Vol.2014-HPC-145 No.45
2014/7/30

OpenACCディレクティブ拡張による データレイアウト最適化

星野 哲也[1]　丸山 直也[3,1,2]　松岡 聡[1,2]

概要：近年増加傾向にある GPU 等のアクセラレータを搭載した計算環境への既存プログラムの移植方法として，CUDA・OpenCL に代表されるローレベルなプログラミングモデルを用いる方法に対し，ディレクティブベースの OpenACC のようなハイレベルなプログラミングモデルを用いる方法が注目されている．このようなディレクティブベースのプログラミングモデルの利点として，元のプログラムを維持したまま移植を行えるために，デバイス間の機能的な可搬性が高いことがあげられる．しかし現状の OpenACC などの High-level なプログラミングモデルは，スカラプロセッサとメニーコアアクセラレータの得意とするデータレイアウトの相違に対応することが出来ず，異なる性質を持ったデバイス間の性能可搬性に問題がある．そこで本研究では，データレイアウトを抽象化し，異なるデバイス間での性能可搬性を向上させるための OpenACC の拡張ディレクティブを試作し，姫野ベンチマークのデータレイアウトをトランスレーターにより変更し，マルチコア CPU，Intex Xeon Phi，K20X GPU のそれぞれで評価を行った．その結果，オリジナルと同一のデータレイアウトと比較して，Intel Xeon Phi では 27%，K20X GPU では 24%の性能向上が得られることを確認した．

1. はじめに

TSUBAME2.5 に代表されるように、CPU に加え GPU 等のアクセラレータを大量に搭載したヘテロジニアスな計算環境が台頭してきている．アクセラレータの演算性能に対する価格・消費電力の低さが買われ，このような計算環境は今後も増えていくものと考えられている．これらのユーザーは多大なプログラミングコストを払い，アプリケーションを複雑化する計算環境に合わせて移植しなければならず，既存のアプリケーションの移植が問題になっている．アプリケーションのアクセラレータ環境への移植手法として現在主流の方法として，ローレベルなプログラミングモデルである CUDA，OpenCL を用いる方法があるが，これらローレベルなプログラミングモデルを使用する場合，ユーザーがアクセラレータのアーキテクチャを意識した記述をする必要があり，プログラムが煩雑になりがちである．

この解決策として，現在のマルチコア CPU 環境において一般的になっている OpenMP と同様のディレクティブベースプログラミングモデルである，OpenACC[6] が注目されている．OpenACC ではソースコードを保持したまま，CPU 向けに作られた既存のアプリケーションに数行の指示文を挿入することにより，アクセラレータ上での実行を可能とする．挿入された指示文を無視すれば元のアプリケーションと同様に CPU 上で実行可能であるため，デバイス間の機能的な可搬性が高いことがディレクティブベースの利点である．しかし，CPU と GPU はそれぞれ最適なデータレイアウトに違いがある等の性能上の異なる特質を持っており，著者らの以前の研究 ([3]) において CUDA・OpenACC を用いて実アプリケーションを移植した際には，Array of Structure (AoS) のデータレイアウトを Structure of Array (SoA) に書き換える必要があった．この問題は，ホスト-アクセラレータどちらで実行する場合にも同一のプログラムを用いる OpenACC のようなプログラミングモデルにおいては，性能可搬性の低下の原因となるために顕著となるが，実行デバイスごとに大きく性能が変化するデータレイアウト等を抽象化することにより解決可能であると考えられる．データレイアウトの抽象化を実現することができれば，ローレベルなプログラミングモデルでは行えないデータレイアウトの自動最適化が可能となり，ハイレベルなプログラミングモデルの利点である異なるデバイス間での可搬性を，機能・性能の両面において達成できるもの

[1] 東京工業大学
Tokyo Institute of Technology
[2] 科学技術振興機構 CREST
JST CREST
[3] 理化学研究所
RIKEN AICS

ⓒ 2014 Information Processing Society of Japan

C 言語
```
  #pragma acc directive-name [clause [[,] clause]…] new-line
     { structured block }
```

Fortran
```
 !$acc directive-name [clause [[,] clause]…]
      structured block
 !$acc end directive-name
```

図 1 OpenACC ディレクティブ

と考えられる．そこで本研究の目的は，データレイアウトを抽象化を検討し，自動最適化に向けた基盤を構築することである．

2. 背景

2.1 OpenACC と CUDA の違い

プログラムの GPU 環境への移植手法として CUDA や OpenCL を使うことが一般的であったが，現在新しいプログラミングモデルとして OpenACC が注目されている．OpenACC は, NVIDIA, Cray, PGI などの複数のベンダーにより規定された，アクセラレータ向けの並列プログラミング規格である．C/C++や Fortran と言った科学技術アプリケーションで多く用いられるプログラミング言語に対して，OpenMP の様にディレクティブを挿入することで，GPU 等のアクセラレータ環境で実行できるプログラムを生成することができる．CUDA や OpenCL を用いる場合，GPU のアーキテクチャを意識した低レベルな記述をする必要があることが，GPU 環境へのプログラム移植の阻害要因になっていたが，ソースコードの直接的な変更の必要がない OpenACC の登場により，GPU 環境への移植の簡素化に期待が高まっている．OpenACC 以前にも，hmpp[2], PGI アクセラレータコンパイラ [9]，OpenMP の CUDA 拡張 OpenMPC[5] などが存在したが，仕様が統一化されたことにより，アクセラレータ，コンパイラなどに依存しないポータビリティが期待されている．例えば OpenACC は，**図 1** のように，並列実行領域に対して，最小で 1 つのディレクティブを挿入することで，アクセラレータ環境での実行プログラムを生成する．

この辺に data ディレクティブとか kernels ディレクティブについて説明

3. 関連研究

Sung らの研究 [8] では，GPU 向けのデータレイアウトとして，Array-of-Structure-of-Tiled-Array(ASTA) を提案，その有効性を評価し，CUDA・OpenCL のような Low-level なアプローチにおいて，Array of Structures 型のデータレイアウトから ASTA への自動変換を実現した．我々の研究ではさらに High-level のプログラミングモデルにおけるデータレイアウトの抽象化を目指している点で差異がある．

Shuai らの研究 [1] では，CUDA・OpenCL で書かれたプログラムを対象とし，データレイアウトを最適化するための指示文ベースの API である Dymaxion++を提供している．Dymaxion++では主に 2 つの指示文，*Reshape* と *Place* を提供している．*Reshape* 指示文では，3 つのデータレイアウトの変更方式，*trasnpose*，*diagonal*，*indirect* を指定することができ，この変換を PCI-E の通信に隠蔽して行うことが出来る．また，*Place* 指示文を用いることで，GPU の持つ on-chip メモリやテクスチャメモリも利用可能である．本研究では High-level なプログラミングモデルを対象とし，異なるデバイス間での性能可搬性を高めることを目的としており，その点において差異がある．

4. データレイアウト抽象化の必要性

4.1 データレイアウトが性能に与える影響

データレイアウトが性能に与える影響を評価するために，いくつかのベンチマーク・実アプリケーションを用いて評価を行った結果を示す．それぞれの性能計測に用いた計算環境を**表 2** に，コンパイル時に指定したオプションを**表 1** に示す．また，マルチコア CPU における実験では TSUBAME の CPU2 ソケット分 (全 12 コア) を用いており，Intel Xeon Phi における実験では 240 スレッドで実行している．Xeon Phi 上での実行時には環境変数として，KMP_AFFINITY=compact を指定している．この指定により，OpenMP のスレッドを順に割り付ける際に，直前のスレッドになるべく近いハードウェアスレッドに割り付けることが出来る．

4.1.1 ストリームベンチマーク

データレイアウトの異なりが与える影響を評価するために，異なるデータレイアウトを用いて Intel CPU, Intel Xeon Phi, NVIDIA Kepler の各デバイス上でストリームベンチマークを実行した結果を示す．ストリームベンチマークに用いたプログラムは，[4] よりダウンロードしたプログラムを本実験のために変更したものである．適用した変更は，OpenACC を用いるための指示文の挿入と，データレイアウトの変更とそれに伴う初期化・アクセス順序の変更である．オリジナルのストリームベンチマークでは，データは 1 次元の配列で宣言されており，先頭の要素から逐次にアクセスされるが，変更後は 2 次元配列的にデータの格納順序を変更し (**図 5**)，2 重ループによるアクセスを行う．データはメモリ上に row major 形式で格納されており，図 5 の左側の場合，オリジナルにおける奇数番目の要素のみを持つ配列と，偶数番目の要素のみを持つ 2 つの配列を持つ Structure of Arrays (SoA) と見なすことが出来，右側の場合 2 要素の構造体の配列，すなわち Array of Structures (AoS) と見なすことが出来る．

このようにデータレイアウトの変更を行った上で，CPU, Xeon Phi, Kepler 上でストリームベンチマークを実行した

ⓒ 2014 Information Processing Society of Japan

表 1　実験環境

CPU	Intel Xeon X5670 6cores 2.93 GHz 2 sockets 54 GB Memory
GPU	NVIDIA Kepler K20X 2688 CUDA cores 6GB Memory
MIC	Intel Xeon Phi 7120X 61 cores 16GB Memory

表 2　コンパイラ，オプション

CPU	icc -O3 -openmp
GPU	pgcc -O3 -ta=nvidia,cc35,kepler
MIC	icc -O3 -mmic -openmp -opt-prefetch-distance=4,1 -opt-streaming-stores always -opt-streaming-cache-evict=0

結果がそれぞれ**図 2**, **図 3**, **図 4** である．ストリームベンチマークで計測しているのは，$Copy : C = A$，$Scale : B = scalar \times C$，$Add : C = A + B$，$Triad : A = B + scalar \times C$ であり，A，B，C はそれぞれ 10M の double 要素を持つベクトルである．グラフの横軸が示しているのは 2 次元配列の短辺の長さ，つまり SoA であれば Array の本数であり，AoS であれば Structure に含まれる要素数を示している．GPU においてはデータレイアウトによる差が顕著に現れており，特にストライド幅が大きくなる AoS のアクセスにおいては性能劣化が著しく，ストライド幅 16 以上では 90%以上性能が低下しいる．MIC においては SoA 型のアクセスにおいては性能劣化は見られないが，AoS 型の配列へのアクセスで性能劣化が見られる．マルチコア CPU の実行においては，AoS 型の配列へのアクセスの方が全体的に性能が高い．

4.1.2　姫野ベンチマーク

実際のアプリケーションに近い形でデータレイアウトの影響を評価するために，理化学研究所より提供されている姫野ベンチマーク [11] を用いて評価を行う．姫野ベンチマークは，非圧縮流体解析プログラムの性能評価のために考案されたもので，ポアソン方程式解法をヤコビ反復法で解く際の主要ループの処理速度を計測するものである．姫野ベンチマークは複数のバージョンが提供されているが，本稿では C の static allocation バージョンをベースとし，本実験のために変更を加えたものである．適用した変更は，マルチコアで並列実行するための OpenMP 指示文の追加，GPU で実行するための OpenACC 指示文の追加，4 次元配列として宣言されている係数配列のレイアウト変更である．また，オリジナルのプログラムでは次のタイムステップで利用する圧力 P を更新するためにコピー処理を行うが，本稿ではダブルバッファリングを用いることにより，ポインターの入れ替えで済ませる様に変更している．さらに，MIC での最適化に際しては，小林らの研究 [10] において述べられている最適化の一部を適用している．

姫野ベンチマークにおける 4 次元の係数配列 a, b, c は，

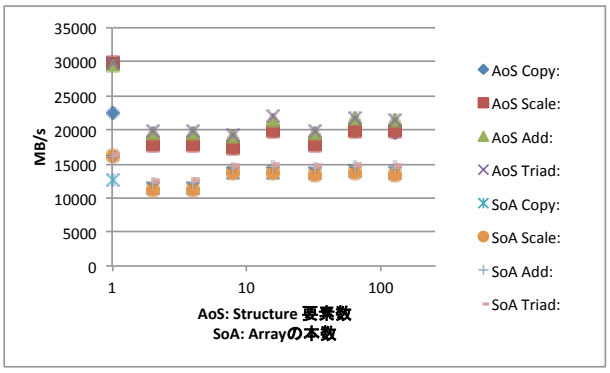

図 2　Intel Xeon CPU 12 コアによるストリームベンチマーク

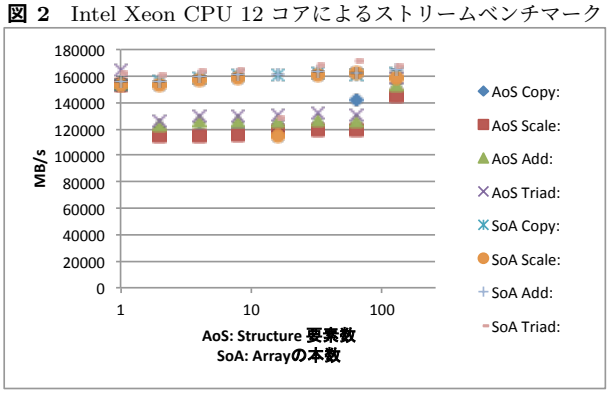

図 3　Intel Xeon Phi 240 スレッドによるストリームベンチマーク

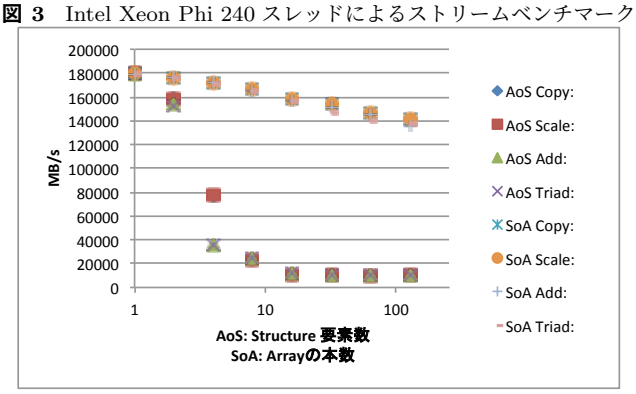

図 4　NVIDIA K20X GPU 上でのストリームベンチマーク

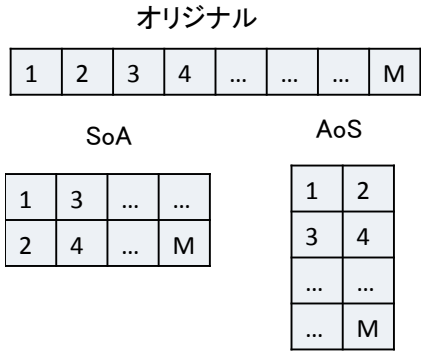

図 5　AoS，SoA それぞれへのレイアウトの変更．データは row major 形式で格納されている．

オリジナルのプログラムでは**図 6** における SOAOS のように定義されている．各 4 次元配列は，最内次元に 3 または 4 要素の構造体を持つ AoS 型のレイアウトと見なすことが出来る．さらにこの 3 つの配列をひとつの構造体とみなし，

ⓒ 2014 Information Processing Society of Japan

```
#if SOAOS
static float  a[MIMAX][MJMAX][MKMAX][4],
              b[MIMAX][MJMAX][MKMAX][3],
              c[MIMAX][MJMAX][MKMAX][3];
#endif
#if SOSOA
static float  a[4][MIMAX][MJMAX][MKMAX],
              b[3][MIMAX][MJMAX][MKMAX],
              c[3][MIMAX][MJMAX][MKMAX];
#endif
#if AOS
static float  abc[MIMAX][MJMAX][MKMAX][10];
#endif
#if SOA
static float  abc[10][MIMAX][MJMAX][MKMAX];
#endif
```

図 6 姫野ベンチマークの係数配列の変更

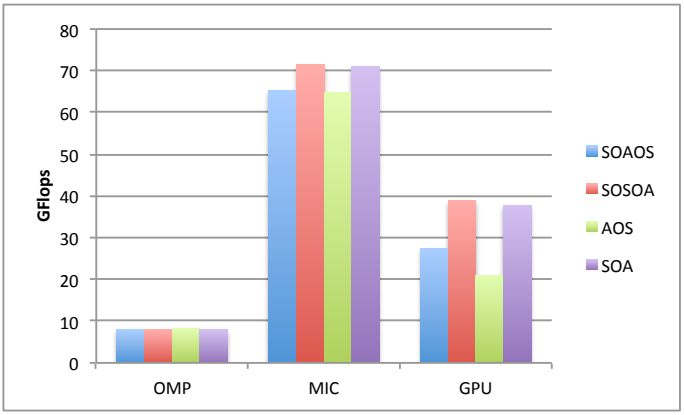

図 7 姫野ベンチマーク データレイアウトの変更による影響

AoS 型の配列の構造体，Structure of Array of Structure (SOAOS) 型と呼ぶこととする．このオリジナルのデータレイアウトに新たに 3 つのデータレイアウトを加え，比較評価を行う．各係数配列を AoS 型から SoA 型に変更したものを SOSOA 型，またこれら 3 つの配列をひとまとめにし，一つの AoS 型の配列，SoA 型の配列に変更したものを追加した．これに伴い，配列の初期化部分と主要ループ処理内における該当配列へのアクセスを変更した．

図 7 に実験結果を示す．OpenMP 版では 12 スレッド，MIC 版では 240 スレッドを利用している．GPU 版では PGI コンパイラにより自動的に選択されたスレッド数を用いており，各スレッドブロックは 64×4 のスレッドからなる．OpenMP 版ではレイアウトの変更により大きな違いは起こらなかったが，MIC，GPU 版においてはオリジナルのデータレイアウトと比較してそれぞれ 10%，40%の性能向上が見られる．

4.1.3 流体アプリケーション UPACS

姫野ベンチマークにおいて用いられているデータレイアウトは比較的シンプルであるが，実際のアプリケーションではさらに複雑なデータ構造を用いる場合が多い．UPACS は独立行政法人宇宙航空研究開発機構により研究開発されている，航空宇宙分野において要求される様々な流体現象の解析に用いることを目的とした流体アプリケーションであるが，多くの流体解析ソルバを内包するためにコードは非常に大規模であり，プログラム全体で更新すべきデータ構造を共有している．故にデータレイアウトも非常に複雑であり，最適なデータレイアウトを求めることは困難でありかつ，書き換えも容易ではない．

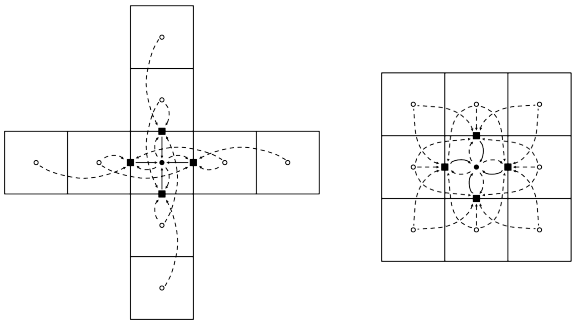

図 8 対流項 (左)，粘性項 (右) のステンシル計算

UPACS の計算フェーズの中でも主要である 2 つのフェーズ，対流項 (Convection) と粘性項 (Viscosity) においてデータレイアウトの変更を行う．**図 8** は対流・粘性項それぞれにおけるステンシル計算を図示したものである．各フェーズはセル中心に定義された物理量からセル表面に定義された値を更新し，更新されたセル表面の値から中心セルの値を更新する．このセル表面に定義されたデータ構造が**図 9** であり，この `cellFaceType` という構造体が 3 次元の配列として定義されている．この 2 つのフェーズはそれぞれ，UPACS の全体の実行時間のうち 25.0%，37.7%を占める計算フェーズであるが，この 2 つのフェーズについてデータレイアウトの変更を適用した．オリジナルの UPACS においてセル表面に定義される構造体の配列図 9 を**図 10**, **図 11** のような AoS 型, SoA 型に変換し，CPU・GPU それぞれで実行したものが**図 12**, **図 13** である．姫野ベンチマークとの相違点であるが，姫野ベンチマークの主要ループでは各 AoS 型の係数行列の構造体の全要素を計算に用いるのと比較して，UPACS のそれぞれのカーネルは `cellFaceType` 中の全ての要素を計算に用いる訳ではない．これらの不要要素の転送がメモリ帯域幅を圧迫しているために，CPU・GPU 双方において，オリジナルのデータレイアウトを用いて実行した場合に最も性能が低いことが分かる．図 12, 図 13 のデータレイアウトではカーネル実行時に全ての要素が使われるが，ストリームベンチマーク・姫野ベンチマークと同様に，GPU では SoA 型のデータレイアウトが最適であった．

この例からわかる通り，実際のアプリケーションにおいては，コードの可読性を重視していたり，物理現象から直感的にわかりやすいデータレイアウトになっているケースもあるため，データレイアウトの変更がより重要になる．

ⓒ 2014 Information Processing Society of Japan

```
type cellFaceType
    real(8)                :: area, nt
    real(8), dimension(3) ::  nv
    real(8), dimension(5) ::  q_r, q_l, flux
    real(8)                ::  shockFix
end type

type(cellFaceType), dimension(:,:,:), pointer :: cface
allocate(cface(-1:in+1,-1:jn+1,-1:kn+1))
```

図 9 オリジナルのデータレイアウト

```
real(8),dimension(-1:in+1,-1:jn+1,-1:kn+1) :: area,nt
real(8),dimension(3,-1:in+1,-1:jn+1,-1:kn+1) :: nv
real(8),dimension(5,-1:in+1,-1:jn+1,-1:kn+1) :: q_r,q_l,flux
real(8),dimension(-1:in+1,-1:jn+1,-1:kn+1) :: shockFix
```

図 10 Array of Structures 型のデータレイアウト

```
real(8),dimension(-1:in+1,-1:jn+1,-1:kn+1) :: area,nt
real(8),dimension(-1:in+1,-1:jn+1,-1:kn+1,3) :: nv
real(8),dimension(-1:in+1,-1:jn+1,-1:kn+1,5) :: q_r,q_l,flux
real(8),dimension(-1:in+1,-1:jn+1,-1:kn+1) :: shockFix
```

図 11 Structure of Arrays 型のデータレイアウト

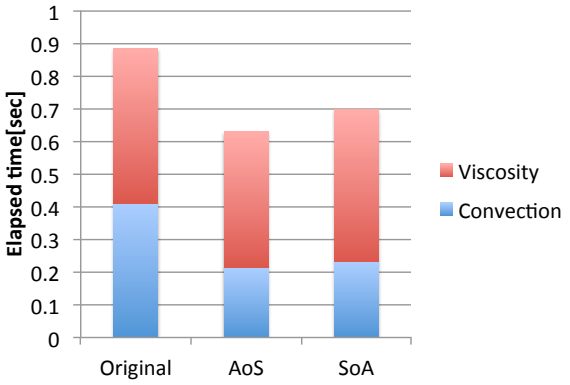

図 12 UPACS の CPU 実行におけるデータレイアウトの影響

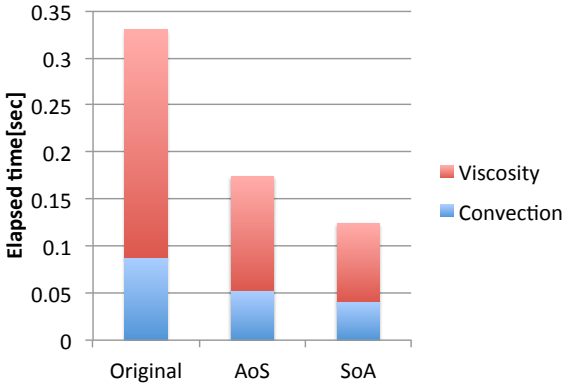

図 13 UPACS の GPU 実行におけるデータレイアウトの影響

5. 提案

本稿における提案は，ハイレベルなプログラミングモデルへのデータレイアウトの抽象化の導入である．前述した通り，プログラムを実行するデバイスの性能特性の違いにより，大きく性能が異なることが知られている．特にホストとアクセラレータにおける，得意とするデータレイアウトの異なりは大きな問題である．この実行デバイスごとに最適なデータレイアウトが異なるという問題は，現在広く使われているローレベルなプログラミングモデルにおいても良く知られた問題であり，CPU 向けに書かれたプログラムのアクセラレータ向けの最適化として，データレイアウト変更の最適化は良く知られている．しかしローレベルなプログラミングモデルにおいては，ホスト-アクセラレータの間での可搬性はそもそも目的としておらず，アクセラレータ専用のプログラムとして書き換えてしまうために，比較的大きな問題にはなっていなかった．その一方でハイレベルなプログラミングモデルである OpenACC では，指示文を無視することで元の CPU のプログラムとしても実行出来るという，元のプログラムを維持したまま移植出来る機能的な可搬性が良いことがメリットであるため，得意とするデータレイアウトの違いによる性能可搬性の低下は解決すべき問題である．

さらにこの抽象化により，ローレベルなプログラミングモデルでは難しいとされていた，データレイアウトの自動最適化が可能になる点が，ハイレベルなプログラミングモデルで行うことの利点であり，本研究で達成されるべき目標である．

5.1 データレイアウトの抽象化方針

データレイアウトの抽象化にあたって，(1) データレイアウトを抽象化する範囲，(2) ユーザーが指示文で与えるべき情報の 2 点を考えるべきである．まず，(1) データレイアウトを抽象化する範囲についてであるが，`acc data` 領域レベルでの抽象化と `acc kernels` 領域レベルでの抽象化が考えられる．`acc data` 領域レベルでの抽象化とは，`acc data` でホスト・デバイスを同一のデータとして取り扱うことが出来るよう抽象化されているのと同様に，ホスト側ではホスト側の得意なデータレイアウト，デバイス側ではデバイス側の得意なデータレイアウトとして保持し，ユーザーにはあたかも同一のレイアウトを使っている様に見せる抽象化である．この場合，ホスト側とデバイス側でそれぞれのデータを関連づけておき，ホスト-デバイス間通信が発生するタイミングでレイアウト変更を行えば良い．しかしその場合，デバイス上で実行すべきカーネルが複数ある場合，あるカーネルにおいて最適なデータレイアウトが他のカーネルにおいても最適であるかどうかは明らかではない．それに対し `acc kernels` 領域レベルでの抽象化とは，各カーネルごとに閉じてレイアウトを抽象化する．この方針では他のカーネルに与える影響が小さいことが利点であるが，場合によってはカーネルが実行されるたびにレイアウトの変更を行わなければならず，レイアウト変更によるオーバーヘッドとカーネルの高速化率からなる最適

ⓒ 2014 Information Processing Society of Japan

```
#pragma acc trans transpose { array_name \
 [start : length][start : length][start : length], [1, 3, 2] }
{
    structured block
}
```

図 14 acc trans ディレクティブ

化問題になる．どちらのモデルが良いかは明らかではないため，検証する必要がある．

次に (2) ユーザーが指示文で与えるべき情報についてであるが，与える情報が多ければ多い程煩雑になり，ハイレベルなプログラミングモデルである利点が失われるため，自動的に最適なレイアウトが選択されることが望ましい．そのため，acc loop ディレクティブのような形式を取ることが望ましいと考えられる．acc loop ディレクティブでは，GPU のスレッドマッピングを調節するパラメーターを gang, worker, vector clause によって明示的に与えることが出来るが，与えない場合はコンパイラが自動的にスレッドマッピングパラメーターを選択する．そこで，本稿で提案するデータレイアウト変更ディレクティブについても，明示的にレイアウトの変更方式を指定することが出来，自動的に選択することも出来るという形式を取るべきであると考える．

5.2 ディレクティブの提案

以上を踏まえ，適切なデータレイアウトを選択するためのディレクティブとして，acc trans(**図 14**) を提案する．acc trans に与えるべき情報は，(1) レイアウト変更をするべき配列名，(2) 配列のデータレイアウト，(3) レイアウトの変換ルール (optional) である．(2) では多次元配列形式で元のデータレイアウトを記述させることにより，C 言語で良く用いられる 1 次元化された配列にも対応可能である．(3) のレイアウト変換ルールについてであるが，この部分についてはユーザーが明示的に指定，またはコンパイラが自動的に判定する．レイアウト変換ルールについてであるが，これはレイアウトの次元数と同数である必要があり．例えば array[I][J][K] のような配列に対し図 14 のように [1,3,2] と指定した場合，1 次元目は 1 次元目に，2 次元目は 3 次元目に，3 次元目は 2 次元目に変更され，指定された領域内では array[I][J][K] は array[I][K][J] とデータレイアウトを変換して取り扱われる．

6. トランスレーターの実装

前述の拡張ディレクティブ acc trans を実現するために，ソース-to-ソースのトランスレーターを実装した．トランスレーターの実装にあたり，ROSE Compiler Infrastructure[7] を用いた．トランスレーターの挙動は以下の通りである．

(1) Rose コンパイラがインプットされたプログラムを解析し，AST を生成する．

```
#pragma bcc trans transpose(foo_a[0:100][0:100][0:3],[1,3,2])
  {
#pragma acc data copy (foo_a[0:100][0:100][0:3], \
foo_b[0:100][0:100][0:3])
    {
#pragma acc kernels
#pragma acc loop gang independent
      for(k = 0;k < 100;k++){
#pragma acc loop vector independent
for(j = 0;j < 100;j++){
          for(i = 0;i < 3;i++){
            foo_b[k][j][i] = foo_a[k][j][i];
          }
}
      }
    }
  }
```

図 15 acc trans サンプルプログラム

(2) トランスレーターが拡張指示文である #pragma acc trans を読み取り，配列名，配列のデータレイアウト，変換方式を取得する．

(3) 取得した配列名から，変更すべき配列の宣言を取得し，データ型等の情報を取得する．

(4) 指示文で指定されたブロックの先頭で新しい配列を宣言し，データの確保を行う．配列宣言はポインター型で行い，多次元配列として宣言されている配列に関しても，1 次元的に取り扱う．また，ブロックの最後で確保したデータの解放を行う．

(5) 配列の次元入れ替えを行う関数を生成，またブロックの先頭/最後で生成した関数を呼び出し，データレイアウトの変更を行う．

(6) #pragma acc trans 領域内で宣言された acc data ディレクティブを新しい配列と差し替える．

(7) 指定領域内で使われている変更すべき全ての配列を新しい配列に入れ替える．この際，新しい配列のインデックスは再計算する．

例えば**図 15** のプログラムをこのトランスレーターにより変換すると，**図 16** を生成する．

しかし，現状の実装で対応しているのは，前述の提案における，acc data レベルでのレイアウトの抽象化のみであり，data ディレクティブ内部で acc trans ディレクティブを用いることは出来ない．また，自動最適化機構も実装出来ておらず，データレイアウトの変換ルールはユーザーが明示的に行う必要がある．これらの実装は今後の課題である．

7. トランスレーターの評価

実装したトランスレーターの評価を行うために，姫野ベンチマークの図 6 における SOAOS 型の配列と AOS 型の配列に対してトランスレーターによる変換を適用し，全通

ⓒ 2014 Information Processing Society of Japan

```
#pragma bcc trans transpose \
( foo_a [ 0 : 100 ] [ 0 : 100 ] [ 0 : 3 ], [ 1, 3, 2 ] )
{
    double *foo_a_generated__1_3_2;
    foo_a_generated__1_3_2 = ((void *)\
    (malloc(sizeof(double ) * 100 * 100 * 3)));
    transpos_foo_a_1_3_2(((double *)\
    foo_a_generated__1_3_2),((double *)foo_a));

#pragma acc data copy (foo_a_generated__1_3_2[0:100 * 100 * 3]\
 , foo_b[0:100][0:100][0:3])
{
#pragma acc kernels
#pragma acc loop  gang independent
      for (k = 0; k < 100; k++) {
#pragma acc loop  vector independent
        for (j = 0; j < 100; j++) {
          for (i = 0; i < 3; i++) {
            foo_b[k][j][i] = foo_a_generated__1_3_2 \
            [((0 * 100 + k) * 3 + i) * 100 + j];
          }
        }
      }
    }
    retranspos_foo_a_1_3_2(((double *)foo_a), \
    ((double *)foo_a_generated__1_3_2));
    free(foo_a_generated__1_3_2);
  }
```

図 16　acc trans サンプルプログラム

りの変換パターンについての計測を表 2 の計算環境で行った．用いたコンパイラとそのオプション，環境変数等も同一である．CPU, Xeon Phi, GPU で評価を行った結果がそれぞれ**図 17**, **図 18**, **図 19** である．4 次元の配列であるため，データレイアウト変換パターンは 24 通りあり，その全てについて評価を行ったが，グラフでは最内次元を動かすパターンのみ抜粋している．各グラフの一番左側の青い棒グラフは，オリジナルと同一のデータレイアウトを用いている．Xeon Phi と K20X GPU においては，[1,2,4,3] の変換ルールを適用したパターン，つまり A[I][J][K][4] を A[I][J][4][K] のように変換したパターンが最も速く，Xeon Phi と GPU でそれぞれ 27%，24%の高速化が得られた．一方で CPU ではデータレイアウトを変換しないパターンが最も高速であった．ただし，データレイアウトを変換していないパターンにおいても，結果的にデータレイアウトは同一になるが，新しい配列の宣言や確保，データのコピー等は行っている．姫野ベンチマークの場合，カーネルが一つしかないために，カーネル全体を `acc trans` の領域内に含めることで，データ転送のコストや変換コストはほとんど無視出来るが，Xeon Phi における結果をトランスレーターによる変換を用いない図 7 と比較すると，変換を用いない場合には最大で 70GFlops 以上の性能が得られていたのに対し，半分程度の性能しか得られなかった．これは，オリジナルでは static な多次元配列として宣言されて

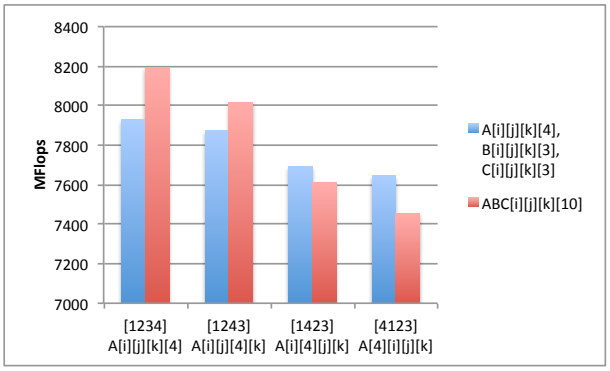

図 17　姫野ベンチマーク on CPU (縦軸の最小値が 0 ではないことに注意)

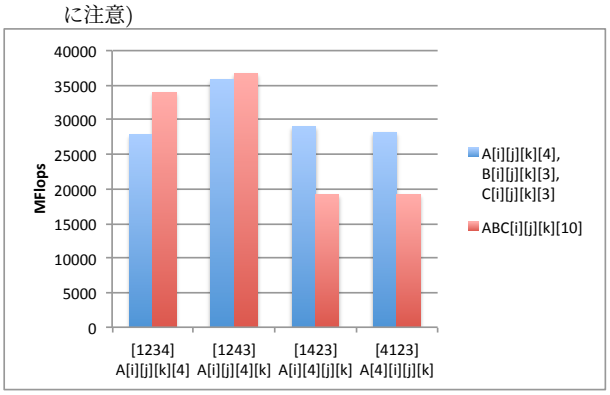

図 18　姫野ベンチマーク on Intel Xeon Phi

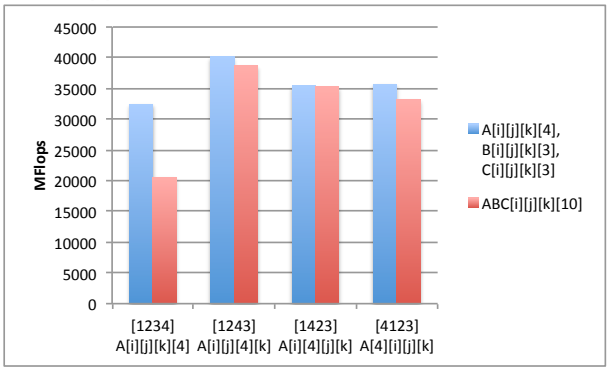

図 19　姫野ベンチマーク on K20X

いた配列を 1 次元化したために，コンパイラによる最適化が効き辛くなったためだと考えられる．

今回の評価では，主要ループが一つしかない姫野ベンチマークのみでしか評価していない．UPACS のように複雑の実アプリケーションにおける評価は今後の課題である．

8. おわりに

本稿では，アーキテクチャにより得意とするデータレイアウトが異なること等が，ディレクティブベースプログラミングモデルである OpenACC の異なるデバイス間における性能可搬性を損ねる原因となることに注目し，データレイアウトの抽象化を行うためのディレクティブを提案し，トランスレーターを実装した．また，姫野ベンチマークに本稿で提案するディレクティブを適応することにより，オリジナルと同一のデータレイアウトと比較して，Intel

ⓒ 2014 Information Processing Society of Japan

Xeon Phi 上で 27%，K20X GPU 上で 24%の性能向上を確認した．

しかし，現状のトランスレーターには自動最適化機構が実装されておらず，本来であれば最適なデータレイアウトは実行するデバイスに合わせて自動的に選択されることが望ましいが，現状ではユーザーが明示的に指定しなければならない．また，姫野ベンチマークの様な単純なベンチマークでなく，複雑な実アプリケーションにおいても効果が得られるかどうか検証する必要があり，これらは今後の課題である．

さらに，実際のアプリケーションでは Array of Structures, Structure of Arrays と言った簡単なデータレイアウトのみでなく，複雑なデータレイアウトを扱う上に，カーネルごとに最適なレイアウトが違うケースも考えられ，データ変換のコストと各カーネルの速度向上からなる最適化問題となる．本研究では，これらのモデル化に取り組むとともに，自動最適化を目指している．

参考文献

[1] Che, S., Sheaffer, J. W. and Skadron, K.: Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, ACM, pp. 13:1–13:11 (online), DOI: 10.1145/2063384.2063401 (2011).

[2] Dolbeau, R., Bihan, S. and Bodin, F.: A Hybrid Multicore Parallel Programming Environment, *High Performance Computing* (Valero, M., Joe, K., Kitsuregawa, M. and Tanaka, H., eds.), Lecture Notes in Computer Science, Vol. 1940, Springer Berlin / Heidelberg, pp. 182–190 (2007).

[3] Hoshino, T., Maruyama, N., Matsuoka, S. and Takaki, R.: CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application, *Cluster Computing and the Grid, IEEE International Symposium on*, Vol. 0, pp. 136–143 (online), DOI: http://doi.ieeecomputersociety.org/10.1109/CCGrid.2013.12 (2013).

[4] in High Performance Computers, S. S. M. B.: http://www.cs.virginia.edu/stream/.

[5] Lee, S. and Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, IEEE Computer Society, pp. 1–11 (online), DOI: 10.1109/SC.2010.36 (2010).

[6] OpenACC-standard.org: The OpenACC Application Programming Interface, (online), available from ⟨http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf⟩ (2011).

[7] Schordan, M. and Quinlan, D.: A Source-To-Source Architecture for User-Defined Optimizations, *Modular Programming Languages* (Böszörményi, L. and Schojer, P., eds.), Lecture Notes in Computer Science, Vol. 2789, Springer Berlin Heidelberg, pp. 214–223 (2003).

[8] Sung, I.-J., Liu, G. and Hwu, W.-M.: DL: A data layout transformation system for heterogeneous computing, *Innovative Parallel Computing (InPar), 2012*, pp. 1–11 (online), DOI: 10.1109/InPar.2012.6339606 (2012).

[9] Wolfe, M.: Implementing the PGI Accelerator model, *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, New York, NY, USA, ACM, pp. 43–50 (online), DOI: http://doi.acm.org/10.1145/1735688.1735697 (2010).

[10] 中田真秀小林広和：スレッド間空間的ブロッキングを利用した Xeon Phi 上の姫野ベンチマークの最適化 (2013).

[11] 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.

ⓒ 2014 Information Processing Society of Japan