Vol.2013-ARC-207 No.22
Vol.2013-HPC-142 No.22
2013/12/17

アクセラレータのメモリ容量を超えるデータをパイプライン処理するためのディレクティブ

中野 瑛仁[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，アクセラレータのメモリ容量を超える大規模データを，単一アクセラレータ上で高速処理するためのディレクティブ仕様 PACC（Pipelined ACCelerator）を提案し，その実装を示す．PACC は，既存のアクセラレータ向けディレクティブ仕様 OpenACC を拡張したものであり，ディレクティブの挿入のみで大規模データの分割処理を可能とする．PACC のトランスレータは，PACC ディレクティブが付加されたプログラムを入力とし，データを分割処理するよう書き換え，OpenACC ディレクティブを挿入したプログラムを出力する．さらに，PACC が出力したプログラムは，分割したデータをパイプライン処理することで，CPU・アクセラレータ間のデータ転送時間の一部をアクセラレータ上の計算時間により隠蔽する．この隠蔽は，分割数を指定するための PACC ディレクティブを用いてチューニング可能である．実験では，CPU 側のメモリ容量が 32 GB，アクセラレータのメモリが容量 5 GB の PC において，27 GB 程度のデータまで処理できることを確認した．また，分割したデータをパイプライン処理することにより，最大で 2.6 倍の高速化を得た．

キーワード：OpenACC，パイプライン処理，ステンシル計算，GPU

Directives for pipelined processing of data exceeding the accelerator memory capacity

Akihito Nakano[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, we present a directive specification, named pipelined accelerator (PACC), and its implementation for acceleration of large-scale computation whose data size exceeds the memory capacity of an accelerator. PACC is an extension of OpenACC, an existing directive specification for accelerators. Our specification realizes partitioning of large-scale data according to inserted directives. Given a program with PACC directives, our PACC translator rewrite the program as an OpenACC program so that data can be partitioned for execution. Furthermore, the generated program processes partitioned data in a pipeline so that data transfer time between the CPU and accelerator can be overlapped with computation time on the accelerator. This overlap can be tuned by using PACC directives that specify the number of data partitions. Experimental results show that PACC realizes large-scale computation with data size of up to 27 GB on a PC whose main and accelerator memory capacities are 32 GB and 5 GB, respectively. We also find that our pipelined execution increases the performance by a maximum factor of 2.6.

Keywords: OpenACC, pipeline, stencil computation, GPU

1. はじめに

GPU（Graphics Processing Unit）などのアクセラレータは，CPU を凌駕する計算性能を有する．この高い計算性能を科学計算の高速化に利用するために，CPU およびアクセラレータを搭載した複合型計算機システムが増加している．

一般に，アクセラレータ向けのプログラムは，それぞれ固有のプログラミング言語を用いて記述する必要がある．また，アーキテクチャの違いにより，高速化のためにはア

[1] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

クセラレータごとに異なる最適化手法を適用する必要がある．したがって，複合型計算機システム上に既存のアプリケーションを移植する際の手間は多い．長期的に高い性能を維持する必要のある科学計算プログラムにおいて，度重なる移植のコストを緩和するためには，複数のシステムにおいて実行可能であり，かつ高い実行性能を達成できる性質（性能可搬性）を持つプログラムを記述することが重要である．

このような背景から，性能可搬性が高く移植コストの小さいディレクティブベースのプログラミングが注目されている．例えば，OpenACC[1] の指示文（ディレクティブ）をCPU 向け逐次プログラムへ挿入することで，そのプログラムはアクセラレータ上でも実行可能になる．また，性能チューニングはディレクティブに適切なパラメータを指定することで適用できる．したがって，プログラムをアクセラレータへ移植・最適化するためには，単にディレクティブを追加または削除し，パラメータを調整するのみでよい．しかし，GPU のように，CPU から独立したメモリ（デバイスメモリ）を持つアクセラレータに対しては，OpenACC ディレクティブを挿入しパラメータを調節するだけでは対応できない．

まず，デバイスメモリ容量が CPU 側のメモリ（ホストメモリ）容量よりも小さいため，アクセラレータ上での実行時にデバイスメモリ不足となり得る．例として，OpenACC のサンプルコードを**図 1** に示す．OpenACC はホストメモリ上の配列 `dst[1:n-2]` および `src[0:n]` に対応する配列をデバイスメモリ上に確保する．これらの配列サイズは，問題サイズ `n` に依存するため，デバイスメモリが不足して実行に失敗しうる．このメモリ不足を防止するためには，配列 `src` および `dst` を複数のデータのかたまり（チャンク）に分割し，チャンクを 1 つずつアクセラレータ上へ転送して計算する必要がある．例えば，**図 2** に示すようにデータを `num_tasks` 個に分割することで，問題サイズによらず実行できる．しかし，この書き換えは`#pragma acc` から始まる行を無視するだけでは取り消せないため，OpenACC 以外のディレクティブと衝突する可能性がある．したがって，OpenACC が未対応の計算機へ移植する際，移植コストが発生する．

次に，CPU・アクセラレータ間のデータ転送時間を隠蔽できない．一般に，アクセラレータ上の計算は CPU・アクセラレータ間の低速なデータ転送を伴う．GPU の場合，CPU・GPU 間のデータ転送を GPU 上の計算とオーバラップすることにより，データ転送時間の一部を隠蔽できる．しかし，処理のオーバラップを実現するためには，独立に処理可能なチャンクが 2 つ以上必要である．この最適化を施せるのは図 2 に示すようにデータを 2 つ以上に分割した場合のみである．前述のように，この書き換えはプログラムを OpenACC 未対応の計算機へ移植する際，移植コストを発生させる．

```
1  #pragma acc parallel loop \
2      copyin(dst[1:n-2]) copyout(src[0:n])
3  for (i = 1; i < n-1 ; i++) {
4    dst[i] = src[i-1] + 2*src[i] + src[i+1];
5  }
```

図 1 デバイスメモリ不足となり得るプログラムの例

```
1  for (j = 1; j < n/num_tasks; j++) {
2  #pragma acc parallel loop \
3      copyin(dst[1+task_size*j:task_size-2]) \
4      copyout(src[task_size*j:task_size])
5    for (i = 1; i < task_size-2; i++) {
6      dst[i+task_size*j] = src[i-1+task_size*j] +
7                           2*src[i+task_size*j] +
8                           src[i+1+task_size*j];
9    }
10 }
```

図 2 図 1 にデータ分割を施したプログラムの例

そこで，本研究では以下の 2 点をディレクティブのみで実現可能とするために，OpenACC を拡張した新たなディレクティブ PACC（Pipelined ACCelerator）を提案し，その実装を示す．

- データ分割による，大規模データのアクセラレータ実行
- パイプライン処理によるデータ転送時間の隠蔽

PACC のディレクティブ仕様を設計するために，OpenACC に不足しているデータ分割のためのディレクティブを定義する．また，パイプライン処理のチューニングに必要な情報およびそれを与えるためのディレクティブを定義する．

トランスレータの実装に関しては，GPU を対象としている．書き換えの内容はデータ分割，OpenACC ディレクティブの挿入，および CUDA[2] 関数の挿入である．なお，CUDA 関数の用途はデバイスメモリ容量の確認のみであるため，OpenACC が対応する GPU 以外のアクセラレータへの移植は容易だと考えられる．

以降では，まず 2 章で関連研究を紹介し，3 章で PACC の設計および実装について述べる．4 章では PACC トランスレータの生成するプログラムを評価し，5 章で本研究をまとめる．

2. 関連研究

アクセラレータを対象としたディレクティブ仕様の 1 つに，OpenMPC[3] がある．OpenMPC は OpenMP の拡張であり，OpenMP プログラムを CUDA プログラムに変換できる．また，チューニングのために追加されたディレクティブを挿入することで，GPU 上で立ち上げるスレッド数など

を指定し，GPU 上の計算を高速化できる．OpenMPC の拡張として，Sabne ら [4] は処理をパイプライン化することで大規模データを GPU 上で実行可能としている．Sabne らは，OpenMPC に対し，新たなディレクティブを追加することなくパイプライン化を実現している．ただし，パイプライン化の対象は`#pragma omp pipeline`で囲まれた領域，つまり GPU 上の計算部分およびそれに伴う CPU・GPU 間通信である．一方，PACC のディレクティブ仕様では，GPU の計算の前後にある CPU の処理もパイプライン処理のステージとして定義できる．また，Sabne らはデータ分割の粒度を自動チューニングし，パイプライン処理によって隠蔽可能な実行時間が最大になるようにしている．このチューニングは，コンカレントカーネルなどの GPU に依存する機能を用いた手法である．PACC は GPU 以外のアクセラレータへも対応することを目的としているため，同様の手法は適用できない．そこで，データの分割数を指定するためのディレクティブを定義し，プログラムの開発者が分割数を調節できるようにしている．

OpenMPC 以外のディレクティブ仕様の 1 つに，XcalableMP[5] がある．XcalableMP は，クラスタなどの分散メモリ環境を対象としたディレクティブ仕様である．ノード内並列に関しては，Nomizu ら [6] がアクセラレータを用いた実装を提案している．また，小田嶋ら [7] は CPU および GPU を協調計算する機構を提案している．PACC の機構であるデータ分割およびパイプライン処理は，XcalableMP におけるノード内並列においても有用だと思われる．

現在の PACC は，対象問題をステンシル計算に絞っている．既存研究において，ステンシル計算を対象とするフレームワークは数多く提案されている．例えば，Christen ら [8] が提供するフレームワークを用いることで，ステンシル計算をマルチコアプロセッサおよびメニーコアプロセッサ向けに自動チューニングできる．また，Luts ら [9] の提案するフレームワークは，マルチ GPU 環境を対象とし，GPU 間通信に関する最適化を施す．Maruyama ら [10] の提案するフレームワークは，GPU を搭載する大規模なスーパーコンピュータを対象として最適化を施す．さらに，Devito ら [11] のフレームワークは，ステンシル計算の一つである偏微分計算に特化したものであり，クラスタ環境などにおいて動作する．これらのフレームワークは，計算機の性能を引き出すための最適化手法を適用する．一方，PACC の目的はデバイスメモリ不足を防止することであり，計算コードのチューニングは対象としていない．また，これらの既存手法は計算コードを記述するためのライブラリを提供しているため，一度に 1 つのフレームワークのみ適用可能である．それに対し，PACC はディレクティブベースであるため，PACC 以外のディレクティブ仕様をプログラムに埋め込める．

3. PACC の提案

OpenACC の主な利点は，高い性能可搬性である．したがって，PACC の設計においても性能可搬性を考慮し，以下の観点から設計する．

(1) 汎用性が高いこと．すなわち，より多くのプログラムに適用可能であること．
(2) より多くの性能チューニングを，ディレクティブの追加あるいはパラメータ調整によって実現できること．

（1）に関しては，OpenACC を適用可能なアルゴリズムに共通した，データ分割およびアクセラレータ上での計算コードの書き換えに必要な情報を定義する必要がある．

現在のディレクティブ仕様は，ステンシル計算を対象として設計している．ステンシル計算は科学計算において重要な計算であること，およびデータ分割が容易であることを考慮し，設計の対象とした．

（2）に関しては，OpenACC の提供する最適化・チューニング手段に加え，CPU・GPU 間のデータ転送時間を GPU 上の計算時間で隠蔽できる．また，分割の粒度をパラメータとして与えることで，隠蔽の効果を高める．

3.1 データ分割に必要な情報

データを分割するためには，分割の形状および袖領域を定義する必要がある．ステンシル計算においてこれらを決定するためには，ステンシル計算の適用領域を表現する配列に関する情報として，次の 2 点を与える必要がある．

(1) 分割可能な幅
(2) 各要素を計算する際の参照パターン

（1）は，分割の形状を決定するために用いる．まず，計算の対象領域を配列で表現する方法は主に 2 種類ある．1 つ目の方法は，n 次元の領域を n 次元配列で表す場合である．もう一方は，n 次元の領域を 1 次元配列で表す場合である．領域内の各要素に対する計算は独立しているため，前者の場合は配列の各次元に関して任意の幅で分割可能である．一方，後者の場合，k 次元方向の要素数を a_k とすると，分割可能な幅は $\sum_{k=1}^{n-1} a_k$ である．例えば，**図 3** は横幅 `cols`，高さ `rows` の 2 次元データに対するステンシル計算のプログラムである．このプログラムにおいて，対象領域は 1 次元配列を用いて表されている．また，対象領域の横方向に並ぶデータを配列の連続番地に並べている．したがって，対象領域は横方向にのみ分割可能であり，その際配列 `src` および `dst` は幅 `cols` で分割することになる．このような分割可能な幅を表す情報は OpenACC のディレクティブでは指定できず，コード解析によって獲得することも困難であるため，PACC のディレクティブとして定義する必要がある．

（2）は，袖領域を定義するために必要である．一般に，ス

```
1  #pragma acc parallel loop \
2      copyin(src[0:cols*rows]) \
3      copyout(dst[0:cols*rows])
4  for (int j = 1; j < rows - 2; j++) {
5    for (int i = 1; i < cols - 2; i++) {
6      dst[i + cols*j] = src[(i-1) + cols*j] +
7                        src[(i+1) + cols*j] +
8                        src[i + cols*(j+1)] +
9                        src[i + cols*(j-1)];
10     dst[i + cols*j] = dst[i + cols*j] & mask[j];
11   }
12 }
```

図 3 二次元ステンシル計算のプログラム

テンシル計算では対象領域の各要素を計算する際に，自身の周囲にある要素を参照する．この参照パターンはすべての要素に対する計算において同一である．したがって，参照パターンの情報をもとに，袖領域の大きさを決定できる．例えば，図 1 では配列 dst[i] の計算において src[i] およびその左右の要素を 1 つずつ参照する．同様に，配列 src を分割する際の袖領域は，左右に 1 つずつである．また，図 3 では配列 src を参照する際，左右に cols 個離れた要素を参照する．したがって，袖領域を src のチャンクの左右に cols 個ずつとる．このように，袖領域の大きさを指定するためには，配列の各次元に関して左右に最も離れた場所の参照地点をディレクティブで指定する必要がある．

3.2 パイプラインの性能チューニングに必要な情報

パイプライン処理の性能を決める重要な要素は，以下の 4 つである．

(1) パイプラインの数
(2) データの分割数（以下，チャンク数）
(3) パイプラインのステージ数
(4) ステージ間の負荷分散

これらのうち，ディレクティブを用いて指定可能とするのは，(1) および (2) である．パイプラインの数は，同時に処理可能なチャンク数を表す．パイプラインが多いほど，より多くのステージを同時に実行可能となるため，実行時間の短縮が期待できる．しかし，パイプラインが多いほど，実行順管理のオーバヘッドが高くなるため，プログラムごとに適切なパイプラインの数を指定する必要がある．

また，強スケールのプログラムにおいてはチャンクを多くすることで各ステージの稼働率を上げられるため，実行時間を短縮できる．チャンクが多いほど，チャンク 1 つあたりの実行時間が短いため，最初に実行するチャンクがパイプラインの 2 段目以降へ到達するまでの時間が短縮される．また，最後のタスクがパイプラインの 2 段目以降へ到達してから実行完了するまでの時間も短くなる．しかし，チャンクが多すぎると，データ転送のレイテンシ増大およびアクセラレータの実行性能低下が起こるため，適切なチャンク数がプログラムによって異なる．

(2) および (3) に関しては，ディレクティブではチューニングできない．パイプラインの段数は，CPU 上の処理，アクセラレータ上における計算，CPU からアクセラレータへのデータ転送，アクセラレータから CPU のデータ転送の最大 4 つである．負荷分散に関しては，ステージごとに使用可能な計算資源が異なるため，パラメータによる調節はできない．

3.3 ディレクティブの設計

3.1 節および 3.2 節で述べたデータ分割およびパイプラインチューニングのためのディレクティブを設計する．設計にあたり，可能な限り OpenACC と同様の記述にすることで，既存の OpenACC プログラムから PACC プログラムへの移植性を高める．

3.3.1 パイプライン化のためのディレクティブ

まず，プログラム内において，データを分割処理したいコード領域を指定する必要がある．また，そのコード領域内において分割可能な配列に関しても，情報を与える必要がある．さらに，パイプライン処理の性能チューニングに関する指示節も，データを分割処理したいコード領域に対して指定する必要がある．

データを分割処理したいコード領域を示すために，pipeline ディレクティブを定義する．その文法を以下に示す．

```
#pragma pacc pipeline 指示節 { 指示節 }
{structured block}
```

データを分割するための情報は，pipeline に対する指示節として指示節 targetin, targetout, targetinout を定義することで与える．指示節 targetin の文法は，以下の通りである．なお，targetout および targetinout に関しても，同様に記述する．

```
targetin(array_name(ls, un, rs)[start:len] \
{(ls, un, rs)[start:len]})
```

ここで，un は分割可能な間隔を表し，ls および rs はそれぞれ左右の袖領域の大きさを表す．start および len は，それぞれ配列の開始番地および長さである．配列を多次元である場合，(ls, un, rs)[start:len] を複数記述することで対応する．なお，この記述は OpenACC における data ディレクティブからの書き換えを想定したものである．pacc pipeline を acc data に置換し，targetin(ls, un, rs) を create に置換することで，OpenACC の記述となる．

3.3.2 パイプライン処理を性能チューニングするためのディレクティブ

性能チューニングのために，パイプライン化の有無およ

```
1  #pragma pacc pipeline \
2    targetin(src(cols,cols,cols)[0:cols*rows]) \
3    targetout(dst(0,cols,0)[cols:cols*rows-2*cols]) \
4      async
5  {
6    // CPUからアクセラレータへのデータ転送
7  #pragma pacc update device(src[0:cols*rows])
8    // アクセラレータ上の計算
9  #pragma pacc parallel loop pipe(1)
10   for (j = 1..rows-2){
11 #pragma pacc loop independent
12     for (i = 1..cols-2) {
13       dst[j*cols+i] = src[j*cols+(i-1)] +
14           src[j*cols+(i+1)] +
15           src[(j-1)*cols+i+1] +
16           src[(j+1)*cols+i+1];
17     }
18   }
19   // CPUからアクセラレータへのデータ転送
20 #pragma pacc update host(dst[cols:col*rows-2*col])
21 }
```

図 4 データ分割可能なプログラムの例

びデータの分割数を指定可能とする．まず，パイプラインの有無を示す指示節として，async 節を定義する．p ディレクティブに対して async 節を付加することで，各ステージの処理をパイプライン化する．

次に，データの分割数を指定するための指示節として，num_tasks(num) 節を定義する．これにより，一度に実行する配列の要素数を ⌈ len / un / num ⌉ × un に固定する．なお，len / un はデータ分割後のチャンク数を表す．

パイプライン化の有無を示す指示節 async 節を定義する．pipeline ディレクティブに対して async 節を付加することで，各ステージの処理をパイプライン化する．

3.3.3 パイプラインのステージを定義するためのディレクティブ

3.2 節で述べたように，パイプラインのステージは 4 つある．これらのステージを定義するためのディレクティブはすべて OpenACC の仕様に存在する．そこで，OpenACC から PACC への書き換えを容易にするために，PACC においても同様の記述法でディレクティブを定義する．例として，図 1 のプログラムに PACC ディレクティブを適用したものを，**図 4** に示す．コード内のディレクティブにおいて，update device，parallel，および update host は，OpenACC のディレクティブと同様の意味を持つ．このように記述することで，CPU からアクセラレータへのデータ転送（update device），アクセラレータ上での計算（parallel）およびアクセラレータから CPU へのデータ転送（update host）をデータ分割し，パイプライン処理できる．また，pipeline ディレクティブの内側に CPU でのコードを記述することで，CPU 上の計算をパイプライン処理のステージに含められる．

図 5 PACC トランスレータの変換手順

3.3.4 計算を分割するためのディレクティブ

ステンシル計算の対象領域を分割する際，計算コードにおいて参照する領域の範囲を修正する必要がある．例えば，図 1 から図 2 への書き換えでは，領域を num_tasks 個に分割することで各領域の要素数が task_size 個になるため，#pragma acc parallel loop の内側にあるループ文において処理する要素数を task_size 個に書き換えている．このように計算を分割するには，ループ文を対象領域の次元と関連付ける必要がある．そこで，関連付けのための指示節として，pipe(dim) 節を定義する．pipe 節を#pragma pacc parallel loop ディレクティブまたは#pragma pacc loop ディレクティブに対して付加することにより，そのディレクティブ直下にあるループ文のインデックスを，領域の dim 次元に対応づける．

3.4 トランスレータの設計

実装には，ROSE Compiler Infrastructure[12] を用いた．**図 5** にトランスレータによる変換手順を示す．変換における手順のうち，抽象構文木の生成は ROSE が持つ機能である．残りのディレクティブ解析およびループ構造や参照先の書き換えは自作した．実装にあたり，このトランスレータを OpenACC が対応する GPU 以外のアクセラレータにも容易に対応できるよう，出力には可能な限り CUDA のライブラリを含めず，OpenACC ディレクティブを用いるようにした．現在の出力ファイルで CUDA を使用しているのは，デバイスメモリの容量を調べる処理のみである．

なお，現在の実装では CPU の処理をパイプラインのステージに含められない．多次元配列にも未対応である．

4. 評価実験

本章では，PACC トランスレータが生成したコードを評価する．評価に用いたアプリケーションは，CAPS 社のサンプルプログラム Black Scholes（BS）および Sobel Filter（SF）である [13].

まず，実行可能なデータサイズおよび実行性能に関して評価する．比較対象を OpenMP 版および OpenACC 版とすることで，トランスレータとしての性能を評価する．次に，処理のパイプライン化による性能向上について評価する．

なお，[13] にて公開しているプログラムは OpenACC 版のみであるため，PACC 版および OpenMP 版は [13] のディレクティブを書き直すことで実装した．実装においては，なるべく OpenACC 版と同様のコード領域を並列化または

```
1  #pragma pacc pipeline \
2    targetin(original_image(cols, cols, cols) \
3      [0:rows*cols]) \
4    targetout(edge_image(cols, cols, cols) \
5      [0:rows*cols]) async
6  {
7  #pragma pacc update \
8    device(original_image[0:rows*cols])
9  #pragma pacc kernels \
10   present(original_image[0:rows*cols], \
11   edge_image[0:rows*cols])
12   {
13 #pragma pacc loop independent pipe(1)
14     for (unsigned long long iy = 1; iy < rows-1;
15         iy++) {
16       #pragma pacc loop independent
17       for (unsigned long long ix = 1; ix < cols-1;
             ix++) {
18         int sum_x=0, sum_y=0, sum=0;
19         /*-------X GRADIENT APPROXIMATION------*/
20         ( 中略 )
21         /*-------Y GRADIENT APPROXIMATION------*/
22         ( 中略 )
23         /*---GRADIENT MAGNITUDE APPROXIMATION----*/
24         sum = abs(sum_x) + abs(sum_y);
25         /* make edges black and background white */
26         edge_image[ ix + iy * cols] = 255 -
27             (unsigned char)(sum);
28       }
29     }
30   }//end of OpenACC kernels region
31 #pragma pacc update host(edge_image[0:rows*cols])
32 }
33 for (unsigned long long iy = 0; iy < rows; iy++) {
34   edge_image[iy * cols] = 255;
35   edge_image[((iy+1) * cols) -1] = 255;
36 }
37 for (unsigned long long ix = 1; ix < cols-1; ix++)
       {
38   edge_image[ix] = 255;
39   edge_image[ix + ((rows-1) * cols)] = 255;
40 }
```

図 6 SF の PACC 版実装

GPU にオフロードするようにディレクティブを挿入した．ただし，SF の PACC 版実装に関してのみ，**図 6** に示すようにプログラムを書き換えている．変更点は，外枠の計算（図 6 の 33 および 37 行目）をオフロードの領域から外していること，および外枠の計算を内側の計算（図 6 の 9 行目）よりも後ろに移動していることである．前者の変更は，各要素を計算する際のステンシルを同一にするためである．後者の変更は，CPU 上で書き換えた配列の値を GPU からのデータ転送で上書きしないためである．

すべての実装において，コンパイラは PGI Compiler 13.7[14] を用いた．PGI Compiler による GPU 用計算コードの生成には，CUDA 5.0 を使用した．実験環境としては，Core i7 3930K，5 GB のデバイスメモリを持つ Tesla K20c，および 32 GB のホストメモリを搭載する Windows 7 Professional 64bit の PC を用いた．GPU のディスプレイドライバは 320.49 である．

4.1 実行可能なデータサイズおよびその実行時間の評価

各実装において実行可能な最大のデータサイズを比較し，PACC によって扱えるデータがどの程度増加したか調査する．BS においては，配列長を 5×10M 個ずつ増やして実験した．また，SF においては，画像の縦横幅を 5×1K 画素ずつ増やして実験した．**図 7** に実行可能なデータサイズおよびその実行時間を示す．

BS の結果において，OpenACC 版は配列長 250M 個以下でのみ実行に成功している．BS においては float 型の配列を 5 つ参照するため，この配列長は $250\mathrm{M} \times 5 \times 4 \approx 4.9\mathrm{GB}$ に相当する．同様に，PACC および OpenMP において実行可能なデータサイズは，それぞれ 26.4 GB および 28.3 GB である．したがって，OpenACC は GPU のメモリ容量の範囲内でのみ実行に成功しているのに対し，PACC はそれよりも大きなデータでも実行に成功している．しかし，OpenMP よりも扱えるデータ量が小さいことから，PACC はホストメモリを十分に使い切れていないことがわかる．この原因は，CPU・GPU 間のデータ転送のために，主記憶上にバッファを用意しているためである．SF に関しても，同様の傾向があった．OpenACC 版が $(50 \times 1\mathrm{K})^2 \times 2 \times 1 \approx 4.9\mathrm{GB}$ であり，PACC 版および OpenMP 版はそれぞれ 25.8 GB および 28.1 GB である．

実行時間に関しては，BS では要素数 150M 個以上の場合に OpenMP よりも最大 2.9 倍高速である．一方，要素数がデバイスメモリに乗り切る範囲においては，OpenACC の方が高速である．SF の方も同様に，SF の方は，要素数 $35 \times 10K$ 個以上で OpenMP より最大 2.9 倍高速である．

どちらのアプリにおいても，デバイスメモリに乗り切る範囲においては，OpenACC が最速である．PACC が OpenACC よりも低速である原因は，主に 2 つある．まず，デバイスメモリの確保に要する時間が挙げられる．PACC はデータがデバイスメモリよりも大きいことを想定しているため，現在の PACC トランスレータが出力するコードは，常にデバイスメモリのほぼ全体を計算用に確保している．したがって，データ量がデバイスメモリ容量よりも小さい場合，メモリ確保にかかる時間が OpenACC と比べて長くなる．次に，ホストメモリ上でのデータコピーである．OpenACC 1.0 の仕様上，ホストメモリ上の任意のアドレスからデバイスメモリ上の任意のアドレスにデータをコピーできない．したがって，OpenACC ではホストメモリ上でのデータコピーを追加している．これらが低速な原因である．なお，今回用いた実験環境においては，デバイスメモ

リ全体を確保するために 0.5 秒程度要していた．また，ホストメモリ上でのデータコピーにはデータサイズ（GB）の 1/12 程度の秒数を要していた．

低速化の原因のうち後者に関しては，少なくとも今回の実験環境ではパイプライン処理によって隠蔽できる．ホストメモリ上におけるメモリコピーの総バイト数は，CPU・GPU 間でのデータ転送の総バイト数に等しい．また，バンド幅に関しても，ホストメモリおよび CPU・GPU 間のデータ転送はどちらも 12 GB/s 程度である．したがって，データコピー時間は CPU・GPU 間のデータ転送時間とオーバラップすることにより，ほぼ隠蔽できるはずである．

4.2 ストリーム数に関する性能比較

本節では各ストリーム数における実行時間を比較することで，性能を評価する．**図 8** に，BS および SF の実験結果を示す．どちらのアプリケーションにおいても，ストリーム数を最大 8，分割数を 10 段階の掛け合わせで実験をした．なお，データ量はどちらも約 20 GB である．

どちらのアプリにおいても，ストリーム数はパイプラインのステージ数と同じ 3 本が適切だといえる．まず，SF 版においては，ストリーム数 3 のときが最速である．この数は，SF におけるストリーム処理のステージ数（CPU・GPU 間転送および GPU 上の計算）に等しい．ストリーム数 4 以上では，オーバラップによって隠蔽される時間がストリーム数 3 の場合と同じであり，ストリーム数の増加に伴う実行順管理のオーバヘッドが増大するため，低速となる．一方，ストリーム数が 3 未満の場合，十分な並列度を確保できないことが原因となり，低速である．

BS 版においては，ストリーム数 2 が最も高速となるケースがある．BS におけるステージ数は SF と同様に 3 であるが，カーネル実行時間はデータ転送時間の 10 分の 1 程度である．したがって，実行時間の大部分は CPU・CPU 間のデータ転送が占めるため，ストリーム数 2 でも十分な並列度となる．そのため，ストリーム数が増えることで隠蔽した実行時間よりもオーバヘッドのほうが大きくなることがある．ただし，ストリーム数 3 の方が高速であるケースの方が多いため，ストリーム数 3 の方が適切である．

4.3 タスク粒度のチューニングによる性能評価

本節では分割数ごとの実行時間を比較し，性能を評価する．**図 9** に，BS および SF の実験結果を示す．

実験の結果から，データの分割数はデータサイズおよびアプリケーションに依存することがわかった．まず，BS においては，データサイズが約 5 GB のとき，パイプライン版は分割数 4,500 および 5,000 において最速であり，実行時間は 0.74 秒であった．また，非パイプライン版は分割数 4,500 において最速であり，実行時間は 1.75 であった．したがって，パイプライン版は非パイプライン版と比較し，

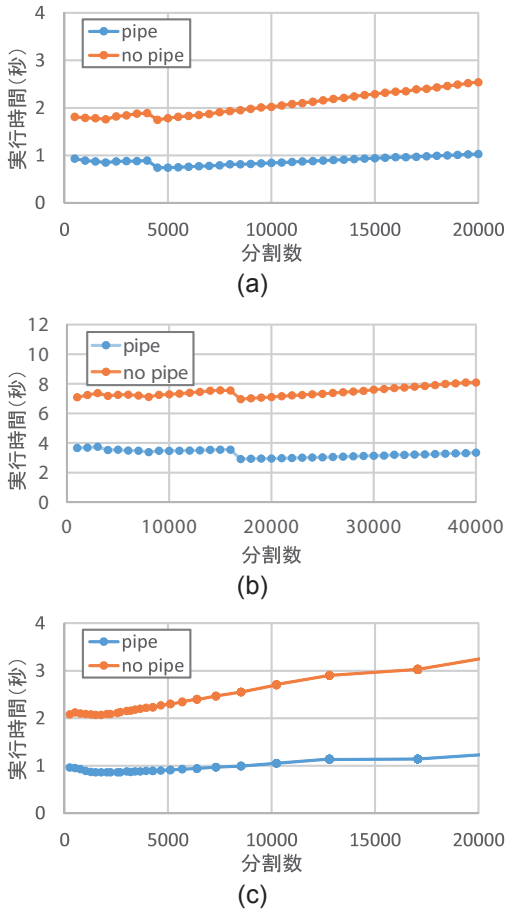

図 9 データ分割数および実行時間の関係．(a) BS（5GB）の結果．(b) BS（20GB）の結果．(c) SF（5GB）の結果．

2.4 倍高速化できた．データサイズが約 20 GB のときは，パイプライン版・非パイプライン版共に分割数 17,000 において最速であり，実行時間はパイプライン版が 2.91 秒，非パイプライン版が 6.96 秒でああった．

また，データサイズが約 20 GB のとき，分割数 17,000 が最も高速であり，実行時間は 2.91 秒であった．データサイズが 4 倍であるにもかかわらず，分割数が 4 倍未満であることから，分割によるオーバヘッドが大きいことがわかる．したがって，最適な分割数はデータサイズに依存するといえる．

次に，SF における結果では，分割数 1506～2695 において最も高速であり，実行時間は 0.86 秒であった．これを BS の 5GB と比較することで，対象問題によっても結果が変わることがわかる．

5. まとめ

本稿では，アクセラレータのメモリ容量を超える大規模データを GPU 上で計算可能とするためのディレクティブを提案し，その実装を示した．このディレクティブ仕様 PACC は，データを分割処理することにより，デバイスメモリ不足に対処している．また，分割後のデータに対してパイプライン処理の機構を適用することで，CPU・GPU 間

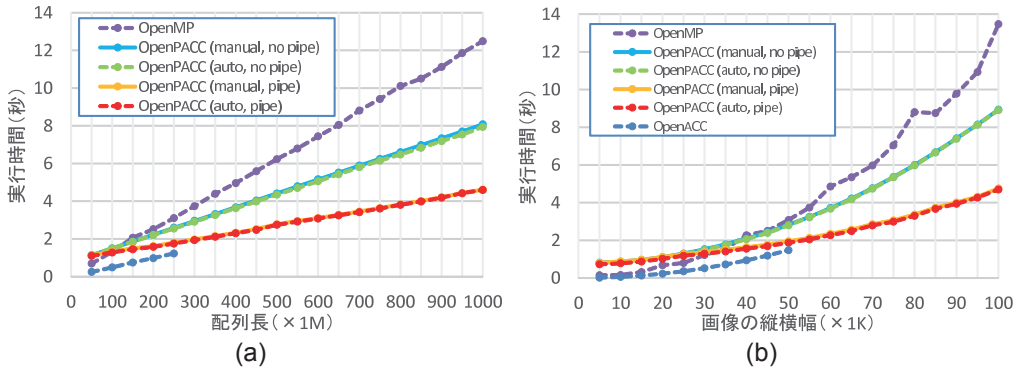

図 7 各実装における実行時間の比較．（a）BS の結果．（b）SF の結果．

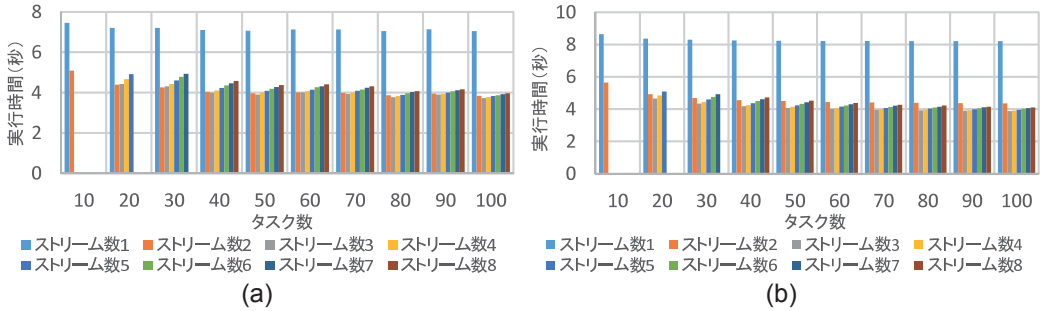

図 8 ストリーム数ごとの実行時間の比較．データサイズはどちらのアプリも 20GB である．（a）BS の結果．（b）SF の結果．

のデータ転送時間を隠蔽する．PACC のディレクティブ仕様は，OpenACC に対して，データ分割を実現するためのディレクティブを追加したものであり，現在一部のステンシル計算のアプリケーションに対して適用可能である．実験においては，デバイスメモリ不足を解消できたことを確認するとともに，処理のパイプライン化によって単純なデータ分割よりも最大 2.6 倍の高速化を実現した．

今後の課題は，PACC の適用可能な問題を増やすことである．

謝辞 本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」，科研費 23300007，および 23700057 の補助による．

参考文献

[1] openacc-standard.org: OpenACC 1.0 (2012). http://www.openacc-standard.org/.
[2] NVIDIA Corporation: CUDA C Programming Guide Version 5.5 (2013). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[3] Lee, S. and Eigenmann, R.: OpenMPC: Extended OpenMP for Efficient Programming and Tuning on GPUs, *International Journal of Computational Science and Engineering* (2012).
[4] Sabne, A., Sakdhnagool, P. and Eigenmann, R.: Scaling large-data computations on multi-GPU accelerators, *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ACM, pp. 443–454 (2013).
[5] xcalablemp.org: XcalableMP. http://www.xcalablemp.org/.
[6] Nomizu, T., Takahashi, D., Lee, J., Boku, T. and Sato, M.: Implementation of XcalableMP Device Acceleration Extention with OpenCL, *Proc. 26th IEEE Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'12)*, pp. 2394–2403 (2012).
[7] 小田嶋哲哉, 朴泰祐, 佐藤三久, 塙敏博, 児玉祐悦, Namyst, R., Thibault, S., Aumage, O.: 並列言語 XMP-dev における GPU/CPU 動的負荷分散機能, 情報処理学会研究報告, 2013-HPC-140 (2013). 7 pages.
[8] Christen, M., Schenk, O. and Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures, *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, IEEE, pp. 676–687 (2011).
[9] Lutz, T., Fensch, C. and Cole, M.: PARTANS: An autotuning framework for stencil computation on multi-GPU systems, *ACM Trans. Architecture and Code Optimization*, Vol. 9, No. 4, p. 59 (2013).
[10] Maruyama, N., Nomura, T., Sato, K. and Matsuoka, S.: Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers, *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'11)* (2011). 12 pages.
[11] DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J. and Hanrahan, P.: Liszt: a domain specific language for building portable mesh-based PDE solvers, *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'11)* (2011). 12 pages.
[12] rosecompiler.org: ROSE compiler infrastructure. http://rosecompiler.org/.
[13] caps entreprise.com: OpenACC examples. http://www.caps-entreprise.com/resources-and-support/openacc-examples/.
[14] The Portland Group: PGI Compiler. http://pgroup.com.