Vol.2016-HPC-155 No.47
2016/8/10

単一GPUコードをマルチGPU環境で実行するための多次元データ分割手法の検討

酒井 亮太郎[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，単一 GPU（Graphics Processing Unit）向けに記述された CUDA（Compute Unified Device Architecture）コードを，マルチ GPU 上で加速するための多次元データ分割手法を検討する．我々の手法は，1 次元データ向けの既存手法の拡張であり，一部のスレッドのサンプリング実行に基づいて GPU メモリに格納しきれない大規模な多次元データを小さなセグメントに分割する．既存手法と比較して，提案手法は小さなセグメントサイズを実現し，GPU メモリの使用量を節約するとともに，CPU・GPU 間のデータ転送量を削減する．行列積を用いた予備実験の結果，提案手法は既存手法よりも GPU メモリ使用量を 80 %削減し，CPU メモリに収まる程度の大規模行列を正しく処理できた．しかし，データ分割に起因するインデックス変換が実効性能を 27 %低下させることが分かった．

キーワード：CUDA，データ分割，マルチ GPU システム

A Preliminary Method for Multi-dimensional Data Decomposition for Executing a Single-GPU code on Multi-GPU Environments

Ryotaro Sakai[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, we present a preliminary method for multi-dimensional data decomposition, aiming at realizing multi-GPU acceleration of the compute unified device architecture (CUDA) code written for a single graphics processing unit (GPU). Our method, namely an extension of a previous method that deals with one-dimensional data, performs a sampling run of selected threads to generate small segments by decomposing large data that cannot be stored entirely in GPU memory. As compared with the previous method, our method realizes smaller segment size, so that it saves the usage of GPU memory and reduces the amount of CPU-GPU data transfer. As a result of preliminary experiments using matrix multiplication, the presented method reduced the usage of GPU memory by 80%, and thereby correctly processed large matrices that can be stored in CPU memory. However, we found that index translation needed for data decomposition dropped the effective performance by 27%.

Keywords: CUDA, data decomposition, multi-GPU system

1. はじめに

GPU（Graphics Processing Unit）[1] とは，グラフィクス処理の加速を目的として，数千個ものスレッドを並列実行できるハードウェアである．昨今では画像処理にとどまらず，流体シミュレーションや機械学習などの科学計算も高速化できる．典型的には，CPU と比べておよそ 10 倍の高速化が期待できる．しかし，GPU メモリの容量は 16 GB にとどまっていて，CPU 実装と同規模の問題を扱うためには，複数の GPU を活用することが不可欠である．

一般に，マルチ GPU コードの開発は，単一 GPU コードよりも大きな労力を要する．例えば，NVIDIA 社の GPU 向け開発環境として多用される CUDA（Compute Unified Device Architecture）[2] では，プログラマはデータならび

[1] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

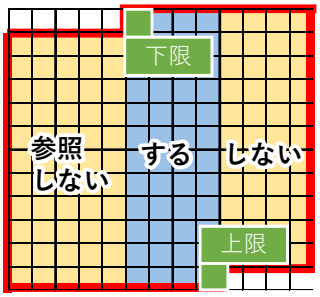


図 1 既存手法 [3] が推定する参照範囲（2 次元配列を列参照する場合）

Fig. 1 Memory access range estimated by the previous method [3]. Rows in a two-dimensional array are accessed.

に計算の分割方法を考え，それらを各 GPU に割り当てるために単一 GPU コードを書き換える必要がある．特に，データ分割の前後でインデックス体系が異なるため，それらの辻褄を合わせる必要があり，マルチ GPU コードの開発を難解なものにしている．

そこで，Kim ら [3] は，OpenCL（Open Computing Language）[4] で記述された単一 GPU コードを，そのままマルチ GPU 上で処理できるシステムを開発した．この既存システムは，データ分割を自動化するために，GPU コードのサンプリング実行によりメモリ参照範囲を解析している．具体的には，GPU コードを呼び出す直前に，一部の GPU スレッドの処理を CPU が模倣し，すべての GPU スレッドの参照範囲を推定している．この推定に基づいて，データを小さなセグメントに自動分割する．しかし，この推定手法は参照範囲を 1 次元のメモリ番地で指定している（**図 1**）．したがって，分割対象のデータが多次元である場合，分割後のセグメントが冗長な領域を含む可能性がある．例えば，2 次元配列に対して一部の列を参照する場合，参照しない列を含めて参照範囲として返してしまう（図 1）．

そのような冗長な領域の除去を目的として，本研究では，多次元データに対するデータ分割手法を検討する．我々の手法は，Kim ら [3] の既存手法を拡張し，次元ごとに参照範囲を指定する．既存手法を単純に多次元データに適用した場合，必ずしも正しいデータ分割を得られないことを示し，正しい分割を得るための拡張を示す．

以降では，まず 2 章で GPU プログラムにおけるデータ分割に関する関連研究を紹介する．次に，3 章で既存手法および多次元データへの単純な適用を示す．その後，4 章で我々の手法を示し，5 章で予備評価の結果を示す．最後に，6 章で本論文をまとめ，今後の課題を挙げる．

2. 関連研究

マルチ GPU およびマルチノード環境を対象として，負荷分散の労力を軽減する研究 [5, 6] がある．Müller ら [5] は CUDASA フレームワークを開発した．CUDA のスレッド階層およびメモリ階層を拡張することにより，マルチノード，マルチ GPU および単一 GPU 向けの統一的な記述を可能とした．Kim ら [6] は，複数 GPU 向け OpenCL コードを GPU クラスタ上で動作させるためのランタイムシステム SnuCL を開発した．SnuCL は，Message Passing Interface（MPI）[7] によるノード間通信を隠蔽する．これらの研究は，プログラマによる明示的なデータ分割を前提としていて，データ分割に関する支援はない．

データ分割の自動化を目的として，Luk ら [8] の Qilin システムは，GPU コードにおけるデータ依存関係を分析し，GPU メモリに収まるようにデータを分割した．また，Ji ら [9] の開発したランタイムライブラリ Region-based Software Virtual Memory（RSVM）は，GPU メモリに乗りきらない大規模データの分割を隠蔽した．ただし，これらはそれぞれのプログラミングモデルに則った記述を要求し，固有の関数を使って GPU を操作する必要がある．結果，元のコードにおけるデータ転送，GPU コードの実行，あるいは GPU コード内でのメモリ参照といった記述を，固有の関数に置き換える必要があり，プログラミングの労力が大きい．

そこで，元のコード構造を維持したまま，データ分割を隠蔽することを目的として，Lee ら [10] は，単一 GPU 向けの OpenCL コードをマルチ GPU 環境で動作させた．ただし，既存手法 [3] と同様に，不連続なメモリ参照パターンに対して，効率のよいデータ分割は難しい．彼らが開発したシステム Virtual Address Space for Throughput processors（VAST）[11] は，大規模データの分割を自動化するために，GPU コードのメモリ参照範囲を解析する．この解析は，あらゆるメモリ参照パターンに対応するために，すべてのスレッドの振る舞いを解析している．したがって，Kim らの既存手法 [3] と比べて，適用範囲は広いものの，解析のオーバヘッドは大きい．

GPU における大規模データ処理を実現するために，CUDA は Unified Memory [2] や Mapped Memory [2] を提供している．前者は，CPU および GPU が共有するメモリ空間を実現し，CPU・GPU 間のデータ転送を隠蔽する．後者は，GPU コードから CPU メモリ上のデータ参照を可能にする．これらは，いずれもデータ参照時にオンデマンドで CPU メモリから GPU メモリへデータを転送していて，データ分割を必要としない．

CPU コードにコンパイラへのディレクティブを挿入することで，GPU コードを自動生成する研究 [12–14] がある．中野ら [12] は，ステンシル計算を対象として大規模データを自動的に分割し，GPU 上でパイプライン処理した．OpenMPC [13] は，マルチコア CPU 向けのディレクティブ Open Multi-Processing（OpenMP）[15] を拡張し，CUDA 特有の記述をプログラマから隠蔽した．Sabne ら [14] は，OpenMPC コードを対象として，大規模データのパイプラ

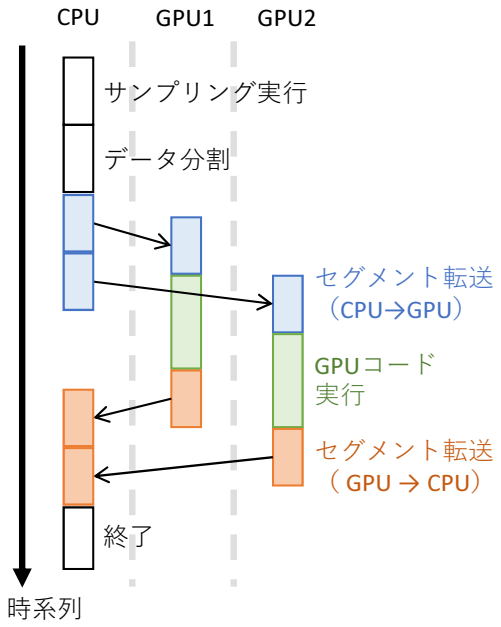


図 2 既存システム [3] における処理の流れ
Fig. 2 Processing flow of the previous system [3].

イン処理を自動化した．これら CPU コードを入力とする研究に対し，本研究はアクセラレータ向けコードを入力としていて，アクセラレータ固有の最適化を施せる．

3. Kim らのデータ分割手法

Kim ら [3] のデータ分割手法は，ある前提条件を満たす単一 GPU 向け OpenCL コード [4] をマルチ GPU 環境で加速する．**図 2** に，その処理の流れを示す．まず GPU コード，すなわちカーネル関数の呼び出し直前に，CPU 上でサンプリング実行を遂行する．これによりタスクごとのメモリ参照範囲を推定する．ここで，タスクとはワークグループの集合に割り当てられた処理を指し，異なるタスクは独立に処理できる．次に，データを分割し，タスク実行に必要なセグメントをタスクごとに得る．分割後のセグメントを CPU メモリから GPU メモリへ転送し，対応するタスクを GPU 上で並列処理する．タスクは独立に処理できるため，それぞれ異なる GPU に割り当てられる．

なお，上述の前提条件は，サンプリング実行すべき GPU スレッドの数を削減するために設けられている．ここで，分割対象の配列を `A` とし，GPU コードにおいて `n` 番目の要素 `A[n]` が参照されていると仮定する（**図 3**）．このとき，その前提条件とは，配列のインデックス `n` がワークアイテム ID(l_1, l_2, l_3) およびワークグループ ID(l_4, l_5, l_6) に関する 1 次関数 $f(l_1, \ldots, l_6)$ で与えられることである．すなわち，

$$f(l_1, \ldots, l_6) = \sum_{i=0}^{6} (a_i \times l_i) \qquad (1)$$

ここで，a_i $(0 \leq i \leq 6)$ は，すべてのスレッドにおいて共通の定数であり，$l_0 = 1$ は記述を簡略化するためのダミー変数である．例えば，図 3 の例であれば，$a_0 = 0$，$a_1 = 1$，$a_2 = s$，$a_3 = w$，$a_4 = sw$，$a_5 = wh$ および $a_6 = swh$ であり，これらはいずれも GPU コードの呼び出し直前には定数とみなせる．このとき，式 (1) に対して定理 1 を適用できる．

```
1  __kernel void func(__global float* A,
2    __global float* B, float d, int s, int w, int h)
3  {
4      int i = get_local_id(0) + get_group_id(0)*s;
5      int j = get_local_id(1) + get_group_id(1)*s;
6      int k = get_local_id(2) + get_group_id(2)*s;
7      int n = i + j*w + k*w*h;
8
9      B[n] = d * A[n];
10 }
```

図 3 GPU コードの例
Fig. 3 An example of GPU code.

```
1  void _samp_func(int _l0, int _l1, int _l2,
2                  int _g0, int _g1, int _g2,
3                  float* A, float* B, float d,
4                  int s, int w, int h)
5  {
6      int i = _l0 + _g0*s;
7      int j = _l1 + _g1*s;
8      int k = _l2 + _g2*s;
9      int n = i + j*w + k*w*h;
10
11     ACCESS_READ(A, n); ACCESS_WRITE(B, n);
12 }
```

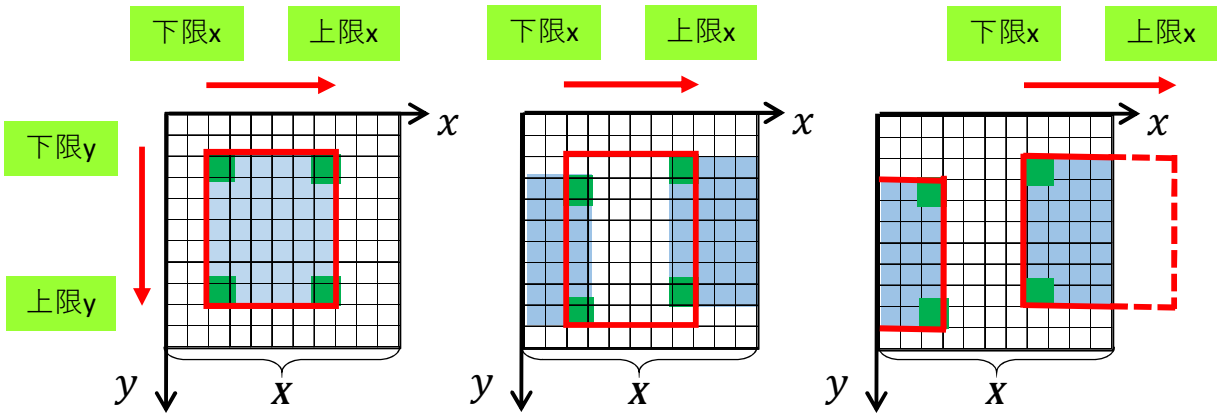
図 4 サンプリングコードの例
Fig. 4 An example of sampling code.

定理 1. *n 個の変数からなる 1 次関数 $f(l_1, \ldots, l_n)$ の上限（あるいは下限）は，すべての l_i が最大（あるいは最小）のときに得られる．*

したがって，GPU コードが先の前提条件を満たしさえすれば，サンプリング実行の対象としてタスク内のすべてのスレッドを選択する必要はなく，l_1〜l_6 が最大および最小のスレッドのみを選択すれば，全体の参照範囲の上限および下限が得られる．

図 4 は，図 3 の GPU コードに対するサンプリングコードであり，CPU 上でタスクごとの参照範囲を推定する．サンプリングコードは，サンプリング対象となるスレッドのワークアイテム ID およびワークグループ ID を引数とし，それらのスレッドの参照メモリ番地を記録する．

既存手法は，参照範囲を 1 次元のメモリ番地で指定しているため，多次元データに対して冗長な領域を含み得る（図 1）．冗長な領域は，GPU メモリの使用量を逼迫するだけでなく，CPU・GPU 間のデータ転送時間を増大させ，全体性能を低減する恐れがある．



(a) 次元ごとの上限および下限　(b) 誤った推定　(c) 正しい推定

図 5　多次元データに対する参照範囲の推定

Fig. 5　Estimating the memory access range for multi-dimensional data.

3.1　多次元データへの安易な適用

多次元データに対する冗長なセグメント領域を最小化するための安易な手法として，推定で得られた 1 次元のメモリ番地を多次元のメモリ番地へ写像することが考えられる．つまり，次元ごとに参照範囲の上限および下限を推定し，その範囲を覆う最小範囲矩形をセグメントとみなせばよい（**図 5(a)**）．

例えば，大きさが $X \times Y \times Z$ の 3 次元配列に対しては，1 次元のメモリ番地 $f(l_1,\ldots,l_6)$ は以下のように次元ごとに写像できる．

$$g_x(l_1,\ldots,l_6) = f(l_1,\ldots,l_6)\,\%\,X \qquad (2)$$

$$g_y(l_1,\ldots,l_6) = f(l_1,\ldots,l_6)\,/\,X\,\%\,Y \qquad (3)$$

$$g_z(l_1,\ldots,l_6) = f(l_1,\ldots,l_6)\,/\,X\,/\,Y \qquad (4)$$

ここで，$g_x(l_1,\ldots,l_6)$，$g_y(l_1,\ldots,l_6)$ および $g_z(l_1,\ldots,l_6)$ はそれぞれ x 次元，y 次元および z 次元のメモリ番地を表し，% は剰余演算子を表す．

しかし，式 (2) および式 (3) が含む剰余演算は，定理 1 の前提条件を満たさない．すなわち，$g_x(l_1,\ldots,l_6)$ および $g_y(l_1,\ldots,l_6)$ は l_1〜l_6 の 1 次関数ではない．結果，**図 5(b)** に示すように，参照範囲が左端と右端の領域に分散している場合，その参照範囲の正しい推定に失敗してしまう．

4.　提案するデータ分割手法

提案手法は，単一 GPU コードおよび分割対象の多次元配列の大きさ $X \times Y \times Z$ を入力として，マルチ GPU コードのためのデータ分割を出力する．既存手法 [3] と同様に，単一 GPU コードは前提条件を満たす必要がある．

4.1　多次元データに対する参照範囲の推定

正しい参照範囲を推定するために，我々の手法は式 (2) および式 (3) における X および Y が GPU コードの実行直前には定数とみなせることに着目する．結果，前提条件を崩していた剰余演算を事前に計算して消去でき，式 (2)〜式 (4) を 1 次関数とみなせる．つまり，式 (2)〜式 (4) はそれぞれ以下のように解釈できる．

$$g_x(l_1,\ldots,l_6) = \sum_{i=0}^{6}((a_i\,\%\,X)\times l_i) \qquad (5)$$

$$g_y(l_1,\ldots,l_6) = \sum_{i=0}^{6}((a_i\,/\,X\,\%\,Y)\times l_i) \qquad (6)$$

$$g_z(l_1,\ldots,l_6) = \sum_{i=0}^{6}((a_i\,/\,X\,/\,Y)\times l_i) \qquad (7)$$

例えば，式 (5) であれば，$a_i\,\%\,X$ をサンプリング実行時に計算し，定数項に置き換えられる．

結果，**図 5(c)** のように参照範囲が分裂していても，提案手法は分裂の有無を判定することで正しい参照範囲を推定できる．分裂の有無は以下のように判定できる．x 次元における上限を U_x として，$X < U_x$ が成立するとき，参照範囲が分裂している．この場合，2 つの領域を 1 つのセグメントとしてまとめる．x 次元における下限を L_x として，範囲 $[0, U_x\%X]$ ならびに $[L_x, X]$ をまとめればよい．

4.2　インデックス変換

一般に，データ分割は GPU メモリの使用量を削減できるが，分割の前後でインデックス体系を変えてしまう．入力となる単一 GPU コードは分割前のインデックス体系 $\mathcal{C}_1$ に基づいて記述されているため，分割後の GPU コードを実行する前に $\mathcal{C}_1$ を分割後のインデックス体系 $\mathcal{C}_2$ へ変換する必要がある．

図 6 に，3 次元配列に対する変換の例を示す．実線で囲まれた直方体が分割前の大きさ $X \times Y \times Z$ の配列を表し，破線で囲まれた直方体が分割後の大きさ $U \times V \times W$ の配列（セグメント）を表す．$\mathcal{C}_1$ における（1 次元の）メモリ番地 f_1 は，$\mathcal{C}_2$ における（1 次元の）メモリ番地 f_2 へ，以下のように変換できる．

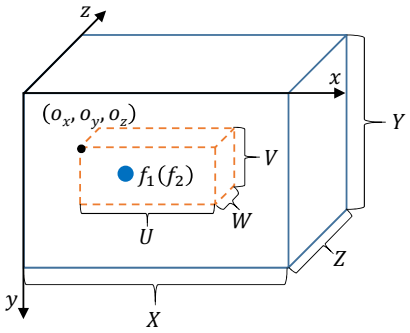


図 6 分割前後の 3 次元配列に対する座標系
Fig. 6 The coordinate systems for a three-dimensional array before and after decomposition.

表 1 実験環境
Table 1 Experimental environment.

項目	仕様
CPU	Intel Xeon E5-2680v2
CPU メモリ容量	512 GB
GPU	Two NVIDIA Tesla K40
GPU メモリ容量	12 GB per GPU
OS	Ubuntu 14.04
コンパイラ	gcc 4.8.4
CUDA バージョン	6.5

$$
\begin{aligned}
f_2 = {} & (f_1 \,\%\, X - o_x) \\
& + (f_1 \,/\, X \,\%\, Y - o_y) \times U \\
& + (f_1 \,/\, X \,/\, Z - o_z) \times U \times V
\end{aligned} \tag{8}
$$

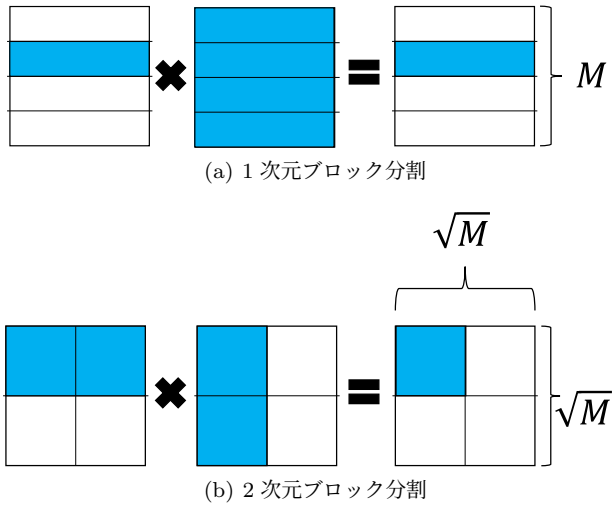
ここで，(o_x, o_y, o_z) は分割前の配列における 3 次元座標を表し，分割後の配列における原点に対応している．

5. 予備評価

提案手法を既存手法 [3] と比較するために，各々を CUDA SDK の行列積コードに手動で適用し，それぞれのメモリ使用量ならびに実効性能を**表 1** のマルチ GPU マシン上で計測した．比較対象とした 3 種のコードを以下にまとめる．

(1) 適用前の単一 GPU コード．大きさ $N \times N$ の行列に対し，行列積 $C = A \times B$ を計算する．
(2) 既存手法が（1）から生成したマルチ GPU コード．行列 A および C に対して 1 次元ブロック分割を施している（**図 7(a)**）．この場合，行列 B は分割できない．
(3) 提案手法が（1）から生成したマルチ GPU コード．すべての行列 A，B および C に対して 2 次元ブロック分割を施している（**図 7(b)**）．

なお，(1) は GPU メモリに格納できる程度の大きさの行列を処理できる．一方，(2) および (3) は行列を分割しているため，GPU メモリ容量を超える大きさの行列を処理できる．ただし，行列は CPU メモリに格納できる必要がある．以降では，各行列の分割数を M とし，M は任意の平方数であるものとする．



(a) 1 次元ブロック分割
(b) 2 次元ブロック分割

図 7 行列積に対する既存手法ならびに提案手法のデータ分割
Fig. 7 Data decomposition for matrix multiplication by the previous method and our method.

また，(2) および (3) は実効性能を高めるために，ダブルバッファを備え，カーネル実行をデータ転送とオーバラップする．各バッファの大きさは GPU メモリ容量の半分，すなわち 6 GB である．タスクは各 GPU にサイクリックに割り当てられる．

5.1 GPU メモリ使用量の解析

各手法の GPU メモリ使用量を評価するために，それぞれの生成するセグメントの大きさから GPU メモリ使用量を解析的に計算した．なお，既存手法は行列 B を分割できないため，その全体をセグメントに含めた．式 (9) および式 (10) に，既存手法（1 次元ブロック分割）および提案手法（2 次元ブロック分割）におけるセグメントサイズを示す．

$$4 \times N^2 \times \left(\frac{1}{M} + 1 + \frac{1}{M} \right) = 4N^2 \frac{M+2}{M} \tag{9}$$

$$4 \times N^2 \times \left(\frac{\sqrt{M}}{M} + \frac{\sqrt{M}}{M} + \frac{1}{M} \right) = 4N^2 \frac{2\sqrt{M}+1}{M} \tag{10}$$

なお，行列の要素は `float` 型であり，セグメントサイズの単位はバイトである．

式 (9) および式 (10) を基に，$N = 40,000$ のときの M およびセグメントサイズの関係を調べた（**図 8**）．行列を分割しない場合，行列全体のメモリ使用量は 19.2 GB に達し，このままでは GPU メモリが枯渇して処理できない．

2 次元ブロック分割を基にする提案手法は，M の増大とともに，セグメントサイズを 0.8 GB に削減できた．したがって，行列の全体を CPU メモリに格納できれば，分割数 M を適切に増大させて行列積を処理できる．例えば，$N = 40,000$ の行列に対しては，$M = 9$ のときにセグメントサイズが 5 GB に減少し，6 GB のバッファに格納できる．

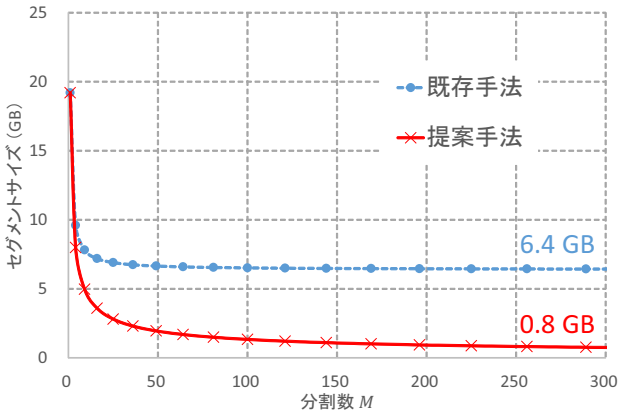


図 8 分割数を変えたときのセグメントサイズ（$N = 40,000$）
Fig. 8 Segment size for different numbers of divisions ($N = 40,000$).

表 2 行列の大きさ $N \times N$ およびそのメモリ使用量
Table 2 Matrix size $N \times N$ and its memory usage.

コード	N の最大値	行列全体のメモリ使用量（GB）
(1)	31,584	11
(2)	38,496	17
(3)	207,680	482

一方，1 次元ブロック分割を基にする既存手法では，そのセグメントサイズが 6.4 GB に留まった．既存手法は行列 B を分割できないため，そのセグメントサイズが $4N^2$ を下回ることはない．したがって，$N = 40,000$ の行列に対しては分割数 M を増やしたとしても，セグメントを 6 GB のバッファに格納できず，処理に失敗する．

5.2 処理可能な行列サイズ

セグメントサイズの削減が処理可能な行列の大きさを増大させることを，実際に各コードを実行して確認した（**表 2**）．分割前のコード（1）は，行列 A，B および C の大きさ $N \times N$ が GPU メモリ容量 12 GB を超えたときに処理に失敗し，その大きさは $N = 31,584$ に達した．提案手法のコード（3）は，この GPU メモリ容量に関する制約を CPU メモリ容量に関する制約に緩和し，行列 A，B および C の大きさ $N \times N$ が CPU メモリ容量に近い 482 GB に至るまで行列積を処理できた．このときの行列の大きさは $N = 207,680$ に達した．

一方，5.1 節で述べた通り，既存手法のコード（2）ではセグメントサイズが $4N^2$ を下回ることはない．したがって，処理可能な行列の大きさは $N = 38,496$ に留まった．式 (9) によれば，このときのセグメントサイズは 6 GB であり，バッファサイズと一致する．

5.3 実効性能

行列の大きさを $6,400 \leq N \leq 64,000$ として，各コード

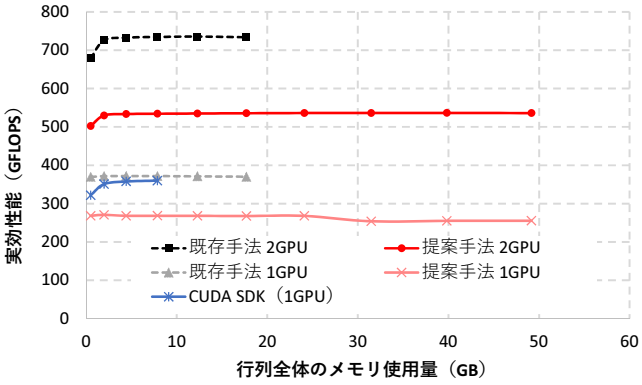


図 9 行列積の実効性能
Fig. 9 Effective performance of matrix multiplication.

（1）～（3）の実効性能を計測した（**図 9**）．

1GPU 実行時の性能を比較すると，（2）は（1）をわずかに上回った．一方，（3）は（2）より性能が最大で 28 %低下した．同様の傾向は，2GPU 実行時にも確認でき，（3）および（2）の性能差は 27 %であった．このように，既存手法が処理できる小さな行列に対し，提案手法の性能は劣る．この理由は，GPU 上でのインデックス変換に関わるオーバヘッドにある．提案手法は，メモリ参照のたびにインデックス変換のための演算を必要とする（4.2 節）．同様のインデックス変換は，既存手法も必要とするが，1 次元ブロック分割時に必要な演算はオフセット値の減算 1 個のみであり，負荷が軽い．一方，2 次元ブロック分割時に必要な演算は，式 (8) に示した剰余演算，除算，乗算および減算である．ゆえに，GPu コードの実行時間が既存手法のものよりも増大していた．したがって，提案手法は既存手法では GPU メモリが枯渇してしまう大規模データに対して有用である．

図 9 では，行列全体のメモリ使用量，すなわち行列の大きさ N に対して実効性能がさほど変動していない．既存手法は N とともに冗長なデータ転送量が増大する．しかし，データ転送およびカーネル実行のオーバラップにより，データ転送時間を隠蔽できたため，冗長性に起因する性能低下は確認できなかった．

1GPU 版に対する 2GPU 版の速度向上率は，いずれの手法においてもほぼ 2 倍であり上限に近い．一般に，分割数 M とともにデータ分割のオーバヘッドが増大するが，M の増大に起因する性能低下は確認できなかった．例えば，$N = 32,000$ に対する（3）の 2GPU 実行の場合，最大の実効性能は $M = 6$ のときの 535 GFLOPS であり，$M = 16$ においてもほぼ同様の 531 GFLOPS を得た．したがって，PCI-Express バスの実効帯域幅が GPU 数とともに増大しさえすれば，GPU 数とともに行列積の実効性能は向上すると考えられる．

なお，分割数 M が著しく少ない場合，計算負荷の不均等が原因で実効性能が低下した．例えば，2GPU 実行にお

ける $M = 3$ のときの実効性能は，提案手法ならびに既存手法ともに最大の実効性能と比べて 10 %低下した．したがって，実効性能を最大化するためには，適切な分割数 M を見つけ出す必要がある．

6. まとめと今後の課題

本論文では，単一 GPU 向けに記述された CUDA コードをマルチ GPU 環境で加速することを目的として，多次元データに対するデータ分割手法を検討した．我々の手法は，1 次元データを対象とする既存手法 [3] を拡張することにより，多次元データを小さなセグメントに分割する．既存手法と同様に，参照先のメモリ番地がスレッド ID およびスレッドブロック ID に関する 1 次関数で表せることを前提として，一部の GPU スレッドのサンプリング実行により，正しい分割を実現する．そのために，前提の成立を阻む剰余演算を事前計算により消去し，次元ごとに参照範囲を推定する．結果，セグメントサイズの削減により，GPU メモリ使用量を節約し，CPU・GPU 間のデータ転送量を削減できる．

予備評価では，CUDA SDK の行列積コードに対して提案手法および既存手法を手動で適用し，それらのメモリ使用量ならびに実効性能をマルチ GPU マシン上で評価した．結果，提案手法は行列の全体を CPU メモリに格納できれば，適切な分割数のもとで GPU メモリ容量を超える規模の行列を処理できた．このときの行列の大きさ $N \times N$ は，分割前と比べて 43 倍大きく，既存手法と比べて 29 倍大きかった．ただし，既存手法が処理できる規模の行列に対し，その実効性能は 28 %低下した．この理由は，提案手法は多次元分割に起因するインデックス変換にあり，GPU コードにおける変換オーバヘッドを削減する必要がある．まとめると，既存手法では GPU メモリが枯渇してしまう大規模データに対して，提案手法は有用である．

今後の課題としては，コード変換の自動化が挙げられる．

謝辞 本研究の一部は，科研費 15K12008，15H01687，16H02801 および JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」の補助による．

参考文献

[1] NVIDIA Corporation: NVIDIA GeForce GTX 980 (2014).
[2] NVIDIA Corporation: CUDA C Programming Guide Version 7.5 (2015).
[3] Kim, J., Kim, H., Lee, J. H. and Lee, J.: Achieving a Single Compute Device Image in OpenCL for Multiple GPUs, *Proc. 16th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'11)*, pp. 277–288 (2011).
[4] Khronos OpenCL Working Group: The OpenCL Specification Version 1.1 (2011). `http://www.khronos.org/registry/cl/`.
[5] Müller, C., Frey, S., Strengert, M., Dachsbacher, C. and Ertl, T.: A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality, *IEEE Trans. Visualization and Computer Graphics*, Vol. 15, No. 4, pp. 605–617 (2009).
[6] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G. and Lee, J.: SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters, *Proc. 26th ACM Int'l Conf. Supercomputing (ICS'12)*, pp. 341–352 (2012).
[7] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, *Int'l J. Supercomputer Applications and High Performance Computing*, Vol. 8, No. 3/4, pp. 159–416 (1994).
[8] Luk, C.-K., Hong, S. and Kim, H.: Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping, *Proc. 42nd IEEE/ACM Int'l Symp. Microarchitecture (MICRO'09)*, pp. 45–55 (2009).
[9] Ji, F., Lin, H. and Ma, X.: RSVM: a Region-based Software Virtual Memory for GPU, *Proc. 22nd Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'13)*, pp. 269–278 (2013).
[10] Lee, J., Samadi, M., Park, Y. and Mahlke, S.: Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems, *Proc. 22nd Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'13)*, pp. 245–255 (2013).
[11] Lee, J., Samadi, M. and Mahlke, S.: VAST: The Illusion of a Large Memory Space for GPUs, *Proc. 23rd Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'14)*, pp. 443–454 (2014).
[12] 中野瑛仁，伊野文彦，萩原兼一：アクセラレータのメモリ容量を超えるデータをパイプライン処理するためのディレクティブ，情処学研報，2013-HPC-142 (2013). 8 pages.
[13] Lee, S. and Eigenmann, R.: OpenMPC: extended OpenMP for efficient programming and tuning on GPUs, *Int'l J. Computational Science and Engineering*, Vol. 8, No. 1, pp. 4–20 (2013).
[14] Sabne, A., Sakdhnagool, P. and Eigenmann, R.: Scaling Large-Data Computations on Multi-GPU Accelerators, *Proc. 27th ACM Int'l Conf. Supercomputing (ICS'13)*, pp. 443–454 (2013).
[15] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. and Menon, R.: *Parallel Programming in OpenMP*, Morgan Kaufmann, San Mateo, CA (2000).