C-003

GPGPU 向けプログラミング環境の比較評価
A Comparison of Programming Environments for GPGPU

小森 遼太 † 渡邊 誠也 † 名古屋 彰 †
Ryota Komori Nobuya Watanabe Akira Nagoya

1. はじめに

近年，GPGPU (General-Purpose computing on GPUs) の技術が注目されている．GPGPU とは，コンピュータシステムにおいてグラフィクス処理に特化した処理装置である GPU (Graphics Processing Unit) をグラフィクス処理以外の汎用計算に応用する技術や考え方である．GPGPU のプログラミング環境として，CUDA，OpenCL，OpenACC が利用されてきたが，近年では新たに OpenMP も用いることができるようになった [1].

OpenACC や CUDA 等を比較した報告は多く存在している [2] [3]．しかし，OpenMP を用いた GPGPU の定量的な評価はまだ少なく，OpenMP と同様のディレクティブ方式のプログラミング環境である OpenACC との違いが設計者には分かりにくい現状がある．

そこで本稿では，GPGPU のプログラミング環境として OpenACC，OpenMP および CUDA の特性の違いを定量的に比較・評価する [4].

本稿の構成は以下のとおりである．まず第 2 章で研究背景として，本研究で用いる GPU を有する計算機向けのプログラミング環境について述べた後，本研究の目的を示す．次に，第 3 章で比較評価に用いるアプリケーションとその実装手法を述べ，第 4 章では評価方法と実行時間の測定結果を述べる．そして，第 5 章で比較評価と考察を行い，今後の課題を述べる．最後に第 6 章で本研究についてまとめる．

2. 研究背景

本章では，GPGPU のプログラミング環境について概要を述べた後，本研究の目的を示す．

2.1 CUDA

CUDA (Compute Unified Device Architecture) とは，NVIDIA 社が同社製の GPU 向けに提供する GPU コンピューティング向けの統合開発環境である．CUDA のプログラミングには C/C++言語をベースに拡張した CUDA C を用いて開発を行う．

2.2 OpenCL

OpenCL は，マルチコア CPU や GPU，DSP (Digital Signal Processor) などの異種混在の計算資源を利用した並列コンピューティングのためのフレームワークである．OpenCL のプログラミングには，C 言語をベースにした OpenCL C，あるいは C++言語をベースにした OpenCL C++を用いて開発を行う．

2.3 OpenACC

OpenACC とは，並列計算機環境を容易に利用するために開発された並列プログラミング環境である．ディレクティブベースのプログラミング手法であり，コードを記述する際にはアクセラレータを用いて並列処理させたい箇所にディレクティブを挿入するだけでよい．

OpenACC では 3 種の並列性の粒度が定義されており，粒度が粗いものから順に gang，worker，vector という．vector に分割されたループは GPU のひとつのスレッドに割り当てられ，gang に分割されたループはスレッドをまとめたスレッドブロックに割り当てられる．

2.4 OpenMP

OpenMP は，主に共有メモリ型計算機用のプログラムを並列化するために標準化された基盤である．OpenACC と同じくディレクティブベースのプログラミング手法を採用しており，既存のプログラムを容易に並列化することができる．従来，マルチコア CPU 等における並列処理を対象としていたが，OpenMP 4.0 より並列処理を外部アクセラレータにオフロードする機能が追加され，2017 年にアップデートされた GCC ver. 7.1.0 より，このオフロード機能がサポートされた．

2.5 本研究の目的

ディレクティブ方式である OpenACC を用いることで，CUDA や OpenCL のような詳細な記述を必要とせず容易に GPU による並列化を行うことができる．OpenACC と CUDA，OpenCL を比較した報告は多く存在しているが，OpenMP を用いた GPGPU の定量的な評価は少なく，同様のディレクティブ方式である OpenACC との違いが設計者には分かりにくい．

そこで本研究では，OpenACC と OpenMP を定量的に比較評価し，特性の違いを明らかにする．各プログラミング環境において GPU の動作をどのように指定可能であり，それが性能へどのような影響を与えるのかを，異なる特性をもつアプリケーションを用いて評価する．CPU による逐次処理と，GPGPU による処理として，OpenACC，OpenMP および CUDA の各プログラミング環境により実装されたものを比較評価し，OpenACC，OpenMP および CUDA の特性の違いを明らかにすると共に，各環境でのプラグマの挿入方法の指針を得ることを本研究の目的とする．

3. 比較評価に用いるアプリケーションとその実装手法

本章では，まず本研究の評価に用いるアプリケーションの説明を述べた後，第 2 章で示した GPGPU の各プログラミング環境での実装について述べる．その後，評価に使用する GPU 動作状況の分析手法について述べる．

†岡山大学大学院自然科学研究科, Graduate School of Natural Science and Technology, Okayama University

表 1: 各実装でのプラグマの指定

実装名	指定したプラグマ
OpenACC1	`acc parallel loop vector`
OpenACC2	`acc kernels`
OpenACC3	`acc kernels` かつ 追加オプション`-Msafeptr`
OpenACC4	`acc parallel` かつ `acc loop independent` かつ `acc loop seq`
OpenACC5	`acc kernels` かつ `acc loop independent` かつ `acc loop seq`
OpenACC6	`acc parallel` かつ `acc loop independent`
OpenMP1	`omp target teams parallel for`
OpenMP2	`omp target parallel for`
OpenMP3	`omp target teams distribute parallel for`
OpenMP4	`omp target parallel for simd`
OpenMP5	`omp target teams distribute parallel for simd`

3.1 本研究の評価に用いるアプリケーション

本研究では，並列計算によって性能の向上が見込めるアプリケーションとして，円周率計算，行列積計算およびラプラス方程式の求解プログラムの 3 種を評価対象のアプリケーションとする．円周率計算は単純な 1 重ループの並列化，行列積計算は多重ループであること，ラプラス方程式はホストとデバイス間で複数のデータ転送が行われることをそれぞれ評価観点とする．なお，これらのアプリケーションにおける数値演算はすべて倍精度の浮動小数点形式を用いる．

3.1.1 円周率計算

区分求積法による円周率計算では，(1) 式を離散化した (2) 式を用いることで，円周率の近似値を求めることができる．

$$\pi = \int_0^1 \frac{4}{1+x^2}dx \tag{1}$$

$$\simeq \frac{4}{n}\sum_{k=0}^{n-1}\frac{1}{1+(k/n)^2} \tag{2}$$

n は分割数を表し，(2) 式の一つの項の計算について並列計算を行う．

3.1.2 行列積計算

$N \times N$ 行列 $\boldsymbol{A}$，$\boldsymbol{B}$ の行列積 $\boldsymbol{C}$ を求める．$\boldsymbol{A}$，$\boldsymbol{B}$，$\boldsymbol{C}$ の i 行 j 列の要素を a_{ij}，b_{ij}，c_{ij} と置くと，c_{ij} は (3) 式で求めることができる．

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \times b_{kj} \tag{3}$$

c_{ij} の値を並列計算によって求める．

3.1.3 ラプラス方程式の求解

定常な場合における熱伝導など拡散方程式を表すラプラス方程式は，(4) 式のヤコビ反復法により収束解を求めることができる．

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4} \tag{4}$$

各座標のもつ値を並列計算によって求める．計算された値を用いて新たな更新点 A_{k+1} の値を求める．

3.2 各アプリケーションの実装

各アプリケーションで指定できる OpenACC と OpenMP のプラグマの組み合わせとして，OpenACC では合計 6 種類，OpenMP では合計 5 種類の実装を行った．それらを示す実装名と各実装でのプラグマの指定を表 1 に示す．

```
1 #pragma acc kernels
2   for (i = 0; i < INTERVALS; i++) {
3     double x = (i - 0.5) * step;
4     sum += 4.0 / (1.0 + x * x);
5   }
```

図 1: 円周率計算の OpenACC2 実装

```
1 #pragma omp target map(tofrom:sum)
2 {
3 #pragma omp teams distribute parallel for
      simd reduction(+:sum)
4   for (i = 0; i < INTERVALS; i++) {
5     double x = (i - 0.5) * step;
6     sum += 4.0 / (1.0 + x * x);
7   }
8 }
```

図 2: 円周率計算の OpenMP5 実装

3.2.1 円周率計算

円周率計算では，分割数 n を 10^9 とした．指定するプラグマの組み合わせとして，OpenACC では表 1 における OpenACC1 と OpenACC2 の 2 種類，OpenMP では OpenMP1 から OpenMP5 の 5 種類の実装を行なった．

OpenACC において `kernels` を指定すると，一般的にループに対して gang 並列と vector 並列の 2 つの粒度での並列化が行われる．しかし，円周率計算の並列化部分は 1 重ループであり，2 つの粒度での並列化は必要なく，vector 並列のみで十分であると考えられる．そこで，`parallel loop vector` により，vector 並列の適用のみ行う OpenACC1 実装と，`kernels` により自動並列化を行う OpenACC2 実装を比較する．

OpenMP においては，`target` および `parallel for` を指定しなければ GPU へオフロードが行われないことから，各実装でその二つのプラグマを記述した．また，`distribute` は `teams` がなければ指定できないため，`distribute` のみの実装は行わず，`simd` 指定しない場合として OpenMP1，OpenMP2 および OpenMP3 の 3 種類の実装を行った．`simd` の効果を調査するため，追加で OpenMP4 と OpenMP5 の実装を行なった．例として，円周率計算の OpenACC2 実装と OpenMP5 実装の並列化部分のコードを図 1 と図 2 にそれぞれ示す．

3.2.2 行列積計算

行列積計算では，対象の行列のサイズを $4,096 \times 4,096$ とした．指定するプラグマの組み合わせとして，OpenACC では表 1 における OpenACC2 から OpenACC5 の 4 種類，OpenMP では円周率計算と同じ組み合わせの 5 種類の実装を行なった．

`kernels` のみ指定した OpenACC2 実装において，3 重ループに対して vector 並列のみしか行われず，期待した性能を得られなかったため，OpenACC3 実装を行なった．`-Msafeptr` は，ポインタと配列間のデータ依存関係をオーバライドする PGI コンパイラのオプションである．GPU アクセラレータに処理をオフロードするループの中で配列にアクセスするために使われているポインタを扱う場合，`-Msafeptr` を付加してコンパイルする必要がある [5]．また，`loop independent` と `loop seq` による指示の効果を調査するために，OpenACC4 と OpenACC5 実装を行なった．

3.2.3 ラプラス方程式の求解

ラプラス方程式の求解プログラムでは，格子点のサイズを $4,096 \times 4,096$ とした．指定するプラグマの組み合わせとして，OpenACC では表 1 の OpenACC2 と OpenACC6 の 2 種類，

表 2: 評価環境

OS	Linux kernel 4.7.0 x86_64
コンパイラ	
C コンパイラ	GCC 7.3.0
OpenMP コンパイラ	GCC 7.3.0
OpenACC コンパイラ	PGI コンパイラ 18.4
CUDA	CUDA 9.0
CPU	Intel Core i7-6700 (Skylake)
コア数	4 コア (論理 8 コア)
動作周波数	3.4GHz
L2 キャッシュ	256KB/コア
L3 キャッシュ	8MB
主記憶	16GB, DDR4-2133
GPU	NVIDIA GeForce GTX 1060 (GP106 Pascal)
SM(Streaming Multiprocessor) 数	10 基
CUDA コア数	1280 コア
ベースクロック	1.5GHz
L2 キャッシュ	1.5MB
デバイスメモリ	6GB, GDDR5

OpenMP では円周率計算と同じ組み合わせの 5 種類の実装を行なった．

OpenACC では，並列化箇所が同じ場合でも性能に違いが生じるのかを調査するため，OpenACC2 の `kernels` による並列化と同じ箇所で並列化が行われるように `parallel` と `loop independent` を指定した OpenACC6 実装を行った．

3.3 GPU 動作状況の分析手法

GPU の動作状況は，NVIDIA 社が提供する NVIDIA Visual Profiler および nvprof コマンドを用いてデータ転送やメモリ使用，スレッドの状況を分析する．

4. 評価方法と実行時間の測定結果

4.1 評価環境と実行時間の測定手法

本研究で用いる評価環境を表 2 に示す．

実行時間の測定方法として，CPU 実装ではプログラム中に `gettimeofday` 関数を使用し測定を行い，CPU は論理 1 コアで動作させた．GPU 実装では，3.3 節で述べた nvprof コマンドにより出力される `duration` の値により測定を行なった．

4.2 各プログラミング環境での実行時間と GPU 動作

各実装での円周率計算の実行時間と GPU の動作状況を図 3 に，行列積計算の実行時間と GPU 動作状況を図 4 に，ラプラス方程式の求解プログラムの実行時間と GPU 動作状況を図 5 にそれぞれ示す．

実行時間グラフの右側の表は，3.3 節で述べたプログラム実行時の GPU 動作状況である．例えば OpenACC1 であれば，実行時のグリッドサイズは (1，1) であり，ブロックサイズは (256，1) であることから，$1 \times 1 = 1$ 個のスレッドブロックと，そのスレッドブロックは $256 \times 1 = 256$ 個のスレッドで並列動作したということを示している．なお，本来グリッドサイズとブロックサイズは最大で 3 次元まで指定されるが，本研究の測定ではどの実装も 3 次元目のサイズはすべて 1 となったため，簡略化して 2 次元での表記を行う．また，動作スレッド数の欄には 1 グリッドで動作した合計スレッドの数を記しており，これは，1 スレッドブロックあたりのスレッド数に 1 グリッドあたりのスレッドブロック数をかけた数である．

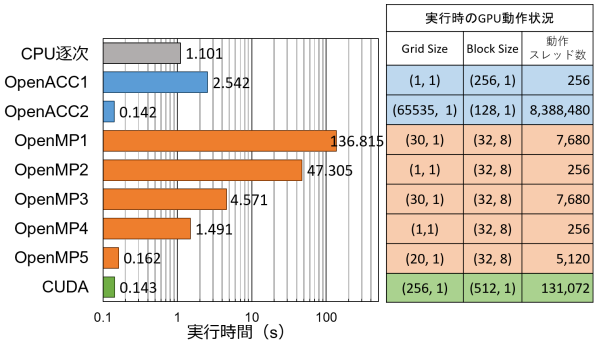

図 3: 円周率計算の各実装での実行時間と GPU 動作状況

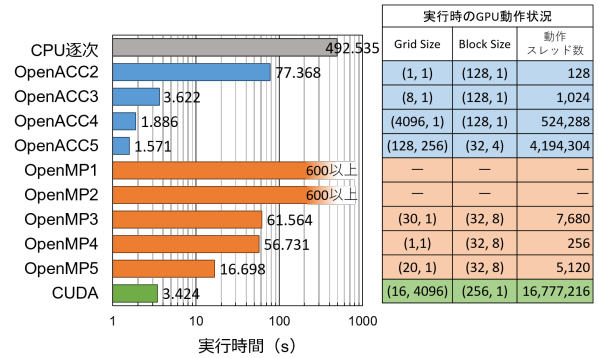

図 4: 行列積計算の各実装での実行時間と GPU 動作状況

実行時間の測定において，プログラムが 600 秒以内に終了しなかったものについては，実行を中止したため，GPU 動作状況の把握はできなかった．

5. 比較評価と考察

5.1 円周率計算について

円周率計算において，図 3 に示す GPU 動作状況のように並列化が行われた．OpenACC では `kernels` により自動並列化を指示した OpenACC2 の場合，gang 並列の適用により 65,535 個のスレッドブロックに分割され，vector 並列の適用により各スレッドブロックは 128 個のスレッドにより並列処理が行われた．

OpenMP では，`teams` によりコンパイラに自動並列化を指示した場合，30 個のスレッドブロックに分割され，各スレッドブロックは $32 \times 8 = 256$ 個のスレッドにより並列処理が行われた．CUDA では，256 個のスレッドブロックに分割され，各スレッドブロックは 512 個のスレッドにより並列処理が行われた．

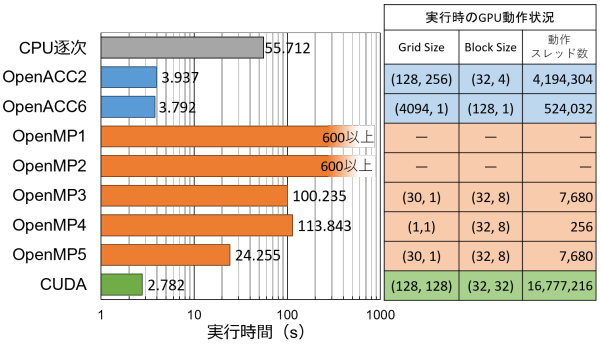

図 5: ラプラス方程式の求解プログラムの各実装での実行時間と GPU 動作状況

OpenACC では，kernels を用いることで gang 並列，vector 並列の適用を行った場合 (OpenACC2)，OpenMP では，teams distribute を指定した上で simd を指定した場合 (OpenMP5) において，それぞれ CUDA と同程度の性能を得ることができた．これらの結果から，1 重ループの単純な並列化であっても，大きな粒度である gang，team によるスレッドブロックレベルでの並列化は有効であると考えられる．

5.2 行列積計算について

OpenACC における行列積計算プログラムの OpenACC2 と OpenACC3 では並列化箇所は指定せず，OpenACC4 と OpenACC5 では，指定した箇所で並列化が行われるようプラグマを挿入した．その結果，各実装で 3 重ループの異なる箇所でそれぞれ並列化が行われた．OpenACC4 の parallel を kernels に変更した OpenACC5 では，i ループと j ループでそれぞれ gang 並列，vector 並列が適用された．OpenACC4 では i ループで gang 並列，j ループで vector 並列の適用と，各ループで並列性の粒度が異なる並列化が行われた．なお，行列積計算の 3 重ループにおいて，i ループが最外ループであり，k ループが最内ループである．これらの結果から，OpenACC において多重ループをもつアプリケーションの並列化を行う場合は，OpenACC5 のように kernels を指定した上で各ループに loop independent を用いてコンパイラに並列化可能であることを指示することで並列度が大きく向上し，性能を向上させることができることが分かった．

OpenMP では，円周率計算の場合と同様に正しいプラグマを挿入することで並列化が行われ CPU 実装と比較して実行時間は短くなるが，OpenACC4 や OpenACC5 のような実行時間短縮は達成できないことから，多重ループに対して OpenACC5 での並列化のような各ループで最適な並列化は行われていない可能性があると考えられる．また，team 数の上限を設定するプラグマである num_teams を使用し，team 数の上限を 30 以上に指定した場合でも，動作スレッドブロック数を 30 個以上にすることができなかった．これは GCC の処理系に問題があると考えられる．

5.3 ラプラス方程式の求解プログラムについて

OpenACC では，ラプラス方程式の求解プログラムにおいて，kernels を用いることで 2 重ループに対してそれぞれ gang 並列, vector 並列が適用された．行列積計算のように，並列化部分が 3 重ループの場合は kernels のみでは各ループで gang 並列，vector 並列とも適用されなかったが，並列化部分が 2 重ループである場合は，kernels によって依存性解析が正しく行われ，gang 並列により並列化されることが判明した．

OpenMP においては行列積計算の場合と同様に，CPU 実装と比較して実行時間は短くなるものの，OpenACC による並列化ほどの性能の向上はなかった．

5.4 考察のまとめ

3 種のアプリケーションの結果から，OpenACC においては kernels を用いることと，3 重ループ以上の多重ループに対しては loop independent によりコンパイラに並列化可能であることを指示することにより，デバイスの並列性を十分に活かす並列化を行うことができるといえる．parallel を指定する際は loop independent や loop seq を追加で指定し，コンパイラに細かく情報を与える必要がある．

OpenMP では，単純なループのアプリケーションに対して適切なプラグマを挿入することで OpenACC や CUDA と同程度の性能を得ることができた．しかし，多重ループのアプリケーションにおいてはループごとに細かく並列化するかどうかを指定できる OpenACC とは異なり，OpenMP では並列化箇所をコンパイラに任せることになり，その結果 OpenACC や CUDA の性能には到達できていないと考えられる．

5.5 今後の課題

GCC では OpenMP は開発段階であり，現在も様々な機能やオプションをサポートしながらバージョンアップを続けているため，本研究で用いたバージョン 7.3.0 で指定できなかった機能が今後サポートされることが考えられる．それらの評価やさらなる高速化のためのプラグマの組み合わせの調査等が必要である．

また，スレッドブロックとスレッドの数による性能の違いについての調査を行うこと，異なる特性をもつアプリケーションによる評価，Intel 社や Cray 社などによる他のコンパイラでの評価が今後の課題として挙げられる．

さらに，CUDA と同様にデータ転送やスレッドの動作を詳細に記述できる OpenCL との比較評価も今後の課題である．

6. おわりに

本研究では，GPGPU のプログラミング環境の比較のため，円周率計算，行列積計算およびラプラス方程式の求解プログラムの 3 種のアプリケーションを OpenACC，OpenMP および CUDA による実装を行った．また，それぞれのプラグマやアプリケーションによる効果について評価を行った．

参考文献

[1] GCC Wiki, Offloading Support in GCC. https://gcc.gnu.org/wiki/Offloading.

[2] 小林 直人, 渡邊 誠也, 名古屋 彰, “GPGPU プログラミング環境としての OpenCL と OpenACC の比較評価,” 第 64 回電気・情報関連学会中国支部連合大会, pp. 258–259, Oct. 2013.

[3] 理化学研究所計算科学研究機構 プログラム構成モデル研究チーム, CUDA vs OpenACC：マイクロベンチマークとアプリケーションによる OpenACC コンパイラの評価, http://atrg.jp/ja/index.php?plugin=attach&pcmd=open&file=04maruyama.pdf&refer=ATTA2012.

[4] 小森 遼太, 渡邊 誠也, 名古屋 彰, “GPU を有する計算機環境向けプログラミング環境の比較評価,” 第 44 回パルテノン研究会資料集, pp. 31–32, Dec. 2018.

[5] SofTek, PGI OpenACC コンパイル用のオプション, https://www.softek.co.jp/SPG/Pgi/TIPS/opt_accel.html.