jh130042-IS02

OpenACC を用いた大規模流体アプリケーションの高速化

松岡聡（東京工業大学）

概要　スーパーコンピュータへの省電力要求の高まりに対応するため、消費電力に対する演算性能の高い GPU(Graphic Processing Unit) 等をアクセラレータとして用いた計算環境が増加している。しかし、従来の CPU 向けに作られた既存のアプリケーションをアクセラレータ上で十分な性能を得るためには、アーキテクチャの特性を理解した最適化を施す必要があり、これらアプリケーションの移植が問題になっている。本研究では、既存のプログラムに指示文を挿入することでアクセラレータへの移植を行うことが出来る OpenACC を用い、JAXA により開発されている UPACS の高速化を行った。適用した最適化の中で特に効果の大きいデータレイアウトの変更の自動化を目指し、トランスレーターを構築し、評価を行った。

1. 研究の目的と意義

スーパーコンピュータの大規模化に伴い、計算機の消費電力制約がより重要な問題となってきている。スーパーコンピュータの省電力のためには、GPU(Graphic Processing Unit)等のアクセラレータを計算機に用いる手法が有効である。実際に、スーパーコンピュータの省電力を競う The Green 500 List(2013 年 11 月版)において、GPU をアクセラレータとして用いる、東京工業大学等が開発した TSUBAME-KFC が 1 位となっており、10 位までを GPU を使ったシステムが独占している。しかし、アクセラレータを用いたシステムにおいては、既存の CPU 向けに作られたアプリケーションを如何に実行するかが問題となり、このような計算環境への移植手法の確立が重要な課題となっている。

このようなアクセラレータを用いた計算環境への移植手法として、OpenACC が注目されている。OpenACC は、近年一般的であるマルチコアの計算環境において広く使われている OpenMP と同様に、ディレクティブベースのプログラミングモデルである。既存の CPU 向けに作られた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。我々は現在までに、アクセラレータ向けのプログラミングモデルとして現状最も広く使われている CUDA と OpenACC を用いてアプリケーションの移植を行い、CPU と GPU のアーキテクチャの特性から最適化方針が異なるために、性能可搬性が問題になることが分かっている。そこで本研究では、性能可搬性について問題になる点を自動最適化により解決することを目的とする。

2. 当拠点公募型共同研究として実施した意義

(1) 共同研究を実施した拠点名および役割分担

- 東京工業大学・理化学研究所
  アプリケーションの移植・性能最適化、性能評価・モデル作成
- 宇宙航空研究開発機構
  流体アプリケーション UPACS の開発

(2) 共同研究分野

- 高性能計算(HPC)
- 数値流体力学

(3) 当公募型共同研究ならではという事項など

本研究では、今後主流になると考えられているメニーコア型のアクセラレータを用いた計算環境への、既存のアプリケーションの移植方法の確立を目的としている。この目的を達成するにあたり、超並列環境においてアプリケーションに要求されることは何か、システム側に求められるものが何かを見極めるために、実際にアプリケーションを作る立場にある数値解析分野の数理科学者と、高性能計算を専門とする計算機科学者の協力が不可

欠である。また、実際にメニーコア環境を用いた実験が不可欠であるが、大規模にアクセラレータを利用出来る環境は現状限られている。数理科学者と計算機科学者の協力のもと、大規模にアクセラレータを用いた成果が得られる点が、当公募型共同研究ならではの特徴である。

3. 研究成果の詳細と当初計画の達成状況

3.1 概要

3.1.1 UPACS

本研究において対象としているアプリケーションである UPACS (Unified Platform for Aerospace Computational Simulation)について説明する。UPACS 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。UPACS では圧縮性流体の数値シミュレーションを並列計算機上で行うために、マルチブロック構造格子法を用いている。マルチブロック構造格子法では、複数の構造格子を非構造的に接続することで、複雑形状周りの計算格子を作成しており、各々の構造格子を 1 ブロック単位としてプロセッサに割り当てることで、並列計算機への適用を可能にしている。この際、各構造格子の大きさはまちまちであるため、プロセッサ単位の計算量が異なり、ロードバランシングが問題になる。

3.1.2 CUDA・OpenACC

アプリケーションのアクセラレータへの移植手法として、現在では CUDA が広く使われている。しかし CUDA は、GPU のアーキテクチャを意識した低レベルな記述を行う必要がある。これに対し OpenACC は、C や Fortran で書かれた既存のプログラムに対し、OpenMP のように指示文(図 1 に例を示す)を挿入することで、アクセラレータ上で実行できるプログラムを生成する。また、CUDA が NVIDIA の GPU のみを対象としていることに対し、OpenACC は Intel や AMD 製のアクセラレータにも対応しているため、特定の計算環境に依存しないポータビリティが期待できる。

```
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 のディレクティブ

3.2 UPACS の OpenACC・CUDA による移植と最適化

中間報告書において報告した通り、UPACS は様々な解法を用いることが出来るが、本研究では実際に使われることの多い解法に対象を絞り、アクセラレータ上へのオフロードを行う。オリジナルのプログラムのプロファイリングは TSUBAME2.5 のノード(表 1)の 1CPU コアを用いて行った。プロファイリングの結果について表 2 に示す。本研究で対象としている UPACS の計算フェーズは 3 つの主要なフェーズからなる。対流項を計算する Convection フェーズ、粘性項を計算する Viscosity フェーズ、時間発展計算を行う Time Integration フェーズの 3 つである。このうち、Convection･Viscosity の両フェーズでは陽解法、Time Integration フェーズでは陰解法を用いている。

表 1 実験環境(TSUBAME2.5 Thin node)

	CPU	GPU
アーキテクチャ	Intel Xeon X5670 Westmere-EP	NVIDIA Tesla K20Xm
周波数	2.93 GHz	0.73 GHz
コア数	6	2688 CUDA cores
メモリ	54 GB	5.6 GB

表 2 UPACS のボトルネックとなるフェーズ

フェーズ	実行時間に占める割合
Convection	25.0%
Viscosity	37.7%
Time Integration	28.5%

主要な 3 つのフェーズの CUDA・OpenACC による移植を行い、CUDA・OpenACC により移植したそれぞれを表 3 の順に段階的にアクセラレータ向けに最適化し、それぞれの最適化の効果や、CPU-GPU 間の性能可搬性への影響について検証を行った。各最適化の詳細は中間報告書にて説明した通りである。

表 3 段階的最適化の適用

	最適化内容
Baseline	CPU-GPU 間のデータ転送コストを最小化
AoS to SoA	データレイアウトの変更
Thread mapping	スレッドマッピングパラメータの調節
Register blocking	ループ間で使い回せるデータのローカル変数を用いた再利用
Loop fusion	複数のループネストの合併

3.3 性能評価

3.3.1 CUDA・OpenACC 版 UPACS の性能

本稿における性能評価は表 1 に示す TSUBAME2.5 の 1 ノードを用いて行った。また、評価に用いたコンパイラを表 4 に示す。

表 4 実験に用いたコンパイラ

	コンパイラ	オプション
OpenMP	ifort	-fast -openmp
OpenACC	pgfortran	-fast -acc -ta=nvidia,cc35
CUDA	nvcc	-O3 -arch=sm_35

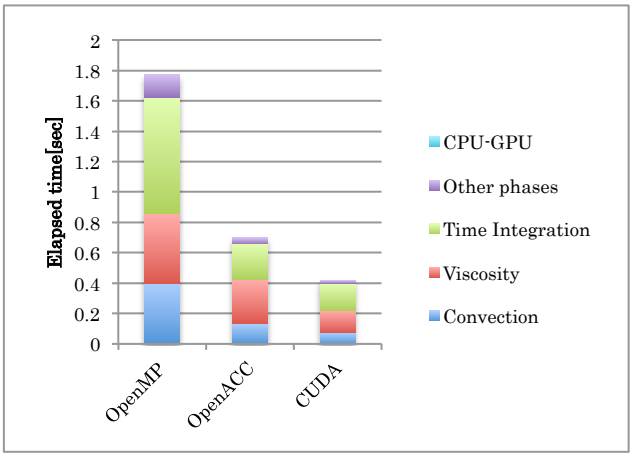

図 2 Baseline における実行時間比較

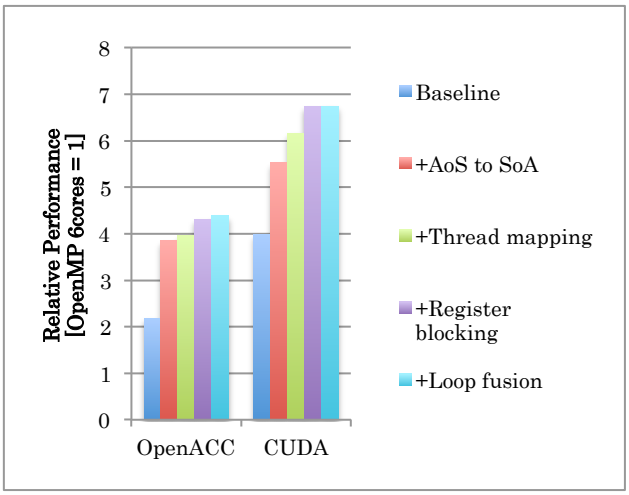

図 3 OpenMP 6 並列の性能を 1 とした各最適化による相対性能

Baseline 版における各フェーズの実行時間内訳の比較を示したものが図 2 である。OpenACC 版は CPU 実行の OpenMP 版と比較して 2 倍以上の性能向上を示したものの、CUDA の 55%程の性能であった。また、前述の最適化を適用した結果を図 3 に示す。図 3 は OpenMP の 6 コアにおける性能を 1 とし、OpenACC・CUDA それぞれの一連の最適化によって得られた相対性能を表している。グラフから分かる通り、OpenACC・CUDA 双方において、データレイアウトの変更による効果が非常に大きい。本研究の当初の計画では、ループフュージョンによる Byte/Flops を下げる様な最適化の効果が大きいものと予想し、これについてのモデル化、自動最適化を行う予定であった。しかし、ループフュージョンにより一部においては性能向上が見られたも

のの、大きな効果は得られなかったため、計画を変更してデータ構造の変更による性能への影響を詳しく調べ、自動最適化へ向けた検証を行った。

3.3.2 データレイアウトが性能にもたらす影響

構造体のデータレイアウトを変更するにあたって、大きく２つの方針がある。AoS 型(図 4)から SoA 型(図 6)に変更するものと、SoAoS 型(図 5)に変更するものである。図 7・8 で示したものが、AoS 型のデータ構造を使用している Convection・Viscosity フェーズの実行時間の合計である。図 7 が CPU6 コアを用いて OpenMP で実行したもので、図 8 が GPU 上で OpenACC を用いて実行したものである。どちらもディレクティブを無視すれば同一のプログラムであるが、CPU で最適なデータ構造が SoAoS 型であることに対し、GPU では SoA 型が最適である。ただし注意として、AoS の場合、各カーネルで構造体中に使われない要素が発生するが(例えば図 2 のカーネルでは構造体の flux 要素以外にはアクセスしない)、SoAoS、SoA では使わない要素へのアクセスはない。

さらに詳細に、それぞれのアーキテクチャにおける性能にデータレイアウトがどのような影響をもたらすかについて、それぞれ違う形のデータレイアウトを持つ配列をコピーする際のメモリバンド幅を計測した。図 9・10 がそれぞれ、CPU・GPU を使った際のメモリバンド幅である。ここで、AoS *N*、SoA *N* の *N* はそれぞれ、構造体に含まれる要素数と構造体中の配列の数を示しており、normal が構造体を用いない通常の配列をコピーした際のメモリバンド幅である。それぞれのデータレイアウトにおいてコピーすべき要素数は同じであり、構造体中に余分なコピーの必要のない余分な要素は含まれていない。グラフから分かる通り、CPU 実行の場合には SoA のコピー時、配列の数 32 の時に最も性能が劣化し、34%の性能低下が起こっているが、GPU 実行の場合には AoS のコピーを行った際に著しく性能が低下しており、通常の配列と比較して 90%の性能低下を示している。

```
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 4 Convection フェーズで用いられる AoS

```
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 5 SoAoS 型

```
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 6 SoA 型

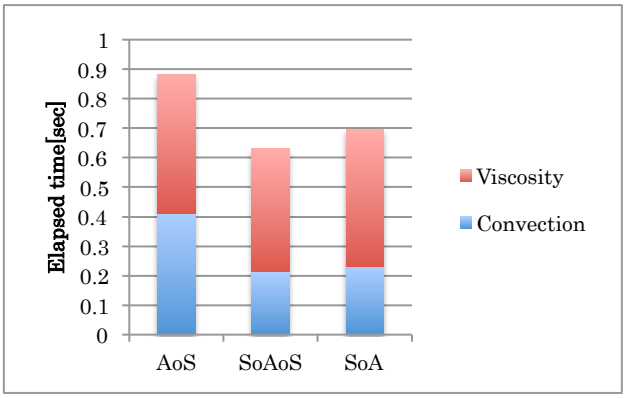

図 7 CPU(OpenMP 6 並列)における、各データレイアウトでの実行時間

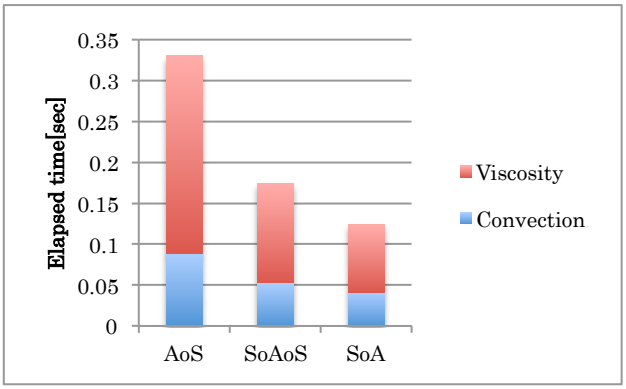

図 8 CPU(OpenMP 6 並列)における、各データレイアウトでの実行時間

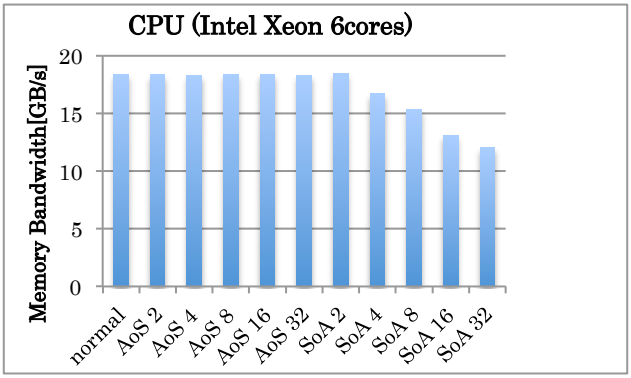

図 9 CPU における AoS・SoA コピー時のメモリバンド幅

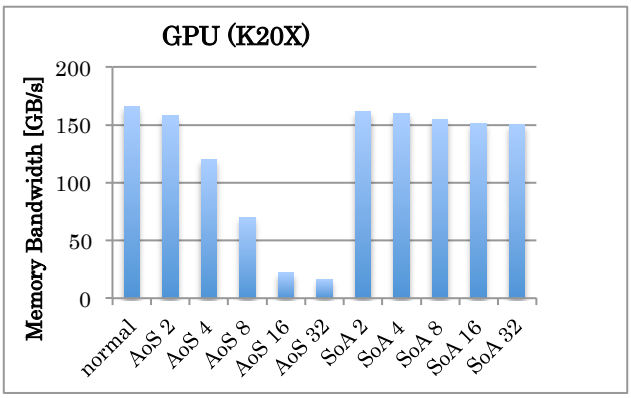

図 10 GPU における AoS・SoA コピー時のメモリバンド幅

3.4 自動データレイアウト変更トランスレーターの作成

3.4.1 トランスレーターの必要性

効果の大きかった最適化である構造体のデータレイアウトの変更についてであるが、データ構造はプログラム全体で共有されている為に、人手による変更はコストが大きい。一般的にデータ構造はプログラム全体で共有されることが多いため、このような自動最適化機構は多くのアプリケーションで有用であると考えられる。

当初の研究計画では、

- Byte/Flop を意識した性能最適化として、陽解法部分へのループフュージョンの適用およびモデル化
- CPU と GPU のアーキテクチャの特性の違いから生じる、最適解法の違いを評価するために、陰解法部分へのレッド・ブラック法を用いた並列化を導入し評価すること
- 負荷の不均衡から生じる性能への影響の評価
- 以上の評価結果から得られた知見を元に、自動最適化機構の提案を行う

という 4 点を予定していた。この自動最適化機構として、データレイアウト変更トランスレーターが多くのアプリケーションに取って有用であると考え、作成を試みた。

3.4.2 トランスレーターの実装

最適なデータレイアウトを選ぶために、本研究では OpenACC の拡張ディレクティブ **acc trans**(図 11)を定義した。このディレクティブによりプログラマが変更したいデータレイアウト、またデータレイアウトを変更すべき領域を指示することで、トランスレーターがディレクティブを解釈し、指定された領域内のデータレイアウトを変更する。現状のトランスレーターは、ROSE Compiler Infrastructure 上に実装されており、拡張ディレクティブを含む OpenACC プログラムを入力とし、ROSE により生成された AST を解析し、ディレクティブの指示を元にソースコードレベルでデータレイアウトを変更し、通常の OpenACC プログラムを出力する(図 12)。現状のトランスレーターの実装においては、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データレイアウトを変更している。例えば、トランスレーターの入力として図 13 の様な拡

張 OpenACC プログラムを用いた場合、図 14 のプログラムを出力する。acc trans 直後の領域内において、変換後の構造体の宣言、メモリ領域の確保、元の構造体からのデータコピー、カーネル部分の変更、変換元のデータ構造へのデータコピー、メモリ領域の解放を行う。しかし、この実装には改善の余地がある。現在の実装では、元のデータ構造から変換後のデータ構造へのコピーは CPU 上で行っているが、これは大きなオーバーヘッドになる。メモリバンド幅の大きい GPU で行う方法や、PCI のデータ通信に隠蔽する方法等で、改善が見込める。

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

図 11 データレイアウト選択のための OpenACC 拡張ディレクティブ

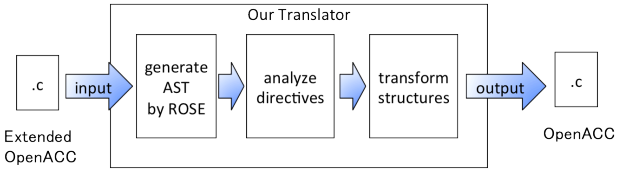

図 12 トランスレーターによるデータレイアウト変換イメージ

```
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 13 acc trans を用いた拡張 OpenACC プログラムの一部

```
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_SoA.b[i] = foo_SoA.a[i];
        }
    }
    memcpy_SoA_to_AoS(foo_SoA, foo, 100);
    free(foo_SoA.a);
    free(foo_SoA.b);
}
```

図 14 トランスレーターが図 13 のプログラムを入力とし、出力する OpenACC プログラム

3.4.3 トランスレーターの評価

トランスレーターの評価を行うために、UPACS の 1 カーネルを抜き出してディレクティブを適用し、評価を行った。図 15 は変換前のオリジナルのデータレイアウトのものと、トランスレーターにより変換されたコードの性能を比較したものである。ただし、データレイアウトの変換は最初と最後の 1 度だけであり、Kernel 部分は 100 回実行している。トランスレーターのデータレイアウト変換によって、Kernel 部分においては、変換前の元のプログラムと比較して 2.4 倍程度の性能向上が得られている。変換後のプログラムにおいては、トランスフォーメーションにかかる時間が大部分を占めているが、UPACS のような時間発展形のアプリケーションの場合、データレイアウトの変換は 1 度だけで良く、タイムステップの刻みが大き

くなれば、このオーバーヘッドは相対的に小さくなると考えられる。

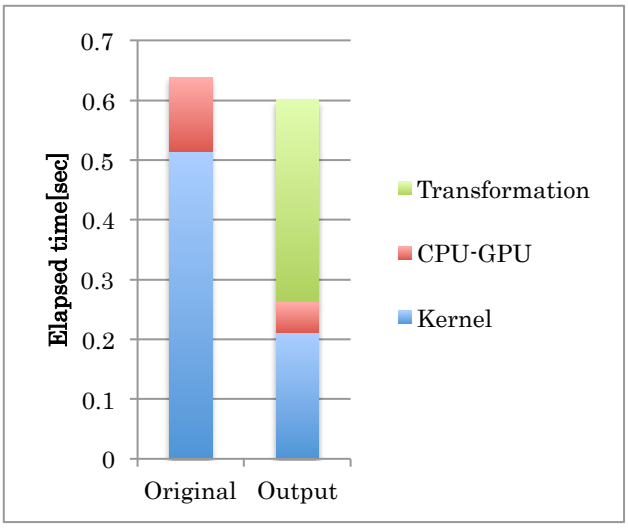

図 15　UPACS の 1 カーネルへの acc trans ディレクティブの適用

当初計画の達成状況について

本提案では、以下の 4 つのフェーズに研究進捗を分け、研究の意義を達成することを目的としてきた。

- フェーズ 1(2013 年 4 月～6 月)
  現状、超平面法を用いて並列化されている陰解法部分(Time Integration フェーズ)への、レッド・ブラック法を用いた並列化の導入および評価。
- フェーズ 2(2013 年 7 月～9 月)
  陽解法部分(Convection・Viscosity フェーズ)のループフュージョンの適用・評価および、自動最適化へ向けてのモデル化
- フェーズ 3(2013 年 10 月～2014 年 2 月)
  大規模実験による、計算負荷の不均一問題への対応
- フェーズ 4(2014 年 2 月～3 月)
  フェーズ 1～3 において得られた知見から、必要とされる自動最適化機構の提案を行う。

フェーズ 2 のループフュージョンに関しては本報告にある通り、効果の乏しいものであったため、データ構造のレイアウトを変更する最適化に注力することとした。またこの問題は UPACS 特有の問題ではなく、メモリバンド幅に律速される多くのアプリケーションをメニーコアアクセラレータ向けに移植する際に生じる問題であり、解決すべき重要な課題であると考えられる。そのため、当初の計画を前倒しして、トランスレーターの構築、評価を行った。

しかし、現状フェーズ 1・3 で行う予定であったアルゴリズムの変更、負荷不均一問題への対応に関しては、当初の計画を達成出来ていない。この点に関しては今後も研究を継続し、解決する予定である。

4.　今後の展望

本研究において構築したトランスレーターはまだ改善の余地があり、PCI の通信等に変換コストを隠蔽することにより、性能の改善が見込める。また、GPU では Kernel 内において、GPU の小容量で高速なオンチップメモリを用いることにより、データレイアウトをカーネル内で変更しながら実行することも可能である。これは現状のトランスレーターでは生成出来ないが、手動による CUDA 実装の評価により、CPU 等での変換コストなく、高速化出来ることが確認出来ているため、このようなコード生成にも対応したい。

また、データ構造はアプリケーションの根幹となる部分であり、後から人手で変更を加えることは容易ではない。そのため、特に OpenACC の様に同一のプログラムを異なるアーキテクチャで実行する場合、データレイアウトを自動的に最適化する仕組みが必要であると考えられる。また、現状のトランスレーターの仕様においては、プログラマが変換後のデータレイアウトを指定する必要があるが、これは本来、自動的に最適なデータレイアウトが選択されるべきである。さらに、最適なデータレイアウトは常に一定とは限らず、アプリケーションによって、または同一のアプリケーション内においても計算フェーズによって異なる場合がある。これからの計算機はレイテンシコアとメニーコアが混在するヘテロジニアスな環境にな

り、さらにはメモリを共有すると考えられているため、計算フェーズによってスカラプロセッサ、メニーコアプロセッサが協調動作する場合も考えられる。すると、最適なデータレイアウトを決定するためには、変換コスト、それぞれの計算フェーズ・アーキテクチャにおける実行コストからなる、最適化問題を解かなければならない。これは計算機科学の立場から見て非常にチャレンジングな課題であり、計算科学分野のアプリケーションにおいても重要な意味を持つため、本研究で構築したトランスレーターを発展させる形で、今後このような自動最適化機構を構築することを目標としている。

また、我々東京工業大学は OpenACC のコミッティに参加しており、本研究で得られた研究成果は、次期仕様策定にフィードバックされている。

5. 研究成果リスト

(1) 学術論文

なし

(2) 国際会議プロシーディングス

Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, Ryoji Takaki: “CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application”, International Symposium on Cluster, Cloud, and Grid Computing(CCGrid 2013)

(3) 国際会議発表(ポスター)

Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka: “OpenACC Performance and Optimization Studies with Kernel and Application Benchmarks”, GPU Technology Conference (GTC 2014)

(4) 国内会議発表

星野哲也, 丸山直也, 松岡聡: 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, HPC143 (2014)