Vol.2016-HPC-155 No.9
2016/8/8

データレイアウト最適化指示文によるOpenACCアプリケーションの高速化

星野 哲也[1,2]　丸山 直也[3]　松岡 聡[2]

概要：アクセラレータ向けの指示文ベースプログラミングモデルである OpenACC を用いて実装されたアプリケーションは，様々な環境において実行可能でありその高い可搬性が特徴と言える．しかし特定のデバイス向けに最適化されたアプリケーションは，別のデバイスにおける実行では十分な性能を達成できない場合がある．異なる性能特性を持つデバイスでは，有効な最適化方針が異なる場合があるためである．我々は，特にデバイスの性能特性により性能に大きく影響を与える得るプログラムのデータレイアウトに着目し，OpenACC の拡張により最適化を可能とした．実アプリケーションである UPACS と CCS-QCD へ適用し，手動による最適化と比較して，それぞれ 99.4%，92.3%の性能を達成した．

1. はじめに

2016 年 6 月の Top500 List[11] にて圧倒的な 1 位を獲得した Sunway TaihuLight 向けの並列プログラミングモデルにも採用されるなど，メニーコアアクセラレータ向けのプログラミングモデルとして OpenACC[7] が脚光を浴びている．OpenACC は，マルチコア CPU 向けの並列プログラミングモデルとして一般的である OpenMP 同様，指示文ベースのプログラミングモデルであり，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．代表的な演算アクセラレータである GPU 向けのプログラミングモデルとして，現在でも主流である CUDA が NVIDIA 社製の GPU のみを対象とする一方，OpenACC は様々なアクセラレータデバイスをターゲットとして選択することができ，その可搬性を売りとしている．しかし，ターゲットとするデバイスはそれぞれ異なる性能特性を持つことが一般的であり，デバイス毎に最適化戦略が異なることはよく知られた問題であるが，OpenACC のプログラミングモデルはその点への対応が十分とは言えない．機能的な可搬性という面から見れば，CUDA 同様広く使われている OpenCL も多数のデバイスをターゲットとしており優れているが，性能的な可搬性という面では，OpenACC と同様問題を抱えている．

著者らは [5] において，CUDA・OpenACC を用いて実アプリケーションの移植・最適化を行った結果，構造体の配列を手動で展開し，配列の構造体に書き換えるという，データレイアウトの変更が最も効果的な最適化であることを確認したが，適切なデータレイアウトは実行デバイス毎に異なるため，他のデバイスで実行した際に最適な性能を得られるとは限らない．この問題は，多数の異なるデバイスを対象とする OpenACC のようなプログラミングモデルにおいては，性能の可搬性の低下原因となるため，CUDA と比較してより顕著な問題となる．

この問題を解決するため，著者らは以前より，OpenACC にデータレイアウトを抽象化する拡張指示文を追加することで，データレイアウトの自動最適化を目指している．[4] において，データレイアウトを抽象化するための指示文の提案を行い，またソース-to-ソースのトランスレータを作成し，ベンチマークプログラムにてその有効性を確認した．しかし，実際のアプリケーションに適用するためには機能が不足しており，また性能最適化も十分ではなかった．本報告では，実アプリケーションに適用するための指示文拡張を行い，実アプリケーションである UPACS と CCS-QCD への適用を行うことで，トランスレータの評価を行った．その結果，手動による最適化には及ばないものの，指示文適用前のベースラインから UPACS と CCS-QCD でそれぞれ 123.5%，120.7%の性能向上を達成した詳細を報告する．

[1] 東京大学
The University of Tokyo
[2] 東京工業大学
Tokyo Institute of Technology
[3] 理化学研究所
RIKEN AICS

2. 背景

2.1 OpenACC

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．対象とする言語は C/C++や Fortran で，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コンパイラ指示文を挿入することで，アクセラレータ環境での実行を可能にする．OpenACC 以前にも，hmpp[2], PGI アクセラレータコンパイラ [12], OpenMP の CUDA 拡張 OpenMPC[6] 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可搬性が期待されている．

同じ指示文ベースのプログラミングモデルである OpenACC と OpenMP の大きな違いのひとつは，採用しているメモリモデルにある．OpenMP は共有メモリモデルであり，各スレッドが同じデータにアクセス可能であることが前提として作られている．その一方で OpenACC はホスト CPU 側と独立したメモリを持つアクセラレータデバイスを対象としている．ホスト CPU 側，アクセラレータデバイス側のメモリを表す概念として，それぞれ `host`，`device` という概念を導入している．また，対象としているアクセラレータデバイスは多数のコアが階層的に管理されていることを想定しており，階層的な並列性に対応するために，OpenACC は 3 階層の並列性を導入している．

例えば図 **1** は OpenACC による行列積の例である．OpenACC を構成する主要な指示文として，並列領域指定指示文，データ移動指示文，ループ指示文の 3 つがある．並列領域指定指示文はアクセラレータで実行するべき領域を指定するためのものであり，`parallel` 指示文，`kernels` 指示文がこれにあたる．図 1 の例では，2 行目の`!$acc kernels` から 16 行目の`!$acc end kernels` でループネストを囲むことにより，並列実行領域を指定している．データ移動指示文は `host-device` 間のデータ移動に用いられ，OpenMP と大きく異なる点のひとつである．OpenACC が対象とするアクセラレータデバイスは，ホスト CPU とは独立したメモリを持っているのが一般的であり，それに対応するためのものである．図 1 の例では，1 行目の`!$acc data` において，入力行列 a, b, c の `host` から `device` へ転送が行われる．ここで `copy(c)` とは，17 行目の`!$acc end data` において，`device` 側から `host` 側へのデータ転送を行うことを指示しており，一方 `copyin(a,b)` と指定した場合，`device` 側からのデータ転送は行わない．ループ指示文は並列化方法や並列粒度の指定を行うためのものである．OpenACC には 3 つの粒度，`gang`，`worker`，`vector` があり，`worker` は `vector` の集合，`gang` は `worker` の集合である．図 1 には 3, 5-6, 9 行目に `loop` 指示文が現れているが，それぞれ

```
1  !$acc data copy(c) copyin(a,b)
2  !$acc kernels
3  !$acc loop gang
4  do j=1,n
5     !$acc loop device_type(nvidia) vector(128) &
6     !$acc& device_type(radeon) vector(256)
7     do i=1,n
8        cc=0
9        !$acc loop seq
10       do k= 1, n
11          cc= cc + a(i,k) * b(k,j)
12       end do
13       c(i,j) =cc
14    end do
15 end do
16 !$acc end kernels
17 !$acc end data
```

図 **1** OpenACC による行列積

4, 7, 10 行目のループ文の粒度を指定している．この例では，最外の `j` ループが最も粗粒度で並列化され，`i` ループが細粒度に並列化される．またこの例では `worker` を指定していないため，`gang` は `vector` の集合である．`seq` の指定された 10 行目のループは並列化が行われず，`vector` 単位で逐次に実行されることとなる．また，最適な粒度はターゲットとするデバイスにより異なる場合がある．これに対応するために，OpenACC は `device_type` 指示子を用意している．`nvidia`，`radeon`，`xeonphi` などの予約語を指定することで，各デバイスごとに最適な粒度を指定することができる．

3. 関連研究

Sung らの研究 [9] 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．本研究では高レベル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いる．

4. データレイアウト最適化指示文の導入

前述の通り，OpenACC には対象とするデバイス毎に並列度を切り替えるための，device_type という指示子が導入されているが，データレイアウトを切り替える仕組みは導入されていない．適した並列度，データの格納順序，データへのアクセス順序は密接に関係し，性能に直結しているため，OpenACC の性能可搬性を高めるためには，並列度を切り替える仕組みのみでは不十分であると考える．

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 CUDA や OpenCL といった低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をアクセラレータ向けに書き換える際の問題として良く知られている．しかし低レベルなプログラミングモデルにおいては，プログラムそのものをアクセラレータ専用に書き換えてしまうことが多いため，比較的大きな問題にはなっていなかった．その一方で OpenACC では，指示文を無視すれば元のプログラムとしても実行可能であり，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重大な問題である．抽象化を行うことで，低レベルなプログラミングモデルでは難しいデータレイアウトの自動最適化を可能とすることが，本研究で達成されるべき目標である．

そこで本稿においては，OpenACC へのデータレイアウト最適化指示文の導入を提案する．OpenACC は host と device という独立したメモリ空間を持ち，データ移動指示文を用いることで host-device 間で適宜データのコピーを行い，データの一貫性を保つ．OpenACC ではこの際，device 側に持つデータは host 側と同じデータレイアウトしか許していないが，図 **2** に示すように，device 側のレイアウトに自由度を持たせ，実行デバイス毎に最適化するというのが基本的な方針である．

本稿における基本的な提案は [4] と同様であるが，主たる成果は，実アプリケーションに対応するための指示文・指示子の追加，Fortran への対応，実アプリケーションへの適用及び評価である．

4.1 データレイアウト最適化指示文 *acc transform*

適切なデータレイアウトを選択するための指示文として，[4] 同様，*acc transform*(図 **3**) を提案する．*acc transform* に与えるべき情報は以下である．

(1) 指示文の対象領域
(2) レイアウト変更をするべき配列名
(3) 配列のデータレイアウト
(4) レイアウトの変換ルール (optional)

図 3 に，すでにデバイス上に確保された多次元配列のデータ格納順序を変更する例を示す．2-4 行目の *acc transform* から 17 行目の *acc end trasnfrom* までが (1) の対象領域である．3,4 行目の *transpose* 指示子がターゲット配列 A, B のデータレイアウトの変更方法を示しており，多次元配列の転置を行うための指示子である．*transpose* の他に，次元数の変更を行う *redim* 指示子，構造体を多次元配列に展開する *expand* 指示子を仕様上は用意している．::(2,1,3) の部分が上記 (4) のレイアウトの変換規則にあたり，ユーザーが明示的に指定するか，またはコンパイラが自動的に判定する．ルールに示す正数値はレイアウトの次元数と同数である必要があり．(2,1,3) であればもともとの 1 次元目は 2 次元目に，2 次元目は 1 次元目に，3 次元目は 3 次元目にといったように変更される．すなわち図 3 の例では，A(l1:u1,l2:u2,l3:u3) は A'(l2:u2,l1:u1,l3:u3) の順に変更され，A(i,j,k) は添え字順を変更して A'(j,i,k) のようにアクセスされる．

また，実アプリケーションに対応するための拡張として，以下の機能を追加した．

(1) !$acc *present_transform*
(2) !$acc *transform_update host/device*

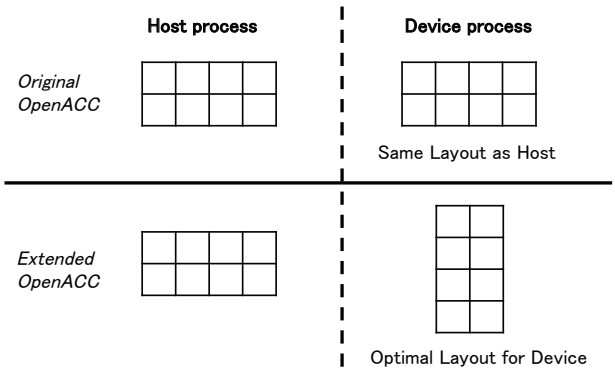

図 **2** データレイアウト最適化のイメージ

```
1  !$acc data copyin(A) copyout(B)
2  !$acc transform &
3  !$acc& transpose(A(l1:u1,l2:u2,l3:u3)::(2,1,3)) &
4  !$acc& transpose(B(l1:u1,l2:u2,l3:u3)::(2,1,3))
5  !$acc kernels present(A, B)
6  !$acc loop gang
7  do k = l3, u3
8     !$acc loop vector
9     do j = l2, u2
10       !$acc loop seq
11       do i = l1, u1
12          B(i,j,k) = A(i,j,k)
13       end do
14    end do
15 end do
16 !$acc end kernels
17 !$acc end transform
18 !$acc end data
```

図 **3** acc transform ディレクティブの一例

(3) !$acc transform *transpose_create/in/out/inout*
(4) !$acc loop *collapse(n)::(rule-list)*
(1) の *acc present_transform* は，*acc transform* 領域内における，関数呼び出しに対応するためのものである．ただし，*acc present_transform* が対象とする領域はこの指示文が適用された関数全体であり，指示文の挿入位置や指示文が対象とする領域は `acc routine` 指示文と同様である．従って，対となる *acc end present_transform* 指示文は用意されていない．(2) の *acc transform_update* は，`acc update` 指示文と似た指示文であり，*acc transform* 領域内においてホスト側，デバイス側をそれぞれ明示的に更新するために用いる．(3) は，`acc data` 指示文の指示子の `create`，`copyin`，`copyout` などと同様に，同期の必要の有無を指示するためのものである．(4) は，ネストループの順番を入れ替えるための，`collapse` 指示子の拡張である．`acc loop` 指示文の `collapse` 指示子は，ネストループを対象として，下に続く n 番目までのループを一つにまとめて並列化するよう指示するための指示子であるが，拡張として *transpose* 同様の規則を加えることにより，ループ順の変更を可能とする．最内側のループを 1 として数え，最外ループを n ループと数える．

5. トランスレータの実装

5.1 ソース-to-ソーストランスレータの実装

前述の拡張ディレクティブ *acc transform* を実現するために，ROSE Compiler Infrastructure[8] を用い，ソース-to-ソースのトランスレータの実装を行った．概要を図 **4** に示す．トランスレータは拡張指示文を含む OpenACC プログラムを入力とし，データレイアウト変換のためのランタイムを含む標準的な OpenACC プログラムを出力する．出力された OpenACC プログラムを PGI などの OpenACC 対応コンパイラでコンパイルし，ランタイムライブラリをリンクすることで実行形式を生成する．入力として許しているのは C または Fortran で書かれたプログラムであるが，一部の指示文は C 言語版では未対応である．拡張指示文のパーザを ROSE コンパイラに追加することで，入力のプログラムを抽象構文木に変換し，抽象構文木上でデータレイアウトの変換を行う．レイアウト変換がなされた抽象構文木を ROSE コンパイラのバックエンドに渡すことで，レイアウト変換がなされた OpenACC プログラムとして出力を行う．この手法についても以前の発表 [4] と同様であるが，前述の追加指示文に加え，Fortran に対応している．ただし，次元数の変更を行うための指示子である *redim* とユーザー定義型の配列を扱うための指示子である *expand* の実装は完成しておらず，実装・評価は今後の課題である．

図 3 を入力とした際のトランスレータによる出力を図 **6** に示す．3,4 行目にてレイアウト変換後配列である `A_tp`，`B_tp` の Host 上における領域確保を行い，Device

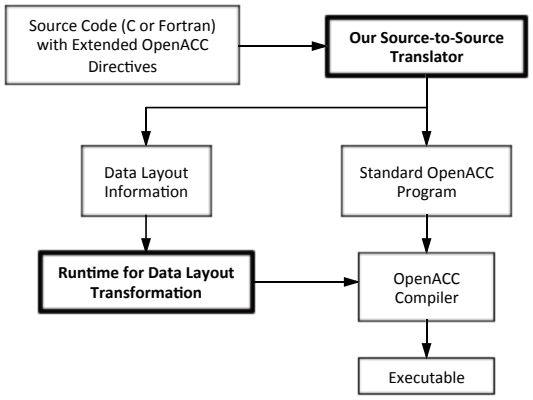

図 4 トランスレーターの概要

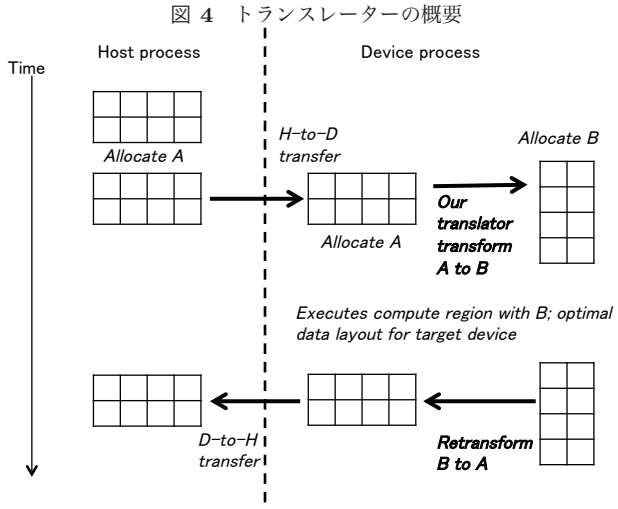

図 5 データレイアウト変更プロセス．オリジナルのデータレイアウト A をデバイス上で B に変更して扱う．

上への領域確保は 17,18 行目で呼び出しているライブラリ内で行っている．ライブラリに渡すべき配列サイズや変換規則などの情報は，5-16 行で行っているように配列に保存しライブラリの引数として受け渡している．また，図 6 中には現れていないが，Fortran の場合の変数宣言は当該ブロックの先頭で行っている．

5.2 ランタイムライブラリの実装

データレイアウトの変換を行うためのランタイムライブラリの実装について説明する．図 **5** に示したように，データレイアウト変更は GPU 上で行う．なおランタイムの実装が完了しているのは，多次元配列をターゲットとする指示子 *transpose* 向けのもののみであり，*redim, expand* 向けのランタイムの実装は今後の課題である．

ランタイムライブラリの実装は OpenACC+CUDA で行われており，データレイアウトの変更を実際に行うカーネルは CUDA によって実装されている．しかし OpenACC のプログラムから呼び出されるインターフェース部分については OpenACC で実装されているため，ランタイムライブラリと出力された OpenACC プログラムは同一のコンパイラでコンパイルする必要が有る．

```
1  !$acc data copyin (A), &
2  !$acc & copyout (B)
3  allocate( B_tp(l2:u2,l1:u1,l3:u3) )
4  allocate( A_tp(l2:u2,l1:u1,l3:u3) )
5  RUT_B_tp(3) = 3
6  UBT_B_tp(3) = u3
7  LBT_B_tp(3) = l3
8  RUT_B_tp(2) = 1
9  UBT_B_tp(2) = u2
10 LBT_B_tp(2) = l2
11 RUT_B_tp(1) = 2
12 UBT_B_tp(1) = u1
13 LBT_B_tp(1) = l1
14 RUT_A_tp(3) = 3
15     ! 中略
16 LBT_A_tp(1) = l1
17 CALL transpose_double_3(B_tp,B,LBT_B_tp,UBT_B_tp,
       RUT_B_tp,1)
18 CALL transpose_double_3(A_tp,A,LBT_A_tp,UBT_A_tp,
       RUT_A_tp,1)
19 !$acc kernels present (A_tp, B_tp)
20 !$acc loop gang
21 DO k = l3, u3
22    !$acc loop vector
23    DO j = l2, u2
24       !$acc loop seq
25       DO i = l1, u1
26          B_tp(j,i,k) = A_tp(j,i,k)
27       END DO
28    END DO
29 END DO
30 !$acc end kernels
31 CALL retranspose_double_3(A,A_tp,LBT_A_tp,UBT_A_tp
       ,RUT_A_tp,1)
32 CALL retranspose_double_3(B,B_tp,LBT_B_tp,UBT_B_tp
       ,RUT_B_tp,1)
33 deallocate( A_tp )
34 deallocate( B_tp )
35 !$acc end data
```

図 **6** 図 3 を入力とした際の出力の一部

transpose 用のランタイムライブラリは，機能的には任意の変換規則に対応できるが，実際に利用されるパターンは限られている．例えば 4 次元の配列を考えた時，arrray4D(I,J,K,L) に変換規則 (3,2,4,1) を適用し，arrray4D(K,J,L,I) のように変更することで高速化するケースは少ないと考えられる．現実に有用なケースとして，例えば X*Y*Z の 3 次元領域に離散化した空間の各点が n 個の物理量を持つような問題を解析の対象とする場合，この 3 次元空間上の物理量を phys(n,X,Y,Z) のような 4 次元配列として表現することは自然であるが，X 次元を分割し GPU のスレッドで並列に解く場合，phys(X,Y,Z,n) または phys(X,n,Y,Z) のように変換したほうが性能上有利であることが多い．phys(n,X,Y,Z) から phys(X,Y,Z,n) への変換は，phys(n,X*Y*Z) から phys(X*Y*Z,n) の 2 次元配列の転置とみなすことができ，phys(X,n,Y,Z) への変換についても同様に，phys(n,X,Y*Z) から phys(X,n,Y*Z) の 3 次元配列の転置とみなすことができる．4 次元以上の配列が対象となっている場合，ランタイムライブラリは変換規則からこのパターンを抽出し，最適化された 2・3 次元配列の転置カーネルを呼び出すことで，高速に転置を行う．

また，*transform* 指示文の指示子として *transpose_create* などが指定された場合，ライブラリは領域確保のみを行い，転置操作は行わない．この操作は，ライブラリの引数として渡している 0 か 1 のフラグにより操作している．

現状の実装の一つのデメリットとして，図 5 に示したように，アクセラレータ上に 2 倍の領域確保が必要になることが挙げられる．

表 **1** 実験環境

CPU	Intel Xeon X5670 6cores 2.93 GHz 2 sockets 54 GB Memory PCI-e Gen2
GPU	NVIDIA Kepler K20X 2688 CUDA cores 6GB Memory

6. 評価

実装したトランスレータの評価を行うために，以下の実験を行った．

- ランタイムライブラリ中で用いている転置カーネルを評価するためのマイクロベンチマーク．
- *present_transform* 指示文，*transpose_create* 指示子を適用した，流体アプリケーションである UPACS による評価．
- *collapse* 指示子の拡張によるループ順変更とランタイムライブラリによる転置の適用を行った，CCS-QCD による評価．

また評価環境を表 **1** に示す．

6.1 micro benchmark

ランタイムライブラリ中で用いる，多次元配列の転置カーネルの実行時間の評価を行う．転置を行うカーネルは，ホストからデバイスへの変換を行うカーネル (transpose) と，デバイスからホストへの変換を行うカーネル (retranspose) の 2 種類がある．図 **7** は，3 次元配列 array3D(1:10,1:1000,1:2000) の転置を行った際の性能である．実験環境は表 1 であり，CUDA で実装されたカーネルは nvcc (version 7.5) コンパイラ (オプション：-O3, -arch=sm_35) によりコンパイルし，OpenACC によるランタイムのインターフェース部分，及びベンチマークプログラムは，pgfortran (version 16.4) コンパイラ (オプション：-acc -Mcuda -O3 -ta=tesla,pinned) を用いている．図 7 の左端，規則 (1,2,3) は転置を行わない，単なる配列のコピーである．転置カーネルは，多次元・任意規則に対応するためナイーブな実装であるが，この中で実

際に使うことの多い転置パターンは規則 (2,1,3) または (2,3,1) であると考えられるため，この 2 つのパターンは個別に最適化を施している．n 次元 (n > 3) の配列であっても，(2,1,3,4,5,6,...,n) や (3,4,5,...,n,1,2) といった規則であれば，3,4,5,...,n のような連続部分では次元の入れ替えは起こらないため，連続部をまとめて扱うことで，それぞれ規則 (2,1,3)，(2,3,1) と同一視することができる．array3D(1:10,1:1000,1:2000) に規則 (2,1,3)，(2,3,1) を適用すると，それぞれ (1:1000,1:10,1:2000)，(1:1000,1:2000,1:10) のサイズを持つ配列に転置される．

規則 (2,3,1) は単なるコピーである規則 (1,2,3) と比較して，transpose の性能は 75.6%であったものの，retranspose は 16.7%の性能であった．さらなる最適化は今後の課題である．ただし，実際にアプリケーションで用いる際に，この変換を行うのは，多くはホスト-デバイス間の通信の発生時であると想定される．そのため，ホスト-デバイス間の通信と比較してこのカーネルが十分に速ければ，オーバーヘッドはほとんど無視できると考えて良い．ホスト-デバイス間のデータ転送を含めた実行時間が図 **8** である．ホスト-デバイス間のデータ転送自体は必ず必要なものであるため，transpose+retranspose カーネルの実行時間が純粋なオーバーヘッドとなる．規則 (2,3,1) の場合，2 つのカーネルの実行時間がホスト-デバイス間の転送時間の 26.5%である．また本実験環境は PCI-e Gen2 であるが，最新の規格では転送速度は 2 倍程度になっているため，オーバーヘッドは相対的に大きくなり，改善が必要であると言える．

しかし，転置の発生するタイミングはホスト-デバイス間の通信時であるため，この転送時間が相対的に小さいアプリケーションにおいては，ライブラリによるオーバーヘッドも小さくなる．さらに，カーネルの実行時間をホスト-デバイス間の通信時間に隠蔽することも原理的には可能である．しかし現在の *acc transform* 指示文は，既にデバイス上に確保済みの配列を主な対象としているため，ホスト-デバイス間の通信がどのタイミングで行われるか関知しておらず，従って変換時間の隠蔽もできない．この問題は，`acc data` 指示文を拡張し，確保と変換を同時に行うよう拡張することで達成できると考えられるが，今後の課題である．

6.2 UPACS

UPACS[10] は独立行政法人宇宙航空研究開発機構 JAXA 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，汎用的な流体アプリケーションである．ただし，今回対象としている UPACS は，JAXA の UPACS をベースとして，株式会社 IHI が開発のために独自の拡張を施したものである．

本評価には株式会社 IHI より提供された実際のデータを

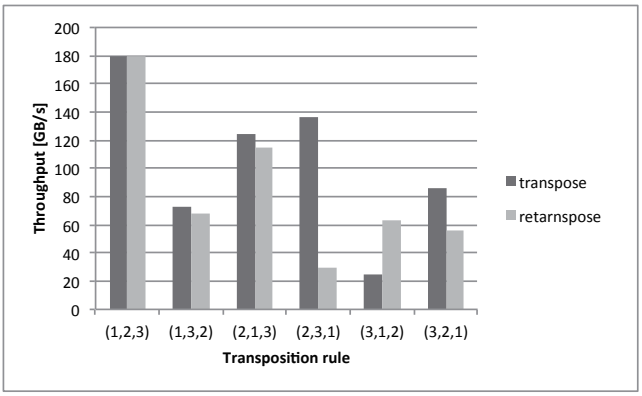

図 **7** transposition カーネルの実行バンド幅 (PCIe 転送時間を含まない)

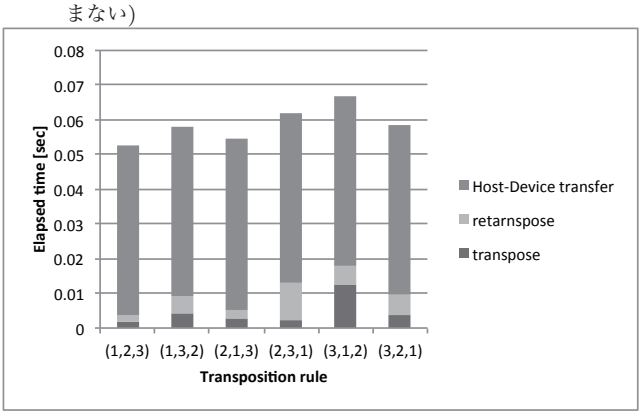

図 **8** PCIe 転送+transpostion カーネルの実行時間

用いる．実験対象データは単翼周りの流れを解析するための 3 次元データであり，格子点数は約 400 万点である．この 400 万点の格子を 7 つのブロックに分割し，翼列まわりへの適合を行っている．7 つのブロックのうち最も大きいものが 83 × 96 × 120 であり，最も小さいものが 110 × 26 × 120 である．なお UPACS ではこれらのブロックは最大 7MPI 並列で計算することができるが，今回は OpenACC の拡張についての評価であるため，MPI による実行は行わず，1 プロセスが全てのブロックの計算を行う．

UPACS の OpenACC 化は [13] にて行っており，手動で書き換えることでデータレイアウト変更の効果について検証している．しかしデータレイアウトの手動による書き換えはコストが高く，またあるデバイス向けに最適化してしまうと他のデバイスでの性能を損ねる原因となるため，自動で最適化できることが好ましい．今回は，データレイアウトの自動最適化に向けての初期評価として，UPACS の実行時間のうちの 6 割程度を占める，対流項計算部と粘性項計算部に拡張指示文を適用し，評価を行った．

ただし，今回対象とする配列は，ユーザー定義型の構造体の配列である．ユーザー定義型の構造体については *expand* 指示子によりサポートを予定しているが実装できておらず，直接のレイアウト変更はできなかった．そのため，オリジナルのユーザー定義型から CPU 向けに手動で書き換えたものをベースラインとする．また，今回対象と

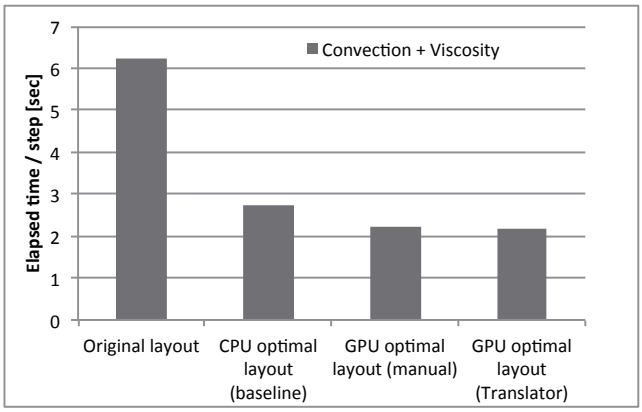

図 **9** UPACS 対流項・粘性項計算部 実行時間

する配列は，対流項・粘性項の計算時に一時的に用いられる作業用の配列である．従って，全て *transpose_create* 指示子の適用で済み，転置カーネルを使う必要はない．レイアウト変換対象とした配列を引数とした関数呼び出しについては，*present_transform* 指示文により対応した．プログラム全体に波及するデータレイアウトについては今後適用を進めていく．

図 **9** に指示文の適用結果を示す．左端がユーザー定義型を用いたオリジナルのデータレイアウトであるが，拡張指示文適用のベースラインとしたのは CPU optimal layout である．指示文適用により，CPU optimal layout と比較して 23.5%の性能向上を得た．また，指示文を適用しない手動による書き換えと比較した際のオーバーヘッドも 1%未満であった．

6.3 CCS-QCD

理研 AICS により整備されている, Fiber Miniapp Suite[3] の中から，実アプリケーションに近いアプリケーションとして，CCS-QCD を用いる．CCS-QCD は，格子 QCD 計算を行う際に現れる，粗行列向けの線形ソルバのベンチマークプログラムであり，Clover 部及び BiCGStab 部の主に 2 つのカーネルから構成される．Fiber Miniapp Suite より提供されている CCS-QCD は既に OpenACC 化が施されており，またデータレイアウトの変更とそれに伴うループ順の入れ替えの最適化が行われている．本実験ではこの最適化を外したものをベースラインとしている．

データレイアウトの変更の対象とした配列と変換規則の一部を表 **2** に示す．なお変換規則については，もともと施されていたレイアウト変更と同様のレイアウトになるように決定している．多くの場合，最内の 2 次元を最外もしくはその一つ内側に移動している．この場合，前述の通り、ライブラリ内部では 3 次元配列の転置とみなして転置が行われる．また，拡張 *collapse* 指示子の適用も行った．適用例を図 **10** に示す．図 10 の例では，最内側から ic, jc, ith, iz, iy, ix のループ順であるが，3 行目の拡張 *collapse* 指示子により ith, iz, iy, ix, ic, jc のループ順に変更される．

表 **2** データレイアウト変換対象と変換規則例 (全 17 配列)

target array	transposition rule
ue_t (1:col,1:col,0:nth,0:nz1,0:ny1,0:nx1,1:ndim)	(3,4,5,6,1,2,7)
fclinve_t (1:clsph,0:nth,1:nz,1:ny,1:nx,1:2)	(2,3,4,5,1,6)
be_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)

```
1  !$acc kernels pcopy (wce_t, wco_t), &
2  !$acc & pcopyin (ue_t, uo_t) async (0)
3  !$acc loop collapse(6)::(3,4,5,6,1,2)
4  !$acc & independent &
5  !$acc & gang vector (128)
6     do ix=ixlow,NX
7     do iy=iylow,NY
8     do iz=izlow,NZ
9     do ith=0,NTH
10    do jc = 1,COL
11    do ic = 1,COL
```

図 **10** 拡張 collapse 指示子の適用

図 **11** と図 **12** が，以上の適用を行った際の実行時間 [sec] と性能 [GFlops] である．なお，問題サイズには CLASS1($8 \times 8 \times 8 \times 32$) のものを使用しており，MPI による並列化は行っていない．実行時間は Clover 部と BiCGStab 部の合計であり，性能はそれぞれのものを示している．左端のグラフがベースラインである，データレイアウト変更とループ順変更の最適化を外したものである．左から 2・3 番目がそれぞれ，手動とトランスレータによるデータレイアウトの変更結果である．左から 4・5 番目が，手動とトランスレータそれぞれにより，データレイアウト変更に加えループ順の変更を行った結果である．トランスレータによりレイアウト変換とループ順変更を行った場合，手動で行った場合と比較すると 92.3%程度の性能であったが，最適化を行わないものと比較すると，120.7%の性能を達成した．

7. 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，指示文ベースの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着目し，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，トランスレータを実装，実アプリケーションを用いて評価を行った．この結果，データのコピーを行う必要のない UPACS においては，手動による最適化と比較して 1%以下のオーバーヘッドであった．また，CCS-QCD による評価においては，データレイアウトの変更に加え，ループ順の変更を適切に行うことで，手動による最適化の 92.3%，最

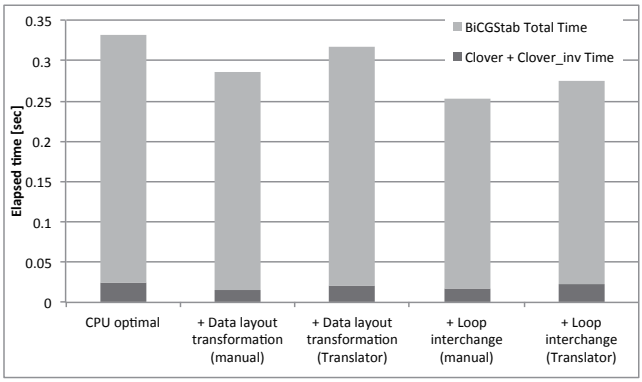

図 **11** CCS-QCD 実行時間

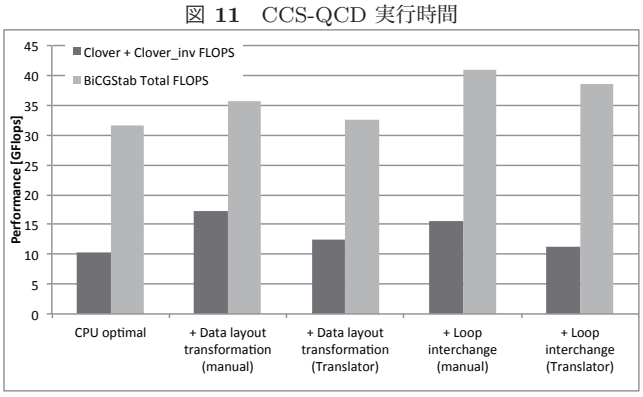

図 **12** CCS-QCD 性能

適化を行っていないものの 120.7%の性能を達成した．

しかし，現状のトランスレータには幾つかの課題も見られた．UPACS では，ユーザー定義型の配列に指示文が対応していないために，本来のオリジナルの配列に対しての適用ができなかった．また，現状のトランスレータには自動最適化機構が実装されていない．今回対象とした UPACS, CCS-QCD は，手動による最適化プログラムが既にあったため，同様の形に変更しただけであった．しかし本来，対象デバイスの最適なデータレイアウトを決定するのは難しく，しかも実行するデバイスごとに変化し得る．このような負担を軽減するために，今後は自動最適化を目指す．

謝辞　本研究は JSPS 科研費 2611834 の助成を受けたものである．また本研究にあたり，アプリケーション及び実験デー タを提供してくださった，株式会社 IHI の平川様をはじめとする皆様に，感謝の意を表する．

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] Fiber Miniapp Suite: http://fiber-miniapp.github.io/.

[4] Hoshino, T., Maruyama, N. and Matsuoka, S.: An OpenACC Extension for Data Layout Transformation, *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, Piscataway, NJ, USA, IEEE Press, pp. 12–18 (online), DOI: 10.1109/WACCPD.2014.12 (2014).

[5] 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).

[6] 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).

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

[8] 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).

[9] 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).

[10] Takaki, R., Yamamoto, K., Yamane, T., Enomoto, S. and Mukai, J.: The Development of the UPACS CFD Environment, *High Performance Computing* (Veidenbaum, A., Joe, K., Amano, H. and Aiso, H., eds.), Lecture Notes in Computer Science, Vol. 2858, Springer Berlin / Heidelberg, pp. 307–319 (2003).

[11] The Top 500 List: `http://www.top500.org/`.

[12] 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).

[13] 星野哲也，松岡　聡：圧縮性流体解析プログラムの OpenACC による高速化，情報処理学会研究報告，Vol. 2016-HPC-153, No. 4, pp. 1–10 (2016).