Vol.2014-HPC-143 No.5
2014/3/3

CPU-GPUそれぞれに最適なデータレイアウトを選択可能にするOpenACCディレクティブ拡張

星野 哲也[1]　丸山 直也[1,2,3]　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Low-level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High-levelなプログラミングモデルを用いる方法が考えられる．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，元のプログラムを壊さずに移植を行えるために，デバイス間の可搬性が高いことがあげられる．しかし現状のOpenACCなどの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等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，評価を行った．

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，Low-level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Low-level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[4]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，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，このようなアーキテクチャごとの違いが性能可搬性の低下の原因となる．これは現状アーキテクチャごとに大きく性能が変化するデータレイアウトなどが抽象化されていないためである．データレイアウトの抽象化を実現することができれば，Low-level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，High-level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性能可搬性を達成できるものと考えられる．そこで本研究の目的は，データレイアウトを抽象化の検討を行い，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

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，CAPSによって開発されている，CPU・GPU環境での並列プログラミング

[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

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

規格である．C/C++や Fortran 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．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[1], PGI アクセラレータコンパイラ [8], OpenMP の CUDA 拡張 OpenMPC[3] 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．

2.2 流体アプリケーション UPACS の GPU 環境への移植

UPACS[7] 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．我々の以前の研究 [9] において，UPACS の CUDA による移植・最適化を行い，CPU と GPU では得意とするデータレイアウトに違いがあり，この変更による性能最適化の効果が大きいことを実証した．また我々の研究 [2] において，OpenACC を用いて UPACS を移植し，CUDA による実装と性能・生産性などの比較を行った結果，CUDA と同様に最適化の効果を得られることを確認した．しかし，CPU と GPU の実行に同一のソースコードを用いるディレクティブベースアプローチである OpenACC による移植の場合，データレイアウトの変更は一方の性能を犠牲にすることとなり，性能可搬性が課題になる．

3. データレイアウトが性能可搬性に与える影響

3.1 データレイアウト変更による CPU・GPU における性能変化

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，

表 1 実験環境 (TSUBAME2.5 Thin ノード)

	CPU	GPU
Type	Intel Xeon X5670 × 2	NVIDIA Kepler K20X × 3
Frequency	2.93 GHz	0.73 GHz
Cores	6	2688 CUDA cores
Memory	54 GB	6GB

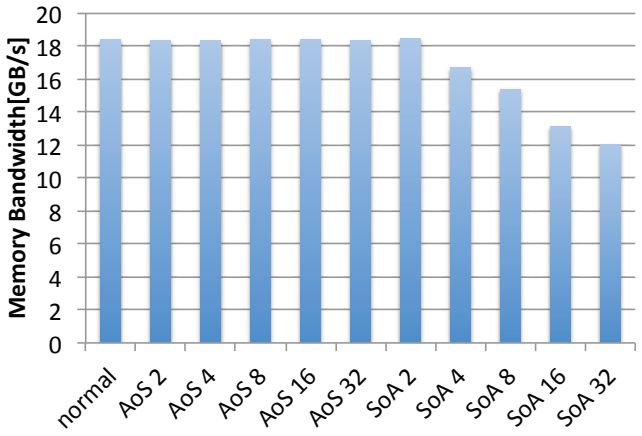

図 2 CPU におけるデータレイアウトの影響

Structure of Arrays 型の配列と Array of Structures 型の配列を用いて，データ転送性能を計測した．**図 2**, **図 3** それぞれ CPU，GPU 上で実行した結果である．実行した計算環境については**表 1** に示す．転送しているデータ量は一定であり，32MB 分のメモリをコピーしている．図中の normal, AoS N, SoA N はそれぞれ，構造体を用いない 4M 要素を持つ double の配列，Array of Structures のデータレイアウト，Structure of Arrays のデータレイアウトを示している．また SoA N, AoS N の N はそれぞれ構造体に含まれる配列の数と構造体の要素数を示している．図 2 から分かる通り CPU の実行では，Array of Structures 型のデータレイアウトではほとんど性能低下は観測できないが，Structure of Arrays 型のデータレイアウトにおいては配列数が増えるに連れて徐々に性能が低下し，配列数 32 の場合においては通常の配列と比較して 34%の性能低下が起きている．それに対し図 3 に示す通り，GPU 上での実行においては Array of Structures 型のデータレイアウトを使用した際に著しく性能が低下し，構造体の要素数 32 の場合において通常の配列のメモリ転送性能比較し，90%程度の性能低下が観測された．

3.2 実アプリケーション UPACS におけるデータレイアウト変更の影響

実アプリケーションにおけるデータレイアウトの影響を調査するために，UPACS のメジャーな計算フェーズのうち 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 4** は対流・粘

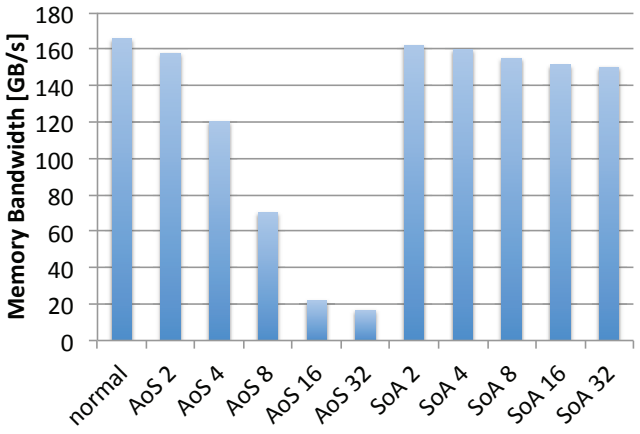

図 3 GPU におけるデータレイアウトの影響

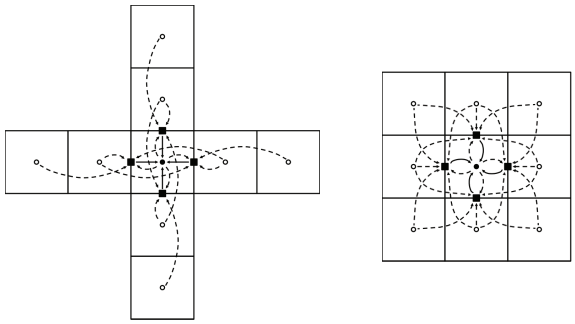

図 4 対流項 (左)，粘性項 (右) のステンシル計算

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 5** 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 5 を**図 6**, **図 7** のような Array of Structures 型, Structure of Arrays 型に変換し，CPU・GPU それぞれで実行したものが**図 8**, **図 9** である．グラフから 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が，これは構造体 `cellFaceType` 中にカーネル実行時に使われない不要な要素を含み，これらの転送がメモリ帯域幅を圧迫しているためである．図 8, 図 9 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，前述のメモリ転送ベンチマークにおける結果同様に，CPU では Array of Structures 型のデータレイアウトが最適であり，GPU では Structure of Arrays 型のデータレイアウトが最適である．

このように，アーキテクチャにより最適なデータレイア

```
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 5 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 6 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 7 Structure of Arrays 型のデータレイアウト

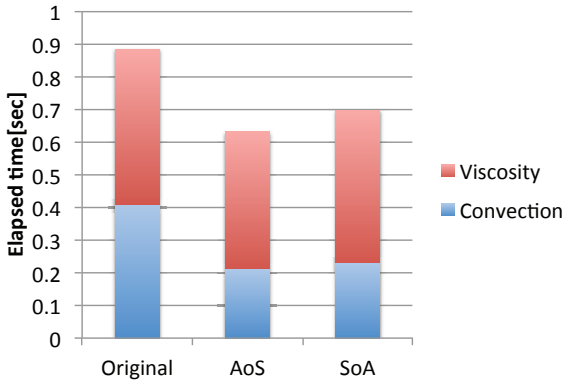

図 8 UPACS の CPU 実行におけるデータレイアウトの影響

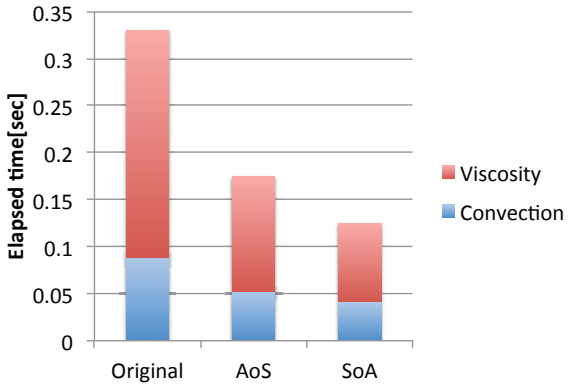

図 9 UPACS の GPU 実行におけるデータレイアウトの影響

ウトは異なる．しかし OpenACC 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においては，CPU・GPU どちらでも同一のプログラムを実行するため，性能可搬性を低下させる要因になると考えられる．

```
#pragma acc trans clause { array_name [ start : length] }
{
    structured block
}
```

図 10 acc trans ディレクティブ

```
struct my_struct *foo;
foo = (void *)(malloc(sizeof(struct my_struct) * 100));
#pragma acc trans aos_to_soa { foo [ 0 : 100 ] }
{
#pragma acc kernels
   for(i = 0;i < 100;i++){
      foo[i].b = foo[i].a;
   }
}
```

図 11 acc trans サンプルプログラム

4. 最適なデータレイアウト選択のための OpenACC ディレクティブの拡張と実装

4.1 ディレクティブの設計

前述の CPU-GPU における最適なデータレイアウトの違いを抽象化するために，OpenACC のディレクティブを拡張する形で，データレイアウト変更のためのディレクティブ `acc trans`(**図 10**) を提案する．`acc trans` は**図 11** のように，プログラム中のデータレイアウトを変更したい領域を囲む形で定義する．さらに `aos_to_soa` のように，どのようにデータレイアウトを変更したいのか指定し，実際に変更するべきデータの名前，データの要素数を指定する．変換目標のデータ型，変換対象の配列名とサイズを受け取ることで，指定された領域内ではデータレイアウトを変更して実行を行う．しかし，このディレクティブの設計は試験的なものであり，議論の余地がある．例えば，現状の設計では変換後のデータレイアウトをプログラマが指定する仕様となっているが，これは本来実行するデバイスによって自動的に決定されるべきである．

4.2 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[5] を用いた．我々のトランスレーターは，`acc trans` を含む拡張 OpenACC のプログラムを入力とし，ROSE により生成された AST を解析し，ディレクティブの情報を元にソースコードレベルでデータレイアウトを変換し，通常の OpenACC プログラムを出力する．例えば図 11 のようなプログラムを入力とした時，**図 12** のようなプログラムを出力する．ただしこのサンプルプログラムでは構造体の宣言部分等は省略している．プログラムの他の部分に影響を与えないために，`acc trans` で指定された領域内で，変換後の構造体の宣言，領域確保，元の構造体からのデータコピー，カーネル部分の実行，元の構造体へのデータコピー，領域の解放を行う．しかしこの実装には改善の余地があり，元の構造体からのデータコピーは現在 CPU 上で行うように実装しているが，これは本来存在しないオーバーヘッドになる．この変換をメモリバンド幅大きい GPU で行う方法や，PCI 通信で隠蔽する方法等で改善が見込める．

```
struct my_struct *foo;
foo = ((void *)(malloc(sizeof(struct AoS) * 100)));
//#pragma acc trans aos_to_soa { foo [ 0 : 100 ] }
{
   struct my_struct_SoA foo_SoA;
   foo_SoA.a = ((void*)(malloc(sizeof(double) *100)));
   foo_SoA.b = ((void*)(malloc(sizeof(double) *100)));
   memcpy_AoS_to_SoA(foo, foo_SoA, 100);
#pragma acc data (foo_SoA.a[0:100], foo_SoA.b[0:100])
   {
#pragma acc kernels
      for(i = 0;i < 100;i++){
         foo[i].b = foo[i].a;
      }
   }
   memcpy_SoA_toAoS(foo_SoA, foo, 100);
   free(foo_SoA.a);
   free(foo_SoA.b);
}
```

図 12 アウトプットプログラム

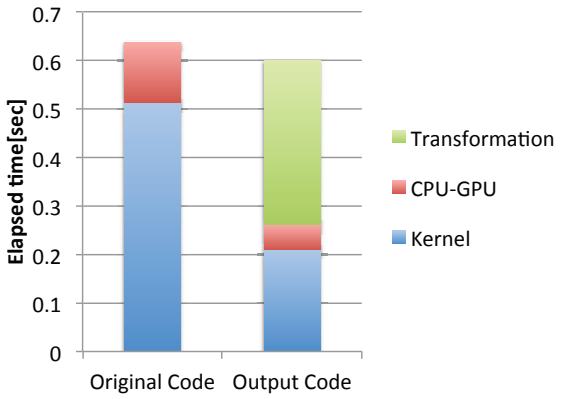

図 13 UPACS の 1 カーネルへの acc trans ディレクティブの適用

5. トランスレーターの評価

作成したトランスレーターの評価を行うために，UPACS の対流項計算フェーズの 1 カーネルを抜き出してディレクティブの適用を行い，TSUBAME2.5 表 1 上で評価を行った．**図 13** に示した結果は，CPU-GPU 間のデータ通信とデータレイアウトの変換を最初と最後に 1 度だけ行い，Kernel 部分を 100 回実行した際の所要時間である．トランスレーターの変換により，Kernel 部分においては元のプログラムと比較して 2.4 倍の性能向上が得られている．UPACS のような時間発展型のプログラムにおいては，トランスフォーメーションのコストはイテレーション回数の増加により相対的に小さくなるため，Kernel 部分における高速化の達成は十分な成果であると言える．

6. 関連研究

Sung らの研究 [6] 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．

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，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 UPACS の 1 部カーネルによる評価を行い，トランスレーターの評価を行い，カーネル部分において 2.4 倍程度の性能向上を確認した．しかし，現状のディレクティブのデザインには改善の余地がある．変換後の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指定しなければならない．またトランスレーターの実装にも改善の余地があり，現在ではデータレイアウトの変更を CPU 上で行っておりオーバーヘッドとなっているが，このコストは PCI 通信とオーバーラップさせる等の方法で隠蔽することが出来るため，トランスレーターの性能改善は今後の課題である．また，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，さらに複雑なデータレイアウトを扱う．さらに，CPU-GPU の混在するヘテロジニアスな環境においては，どのようなデータレイアウトをとりどのデバイスで実行するのが最適であるかは明らかでないため，これらのモデル化に取り組むとともに，自動最適化を目指すことが今後の課題である．

謝辞 本研究にあたり，UPACS を提供してくださった，独立行政法人宇宙航空研究開発機構准教授の高木亮治先生をはじめとする皆様に，感謝の意を表する．

参考文献

[1] 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).

[2] 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).

[3] 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).

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

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

[6] 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).

[7] 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).

[8] 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).

[9] 星野哲也，丸山直也，松岡 聡：大規模流体アプリケーションの CUDA・OpenACC への移植性の評価，情報処理学会研究報告，Vol. 2012-HPC-135, No. 42, pp. 1–9 (2012).