013.
[10] Okwan Kwon, Fahed Jubair, Seung-Jai Min, Hansang Bae, Rudolf Eigenmann, and Samuel Midkiff. Automatic Scaling of OpenMP Beyond Shared Memory. In *the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2011.
[11] William Blume, and Rudolf Eigenmann. Symbolic range propagation. In *the 9th International Symposium on Parallel Processing (IPPS)*, 1995.
[12] Omni Compiler Project. XcodeML. http://omni-compiler.org/xcodeml.html.
[13] Omni Compiler Project. Omni Compiler. http://omni-compiler.org.
[14] 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/supercom/himenobmt/.
[15] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html.
[16] Center for Computational Sciences, University of Tsukuba. HA-PACS Project. https://www.ccs.tsukuba.ac.jp/research_project/ha-pacs/.