Vol.2012-HPC-135 No.42
2012/8/3

# 大規模流体アプリケーションの CUDA・OpenACC への移植性の評価

星野 哲也[1]　丸山 直也[1,2,3]　松岡 聡[1,2,4]

**概要**：地震や気象予測，航空機や高層ビル設計といったシミュレーションに利用される数値流体力学アプリケーションは，近年一般的になりつつある GPU を用いたスーパーコンピュータにおいて，目覚ましい成果を上げている．しかし，GPU を用いたプログラミングは，高い性能を得ること難しいと言われており，レガシープログラムの GPU 環境への移植が問題となっている．本稿では，実際に利用されている大規模流体アプリケーションである UPACS を手動により CUDA 化し，性能と移植コストの面から評価を行った．また，プログラムの移植性を解決すると期待されている，OpenACC の予備評価を行った．これら評価の結果を示し，今後解決すべき課題について述べる．

# Evaluation of Portability for a Real-world CFD Application with CUDA and OpenACC

Tetsuya Hoshino[1]　Naoya Maruyama[1,2,3]　Satoshi Matsuoka[1,2,4]

***Abstract:*** Computational fluid dynamics (CFD) applications used for an earthquake and meteorological simulation are one of the most important application executed with high-speed supercomputers. Especially, GPU-based supercomputers have been showing remarkable performance of CFD applications. However, GPU-programing is still difficult to obtain high performance, which prevents legacy applications from being ported to GPU environment. We apply classical optimizations to a real-world CFD application UPACS and evaluate it's performance and porting costs, and we also evaluate OpenACC expected to provide portability across CPUs and GPUs. We demonstrate these results of evaluation and mention performance problems should be resolved in the future.

## 1. はじめに

計算機の急速な性能向上とともに，飛躍的に発展してきた数値流体力学アプリケーションは，地震や気象予測，航空機や高層ビルの設計といったシミュレーションに利用され，スーパーコンピュータ上で実行されるアプリケーションとして，最も重要なアプリケーションのひとつである．流体アプリケーションの支配方程式である，偏微分方程式を解析的に解く手法としてステンシル計算が頻出するため，これを高速に解くことが流体アプリケーションの性能上の鍵となる．ステンシル計算は多くの場合メモリバンド幅に律速されるため，CPU と比較しメモリ帯域の大きい GPU を利用する研究 [4] は盛んに行われており，その有効性が示されている．さらに，大規模に GPU を搭載したスーパーコンピュータ，東京工業大学に設置された TSUBAME2.0 などにおいて，流体アプリケーションの目覚ましい成果が上げられている [7][1].

GPU の性能に対する価格・消費電力の低さから，今後も GPU を利用した計算環境は増加することが予想される．ここで問題になるのが，既存アプリケーションの GPU 環境への移植である．既存アプリケーションの移植方法として，CUDA，OpenCL などを用いる方法が考えられる．し

[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
[4] 国立情報学研究所
National Institute of Informatics

かし，CUDA や OpenCL を用いる場合，ユーザーが GPU のアーキテクチャを意識した詳細な記述をしなければ，十分な性能が得られず，プログラムが複雑になりがちであり，アーキテクチャの変遷に伴う，アプリケーションの連続性が課題になっている．この解決策として，ソースコードを保持したまま，いくつかの指示文の挿入により GPU 環境で実行することが出来る，OpenACC[5] なども発表されているが，OpenACC を用いた研究はあまりなされておらず，実際の移植コスト，CUDA などに対する性能，解決すべき課題などは明らかになっていない．

本研究では，既存アプリケーションの GPU 環境への移植時の，性能・コスト・課題について明らかにし，定量的評価の指標を作ることを目的とし，実際に利用されている大規模流体アプリケーションである UPACS[8] を例として，GPU による高速化を行った．まず，手動により段階的に最適化を行いつつ CUDA 化を行い，この手動実装について，TSUBAME2.0 の単一ノードを用いての性能評価，書き換え行数をもとにしたコストの評価を行った．さらに，OpenACC を用いて実装した場合の性能評価を行った．これらの結果を示し，大規模流体アプリケーションへの GPU 適用における，性能上の課題を示す．

## 2. 背景

### 2.1 UPACS

本研究において対象としている大規模流体アプリケーションの例として，今回対象とした UPACS について説明する．UPACS[8] 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，複雑形状まわりの計算格子を作成し，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．さらに UPACS は，様々な流体解析プログラムを共通的に利用できるプラットフォームを確立することを目的としているため，コードの階層化，データおよび計算手法のカプセル化といった，オブジェクト指向的な考え方を用いて設計されている．これにより，数百ものモジュールを共通のデータ構造を用いて管理することを可能としている．

UPACS のように，格子法を用いた流体アプリケーションは，メモリへのアクセス量が多く，メモリバンド幅が性能律速の要因となることが多い．また，大規模なアプリケーションが一般に持つ性質である，オブジェクト指向的な考え方を用いて設計されているため，それぞれの計算カーネルの書き換えは独立に行うことが出来るが，データ構造などの，プログラム全体で共有されている要素を書き換える場合，書き換え範囲が広範囲に及ぶことが考えられる．

```
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 ディレクティブ

### 2.2 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，昨年，新しいプログラミング規格である OpenACC が発表された．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[2], PGI アクセラレータコンパイラ [9], 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．

## 3. GPU 環境への移植による性能・コストの評価手法

実際の大規模流体アプリケーションを GPU 環境へ移植する際の性能と実装コストを評価するために，UPACS に対して手動による CUDA 化を施し評価する．UPACS は開発言語が Fortran であるため，CUDA Fortran を使う方法も考えられるが，本研究では一度 C 言語に書き換えた上で CUDA に書き換える方法を取る．

まず，UPACS の既存 CPU 実装に対してプロファイリングを行い，性能上のボトルネックを特定する．UPACS は様々な解法を用いることが出来るが，本研究では解法は固定しており，GPU 環境への移植もこの解法で用いられる部分のみとする．CPU 実装のプロファイリングは，TSUBAME2.0 の 1 ノード (**表 1**) を用いて行った．ただ

**表 1** 実験環境 (TSUBAME2.0 Thin ノード)

| | CPU | GPU |
|---|---|---|
| Type | Intel Xeon X5670 × 2 | NVIDIA Tesla M2050 × 3 |
| Frequency | 2.93 GHz | 1.15 GHz |
| Cores | 6 | 14SM / 448 cores |
| Memory | 54 GB | 3GB |

**表 2** 1CPU での実行における，ボトルネックとなっているソルバと主要なサブルーチン

| ソルバ | 主要サブルーチン | 割合 (サブルーチン) | (ソルバ) |
|---|---|---|---|
| 対流項 | muscl | 10.7% | 25.0% |
| | flux | 4.4% | |
| | update | 3.3% | |
| 粘性項 | cellfacevariables | 27.2% | 37.7% |
| | flux_vis | 4.4% | |
| | update | 3.1% | |
| 時間項 | mfgs | 26.7% | 28.5% |
| | update | 0.6% | |

**表 3** 段階的な手動による最適化の適用

| | |
|---|---|
| V1 | ボトルネックのソルバ内の主要ルーチンを CUDA 化 |
| V2 | V1 + ボトルネックのソルバ内の全 CUDA 化 |
| V3 | V2 + 定数データの転送除去 |
| V4 | V3 + ボトルネックのソルバ以外も CUDA 化 |
| V5 | V4 + 対流項・粘性項ソルバ内ローカル配列のデータ構造変更 |
| V6 | V5 + MFGS ソルバの並列性の増加 |
| V7 | V6 + その他 (アドレス計算削減，共有メモリ利用等) |

し，UPACS ver.2.0 においては，ノード内並列化が施されていないため，1CPU ソケット内の 1CPU コアによりプロファイリングを行った．また，入力データの格子サイズは 120×120×120 であり，ナビエ-ストークス方程式を解く．この入力に対するプロファイリング結果を**表 2** にまとめる．

ボトルネックとなっているソルバ内の主要なサブルーチンを CUDA 化したものを V1 とし，これを手動 CUDA 実装のベースラインとする．V1 をベースに性能評価を行いながら，既知の最適化を段階的に適用したものを V2∼V7 と定義し (**表 3**)，これらについて性能・コストの比較評価を行う．

## 4. 実装・評価

### 4.1 CUDA 実装の性能・コスト評価

前述の実装を用いて TSUBAME2.0 の 1GPU で実験を行い，1 タイムステップにかかる実行時間を計測した．入力データはプロファイル時と同様のものである．また，流体系のアプリケーションの性能はメモリバンド幅に律速されることが多い．そのため，GPU での実行と比較するためには，CPU1 ソケット (6 コア) + メモリを一組と考えて

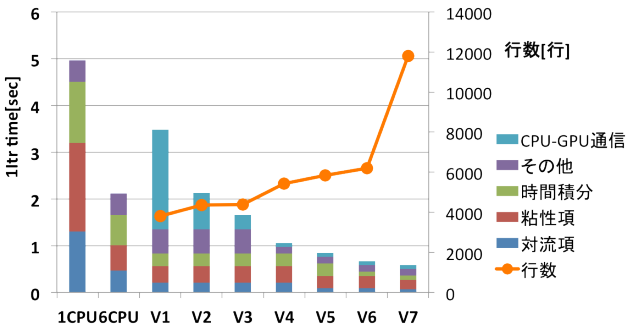


**図 2** 1 タイムステップ辺りの実行時間 (左軸)
書き換え行数 (右軸)

比較するべきである．よって各主要サブルーチンに対し，OpenMP による並列化も行った．ただし，通常の mfgs は陰解法であり，並列化するためにはアルゴリズムを変更する必要があったため，コンパイラによる自動並列をサポートするために UPACS が提供している mfgs_vector サブルーチンを用いて並列化を行った．オリジナルの 1CPU コアでの実行と比べて，6CPU コアでの実行では 2.5 倍程度高速化している．**図 2** はオリジナルの実装，OpenMP による 1 ソケット (6CPU コア) での実装，表 3 における各実装の 1 タイムステップにかかる実行時間と，書き換えに要した行数を表している．ただし，書き換えに要した行数には，Fortran から C 言語に書き換えた際の行数の増加分も含まれている．

OpenMP による 6 並列と V1 を比較すると，各ソルバにおいては性能向上が見られるのものの，全体としてはむしろ遅くなっている．原因は各サブルーチンの前後で行っている CPU-GPU 通信である．各サブルーチンの間に存在する CPU で実行していたサブルーチンが，GPU で実行するサブルーチンの結果と依存し，CPU-GPU 間で同期を取る必要があったためである．これを受けて V2 ではソルバ内を全て GPU 上で実行することで同期コストを減らし，V3 では実行中に変わらない定数のメモリの転送を除去し，V4 では主なボトルネックとなっていたソルバ以外の部分を GPU で実行することで，同期回数をを減らした．V2∼V4 では主に CPU-GPU 間の通信の削減をしているが，書き換えのコストは比較的に小さく，V1 から 3 倍程度性能向上している．

V5 以降では，GPU で実行するカーネル部分の最適化を行っている．まず V5 では，対流項・粘性項で局所的に使われていたデータ構造を Array of Structure から Structure of Array に変更した (**図 3**)．Array of Structure の場合，メモリ上に構造体である cface が連続して並ぶことになる．そのため，cface の要素である cface%area へのアクセスはストライド状になる．各スレッドが連続領域へのアクセスでない場合，グローバルメモリへのアクセスがコアレッシングされないため，データの転送効率が低下する．さらに，

```
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))

          ↓   ↓   ↓   ↓   ↓

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 3** Array of Structure 型の配列 (上) から，Structure of Array 型の配列 (下) への変更

各ルーチンでは cface の要素全てを使う訳ではないため，キャッシュに必要のないデータが読み込まれ，キャッシュヒット率の低下につながる．そこで，各要素をそれぞれ別の配列にすることで，これらの問題を解決した．**図 6** は，対流項・粘性項の主要サブルーチンの，データ構造変更に伴う実行時間の変化を示している．サブルーチンごとに差はあるが，最大で 4 倍程度の性能向上が得られていることがわかる．V6 では時間積分ルーチンの並列性を増加させるよう変更を行った．mfgs ルーチンは陰解法であり，対流項・粘性項ほどの並列性がないためである．mfgs では，更新するセルの値は前後・左右・上下の 6 セルの値を用いて計算されるが，この 6 セルの値をそれぞれ別のスレッドを用いて処理し，共有メモリを使ってリダクションすることによって並列性を増加させた．この変更により，時間積分部分では 2 倍程度の高速化が得られた．V7 では，細かなコードの変更による最適化を行っている．除算の除去，冗長な演算の除去，アドレス計算の削減などの細かい演算の最適化や，差分計算を行うサブルーチンを，i，j，k 方向それぞれの差分計算サブルーチンに分割することによる (**図 4**，**図 5**)，レジスタ使用量の削減，共有メモリの適用などを行った．これにより 1.2 倍程度性能向上したが，コード量が倍増してしまっている．

V1∼V4 で行ったのは，主にデータ転送の最適化であり，計算カーネルにはほとんど手を加えておらず，比較的単純な変更のみで 3 倍程度性能向上しているが，V5∼V7 では，データ構造の変更やアルゴリズムの変更を伴う計算カーネルの最適化であり，自動最適化を考えた場合，難しい変更であると言える．

### 4.2 メモリバンド幅計測による，現状の CUDA 実装の性能評価

現状の手動による CUDA 実装により，どの程度性能が得られているのかを確かめるために，各ソルバ内の主要な

```
do n=1,5
  do k=1,kn
    do j=1,jn
      do i=1,in
        ! idelta は i, j, k 方向それぞれの差分計算で
        ! (/1,0,0/),(/0,1,0/),(/0,0,1/) をとる
        im = i-idelta(1)
        jm = j-idelta(2)
        km = k-idelta(3)
        dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
            * ( flux(n,i,j,k) - flux(n,im,jm,km) )
      end do
    end do
  end do
end do
```

**図 4** 対流項 update サブルーチン

```
do n=1,5
  do k=1,kn
    do j=1,jn
      do i=1,in
        dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
            * ( flux(n,i,j,k) - flux(n,i-1,j,k) )
      end do
    end do
  end do
end do
```

**図 5** 各方向ごとに分割した update ルーチン (i 方向の差分計算)

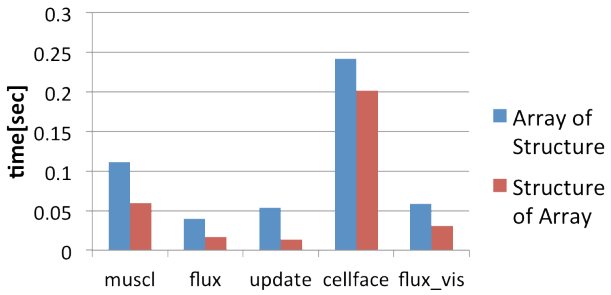


**図 6** データ構造の変更に伴う各サブルーチンの性能向上

**表 4** 主要サブルーチンのメモリバンド幅

| ソルバ | 主要サブルーチン | メモリバンド幅 (サブルーチン) | (ソルバ) |
|---|---|---|---|
| 対流項 | muscl | 41.8 GB/s | 6.0 GB/s |
| | flux | 46.8 GB/s | |
| | update | 47.9 GB/s | |
| 粘性項 | cellfacevariables | 9.6 GB/s | 2.7 GB/s |
| | flux_vis | 56.5 GB/s | |
| | update | 47.6 GB/s | |
| 時間項 | mfgs | 10.8 GB/s | 6.6 GB/s |
| | update | 49.8 GB/s | |

サブルーチンのメモリバンド幅を計測した．V7 の各サブルーチンに対して計測したメモリバンド幅を**表 4** に示す．

ここで，各サブルーチンのメモリバンド幅は，(入出力メ

モリ総量) / (実行時間) で求めており，読み込んだメモリが全てキャッシュに乗る，理想的な状態を仮定している．なお，実験に用た GPU(M2050) のメモリバンド幅の理論性能値は 148GB/sec である．サブルーチンごとのメモリバンド幅を比較すると，cellfacevariables と mfgs ルーチンにおいて，十分な性能が得られていないことがわかる．mfgs は陰解法であり，他のサブルーチンと比較して並列性が少ないために，十分なメモリバンド幅が得られなかったと考えられる．cellfacevariables の律速原因はいくつか考えられ，ひとつは，保持しておくべき変数が多く，レジスタ数が足りていないために，レジスタスピルを起こしていること．二つ目は，多量の配列からデータを読み込むために，キャッシュミスが非常に多くなっていること．さらに，扱うデータ構造が適切でない可能性などが考えられる．

また，各ソルバ全体のメモリバンド幅を計測した結果を表 4 の右列に示した．各サブルーチン同様，(入出力メモリ総量) / (実行時間) から求めているが，サブルーチンのメモリバンド幅と比較して，遥かに効率が悪いことがわかる．これは，各サブルーチン間のデータのやり取りをグローバルメモリを介して行っていること，同一メモリを複数回読み込んでいることが原因と考えられる．解決方法として，それぞれのサブルーチンのループ融合が考えられる．これにより，先の無駄なメモリアクセスを削減することが出来，大幅な性能向上が見込める．例えば，対流項全てのサブルーチンを融合した場合，入出力メモリ量は 1/7 程度まで押さえることが出来，理論上 7 倍程度の高速化が見込める．しかしその場合，配列へのアクセスパターンも変化するために，適切なデータ構造が変化する可能性があると考えられる．

### 4.3 CPU と GPU における最適なデータ構造の違い

UPACS のデータ構造に関してであるが，先行研究 [11] では，配列の最外次元が変化するようなデータアクセスを行うルーチンに対し，配列の軸を最内次元に入れ替えるデータ構造の変更 (**図 7**) が，現在主流であるスカラーマシン向けのチューニングとして有効であることが示されている．しかし，GPU ではこのような変更が効果的ではない場合がある．

muscl，cellfacevariable，mfgs，update ルーチンは，最外次元が変化するようなデータアクセスが多用されるルーチンであるが，muscl, update ルーチンにおいて図 7 のようなデータ構造変更を行ったところ，それぞれ 22GB/s, 25GB/s 程度まで性能が低下した．また，muscl はさらに 2 つのルーチンに分割できるが，一方では著しく性能が低下したものの，もう一方では 5GB/s 程度の性能向上を示した．このように，サブルーチンごとに最適なデータ構造が異なる場合があると考えられる．

本研究で用いている UPACS ver2.0 は，先行研究 [11] における最適化が適用される以前のものであるため，データ構造の変更を必要としなかったが，現在主流のスカラープロセッサ向けのアプリケーションでは，上記のような配列を用いたものが多いと考えられる．スカラープロセッサ向けのアプリケーションの GPU 環境への移植を考えたとき，上記のようなデータ構造の変更が必要とされるが，データ構造の変更はプログラム全体に影響を及ぼすため容易ではなく，アーキテクチャ間での移植性が問題になると考えられる．

```
do k = 1, kmax
   do j = 1, jmax
      do i = 1, imax
         . . .
        nv(:) = fNormal(i,j,k,1,:) → fNormal(:i,j,k,1)
        q(:)  = q(i,j,k,:)         → q(:,i,j,k)
        dq(:) = dq(i,j,k,:)        → dq(:,i,j,k)
         . . .
      end do
   end do
end do
```

**図 7** 配列の軸入れ替えによる，スカラープロセッサ向けチューニング

**表 5** OpenACC コンパイラバージョン

| | C コンパイラ | Fortran コンパイラ | バージョン |
|---|---|---|---|
| Cray | CC | ftn | 8.1.0.143 |
| PGI | pgcc | pgfortran | 12.4-0 |
| CAPS | hmpp | hmpp | 3.1.0 |
| (host) | icc | ifort | 11.1 |

## 5. OpenACC の予備評価

手動での CUDA 実装との比較のために，OpenACC の予備評価を行った．東京工業大学では現在，Cray，PGI，CPAS の 3 社全ての OpenACC コンパイラが利用できる．これら 3 社の OpenACC コンパイラを用いて，予備実験として，行列積プログラム，微分方程式を解く際に用いられる diffusion プログラムに対して，手動の CUDA 実装と比較して，どの程度の性能が得られるかを検証した．実験に用いた各社のコンパイラバージョンを**表 5** に示す．

### 5.1 行列積，diffusion プログラムへの OpenACC 適用

行列積プログラム，diffusion プログラムに対する，各 3 社のコンパイラを用いた OpenACC 実装と，最適化されていない CUDA プログラム，手動で最適化を加えた CUDA プログラムを用意し，演算性能，メモリバンド幅を比較した．なお，CUDA Fortran は PGI コンパイラ，CUDA C は nvcc コンパイラを用いている．ただし，Cray のみ実行環境が異なり，使用している GPU は X2090 である．X2090 は演算，メモリバンド幅の理論性能値がそれぞれ 665GFlops(double)，178GB/s であり，M2050 の 515GFlops(double)，148GB/s と比較して高い性能を誇る．

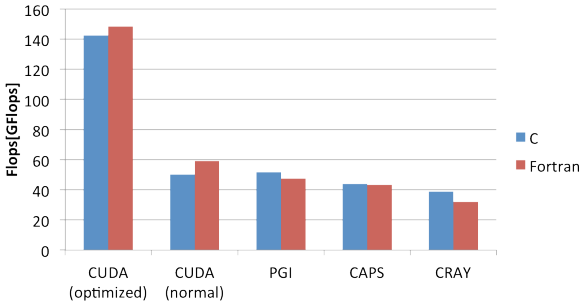


**図 8** 行列積 2048×2048

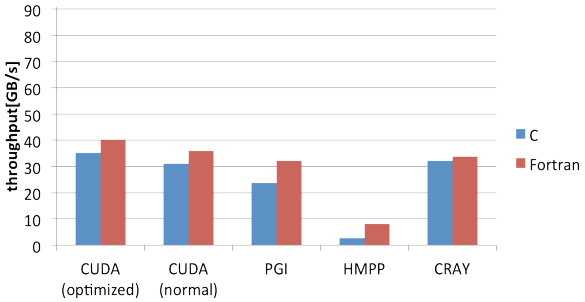


**図 9** diffusion 64×64×64 (single)

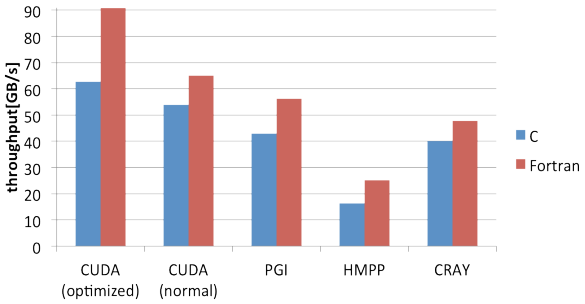


**図 10** diffusion 128×128×128 (single)

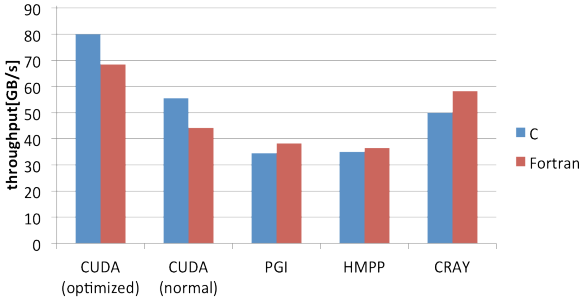


**図 11** diffusion 256×256×256 (single)

行列積プログラムにおいて，2048 × 2048 の行列積を計算したそれぞれの性能を**図 8** に示す．

CUDA の最適化済みプログラムは，ループアンローリング，共有メモリを用いたブロック化を行っている．**図 12** に示したプログラムは，PGI 版の行列積ルーチンである．まず，行列積サブルーチンの呼び出し元において，data ディレクティブを用いてあらかじめ GPU 側へデータ転送を行う．kernels ディレクティブを用いて並列実行領域を指定し，present によってデータが既にデバイス上にある

```
void matmul(
  double * restrict a,
  double * restrict b,
  double * restrict c,
  const int n){
  int i,j,k;
#pragma acc kernels present(a[0:n*n],b[0:n*n], c[0:n*n])
#pragma acc loop private(c[0:n*n]) gang vector(4)
  for(i=0; i<n; i++){
#pragma acc loop gang vector(64)
    for(j=0; j<n; j++){
      double Cij = 0;
      for(k=0; k<n; k++){
        Cij += a[POS(i,k,n)] * b[POS(k,j,n)];
      }
      c[POS(i,j,n)] = Cij;
    }
  }
}

int main(int argc, char *argv[]){
   /*中略*/
#pragma acc data copyin(a[0:n*n], b[0:n*n]) copy(c[0:n*n])
  {
    matmul(a, b, c, n);
  }
   /*中略*/
}
```

**図 12** C 言語における，行列積プログラムへの OpenACC 適用 (PGI 版)

ことを指示，さらに loop ディレクティブを用い，配列 C がループ間で依存しないことを指示，vector によってスレッドブロックサイズを (64, 4) にすることを指示し，gang によってそれらを SM に割り当てている．”PGI 版”と書いたのは，図 12 は PGI コンパイラ以外ではコンパイルできないためである．各社によって作られた OpenACC コンパイラは，現状では未完成であるために，いくつかの機能が未実装であったり，仕様の解釈が異なるためである．そのため，現状の結果は全く同じ実装による性能を示すものではない．Cray の実装では，図 12 における kernels ディレクティブの代わりに parallel ディレクティブを用いた．private なしで並列化できたため，private は用いておらず，vector は無視されるため，スレッドブロックサイズの設定は行っていない．CAPS の実装では，gang，worker，vector を用いて設定した場合より，デフォルト設定の方が高速であったため，ループディレクティブを#pragma acc loop independent とした．independent により，private を用いずに，ループが並列化可能であることを指示している．

次に，diffusion プログラムに対して OpenACC を適用した結果を**図 9** **図 10** **図 11** に示す．diffusion は 7 点のステンシル計算であり，メモリ読み込みに律速されるため，グラフの縦軸はメモリバンド幅を示している．なお，実験は単精度で行っている．**図 13** に示したのは，CAPS 版の diffusion プログラムである．元の diffusion プログラムで

はzのループが最外ループであるが，hmppコンパイラでは，x, y軸ループを2次元スレッドにマッピングしたものをz回呼び出すという実装になってしまい，効率が悪かったため，CAPS実装においてのみ，やむなくz軸ループを最内ループとした．またhmppコンパイラは，並列実行部に到達する度に，スカラー値をデバイスメモリ上にallocate, upload, download, freeを繰り返すため，createにより領域確保，updateによりメモリへのコピー，さらにC言語版ではスカラー変数にconstを付けることで対応した．しかし，プロファイラにより計測したところ，未だに必要回数以上のメモリコピーをしているようであり，そのために他の実装と比較して性能が得られていないと考えられる．Cray版では，data presentディレクティブ，parallelディレクティブと，各ループにloopディレクティブを与えることで実装した．PGI版では，data presentディレクティブ，kernelsディレクティブと，z軸ループに!$acc loop seq，y軸ループに!$acc loop gang vector(4)，x軸ループに!$acc loop gang vector(64)を与えることで実装した．

行列積プログラムでは，サイズの変更による各コンパイラの性能比に大きな差は見られなかったが，diffusionでは大きな差が見られた．サイズ$256^3$におけるPGIの性能低下は，L1キャッシュによる影響と考えられる．CUDA実装においては，キャッシュサイズをデフォルトの16KBから48KBに変更しているが，OpenACCにおいては，明示的なキャッシュサイズ変更をしておらず，また，キャッシュサイズ変更用の関数も用意されていないため，デフォルトの16KB設定で動作していると考えられる．CAPS, Crayにおいては，サイズ$128^3$と比較して，性能向上が見られる．

また，コンパイラによって得意とする配列の形に違いが見られた．Fortran版の実装において，CAPSの場合には配列インデックスを1次元的に扱っている．この場合，independentを与えることが必須になるが，3次元配列として扱う場合よりも最大25%程度高速であった．また，Crayの実装においても配列インデックスを1次元的に扱っている．Crayはこの場合でもindependentなしで並列化でき，3次元配列として扱うよりも最大30%程度高速であった．それに対し，PGIの場合には，3次元配列として扱った場合の方が最大で30%程度高速であった．グラフは，それぞれ性能の良かった実装のものである．

### 5.2 UPACSへのOpenACCの適用

また，OpenACCを用いた評価を対流項ソルバの各主要ルーチンに対して行った．対流項の各サブルーチンを抜き出し，それぞれに対してPGIのOpenACCコンパイラを用いてOpenACCの適用を行い，手動によるCUDA実装と性能を比較した．比較結果を**表 6**に示す．updateルーチンでは7割程度の性能が得られているが，muscl ルーチンでは2割強程度の性能しか得られていない．**図 14**はupdateルーチンに対してのOpenACCの適用方法である．PGIのOpenACCコンパイラの場合，図4のように，配列インデックスにi, j, kに依存するような変数を用いた場合，ループに依存関係があると判断され，並列化できなかったために，i方向差分，j方向差分，k方向差分それぞれのサブルーチンに分解している．構造体の使用が許されていないため，図3の変更を行うことで，構造体を使わない形に書き換えている．このように，ディレクティブの挿入以外の書き換えが必要になってしまったため，表6に示した様に，書き換え行数が増加してしまっている．

```
!$acc data present(f1,f2) &
!$acc create(nx, ny, nz, ce, cw, cn, cs, ct, cb, cc, dt)
!$acc update &
!$acc device(nx, ny, nz, ce, cw, cn, cs, ct, cb, cc, dt)
do while(time + 0.5*dt < 0.1)
!$acc kernels
!$acc loop independent
    do y = 0,ny-1
!$acc loop independent
       do x = 0,nx-1
          do z = 0,nz-1
             ! 略 境界処理
             f2(c) = cc * f1(c) + cw * f1(w) + ce * f1(e)
             + cs * f1(s) + cn * f1(n) + cb * f1(b) + ct * f1(t)
          end do
       end do
    end do
!$acc end kernels
    ! 略
!$acc kernels present(f1,f2)
    ! f1，f2 を入れ替え，上記と同様の処理を行う．
!$acc end kernels
end do
```

**図 13** Fortranにおける，diffusionプログラムへのOpenACC適用 (CAPS版)

プログラムがFortranである場合，allocatable，pointer属性のついた配列のメンバを受け取れないという制約がある．サブルーチンなどでの引き渡しも許されていないため，UPACSに適用するためには，サイズの明示された配列にいったんコピーするか，サブルーチンを全てインライン展開する必要がある．しかし，プログラムが大規模であれば，後者の方法はあまり現実的ではないと思われる．

OpenACCの利点のひとつに，CPU，GPU環境間での移植性があげられるが，4節で書いた通り，CPUとGPUで得意とするデータ構造が異なる場合がある．このような問題をOpenACCでは解決できないため，解決策が必要になると考える．

## 6. 関連研究

大規模アプリケーションの性能最適化の研究として，南らの研究[10]があげられる．南らの研究では，流体アプリ

```
!$acc kernels
    do n=1,5
      do k=1,kn
!$acc loop gang vector(4)
        do j=1,jn
!$acc loop gang vector(16)
          do i=1,in
            dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
                * ( flux(n,i,j,k) - flux(n,i-1,j,k) )
          end do
        end do
      end do
    end do
!$acc end kernels
```

**図 14** update ルーチン i 方向の差分計算への OpenACC 適用

**表 6** 対流項サブルーチンのメモリバンド幅と OpenACC 適用に伴う変更行数 ( () 内は変更行数のうち挿入した指示文の組数)

| サブルーチン | バンド幅 (CUDA) | (OpenACC) | 変更行数 全体 (指示文) |
|---|---|---|---|
| muscl | 41.8 GB/s | 10.5GB/s | 157(13) |
| flux | 46.8 GB/s | 27.3GB/s | 48(4) |
| update | 47.9 GB/s | 33.7GB/s | 44(11) |

ケーションでよく用いられ，メモリ律速となる疎行列ベクトル積に焦点を当て，プログラムの要求する Byte/Flops の値から，実行するマシン上で期待される性能値を予測するモデルを作り，実際にアプリケーションのチューニングを行い，モデルに近い性能が得られることを示している．しかし，プログラムの移植性などについては議論されていない．また，GPU によるステンシル計算の最適化研究として，[4] があげられる．3.5D-blocking algorithm を提唱し，バンド幅要求を減らすことで計算速度をあげているが，実際に使われるようなアプリケーションへの適用は行っていない．GPU でのアプリケーション最適化の研究として，Ryoo らの研究 [6] があげられる．Ryoo らの研究では，様々なアプリケーションを GPU 環境に移植し，性能評価を行い，様々な性能の律速原因について述べている．その中でデータ構造について言及しているが，具体的な解決策の提案などは行っていない．

## 7. おわりに

大規模流体アプリケーションの GPU による高速化手法を評価するために，一例として UPACS の手動による CUDA 化を行い，既知の最適化を段階的に適用した．その結果，1CPU コアの 8.5 倍，6CPU コアの 3.5 倍程度の性能向上が得られることを確認し，各最適化に伴うコストを評価した．さらに，ディレクティブベースプログラミング言語である OpenACC を用いた予備評価を行った．性能上の課題として，CPU と GPU それぞれの実装において，最適なデータ構造が違う場合があることを確認した．しかし，一般的に大規模なアプリケーションにおいては，プログラムは機能ごとにカプセル化されているが，データ構造は共有されているために，データ構造の書き換えは困難を伴う．このような問題を解決するためには，現状の OpenACC などでは十分でなく，アーキテクチャごとに最適なデータ構造を選択し，プログラム移植性をサポートする，データ構造を関心ごととした新しい仕組みが必要であると考える．今後の課題として，まずはデータ構造の変更が，プログラムの性能にどれだけの影響を与えるのか，定量的に評価する必要がある．その上で，OpenACC のように指示文を用いるなどの方法で，プログラムの移植性を保ちつつ，最適なデータ構造を選択できる仕組みを検討する．

**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] Bernaschi, M., Bisson, M., Endo, T., Matsuoka, S. and Fatica, M.: Petaflop biofluidics simulations on a two million-core system, *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1 –12 (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] 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] Nguyen, A., Satish, N., Chhugani, J., Kim, C. and Dubey, P.: 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and 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. 1–13 (online), DOI: http://dx.doi.org/10.1109/SC.2010.2 (2010).

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

[6] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B. and Hwu, W.-m. W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, New York, NY, USA, ACM, pp. 73–82 (online), DOI: http://doi.acm.org/10.1145/1345206.1345220 (2008).

[7] Shimokawabe, T., Aoki, T., Ishida, J., Kawano, K. and Muroi, C.: 145 TFlops Performance on 3990 GPUs of TSUBAME 2.0 Supercomputer for an Operational Weather Prediction, *Procedia Computer Science*, Vol. 4, No. 0, pp. 1535 – 1544 (online), available from

⟨http://www.sciencedirect.com/science/article/pii/S1877050911002249⟩ (2011).

[8] 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).

[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] 南　一生，井上俊介，堤　重信，前田拓人，長谷川幸弘，黒田明義，寺井優晃，横川三津夫：「京」コンピュータにおける疎行列とベクトル積の性能チューニングと性能評価，Vol. 2012, pp. 23–31 (2012).

[11] 高木亮治：三次元圧縮性流体解析プログラム UPACS の性能評価，（オンライン），入手先 ⟨http://www.ssken.gr.jp/MAINSITE/download/wg_report/smpt/2.2_takaki.pdf⟩ (2006).