Introduction To GPU. And Its Application

特集 | GPUとその応用

3

OpenACCプログラミング

丸山直也[†], 星野哲也^{††}

キーワード アクセラレータ, OpenACC, GPU

1 ま え が き

C言語等で記述された既存CPU向けプログラムをGPU上で動作させるためにはコードの変更, 移植作業が必要であり, 現在のところCUDAやOpenCLのような専用言語による明示的なGPU向けプログラミングが主流である¹⁾²⁾.

CUDAはNVIDIAにより設計, 開発が進められているC/C++の拡張であり, 特にNVIDIA GPUアーキテクチャに最適化された設計となっている. またCUDAのFortran版もPGI社より提供されている. デバッガやプロファイラといった周辺ツールも整備されており, 比較的成熟したプログラミング環境と言える.

しかしながら, CUDAやOpenCLのような明示的なプログラミングを必要とするプログラミングモデルはその低レベルなプログラミングモデルのために既存プログラムの移植コストが大きい. 既存プログラムの移植のためには, ①GPU上で実行する箇所の特定, ②特定した箇所のGPUコードの記述, ③CPU・GPU分散メモリー空間の管理, の3点が主に必要である. ①および②は通常GPUアプリケーションではプログラム全体をGPU上で実行するわけではなく, CPUからGPU実行に適した箇所をオフロードする形態をとる場合が多いためである. またこのようなオフロードアプローチではオフロード箇所が必要とするデータがGPUメモリーに存在する必要がある. これは現在の主なGPUアクセラレータではCPU側メモリーとGPU側メモリーが物理的および論理的に分離された構成となっているためである. 現在においても一部のGPUではCPUとメモリーシステムを部分的に統合したアーキテクチャとなっており, 今後そのような統合型メモリーアーキテクチャの採用が進む可能性があるが, 現在の主流アクセラレータではCPU・GPUメモリー空間の管理はプログラマにまかされている.

上述のGPU移植コストの問題を解決するために種々の研究開発が現在行われており, 特に指示文によるアクセラ

レータへのオフロード型プログラミングモデルが実用段階へと進められている. OpenACCはNVIDIA, PGI, Cray, CAPSの4社により定められた指示文の仕様であり³⁾, 対応コンパイラがPGI, Cray, CAPSの各社より提供されている. すでに同様な取組みとしてPGI社によるPGI Accelerator CompilerやCAPS社によるHMPPなどが製品として提供されているが, OpenACCはベンダー間で統一された仕様である点が異なる.

OpenACCにおいても上記のCUDA等による明示的なプログラミングと同様にGPU上にオフロードする箇所の特定はプログラマにまかされている. 一方で特定した箇所のGPU向けコードの記述およびデータ管理については既存CPUプログラムに対する指示文の追加のみで実現可能であり, 既存プログラムの再利用性およびメンテナンス性の大幅な向上が期待される. CUDAプログラミングにおける③のデータ管理では, 例えば個々の配列データについてCPUメモリー上データとGPUメモリー上データをそれぞれ明示的な宣言および一貫性の管理をする必要があるが, OpenACCではCPUメモリー上データをそのままオフロード箇所で行うことが許されており, 一貫性の管理はコンパイラにまかせることも可能である. データ管理は特にプログラムの煩雑さを増すものであり, バグの原因となる場合が多いが, このようなコンパイラによる自動処理により生産性の向上が期待される.

本稿ではOpenACCバージョン1.0の仕様にに基づきその指示文について概要を説明し, 実際にOpenACCによりGPUプログラムを記述する方法を紹介する. また, 実際のOpenACCコンパイラを用いた予備的な性能評価結果を示し, 最後に今後の展望を議論してまとめる.

2 OpenACCプログラミング概要

OpenACCはOpenMPと同様に主にループを並列実行するための指示文を定義した仕様である. 指示文のシンタックスを図1に示す.

指示文は大きく分けて計算領域指示文とデータ領域指示文に分類される(表1). 前者ではアクセラレータにオフロードする計算領域を指定するものであり, 後者は同様にア

[†] 独立行政法人理化学研究所 計算科学研究機構

^{††} 東京工業大学 大学院情報理工学研究科

"OpenACC Programming" by Naoya Maruyama (RIKEN Advanced Institute of Computational Science, Kobe) and Tetsuya Hoshino (Graduate School of Information Science and Engineering, Tokyo)

表2 parallel指示文のオプション

if	async	num_gangs	num_workers	vector_length	reduction
copy	copyin	copyout	create	present	present_or_copy
present_or_copyin	present_or_copyout	present_or_create	deviceptr	private	firstprivate

```

C言語
#pragma acc 指示文 [オプション]
{
    // C言語コード
}

Fortran言語
!$acc 指示文 [オプション]
...! Fortranコード
!$acc end 指示文

```

図1 OpenACC指示文シンタックス

表1 OpenACC v1.0主な指示文一覧

parallel		指定領域を一括オフロード
kernels	計算領域指示文	指定領域を複数のカーネルとしてオフロード
loop		指定ループをオフロード
data	データ領域指示文	指定領域においてアクセラレータヘデータ転送を指定

アクセラレータに転送するデータを指定するものである。以下それぞれについてOpenACC仕様バージョン1.0に基づき説明する。

2.1 計算領域指示文

アクセラレータへオフロードする計算領域指示文はOpenACCではparallelとkernelsが用意されている。parallel指示文は指定された領域を一括してオフロードする指示文であり、OpenACCにおいて最も基本的な指示文である。OpenACCコンパイラは指定された領域を並列実行するコードを生成する。OpenACCにおける並列化はスレッド並列が基本であり、単一スレッドをワーカーと呼ぶ。複数スレッドのグループをギャングと呼び、CUDAやOpenCLと同様に多数スレッドの階層的な制御が可能である。スレッド数等の制御は指示文に対するオプションとして指定可能である。

parallel領域においてアクセスされるデータはアクセラレータ上に存在する必要がある。一般的にアクセラレータはホストメモリーとは分離されたメモリーを持つため、parallel領域開始時にホストメモリーから必要データを転送し、終了時にホストメモリーへ転送する必要がある。上述の通り、CUDAではプログラマが明示的にデータ管理を行う必要があるが、OpenACCではコンパイラにまかせることも可能である。OpenACCの仕様では配列データについては明示的な指定がない場合の標準動作としてアクセラレータに自動転送することが定められており、以下に説明するpresent_or_copyが指定された場合と等価である。スカラ

ー値についてはギャング毎に個別の変数としてアクセラレータ上に作成される。

parallel指示文ではアクセラレータ上での計算をより詳細に制御するためのオプションが用意されている。表2にオプション一覧を示す。ifオプションはオフロードを条件が真の場合のみ行うためのオプションである。例えば計算対象問題サイズが一定サイズ以上の場合にのみオフロードする等の指定が可能である。オフロードにはPCIデータ転送のオーバーヘッドがかかるためより計算コストの高い場合の方がアクセラレータによる高速化を達成しやすいが、ifオプションを用いることで高速化が見込まれる場合のみオフロードさせることが可能である。asyncオプションはオフロード領域の実行終了を待たずにホスト側において次の処理に進むことを指定する。num_gangsは並列スレッドのグループであるギャングの数を指定し、num_workersは単一ギャング内のワーカー数（スレッド数）を指定する。vector_lengthはベクトル長を指定するものであり、以下に紹介するloop指示文においてvector並列を指定した場合およびOpenACCコンパイラによって自動ベクトル化される場合に使われる値である。reductionはparallel領域内における縮約操作を指定するものであり、加算、乗算等の2項演算子およびスカラー変数を指定する。copyオプションは指定された変数を領域開始時にホストメモリーからアクセラレータメモリーへ転送し、終了時にホストメモリーに戻すことを指定する。同様に、copyinおよびcopyoutでは前者は開始時のみ、後者は終了時にのみそれぞれホスト・アクセラレータ間転送を行う。createは指定された変数をアクセラレータ上に作成することを指定するものであり、作成された変数はホストメモリーとは別個のものとして扱われ、データ転送は行われない。presentオプションは指定された変数がすでにアクセラレータ上に存在することを指示するものであり、以下に説明するデータ領域指示文とともに使われる。presentオプションが指定された変数はすでにアクセラレータ上に存在するデータを使うためホスト・アクセラレータ間転送は発生しない。presentオプションとの組合せであるpresent_or_copyやpresent_or_copyin等はアクセラレータ上にデータが存在する場合はpresentオプションが指定されたものと解釈し、存在しない場合はcopyやcopyinとして解釈される。deviceptrは指定された変数はアクセラレータ上変数であることを指定するものであり、ホストメモリーからのデータ転送は行われない。privateオプションおよびfirstprivateオプションは指定された変数が各ギャングに独立に用意され、後者では領域開始

表3 loop指示文のオプション

collapse	gang	worker	vector
seq	independent	private	reduction

時のホストメモリー上の値がアクセラレータ上変数にコピーされる。

kernels指示文はparallel指示文と同様に指定された領域をアクセラレータ上で計算することを指示するものである。parallel指示文との違いはkernels指示文では複数のカーネルへと分割されてオフロードされることである。典型的には複数のループをオフロードする場合に用いられ、並列実行領域をまとめてkernels領域として指定することで各ループについてコンパイラが個別にオフロードされるカーネルコードを生成し並列化を行う。オフロード方式のオプションとしてはparallel指示文とほぼ同様のものが利用可能である。

loop指示文はparallel指示文もしくはkernels指示文で指定された領域内で用いるものであり、ループの並列化の詳細な制御を行うための指示文である。loop指示文を利用可能な箇所はparallel領域もしくはkernels領域内のループに対してのみであり、オプションとして表3に示すものが利用可能である。collapseオプションは完全ネストされた多重ループに対して有効であり、多重ループをまとめて並列化することを指定する。gangオプションはループをギャング間で並列処理することを指定し、同様にworkerオプションはループを単一ギャング内の複数ワーカー間で並列処理することを指定する。seqオプションはループを逐次実行することを指定し、vectorオプションはループをベクトル並列やSIMD並列で実行することを指定するものである。independentオプションはOpenACCコンパイラにループに依存関係が存在せず並列化可能であることを指定するものである。並列化可否の判断はコンパイラの解析によって自動的に行われるが、コンパイラは一般的に保守的な仮定のもとで解析するため依存関係が存在すると誤って判断する場合があります。independentオプションによりコンパイラの解析を無視させることもできる。

2.2 データ領域指示文

data指示文はアクセラレータメモリー上へのデータのオフロード領域を指定するものである。data指示文で指定された領域では指定された配列やスカラー値がアクセラレータメモリーに存在することが保証される。典型的にはdata領域内に複数のparallel領域やkernels領域を持たせ、各計算領域の開始、終了毎にホスト・アクセラレータ間の転送を行うのではなく、複数の計算領域にまたがってアクセラレータにデータをオフロードするために用いる。

data指示文のオプション一覧を表4に示す。それぞれの意味は上述のparallel指示文の場合と同様である。

表4 data指示文のオプション

if	copy	copyin	copyout
create	present	present_or_copy	present_copyin
present_or_copyout	present_create	deviceptr	

3 性能事例

OpenACCによるGPUプログラムの性能を評価するために、行列積および7点ステンシルをCUDAおよびOpenACCによる実装を行いその性能を比較する。言語による違いが存在するか確認するためにC言語での実装およびFortran言語での実装の両バージョンを作成する。OpenACCコンパイラとして、Crayコンパイラバージョン8.1.0.143, PGIコンパイラバージョン12.4, CAPS社HMPPコンパイラバージョン3.1.0を用いる。HMPPコンパイラではホストCPU向けコンパイラを共用するが、Intelコンパイラバージョン11.1を用いる。CUDA環境としてはCUDA Cバージョン4.1, およびPGIコンパイラバージョン12.4付属のCUDA Fortranを用いる。それぞれのコンパイルオプションを表5に示す。

評価アクセラレータとしては東京工業大学TSUBAME2.0スーパーコンピュータのNVIDIA Tesla M2050を用いる。ただし、CrayコンパイラについてはCray XK6のTesla X2090を用いる。

3.1 行列積

倍精度行列積の性能を2048²の正方向行列を用いて評価する。図2～図4に行列積カーネルのOpenACC用コードを示す。本評価では各コンパイラ向けに異なるバージョンを作成した。これはコンパイラの動作の違いにより、一部コンパイラではindependentオプションにより明示的に依存関係が存在しないことを指示する必要があることや最高性能を達成する指示文オプションが異なったためである。図5にこれらのカーネルの呼び出し元となる箇所を示す。呼び出し元においてデータ領域を指定し、matmul関数内部において行列積ループをGPUにオフロードする。ループの並列化のためにはPGIおよびCAPSではindependentオプションが必要であったが、Crayコンパイラは依存関係が存在しないことを自動判別可能であった。CUDA版は各スレッドが内積を計算し、i, jループを並列化した。

図6に各コンパイラを用いた場合の性能を示す。ホスト・GPU間のPCIデータ転送時間は計測に含まない。グラフよりCUDA版が最速であったが、PGI版は1割程度の性能

表5 data指示文のオプション

	コンパイルオプション	
Cray	-O3 -h pragma=acc	
PGI	-fast -acc -ta=nvidia,cc20 -lpgacc	
CAPS(host)	-nvcc-options -arch,sm_20 -nvcc-options -O3	
	-O3 -restrict	
CUDA	(nvcc)-O3	(pgfortran)-Mcuda=cc20

```

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 independent gang vector(4)
    for(i=0; i<n; i++){
    #pragma acc loop independent gang vector(64)
        for(j=0; j<n; j++){
            double tmp = 0;
            for(k=0; k<n; k++){
                tmp += a[i*n+j] * b[i*n+j];
            }
            c[i*n+j] = tmp;
        }
    }
}

```

図2 行列積カーネル (PGI向け)

```

void matmul
(double *restrict a, double *restrict b, double *restrict c, const int n)
{
    int i,j,k;
    #pragma acc parallel present(a[0:n*n],b[0:n*n],c[0:n*n])
    #pragma acc loop
    for(i=0; i<n; i++){
    #pragma acc loop
        for(j=0; j<n; j++){
            double tmp = 0;
            for(k=0; k<n; k++){
                tmp += a[i*n+j] * b[i*n+j];
            }
            c[i*n+j] = tmp;
        }
    }
}

```

図3 行列積カーネル (Cray向け)

```

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 independent
    for(i=0; i<n; i++){
    #pragma acc loop independent
        for(j=0; j<n; j++){
            double tmp = 0;
            for(k=0; k<n; k++){
                tmp += a[i*n+j] * b[i*n+j];
            }
            c[i*n+j] = tmp;
        }
    }
}

```

図4 行列積カーネル (CAPS向け)

低下に抑えられることがわかった。ただし、CUDA版は最適化を一切行っていないコードであり、CUBLASのDGEMM性能は312GFLOPSであった。Cray版の評価にはX2090を用いており、理論ピーク性能がM2050の

```

int n = 2048;
double *a,*b,*c;
a = (double *)malloc(sizeof(double) * n * n);
b = (double *)malloc(sizeof(double) * n * n);
c = (double *)malloc(sizeof(double) * n * n);

#pragma acc data copyin(a[0:n*n], b[0:n*n]) copy(c[0:n*n])
{
    matmul(a, b, c, n);
}

free(a);
free(b);
free(c);

```

図5 行列積カーネル呼び出し元

図6 行列積 (2048²) 性能比較

515GFLOPSに対して665GFLOPSであるが、CrayコンパイラによるOpenACC版の性能が最も劣る結果となった。

3.2 7点ステンシル

単精度7点ステンシルカーネルの性能をCUDA版、OpenACC版でそれぞれ評価する。行列積と同様に各OpenACCコンパイラによって最適指示文構成が異なったためそれぞれ別個のバージョンを作成した(図7～図9)。典型的にはステンシル計算ではユニットストライドであるx軸およびy軸を並列化し、z軸を各スレッドが逐次処理する並列化方式が高効率となる場合が多いため、CUDA版およびOpenACC版ともにそのようにスレッドを割当てた。行列積の場合と同様にOpenACC指示文ではコンパイラ毎に指示文の使い方が異なり、さらにCAPS版ではループ順序の入れ替えが必要であった。また、通常はステンシルの入力配列と出力配列の入れ替えをCではポインタの交換により実現可能だが、ポインタ交換ではすべてのコンパイラにおいて並列化されなかったため図7～図9のとおり直接書き下すことで対処した。

図10に各バージョンにおけるメモリアクセススループットを比較したグラフを示す。行列積と同様にCUDA版が最速であったが、OpenACC版との性能差は特にFortran版において行列積と比較してより大きいものであった。ステンシル計算はメモリアクセススループットがボトルネッ

```

static void
diffusion(float *restrict f1, float *restrict f2, int nx, int ny, int nz,
          float ce, float cw, float cn, float cs, float ct,
          float cb, float cc, float dt,
          float **f_ret, float *time_ret, int *count_ret) {
    {
        float time = 0.0;
        int count = 0;
        float *restrict f1_t = f1;
        float *restrict f2_t = f2;
#pragma acc data present(f1_t[0:nx*ny*nz], f2_t[0:nx*ny*nz])
        {
            do {
#pragma acc kernels
                {
                    int x, y, z;
#pragma acc loop seq
                    for (z = 0; z < nz; z++) {
#pragma acc loop independent gang vector(4)
                    for (y = 0; y < ny; y++) {
#pragma acc loop independent gang vector(64)
                    for (x = 0; x < nx; x++) {
                        int c, w, e, n, s, b, t;
                        c = x + y * nx + z * nx * ny;
                        w = (x == 0) ? c : c - 1;
                        e = (x == nx-1) ? c : c + 1;
                        n = (y == 0) ? c : c - nx;
                        s = (y == ny-1) ? c : c + nx;
                        b = (z == 0) ? c : c - nx * ny;
                        t = (z == nz-1) ? c : c + nx * ny;
                        f2_t[c] = cc * f1_t[c] + cw * f1_t[w] + ce * f1_t[e]
                            + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
                    }
                }
                time += dt; count++;
                if(time + 0.5*dt >= 0.1)
                    break;
#pragma acc kernels
                {
                    int x, y, z;
#pragma acc loop seq
                    for (z = 0; z < nz; z++) {
#pragma acc loop independent gang vector(4)
                    for (y = 0; y < ny; y++) {
#pragma acc loop independent gang vector(64)
                    for (x = 0; x < nx; x++) {
                        /*f1_t, f2_t を入れ替えて同様の処理*/
                    }
                }
                time += dt; count++;
            } while (time + 0.5*dt < 0.1);
        }
        *f_ret = f1_t;
        *time_ret = time;
        *count_ret = count;
    }
    return;
}

```

図7 7点ステンシルコード (PGI版)

```

static void
diffusion(float *restrict f1, float *restrict f2, const int nx, const int ny,
          const int nz, const float ce, const float cw, const float cn,
          const float cs, float ct, const float cb, const float cc, const float dt,
          float **f_ret, float *time_ret, int *count_ret) {
    {
        float time = 0.0;
        int count = 0;
        float *restrict f1_t = f1;
        float *restrict f2_t = f2;
#pragma acc data present(f1_t[0:nx*ny*nz], f2_t[0:nx*ny*nz])
#pragma acc data create(nx, ny, nz, ce, cw, cn, cs, ct, cb, cc)
#pragma acc update device(nx, ny, nz, ce, cw, cn, cs, ct, cb, cc)
#pragma acc data present(f1_t[0:nx*ny*nz], f2_t[0:nx*ny*nz])
        {
            do {
#pragma acc kernels
                {
                    int x, y, z;
#pragma acc loop independent gang vector(4)
                    for (y = 0; y < ny; y++) {
#pragma acc loop independent gang vector(64)
                    for (x = 0; x < nx; x++) {
                        for (z = 0; z < nz; z++) {
                            int c, w, e, n, s, b, t;
                            c = x + y * nx + z * nx * ny;
                            w = (x == 0) ? c : c - 1;
                            e = (x == nx-1) ? c : c + 1;
                            n = (y == 0) ? c : c - nx;
                            s = (y == ny-1) ? c : c + nx;
                            b = (z == 0) ? c : c - nx * ny;
                            t = (z == nz-1) ? c : c + nx * ny;
                            f2_t[c] = cc * f1_t[c] + cw * f1_t[w] + ce * f1_t[e]
                                + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
                        }
                    }
                }
                time += dt; count++;
                if(time + 0.5*dt >= 0.1)
                    break;
#pragma acc kernels
                {
                    int x, y, z;
#pragma acc loop independent gang vector(4)
                    for (y = 0; y < ny; y++) {
#pragma acc loop independent gang vector(64)
                    for (x = 0; x < nx; x++) {
                        for (z = 0; z < nz; z++) {
                            /*f1_t, f2_t を入れ替えて同様の処理*/
                        }
                    }
                }
                time += dt; count++;
            } while (time + 0.5*dt < 0.1);
        }
        *f_ret = f1_t;
        *time_ret = time;
        *count_ret = count;
    }
    return;
}

```

図8 7点ステンシルコード (CAPS版)

クであり、OpenACCコンパイラのメモリアクセスに対する最適化が比較的未成熟と言える。

4 むすびと今後の展望

本稿では指示文によるアクセラレータ向けプログラミングAPIであるOpenACCについてそのプログラミング方法の概要を紹介し、NVIDIA GPUを用いた性能評価結果を示した。OpenACCはCUDA等の明示的なGPUプログラミングとは異なり、既存CPU向けプログラムに対して指示文の追加のみでGPU向けプログラムとすることが可能であり、

既存アプリケーションのアクセラレータへの移行コストを大きく軽減することが期待されている。本稿ではOpenACC仕様バージョン1.0にて定義されている主な指示文について概要を述べたが、本稿で紹介した指示文以外にもより詳細なオフロードの制御のための指示文やランタイムAPIが用意されており、それらを活用することで特にホスト・アクセラレータ間のデータ移動の詳細な制御が可能である。一方でparallel領域やkernels領域のアクセラレータ向けコードへの変換はコンパイラにほぼまかされており、CUDAやOpenCL等で記述した場合に比較して記述の

```

static void
diffusion(float *restrict f1, float *restrict f2, int nx, int ny, int nz,
          float ce, float cw, float cn, float cs, float ct,
          float cb, float cc, float dt,
          float **f_ret, float *time_ret, int *count_ret) {
    {
        float time = 0.0;
        int count = 0;
        float *restrict f1_t = f1;
        float *restrict f2_t = f2;
        #pragma acc data present(f1_t[0:nx*ny*nz], f2_t[0:nx*ny*nz])
        {
            do {
                #pragma acc parallel
                {
                    int x, y, z;
                    #pragma acc loop
                    for (z = 0; z < nz; z++) {
                        #pragma acc loop
                        for (y = 0; y < ny; y++) {
                            #pragma acc loop
                            for (x = 0; x < nx; x++) {
                                int c, w, e, n, s, b, t;
                                c = x + y * nx + z * nx * ny;
                                w = (x == 0) ? c : c - 1;
                                e = (x == nx - 1) ? c : c + 1;
                                n = (y == 0) ? c : c - nx;
                                s = (y == ny - 1) ? c : c + nx;
                                b = (z == 0) ? c : c - nx * ny;
                                t = (z == nz - 1) ? c : c + nx * ny;
                                f2_t[c] = cc * f1_t[c] + cw * f1_t[w] + ce * f1_t[e]
                                    + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
                            }
                        }
                    }
                    time += dt; count++;
                    if (time + 0.5 * dt >= 0.1)
                        break;
                    #pragma acc parallel
                    {
                        int x, y, z;
                        #pragma acc loop
                        for (z = 0; z < nz; z++) {
                            #pragma acc loop
                            for (y = 0; y < ny; y++) {
                                #pragma acc loop
                                for (x = 0; x < nx; x++) {
                                    /*f1_t, f2_t を入れ替えて同様の処理*/
                                }
                            }
                        }
                        time += dt; count++;
                    } while (time + 0.5 * dt < 0.1);
                }
                *f_ret = f1_t;
                *time_ret = time;
                *count_ret = count;
            }
            return;
        }
    }
}

```

図9 7点ステンシルコード(Cray版)

自由度は大幅に制限され、アーキテクチャ固有の最適化等は必ずしも表現できるとは限らない。3節で見たように比較的単純なコードではOpenACCコンパイラを用いた場合でも最適化されていないCUDAコードと10%程度の性能差に抑えられる場合もあるが、より複雑かつ大規模なアプリケーションにおける有効性を今後評価する必要がある。また、各コンパイラの違いにより並列化および最適化のための指示文の組合せが異なったが、自動チューニング等の

図10 7点ステンシル性能比較

ソフトウェア技術との組合せによる指示文組合せ自動選択等の研究開発が有効であろう。今後コンパイラの発展、成熟により性能最適化が進むことが期待される。

OpenACCは2011年にそのバージョン1.0仕様が公開され、現在においてはPGI, Cray, CAPSの各社から対応コンパイラが提供されているが、それぞれサポートしているアクセラレータはNVIDIA GPUに限定されている。OpenACCの仕様自体はNVIDIA GPUに限定されるものではないため、AMD GPUやIntel MIC等の他のアクセラレータへOpenACCプログラムをコンパイルすることも可能であり、今後より多くのアクセラレータデバイスがサポートされることが期待される。また、OpenACC仕様の策定とともにOpenMPのアクセラレータ対応が進められており、2012年冬に公開予定のOpenMPバージョン4.0ではOpenACCの一部を取り入れた仕様となる可能性が高く、今後指示文によるアクセラレータプログラミングの普及が進むと思われる。

(2012年7月20日受付)

〔文 献〕

- 1) NVIDIA Corporation, CUDA Programming Guide Version 4.1
- 2) Khronos OpenCL Working Group, the OpenCL Specification Version 1.2
- 3) The OpenACC Application Programming Interface Version 1.0 (Nov. 2011)

丸山 直也 2003年、東京工業大学大学院修士課程修了。2008年、東京工業大学大学院博士課程修了。東京工業大学研究員、助教を経て、2012年より、理化学研究所計算科学研究機構チームリーダー。高性能計算、耐故障性、プログラミング言語の研究に従事。理学博士。

星野 哲也 2012年、東京工業大学卒業。現在、同大学大学院修士課程に在学中。GPUプログラミングの研究に従事。