OpenACC による GPU デバイスメモリ管理についての考察

渡邉孔英 [1]　菊池航平 [2]　柏野隆太 [1]　綱島隆太 [1]
藤田典久 [3,1]　小林諒平 [3,1]　朴泰祐 [3,1]

概要：アプリケーションの GPU 化によって高速化を図るとき，CPU メモリと GPU メモリの間のデータ移動管理が必要になる．OpenACC で記述されたプログラムを PGI コンパイラでコンパイルするとき，データ移動の管理は自動的に行わせるか，プログラマが記述するかを選択することができる．本研究では，両方の方法によるデータ移動管理とその性能について，実験を行って比較および考察した．その結果，データアクセスのパターンによっては，データ移動管理を自動的に行わせる方がデータ転送を削減でき，高速化に役立つ場合があることがわかった．

キーワード：アプリケーション GPU 化，OpenACC，PGI Compiler，CUDA Unified Memory

1. はじめに

近年の高性能計算システムにおいて，消費電力を抑えながら極めて高い性能を達成するという事が最も重要な課題の一つとして挙げられる．現在，広く用いられている Intel Xeon に代表される汎用マルチコア CPU は，プログラミングの自由度が高い反面，性能あたりの消費電力が高いという問題がある．この問題に対する解決策として，CPU と併用して演算加速装置を利用することが主流になりつつある．実際, 2020 年 11 月に発表された Top500[1]では，上位 10 機中 7 機が演算加速装置を搭載している。そして、その中で最もよく使用されているものは Graphics Processing Unit (GPU)である．

GPU は，高い演算性能とメモリバンド幅を持ち，データ並列性を特徴とするアプリケーションに対して高い性能を実現することが期待できる．また，GPU は 1 つの命令ストリームの下で多数の計算コアを同時並列に実行するという Single Instruction Multiple Threads (SIMT) 方式によって動作するため，消費電力あたりの性能が極めて高くなる．そのため，スーパーコンピュータ向けのアプリケーションの実行に適しており, 既存の汎用 CPU 向けに書かれたアプリケーションを GPU 化することで，消費電力を抑えつつ十分な性能向上が期待できる．しかしながら，GPU を利用するプログラミングは複雑であり，その実装には実行性能のみならずプログラムの生産性が求められている．

そのような要求を満たす演算加速装置向けのプログラミング言語として，OpenACC[2] が知られている．OpenACC はディレクティブと呼ばれる指示文を既存の逐次実行プログラムに挿入することで比較的簡単にコードの演算加速が可能になる．特に，GPU 向けの利用が広がっている．ディレクティブにより特定部分の演算の GPU へのオフロードが可能であるため, 既存の CPU コードを最大限維持しつつインクリメンタルに開発を進めることができるというメリットがある．

また, OpenACC で記述されたプログラムを扱えるコンパイラのうち PGI コンパイラでは， CPU と GPU の間のデータ転送を自動化し，ユーザが Unified Memory と呼ばれる CPU と GPU で共通のアドレス空間を使えるようにする機能がある．本来 OpenACC による GPU 化では，CPU と GPU の間のデータ転送はプログラムがディレクティブの形で明示し管理するが，Unified Memory 機能を使用することで，プログラマはプログラム中のデータ管理を指示する必要がなくなり，複雑なデータ構造のプログラムを GPU 化する際の生産性を向上させることができる．

Unified Memory により，ユーザ側のデータ管理が不要になる反面，実際のデータ移動のための通信の挙動が隠蔽されてしまう．そのため，ユーザの想定通りの性能が得られないような事象が起こり得る．このような場合に，Unified Memory の挙動を理解しておくことは重要になる．しかし，筆者らが調べた限り，この OpenACC の Unified Memory の扱いとそれによる性能への影響を詳細に調べた文献はほとんどない．いくつかのアプリケーションを Unified Memory で実装した場合の総合的な性能評価だけでは，これを使おうとするプログラマに対する明確な指標とならない．

本稿では, 典型的な HPC アプリケーションにおいて発生するデータアクセスに応じた Unified Memory の挙動を調査し，報告する．2 章では関連する研究とその内容について述べる．3 章では本調査の中心となる技術である OpenACC および Unified Memory の概要を説明する．4 章では，総和計算プログラムとステンシル計算の袖領域通信プログラムを用いた Unified Memory の調査実験の手法を説明する．5 章では，実際に実験を行って得られた結果を示す．6 章では，得られた結果から OpenACC における Unified Memory の性能評価とこれを扱ううえでの注意点について考察を行う．

1 筑波大学 情報理工学位プログラム
2 筑波大学 情報科学類
3 筑波大学 計算科学研究センター

ⓒ2020 Information Processing Society of Japan

2. 関連研究

NVIDIA GPU の Unified Memory の性能を評価した研究として，Landaverde らによる[3]がある．多くのアプリケーションでは Unified Memory 利用に伴うオーバヘッドがボトルネックになりうるが，アプリケーションの種類とデータサイズにより Unified Memory を用いる実装の方が高速になる事例があることを示している．

また，OpenACC における Unified Memory の利用に関する研究として，土井による[4]が挙げられる．この研究では，姫野ベンチマークおよび CCS QCD ベンチマークを利用して，PCIe 環境および NVLink 環境における Unified Memory の性能測定を行っている．そして，Unified Memory は NVLink 環境であってもレイテンシのオーバヘッドが性能に影響するが，問題サイズとアプリケーションの種類によっては Unified Memory が上回ることを示している．

OpenACC および CUDA の実装に関しての研究として，Deldon らによる[5]がある．Unified Memory の実装として，ホスト側に Unified Memory Pool 領域を用いることで，Unified Memory の性能向上を行っている．SPEC ACCEL 1.2 Benchmarks における性能評価では，通常の実装と比べて遜色ない結果が得られたと報告されている．

研究[3][4][5]ではベンチマークアプリによる性能評価および比較を行っている．そのため，Unified Memory がどのようなアクセスパターンによって性能がどう変化するかを明確にしていない．そのため，メモリアクセスパターンに対する振る舞いを調べることは意味があると考えられる．また，[4]ではデータ管理を明記した場合と Unified Memory の比較に NVIDIA の提供する高速インターコネクト NVLink 環境を使用している．NVLink 環境に対応している CPU アーキテクチャは限られるため，より多くの CPU アーキテクチャに対応している PCIe 環境下における Unified Memory の性能の調査には意味があると思われる．そこで，本研究では，PCIe 環境の下で Unified Memory を利用し HPC アプリケーションにおける典型的なメモリアクセスパターンにおける性能を調査した．

3. 前提となる事柄

3.1 OpenACC

OpenACC は，ディレクティブを用いてプログラマが GPU などの演算加速装置を簡単に利用することができるように作られた指示文ベースのプログラミング言語である．共有メモリ計算機における並列化に用いられる OpenMP と同様に，データの転送やループの並列化領域をディレクティブで指定することで，コンパイラが自動的に演算加速装置用のコードを生成する．これにより，ユーザは元の CPU 向けのコードに対して比較的少ないコードの追加，変更を行うだけでアプリケーションを移植することができる．

CUDA と比較すると，実行性能では最適化された CUDA コードには及ばないが，ディレクティブベースのプログラミングによって段階的に GPU 化を進められることや Unified Memory を活用することでデータ管理を自動化できることなど，プログラムの生産性，移植性についての利点が多い．

3.2 Unified Memory

GPU の開発における大きな負担の一つが，CPU と GPU の間のデータ転送の記述である．GPU の計算に使用するデータ構造が複雑な場合は，ユーザが複数のディレクティブを用いてデータ転送を記述する必要がある．この開発の負担を軽減する機能として，Unified Memory が存在する．

Unified Memory とは，CPU や GPU など複数デバイス上で動作するコードにおける，データの割当やアクセスの管理を簡単にするための仮想メモリの仕組みである．Unified Memory は CPU と GPU の間の通信を自動化し，統一されたアドレス空間でデータを扱うことができる．

Pascal 世代以降の GPU における Unified Memory の実装では，ホスト側である CPU またはデバイス側である GPU 側のいずれかでページフォルトが発生すると Unified Memory のランタイムライブラリが自動的にページ単位での通信を行うという仕組みになっている．

明示的なデータ転送の記述が不要になるため，ユーザ側にとってはプログラムの実装の容易さや移植性，保守性などに大きなメリットがある．しかしながら，ページフォルトのハンドリングなどに伴うオーバヘッドがプログラムに影響を与えることが考えられる．Unified Memory の利用環境として，PCIe を用いる場合と NVLink を用いる場合が考えられる．前者の環境は多数の CPU に適応可能であるが，後者は IBM Power アーキテクチャなどに限られる．その反面，後者の方が CPU-GPU 間通信性能は高速である．多くのユーザにとって利用可能であり，同時に通信オーバヘッドの影響が大きい PCIe 環境における Unified Memory の影響を調べることには意味があると考えられる．

3.3 OpenACC における data directive を用いたデータ管理

本研究では，NVIDIA 社により提供される PGI コンパイラ[5]を用いて OpenACC プログラムのコンパイルを行う．OpenACC はディレクティブを用いることで GPU への演算のオフロードを記述する．OpenACC プログラミングモデルでは大きく分けて 2 つの種類のディレクティブが存在する．1 つ目は計算に関わる kernel/parallel directive であり，２つ目はデータ移動に関わる data directive である．

kernel/parallel directive は，GPU へオフロードする計算部分のコード領域を指定する他，GPU における計算性能を引き出すための付加情報を与える等の目的で使用される．

ⓒ2020 Information Processing Society of Japan

リスト 1 OpenACC における data directive のコード例

```
1  #pragma acc data copy(a[0:bufsize])
2  {
3      #pragma acc parallel loop
4      for (int i=0;i<bufsize;i++){
5          a[i]+=i`;
6      }
7  }
```

data directive は，ホスト側である CPU と演算加速器側である GPU 間のデータ転送を明示的に記述するために存在する．CPU と GPU は物理的には異なるメモリを用いて計算を行うため，何らかの形でデータ転送を行う必要がある．data directive に続けて，特定の clause を付与することで，対象のメモリがどのように振る舞うかを指定することができる．主に次の 3 つの clause が用いられる．

- copy clause: data directive が付与されているコード領域の開始時点で CPU から GPU 側へメモリをコピーし，領域の終了時点で GPU から CPU へデータをコピーする．
- copyin clause: data directive が付与されているコード領域の開始時点で CPU から GPU 側へメモリをコピーするが，GPU 側からのコピーは行わない．
- copyout clause: data directive が付与されているコード領域の終了時点で GPU から CPU へのデータコピーを行う．開始時点では CPU 側からのコピーを行わない．

OpenACC における明示的なデータ転送は，data directive と上記に代表される clause によって記述することができる．具体的なコード例がリスト 1 である．ここでは，CPU 側の配列 a を GPU に転送して，値を代入し，再度 CPU へ書き戻すという処理を行っている．parallel directive により代入処理の GPU 化を行い，data directive によりデータ転送を明記している．data directive 領域の開始と終了でコピーを実行するため，copy clause を用いて記述している．

data directive では連続したデータの送信が基本であり，非連続なデータの転送は, OpenACC の仕様上幾つかの制約が存在するため難しい． 例えば，data directive における多次元配列の送信では，最上位の次元のみが部分指定可能であり，他の次元では全ての領域を指定しなければならない [2].

非連続なデータの転送が発生する例として，多次元配列における並列ステンシル計算を行う場合が考えられる．この計算では袖領域通信を行う必要があり，連続でない次元の袖領域を交換しようとする際は，ストライドアクセスが発生する．連続データを指定しようとナイーブな実装をすると，割り当てられた領域全体を転送することになる．そのため，実際の実装では交換する袖領域のみを GPU において一次元配列に詰めかえ，連続データに変換してから送信を行う方が，転送が必要なデータ量が減って効率が良くなると考えられる．

3.4 OpenACC における Unified Memory を用いたデータ管理

OpenACC で Unified Memory を有効にするには，PGI コンパイラに対して，-ta=tesla:managed オプションを付与してコンパイルすればよい．Unified Memory を利用することでデータ管理が自動で行われ data directive を記述する必要がなくなるが，既に data directive が記述されたコードに先のオプションを付与してコンパイルすると， data directive は無視され出力されるコードでは Unified Memory による自動的なデータ移動管理が行われる[6].

Unified Memory によるデータ移動の自動管理はページングによる実装になっており，ページフォルトの発生毎に一定サイズのデータ転送を行う．このため連続領域の配列データの転送については，data directive による転送に比べてオーバヘッドの大きい Unified Memory では性能が低くなると考えられる． 一方で data directive を用いた記述では，3.3 節で述べたように連続する範囲のデータ転送しか記述できないという転送形状の制限がある．この制限のために，たとえば上記の袖領域通信を含む多次元配列のステンシル計算のような場合に対しては，必要となったデータのみ転送するという特徴から Unified Memory による自動管理の方がデータ転送にかかる時間が少なくなるケースが起こると考えられる．

4. 実験

4.1 実験環境

実験は筑波大学計算科学研究センターが運用するスーパーコンピュータ Cygnus において行った．

Cygnus には演算加速装置として GPU および FPGA の 2 種が搭載されたノードと，GPU のみが搭載されたノードがあるが，今回は GPU のみが搭載されたノードにおいて実験を行った．実験に使用したノードの構成を表 1 に示す．すべての実験は 1 台のノード上で行った．

図 1 に Cygnus の 1 ノードの構成を示す．Cygnus の各ノードには 2 ソケットの CPU が搭載されているが，演算加速装置はすべて PCIe スイッチ経由で片方の CPU へ接続されている．

実験においては， GPU が 2 ソケットの CPU のどちらに接続されたメモリとデータ転送を行うかによって，データ転送のコストが変化する点を考慮し，GPU が接続されている方のソケットにプロセスを配置し，そのソケットに接続されたメモリとの間で転送を行った場合の測定を行った．

ⓒ2020 Information Processing Society of Japan

表 1 Cygnus の構成

CPU	Intel Xeon Gold 6126 (12 cores) x 2
CPU メモリ	DDR4-2666 192GiB (16GiB x 6 channels x 2 sockets)
GPU	NVIDIA Tesla V100 (PCIe) with 32GiB memory x 4
Interconnect	InfiniBand HDR100 x4
OS	CentOS 7.6
コンパイラ	PGI Compiler 19.10
MPI ライブラリ	OpenMPI 3.1.6
CUDA	10.2

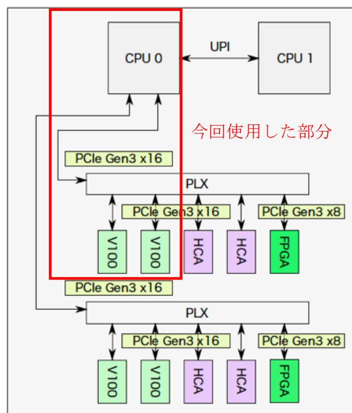

図 1 Cygnus ノード内のブロックダイアグラム

4.2 実験の説明

本研究においては，次の各項目について実験を行った．

(1) data directive が記述されたコードを managed オプション付きでコンパイルした際の挙動の調査

3.4 節で述べた data directive のあるコードを managed オプション付きでコンパイルしたときの挙動を確認するため，データ配列に対して連続領域アクセスを行うプログラムの実行時間を測定し，データ移動を手動管理した場合および自動管理した場合と比較した．

連続領域アクセスを行うプログラムの概要をリスト 2 に示す． CPU メモリにおいて初期化された 8GByte のデータ配列へ，先頭から全ての要素へ順に GPU によって定数を加算し，その後 CPU によってデータ配列の総和を計算する． このプログラムでは初期化されたデータ配列を CPU メモリから GPU メモリへ，定数を加算されたデータ配列を GPU メモリから GPU メモリへ転送する必要がある．

(2) データ移動の手動管理と自動管理の性能比較

3.4 節で述べた data directive による転送と Unified Memory によるオンデマンドの転送の性能面を比較するため，大きなデータ配列がアクセスされる要素数の割合に対するそれぞれの転送方法の性能を調べた．

リスト 2 連続領域アクセスを行うプログラムの概要

```
1  const size_t bufsize=1024*1024*1024;
2  double* a;
3  a=new double[bufsize];
4  for(int i=0;i<bufsize;i++){
5  a[i]=i%3343;
6  }
7  #pragma acc data copy(a[0:bufsize])
8  {
9  #pragma acc parallel loop
10 for(int i=0;i<bufsize;i++){
11 a[i]+=1;
12 }
13 }
14 for(int i=0;i<bufsize;i++){
15 sum+=a[i];
16 }
```

はじめに, データ配列に対して連続領域アクセスを行うプログラムで実験を行った．このプログラムは，(1)に示した連続領域アクセスプログラムが GPU によって定数を加算する要素の数を変えながら，初期化開始から総和計算完了までにかかった実行時間を測定する．

次に, データ配列に対してストライドアクセスを行うプログラムで実験を行った．ストライドアクセスを行うプログラムの概要をリスト 3 に示す．このプログラムでは，はじめに CPU メモリにおいて初期化された 8GByte のデータ配列へ，先頭から一定間隔にある要素へ順に GPU によって定数を加算し, その後 CPU によってデータ配列全体の総和を計算する．

いずれのプログラムでも， GPU による定数加算と CPU による総和計算を行う際に， CPU メモリと GPU メモリの間でデータ転送が必要となる．手動管理の場合の data directive では，copy clause を用いてデータ配列すべてを転送するように指定した．

(3) 並列ステンシル計算における袖通信の性能比較

最後に, CPU-GPU 間のデータ転送が発生するユースケースにおけるデータ転送性能の比較のため，2 次元のステンシル計算を並列に行うプログラムを作成し，その実行時間を測定した．

このプログラムでは，正方形の 2 次元領域を縦長の長方形 2 つに分割し， 両領域に 1 プロセスずつ割り当て，各プロセスが GPU を 1 つずつ使って 4 点ラプラシアン計算を行う．袖通信は MPI を用いて，CPU メモリ経由で行う．これによって必要となる，GPU メモリ-CPU メモリ間のデータ転送に注目し，その実行時間を測定した．

ⓒ2020 Information Processing Society of Japan

リスト 3　ストライドアクセスを行うプログラムの概要

```
1  const size_t bufsize=1024*1024*1024;
2  const size_t pagesize=4096;
3  double* a;
4  double asum;
5  a=new double[bufsize];
6  for(size_t
   pagenum=1;pagenum<100;pagenum+=prep){
7  for(size_t i=0;i<bufsize;i++){
8  a[i]=i%3343;
9  }
10 #pragma acc data copy(a[0:bufsize])
11 {
12 #pragma acc parallel loop
13 for(size_t
   i=0;i<bufsize;i+=(pagesize/sizeof(double))*pagenum)
   {
14 a[i]+=1;
15 }
16 }
17 double sum=0;
18 for(size_t i=0;i<bufsize;i++){
19 sum+=a[i];
20 }
21 asum+=sum;
22 }
```

縦長の長方形 2 つに分割された 2 次元領域を row-major で保持するため，袖領域は連続していない．このため，通信の前後で MPI 関数に渡す 1 次元バッファとデータ配列の間でデータを詰めかえる pack/unpack 処理を行う．

発生するデータアクセスと実行時間の関係を調べるため，データ配列を CPU-GPU 間で転送して pack/unpack 処理を CPU 側で行うプログラムと， pack/unpack 処理を GPU 側で行って通信バッファに格納するデータのみを転送するプログラムのそれぞれについて，データ転送を手動管理した場合と自動管理した場合の実行時間を測定した．リスト 4 に CPU 側で pack を行うプログラムの概要を示す．

手動管理の場合の data directive によるデータ転送の指示は，pack/unpack を CPU で行うプログラムでは copy clause を用いてデータ領域を全て転送するように指定した．pack/unpack を GPU で行うプログラムでは copyout clause を用いて送信バッファに格納するデータを転送し，copyin clause 用いて受信バッファに格納されたデータを転送することで行った．また，自動管理の場合のデータ転送にかかった時間が測れるよう，GPU 側で packing を行うプログラムでは GPU 側で packing したデータを CPU 側で一度別の配列に移し替え，MPI 関数へ渡されるデータが CPU メモリに転送されているようにした．

問題領域の各点は倍精度浮動小数点数で表され，周期境界条件で，反復回数は 20 回とした．問題領域の大きさは一辺の長さが 30720 点の正方形とした場合と，61440 点の正方形とした場合の両方について実験した．

リスト 4　2 次元ステンシル計算プログラムの概要

```
1  const size_t xsize=16*1024;
2  const size_t ysize=32*1024;
3  const size_t halo=1;
4  const size_t cbufsize=32*1024;
5  double **a,**anew;
6  double **sendbuf,**recvbuf,**sendbuf_d;
7  MPI_Request req[8];
8  int dest[4],src[4];
9  int destid[4];
10 int myrank,mydevice;
11 for(int i=0;i<ysize+(2*halo);i++){
12 for(int j=0;j<xsize+(2*halo);j++){
13 a[i][j]=(i+j)%3343;
14 }
15 }
16 #pragma acc enter data
   copyin(a[0:ysize+(2*halo)][0:xsize+(2*halo)])
   create(anew[0:ysize+(2*halo)][0:xsize+(2*halo)])
17 for(int iter=1;iter<21;iter++){
18 #pragma acc data
   present(a[0:ysize+(2*halo)][0:xsize+(2*halo)],anew[0:ysize+
   (2*halo)][0:xsize+(2*halo)])
19 {
20 #pragma acc parallel
21 {
22 #pragma acc loop
23 for(int i=halo;i<ysize+halo;i++){
24 #pragma acc loop
25 for(int j=halo;j<xsize+halo;j++){
26 anew[i][j]=(a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1])*0.25;
27 }
28 }
29 #pragma acc loop
30 for(int i=halo;i<ysize+halo;i++){
31 #pragma acc loop
32 for(int j=halo;j<xsize+halo;j++){
33 a[i][j]=anew[i][j];
34 }
35 }}}
36 #pragma acc exit data
   copyout(a[0:ysize+(2*halo)][0:xsize+(2*halo)])
37 for(int i=0;i<ysize;i++){
38 for(int h=0;h<halo;h++){
39 sendbuf[0][i+(h*ysize)]=a[i+halo][halo+h];
40 sendbuf[1][i+(h*ysize)]=a[i+halo][xsize+h];
41 }
```

```
42  }
43  for(int i=0;i<2;i++){
44  MPI_Isend(sendbuf[i],ysize*halo,MPI_DOUBLE,dest[i],desti
    d[i],MPI_COMM_WORLD,&req[i]);
45  MPI_Irecv(recvbuf[i],ysize*halo,MPI_DOUBLE,src[i],i,MPI
    _COMM_WORLD,&req[i+2]);
46  }
47  for(int i=0;i<ysize;i++){
48  for(int h=0;h<halo;h++){
49  a[i+halo][h]=recvbuf[0][i+(h*ysize)];
50  a[i+halo][halo+xsize+h]=recvbuf[1][i+(h*ysize)];
51  }
52  }
53  #pragma acc enter data
    copyin(a[0:ysize+(2*halo)][0:xsize+(2*halo)])
54  }
55  #pragma acc exit data
    copyout(a[0:ysize+(2*halo)][0:xsize+(2*halo)])
56  double sum=0;
57  for(int i=1;i<ysize+halo;i++){
58  for(int j=1;j<xsize+halo;j++){
59  sum+=a[i][j];
60  }
61  }
```

5. 実験結果

5.1 data directive が記述されたコードを managed オプション付きでコンパイルした際の挙動

データ配列に対して連続領域アクセスを行うプログラムの実行時間を測定し，data directive を記述したコードを managed オプションなしでコンパイルしたもの，data directive の無いコードを managed オプション付きでコンパイルしたもの，data directive を記述したコードを managed オプション付きでコンパイルしたものの 3 つについて比較した．図 1 にその結果を示す．HtoD copy+D computation は CPU メモリから GPU メモリへのデータ転送と GPU によるデータ加算にかかった時間を，DtoH copy+H computation は GPU メモリから CPU メモリへのデータ転送と CPU による総和計算にかかった時間を表している．

図 1 より，data directive を記述したコードを managed オプションなしでコンパイルする手動管理の場合は，data directive の無いコードを managed オプション付きでコンパイルした自動管理の場合に比べて実行時間が短くなっていることがわかる．また，data directive を記述したコードを managed オプション付きでコンパイルした場合の合計実行時間は，data directive を記述したコードを managed オプションなしでコンパイルした手動管理の場合に比べて 1.44 倍なのに対し，data directive の無いコードを managed オプション付きでコンパイルした自動管理の場合と比べると 0.99 倍と実行時間がほぼ同じになっていることがわかる．

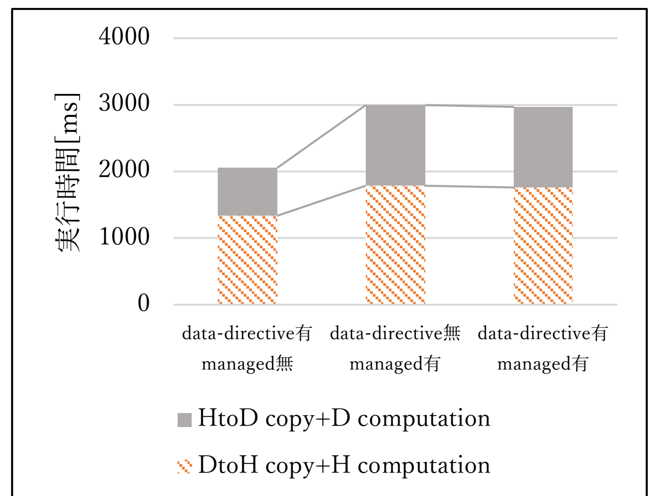

図 2 データ移動管理の方法とプログラム実行時間の関係

5.2 データ移動の性能比較

図 2 にデータ移動が手動管理の場合と自動管理の場合の連続領域アクセスプログラムの実行時間を示す．また，図 3 にデータ移動が手動管理の場合と自動管理の場合のストライドアクセスプログラムの実行時間を，ストライドが 4[KByte]の場合から 396[KByte]の場合まで示す．これらの実行時間には，CPU メモリ-GPU メモリ間のデータ転送にかかった時間のほかに，GPU による定数加算と CPU による総和計算にかかった時間が含まれる．これは，Unified Memory におけるデータ転送はデータアクセスが発生した時点で行われるため，計算と通信の時間を分離することが難しいという理由による．

図 2 より，GPU 側からアクセスした要素の割合がおよそ 60%以下のとき，データ移動を自動管理した場合の方が手動管理した場合よりも実行時間が短くなっていることがわかる． GPU による定数加算および CPU によるデータ配列の総和計算にかかる時間はデータ移動の管理方法によって変化しないと考えられるため，実行時間の差は CPU メモリと GPU メモリの間のデータ転送にかかった時間とほぼ同じとみることができる．したがって，GPU 側からアクセスした要素の割合が少ない場合は CPU-GPU 間のデータ転送にかかった時間が短く，すなわち転送したデータ量が少なくなっているということができる．

図 3 より，アクセスするデータの間隔が 128KByte 以上のとき実行時間は手動管理の場合の 2030 ミリ秒を自動管理の場合が 1650 ミリ秒と下回り，そこから間隔が広がるにつれて実行時間が短くなっていることがわかる．一方で，アクセスするデータの間隔が 120KByte 以下のとき，自動管理の場合の実行時間はおよそ 2600 ミリ秒で一定になっており，手動管理の場合よりも実行時間が長くなっている．これらのことから，データ配列の要素へストライドアクセスを行う場合，その間隔が約 128KByte 以上開いている場合には自動管理の方が手動管理よりも高速になるとわかる．

ⓒ2020 Information Processing Society of Japan

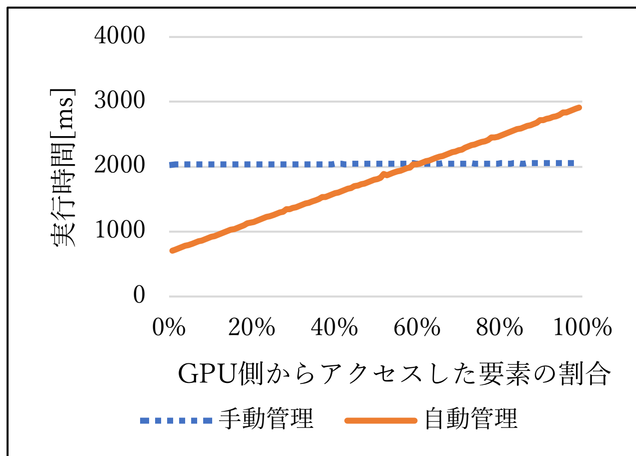

図 3 連続領域アクセスプログラムの実行時間

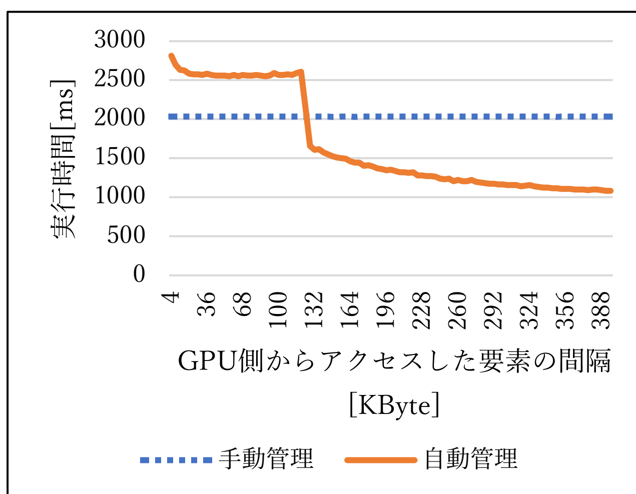

図 4 ストライドアクセスプログラムの実行時間

5.3 並列ステンシル計算における袖通信の性能比較

図 5 に CPU 側で pack/unpack を行う並列ステンシル計算プログラムの実行時間を，図 6 に GPU 側で pack/unpack を行う並列ステンシル計算プログラムの実行時間を示す．問題領域の一辺の長さが 30720 点の場合をサイズ小, 61440 点の場合をサイズ大として示してある．これらの実行時間にはステンシル計算，pack とそれを行うための GPU メモリから CPU メモリへのデータ転送, unpack とその後の CPU メモリから GPU メモリへのデータ転送のそれぞれにかかった時間が含まれ，MPI 通信にかかった時間は含まれていない．

図 5 より，CPU 側で pack を行うケースを自動管理と手動管理で比較すると，問題サイズが小さい場合は手動管理の，問題サイズが大きい場合は自動管理の実行時間が短くなっていることがわかる．問題サイズが小さい場合は，手動管理に比べ自動管理の場合でかかる時間は 1.87 倍になっている．それに対して，問題サイズが大きい場合は手動管理に比べ自動管理の場合でかかる時間は 0.782 倍になっている．

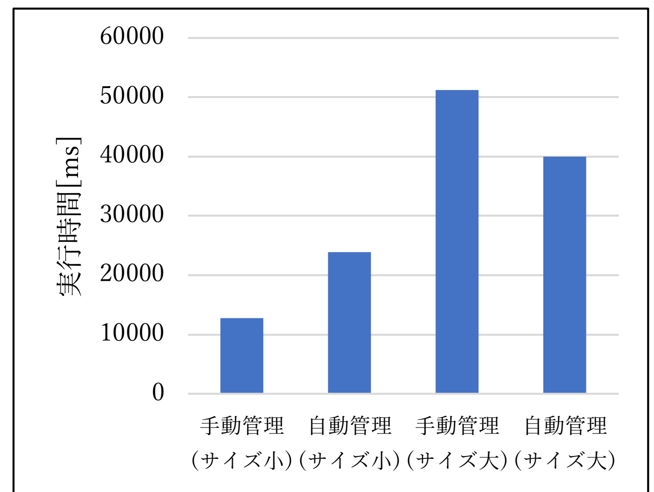

図 5 CPU 側で pack/unpack を行う
並列ステンシル計算プログラムの実行時間

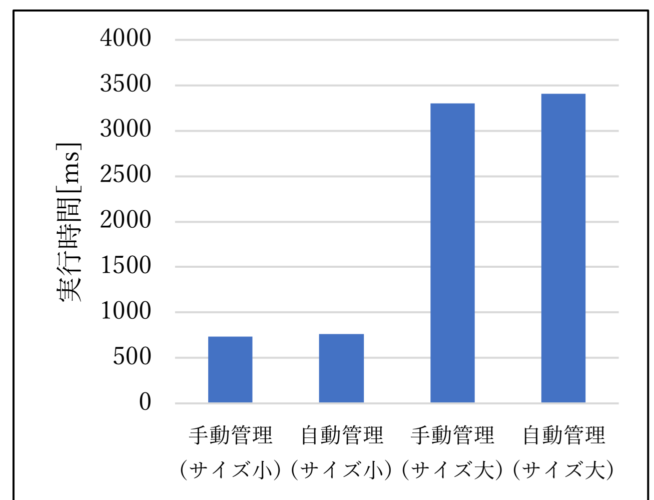

図 6 GPU 側で pack/unpack を行う
並列ステンシル計算プログラムの実行時間

図 6 より，GPU 側で pack を行うケースでは，手動管理に比べて自動管理の実行時間は，問題サイズが小さい場合で 1.03 倍，問題サイズが大きい場合で 1.04 倍となっており，手動管理と自動管理の実行時間の差がほとんどないことがわかる．

6. 考察

6.1 data directive が記述されたコードを managed オプション付きでコンパイルした際の挙動

5.1 節において述べた通り，data directive を記述したコードを managed オプション付きでコンパイルした場合は，data directive を記述し managed オプションなしでコンパイルした場合よりも，data directive を記述せずに managed オプション付きでコンパイルした場合に実行時間がほぼ同じになっている．この結果は，data directive の有無によらず，managed オプション付きでコンパイルした際は Unified

Memory による自動的なデータ移動管理が行われるという予想と矛盾しない．

6.2 データ移動の性能

5.2 節に示した結果から，先頭から連続領域アクセスを行う場合は参照される要素が全体の約 60%以下，ストライドアクセスを行う場合はアクセスされる要素の間隔が約 128KByte 以上のときには，データ移動を Unified Memory による自動管理で行う方が，手動管理でデータ配列全体を転送するよりも，データ転送にかかる時間が短くなるということができる．

ここで，ストライドアクセスで自動管理と手動管理の性能が逆転する閾値である 128KByte 間隔は，8GByte のデータ配列ではアクセスされる要素の割合にして1.53×10^{-5}でしかなく，連続領域アクセスの閾値である約 0.6 に及ばない点について考える．

Unified Memory はページングによる実装になっており，データ転送がページ単位で行われる．例えば 1 つの要素だけを参照する際にも，その要素が格納されたページ全体が転送されることになる．このため，特にストライドアクセスでは，ページサイズが大きいほど，アクセスされる要素の数よりも転送される要素の数が大きくなる．これに基づくと，アクセスされる要素の間隔がページサイズ以下の場合はデータ配列の全要素が転送され，間隔がページサイズ以上の場合にはじめて転送が必要な要素が減ることになる．なお，連続領域アクセスの場合は，転送が必要なデータの割合と GPU 側からアクセスした要素の割合はほとんど同じになる．

これをふまえて，128KByte 間隔のストライドアクセスにおいて転送が必要な要素の割合が 0.6 となるページサイズを求めると，$128 \times 0.6 = 76.8$より 76.8[KByte]となる．しかし，図 3 をみると，アクセスした要素が 76.8[KByte]から 120[KByte]の間は転送が必要な要素が減ると予想されるにもかかわらず，ほとんど実行時間の減少がみられない．この点から，正確なページサイズについてはより詳細な調査をもとにした検討が必要だと考えられる．

6.3 並列ステンシル計算における袖通信の性能

CPU 側で pack/unpack を行う場合，手動管理では小さい問題サイズで 3.5GByte，大きい問題サイズで 14GByte になる領域すべての転送が行われる．自動管理では小さい問題サイズでおよそ 120KByte，大きい問題サイズでおよそ 240KByte の間隔で 1 要素ずつの転送が必要になると考えられる．自動管理で小さい問題サイズについて実行した場合，アクセスされるデータの間隔が 6.2 節で触れた閾値とほとんど同じかそれより低くなるため，データ転送量の削減効果が低いと考えられる．それに対して自動管理で大きい問題サイズについて実行した場合，アクセスされるデータの間隔が閾値よりも明らかに大きくなるため，データ転送量が削減され，結果として手動管理の場合よりも実行時間が短くなったと考えられる．

GPU 側で pack/unpack を行う場合は，手動管理と自動管理のいずれの場合も，小さい問題サイズでは連続する 32KByte，大きい問題サイズでは連続する 64KByte の転送が必要となる．こちらは自動管理でも転送されるデータ量が 1 か 2 ページ分で済むため，自動管理において発生するオーバヘッドもその分小さくなり，データ転送にかかる時間の割合が計算にかかる時間に対して少なくなると考えられる．このことから，転送が必要なデータ量が計算量に対して小さいときは，データ移動管理が手動の場合と自動の場合の実行時間の差が小さくなると考えられる．

7. まとめ

ここまでに，まず data directive が記述されたコードを managed オプション付きでコンパイルした際の挙動を確認した．その後，連続領域アクセスとストライドアクセスの 2 種類のアクセスパターンについて，データ移動の性能を手動管理と自動管理で測定し，比較した．最後に，並列ステンシル計算における袖通信を題材として，データ移動管理手法によるデータ転送性能を，pack/unpack の方法と扱う問題サイズを変えて測定し，比較した．

これらのことから，Unified Memory による自動管理の方が有利になると思われた，「大きいデータ配列のうち一部のみが参照されるものの，参照される部分が明らかでないために配列の全部を転送している」という場合では，不連続にアクセスされる要素の間隔が 128KByte よりも広くなる場合に，データ転送が削減され，データ転送にかかる時間が短くできると考えられる．実際に並列ステンシル計算における袖通信を題材として行った実験では，アクセスされる要素の間隔がおよそ 240KByte となるサイズの問題では手動管理よりも自動管理の実行時間が短くなった．

転送が必要なデータが小さい場合には，データ移動管理の方法によるプログラム全体の実行時間の変化はわずかにとどまる．そのような場合については，Unified Memory によるデータ移動の自動管理を利用することで，より開発コストを抑えた GPU の利用が可能になると考えられる．

6.2 節において，Unified memory による自動的な転送の単位となるページサイズを簡易的に予想した．しかし，これをより正確に求められれば，より自動管理と手動管理のどちらが高速になるかの予想が立てやすくなると考えられる．このため，今後は転送単位についての詳細な検討が課題として挙げられる．

参考文献

[1]“NOVEMBER 2020 | TOP500”．
https://www.top500.org/top500/lists/2020/11/, (参照 2020-11-17).
[2]“OpenACC Specification”
https://www.openacc.org/specification

[3] Landaverde R, Zhang T, Coskun AK, Herbordt M (2014) An investigation of Unified Memory access performance in CUDA. In: Proceedings of 2014 IEEE High Performance Extreme Computing Conference
[4] Sebastien Deldon, James Beyer, and Douglas Miles. (2018) OpenACC and CUDA Unified Memory. In: Proceedings of Cray User Groups　2018
[5] 土井 淳：NVLink における Unified Memory と OpenACC によるプログラミングの性能評価．研究報告ハイパフォーマンスコンピューティン グ（HPC），2017-HPC-159, No.5.
[6]“PGI Compilers&Tools”. https://www.pgroup.com/index.htm, (参照 2020-11-16).
[7] “PGI Compiler User's Guide Version 19.10 for x86 and NVIDIA Processors　“. https://www.pgroup.com/resources/docs/19.10/x86/pgi-user-guide/index.htm#acc-mem-unified, (参照 2020-11-16).

ⓒ2020 Information Processing Society of Japan