Vol.2016-HPC-155 No.41
2016/8/10

入力サイズより出力サイズが十分小さい場合のGEMM演算のGPU上での実装

服部 大士[1,a)]　藤本 典幸[1,b)]

概要：BLAS の GPU 向け実装に cuBLAS や MAGMA BLAS が存在する．それらの実装の GEMM ルーチンは与えられた行列のサイズに最適な内部ルーチンを呼び出すことによって多くの場合，理論最大性能に近い高速な計算を実現している．しかし，これらの実装では，サイズ m × k の行列と k × n の行列の GEMM 演算は m，n が小さい場合低速である．本論文では，転置を伴わない GEMM 演算において，k が特に大きく，m，n が小さい場合を対象とした実装手法を提案し，提案手法が既存手法より最大で 2 倍程度高速であることを示す．

1. はじめに

行列積は最も基本的で重要な線形代数演算の一つであり，様々なアルゴリズムの研究が行われている．n 次正方行列の場合，行列積の定義に基づく基本的なアルゴリズムの計算量は $O(n^3)$ であり，それよりも小さな計算量もつ代表的なアルゴリズムとして，計算量 $O(n^{\log_2 7})$ であるシュトラッセンのアルゴリズム [12] が存在する．このように，ある程度計算量が大きいので，あるアルゴリズム中で行列積をサブルーチンとして用いている場合，計算時間のうちの多くを行列積が占めることが想定される．すなわち行列積計算を高速化することはそれ自体の高速化にとどまらず，行列積をサブルーチンとして用いる多くのアルゴリズムを高速化することにつながるため行列積は重要な問題である．また，行列積を拡張した GEMM 演算 [6] も，広い範囲に応用されるため，行列積と同様に重要である．また，計算過程にそれぞれ独立な部分が多く存在することから，多数のプロセッサを持つ計算機を想定した並列アルゴリズムの研究も多く行われており，Fox のアルゴリズム [5] をはじめ，Fox のアルゴリズムを基にした PUMMA [3]，SUMMA [2], [13] と DIMMA [2] などが提案されている．

その中で，Demmel らは，行列積を計算機がもつメモリの残量に応じて深さ優先探索と幅優先探索を切り替えつつ再帰的に分割し，並列に計算するアルゴリズム CARMA [4] を提案した．CARMA は必要な通信回数が下界に漸近的に一致する通信最適なアルゴリズムであり，キャッシュミスの発生を抑えることで，サイズ $m \times k$ の行列とサイズ $k \times n$ の行列の行列積計算において k の値が m, n に対して十分大きい場合に，特に従来のアルゴリズムより高速な計算を実現している．

これらの特徴から行列積は，多数のプロセッサを持ち並列計算に対して高い性能を発揮する GPU によって大幅な高速化が可能な問題である．GPU は本来画像処理に用いられるプロセッサであるが，汎用計算に用いることも可能であり，その理論性能に対して安価な計算資源である．NVIDIA 社は，線形代数演算ライブラリ BLAS [8] を基に，自社の GPU 向けに高速な線形代数演算ライブラリ cuBLAS [10] を実装した．このうち，行列積の計算は各種 gemm 関数を用いることで計算できる．これらの関数は与えられた行列のサイズに最適な内部ルーチンを呼び出すことによって多くの場合，最大性能の理論値に近い高速な計算を実現している．しかし，出力サイズが小さい行列積計算は，入力サイズより出力サイズが大きい行列積と比べて並列性の確保が難しいため低速な傾向がある．その中で，GPU 向け線形代数演算ライブラリ MAGMA BLAS[9] の開発の一環として，比較的小規模な GEMM 演算の高速化 [1] が行われている．その他にも，多数の出力サイズが小さい GEMM 演算を同時に計算することで高速化を行う [7] といった研究がなされており，GPU が不得意とする小規模な行列積の GPU 実装の研究が行われている．本研究では，入力より出力が十分小さい GEMM 演算について，GPU 向けの新しい行列積アルゴリズムを提案，実装し，cuBLAS と性能比較を行い，高速化の可能性を検証する．提案手法の実装は素朴な実装に加えて命令レベル並列性を考慮した

[1] 大阪府立大学大学院工学研究科
Graduate School of Engineering, Osaka Prefecture Uniersity
[a)] swb01128@edu.osakafu-u.ac.jp
[b)] fujimoto@cs.osakafu-u.ac.jp

© 2016 Information Processing Society of Japan

実装に対して，提案手法による変更を加えたものを用いた．

本論文の構成は以下の通りである．第 2 章で提案手法の前提知識として必要な，GPU と NVIDIA 社の GPU を対象とした GPGPU 向けプログラミング環境である CUDA の説明，GEMM 演算の説明，GPU における行列積の素朴な実装について説明を行う．次に，第 3 章で提案手法を示し，第 4 章で評価実験について述べ，第 5 章でまとめを行う．

2. 準備

2.1 GPU と CUDA

2.1.1 CUDA の概要

CUDA とは，NVIDIA 社が提供する GPU 向けの統合開発環境であり，C 言語と C++言語を拡張した CUDA C/C++という言語により GPU プログラミング向けの機能を記述できる．ここで，CUDA によって操作できる NVIDIA 社製の GPU の構造について説明する．内部に複数のストリーミングマルチプロセッサ (SM) をもち，それぞれが GPU のメモリであるデバイスメモリに接続されている．その SM の中には演算を担当する多数のコアが搭載されている．また，SM の内部にもいくつかのメモリ領域が存在し，そのうち共有メモリは同一 SM 内においてデータを共有することができ，コアに近い位置に存在するため高速にアクセスできる．レジスタはすべてのメモリのうち最も高速にアクセスできる領域であり，各コアが独立した領域を用いる．

2.1.2 CUDA によるマルチスレッドプログラム

CUDA は多数存在するコアが並列にプログラムを実行することによって高速な計算を実現している．並列なプログラム実行の最小単位としてスレッドが存在する．各スレッドにスレッド番号が与えられ，基本的に同じプログラムを実行するが，その番号によって異なるメモリ領域にアクセスする．スレッドはブロックと呼ばれる 3 次元のスレッド集合によってまとめられ，ブロックはグリッドと呼ばれる 3 次元のブロック集合によってまとめられる．これらの変数は $dim3(x, y, z)$ 型の変数を用いて一括で管理できる．2 次元や 1 次元の $dim3$ 型変数を宣言する場合は第 2，第 3 変数を省略することで表現できる．この時，同じブロックに存在するスレッドはすべて同じ SM 内で実行され，共有メモリの確保もブロック単位で行われる．異なるブロックのスレッドとデータの交換を行う場合はデバイスメモリを通じて行う．また，各スレッドはワープというグループでまとめられており，同じワープ内の 32 スレッドが同じ命令を実行するようになっている．ブロック内のスレッドの同期をとる必要がある際には__syncthreads() 関数を用いればよい．

2.1.3 カーネル関数

カーネル関数とは，CUDA において GPU での処理を行う際に呼び出す関数であり，$FunctionName\langle\langle\langle Blocks, Threads\rangle\rangle\rangle(Arguments)$ という構文で呼び出す．$Blocks$ で 3 次元までのブロック数を指定し，$Threads$ で 3 次元までのスレッド数を指定し，$Arguments$ で任意の数の引数を指定する．カーネル関数が呼び出されたとき，指定された数のスレッドが呼び出されたカーネル関数を並列実行する．

2.1.4 atomic 関数

atomic 関数 [11] とは，あるスレッドがグローバルメモリ，または共有メモリ上のあるアドレスに存在するデータの読み込み，修正，書き込みの一連の処理を行う際に，その処理が終了するまで他のスレッドからの同一アドレスへのアクセスを防止できる関数である．atomic 関数は 32 ビット，または 64 ビットの整数型と，一部の関数において単精度浮動小数点型のみ提供されている．そのため，倍精度浮動小数点型で atomic 関数を利用する場合は，atomicCAS() 関数を用いて実装する必要がある．倍精度浮動小数点型における原子性が保証された加算処理を行う atomicAdd() 関数の実装も可能である [11]．

2.2 GEMM 演算

GEMM 演算 [6] は以下のように定義される．

$$
\begin{aligned}
C &\leftarrow \alpha op(A) op(B) + \beta C \\
trans(X) &= \{N, T, H\} \\
op(X) &= \begin{cases} X & (trans(X) = N) \\ X^T & (trans(X) = T) \\ X^H & (trans(X) = H) \end{cases}
\end{aligned}
$$

ここで，A は m 行 k 列の行列，B は k 行 n 列の行列，C は m 行 n 列の行列，α, β はスカラー，X は任意の行列，X^T は行列 X の転置行列，X^H は行列 X のエルミート転置を指す．GEMM 演算は，行列積と同様様々な分野で用いられている．

2.3 GPU における行列積の素朴な実装

以下の 2 つの区分行列を考える．ここでは，区分行列中の小行列は同じサイズの正方行列であるとする．

$$
A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1r} \\ A_{21} & A_{22} & \dots & A_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pr} \end{pmatrix}
$$

$$
B = \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1q} \\ B_{21} & B_{22} & \dots & B_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ B_{r1} & B_{r2} & \dots & B_{rq} \end{pmatrix}
$$

© 2016 Information Processing Society of Japan

このとき，$C = AB$ の計算結果は式 (1) ように表すことができ，各小行列の計算結果 C_{ij} は式 (2) で表せる．

$$C = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1q} \\ C_{21} & C_{22} & \dots & C_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ C_{p1} & C_{p2} & \dots & C_{pq} \end{pmatrix} \quad (1)$$

$$C_{ij} = \sum_{h=1}^{r} A_{ih} B_{hj} \quad (2)$$

このとき，各小行列 C_{ij} の計算は他の小行列の計算に影響を及ぼさず，互いに独立である．ゆえに，それぞれの計算を並列に行える．

行列積のアルゴリズムが多数提案されているのと同様に，GPU における行列積の実装も多数存在する．このうち，基本的な実装の一つが式 (1)，(2) を用いて小行列を並列に計算する方法である．そのアルゴリズムを示す．なお，ここで示すアルゴリズムは列優先で格納された行列を対象としたアルゴリズムであり，Algorithm 1 はカーネル関数，Algorithm 2 は Algorithm 1 を呼び出して行列積を計算する CPU アルゴリズムである．

Algorithm 1，2 を用いて行列積の計算を行った場合，式 (1) における小行列の数と Algorithm 1 で使用するブロック数が一致する．ゆえに，行列積の出力サイズが大きいほど，計算時のブロック数が増加し，高い並列度が得られ，性能が上昇する．しかし，出力サイズが小さい場合ブロック数も減少し，並列度も低下する．すると，ハードウェア資源を十分に使い切ることができなくなり，ハードウェアが持つ最大性能を出すことが困難になる．

3. 提案手法

出力サイズが小さい行列積は，素朴な実装を用いた場合並列性が低い．そのため，出力が小さい状況でも並列性を確保できる方法を考える必要がある．そのために，以下の区分行列を用いた行列積について考える．

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1r} \end{pmatrix} \quad (3)$$

$$B = \begin{pmatrix} B_{11} \\ B_{21} \\ \vdots \\ B_{r1} \end{pmatrix} \quad (4)$$

$$C_h = A_{1h} B_{h1} \quad (5)$$

$$C = \sum_{h=1}^{r} C_h \quad (6)$$

ここでは，A と B の各小行列は C_h を定義可能な行列であるとする．このとき，各 C_h は独立して計算できる．つまり，並列に計算できる．しかし，式 (6) で各 C_h の C への加算を同時に行うと，あるスレッドによる計算結果があるワードに書き込まれる前に，他のスレッドがそのワードにアクセスし，その計算結果を書き込むことで，片方のスレッドの計算結果しか反映されていない状態になる恐れがある．ここで，atomicAdd() 関数を用いれば，C_h の合算を行い C を計算する際に，同一アドレスに複数のスレッドが同時にアクセスすることを防げる．これにより，式 (5) の計算を並列に行いながら，式 (6) を正しく計算できる．これらの式 (5) と式 (6) の性質は，GPU における転置を伴わない GEMM 演算にも適用できる．以上の性質を踏まえ

Algorithm 1: 素朴な行列積の GPU アルゴリズムの GPU 部分

MatrixMultiply
$\langle\langle\langle(\frac{m}{b_s}, \frac{n}{b_s}), (b_s, b_s)\rangle\rangle\rangle(A, B, C, m, n, k, b_s)$

入力: サイズ $m \times k$ の行列 A,
サイズ $k \times n$ の行列 B,
グリッドサイズ (b_x, b_y),
ブロックサイズ (b_s, b_s)
出力: $C = AB$

```
1: tx = threadIdx.x, ty = threadIdx.y
2: bx = blockIdx.x, by = blockIdx.y
3: index = m * (by + ty) + bx + tx
4: 担当成分の計算結果を記憶するために
   領域 temp をレジスタに確保する
5: temp = 0
6: 小行列の A_ih, B_hj 確保のために,
   サイズ b_s × b_s の
   共有配列 S_A[b_s][b_s], S_B[b_s][b_s] を宣言する
7: for t = 0 to k − 1 step b_s do
8:     S_A[tx][ty] = A[bx + tx + (ty + t) * m]
9:     S_B[tx][ty] = A[tx + t + (by + ty) * k]
10:    __syncthreads()
11:    for i = 0 to b_s − 1 do
12:        temp ← temp + S_A[tx][i] × S_B[i][ty]
13:    end for
14: end for
15: C[index] = temp
```

Algorithm 2: 素朴な行列積の GPU アルゴリズム

入力: サイズ $m \times k$ の行列 A,
サイズ $k \times n$ の行列 B,
小行列のサイズ b_s
出力: $C = AB$

```
1: グローバルメモリに A, B, C を記憶する領域を確保する
2: CPU 側のメモリから，A, B をグローバルメモリに
   確保した領域にそれぞれコピーする
3: 二次元のブロック dim3(b_s, b_s) を宣言する
4: 二次元のグリッド dim3(m/b_s, n/b_s) を宣言する
5: MatrixMultiply⟨⟨⟨(m/b_s, n/b_s), (b_s, b_s)⟩⟩⟩
   (A, B, C, m, n, k, b_s)
6: 全てのスレッドの計算が終了するまで待機する
7: グローバルメモリ上の C の計算結果を
   CPU 側のメモリにコピーする
8: グローバルメモリ上に確保した全ての領域を
   解放する
```

© 2016 Information Processing Society of Japan

Algorithm 3: 提案手法による GEMM 演算の概要

入力: サイズ $m \times k$ の行列 A，サイズ $k \times n$ の行列 B，
サイズ $m \times n$ の行列 C，スレッド数 t
小行列のサイズ b_s，分割数 p，スカラー α, β
出力: $C \leftarrow \alpha AB + \beta C$
1: betaC-calcurator$\langle\langle\langle \frac{mn}{t}, t\rangle\rangle\rangle(C, \beta)$
2: alphaAB-calcurator$\langle\langle\langle(\frac{m}{b_s}, \frac{n}{b_s}, p), (b_s, b_s)\rangle\rangle\rangle$
$(A, B, C, m, n, k/p, \alpha)$

Algorithm 4: betaC-calcurator$\langle\langle\langle \frac{mn}{t}, t\rangle\rangle\rangle(C, \beta)$

入力: サイズ $m \times n$ の行列 C，スカラー β，スレッド数 t
出力: $C \leftarrow \beta C$
1: $i = threadIdx.x$
2: $j = blockIdx.x$
3: $index = blockDim.x * j + i$
4: $C[index] \leftarrow \beta C[index]$

たうえで，提案手法のアルゴリズムの概要を Algorithm 3 に示す．なお，ここでは CPU からのデータ転送については省略し，転置を伴わない GEMM 演算の方法のみ示す．ここで，t は任意のスレッド数を表し，分割数 p は式 (3)，(4)，(6) の r と一致する．betaC-calcurator() カーネル関数は $C \leftarrow \beta C$ を計算する．また，alphaAB-calcurator() カーネル関数は $C \leftarrow C + \alpha AB$ を計算する．素朴な実装に GEMM 演算の機能を追加する場合や，MAGMA BLAS[9] の GEMM 関数の一部のカーネル関数においては，小行列 C_{ij} の計算を行った後に $C \leftarrow \alpha AB + \beta C$ を最後に計算する方法を用いることができる．しかし，提案手法においては各小行列に対して複数のブロックが計算を行う．この時，alphaAB-calcurator() カーネル関数の全てのブロックが $C \leftarrow \alpha AB + \beta C$ を計算してしまうと，C の各要素の値はそれまでに加算された数値の合計が用いられてしまうため，実際より大きい計算結果が得られてしまう．これを避けるために，先に βC の計算結果を計算し，終わり次第，小行列 $C_h = \alpha A_{1h} B_{h1}$ の計算結果を C に加算する．betaC-calcurator() カーネル関数と alphaAB-calcurator() カーネル関数のアルゴリズムをそれぞれ Algorithm 4,5 に示す．これらのアルゴリズムは CARMA[4] が k が大きい際に用いている行列積の分割方法から着想を得ている．

Algorithm 5 は，4 段の命令レベル並列性 [14] およびプリフェッチを考慮することで，さらに高速化できる．プリフェッチは Algorithm 5 の 9 行目の for ループの繰り返し中に，次の繰り返しで各スレッドがグローバルメモリにコピーするデータをレジスタにコピーすることで実装できる．これにより若干の性能上昇が得られる．なお，Algorithm 5 は Algorithm 1 を基に記述されているが，Algorithm3 の 2 行目における小行列 C_h の計算方法は任意である．ただし，atomicAdd() 関数を用いるなどして，各 C_h の C への加算を行う際に演算の原子性が保障されなければならない．

Algorithm 5: alphaAB-calcurator
$\langle\langle\langle(\frac{m}{b_s}, \frac{n}{b_s}, p), (b_s, b_s)\rangle\rangle\rangle(A, B, C, m, n, k/p, \alpha)$

入力: サイズ $m \times k$ の行列 A，サイズ $k \times n$ の行列 B，
サイズ $m \times n$ の行列 C，
小行列のサイズ b_s，分割数 p
出力: $C \leftarrow C + \alpha AB$
1: $tx = threadIdx.x, ty = threadIdx.y$
2: $bx = blockIdx.x, by = blockIdx.y$
3: $bz = k/p * blockIdx.z$
4: $index = m * (by + ty) + bx + tx$
5: 担当成分の計算結果を記憶するために
領域 $temp$ をレジスタに確保する
6: $temp = 0$
7: 小行列の A_{ih}, B_{hj} 確保のために，
サイズ $b_s \times b_s$ の
共有配列 $S_A[b_s * b_s], S_B[b_s * b_s]$ を宣言する
8: **for** $t = 0$ to $\frac{k}{b_s} - 1$ **step** b_s **do**
9: $S_A[b_s * ty + tx] = A[bx + tx + (bz + ty + t) * m]$
10: $S_B[b_s * ty + tx] = B[bz + tx + t + (by + ty) * k]$
11: $\_\_syncthreads()$
12: **for** $i = 0$ to $b_s - 1$ **do**
13: $temp \leftarrow temp + S_A[b_s * i + tx] \times S_B[b_s * ty + i]$
14: **end for**
15: **end for**
16: $C[index]$ =atomicAdd$(\alpha * temp)$

4. 評価実験

4.1 実験の設定

提案手法の性能を評価するために，入力サイズより出力サイズが十分に小さい状況における提案手法の性能評価および，cuBLAS との性能比較を行う．

本論文では二つの実験を行った．一つ目の実験においては，提案手法においていくつかの異なるサイズで小行列への分割を行い，性能の変化を追った．二つ目の実験では，提案手法と cuBLAS の性能比較を行った．

本論文では倍精度浮動小数点による実験を行う．それぞれの実験において，いくつか m, n の値に対して k, p の値の変化による提案手法の性能変化を調査する．なお，それぞれの値に対して 50 回の試行を行い，それによって算出された性能の平均値を最終的な性能とした．

4.2 実験環境

今回の実験は，CPU は Intel Core i3-2100(3.10GHz)，GPU は NVIDIA Tesla K20c(単精度 2496 コア，倍精度 832 コア，ピーク性能 3.52TFLOPS(単精度)，1.17TFLOPS(倍精度))，OS は Windows7 Professional 64bit，CUDA のバージョンは 7.5，コンパイラは Visual Studio 2012 Ultimate をそれぞれ用いた．

提案手法は入力サイズに対して出力サイズが十分小さい場合の高速化を目的としている．具体的には，式 (5) と式 (6) を並列に計算している．ここで，最大限高速化する

© 2016 Information Processing Society of Japan

ために，小行列への分割をどの程度行えばよいのか検討する．今回対象とする入力サイズより出力サイズが十分小さい場合，同時に実行されるスレッドブロックの数が減少する．すると，同時に実行可能なブロック数の最大値に届かず，ハードウェア資源を使い切れていない状態になる．この問題を解決するには，少なくとも GPU が同時実行できるスレッドブロックの数以上のブロックを生成すればよい．このとき，提案手法のカーネル関数のブロックあたりのスレッド数を t，共有メモリの使用量を s，利用するレジスタ数を r とする．さらに，SM あたりの共有メモリの容量を S，SM あたりのレジスタ数を R，1 つの SM において同時起動できるスレッド数を T，GPU が持つ SM の数を M，SM で同時起動できるブロック数の最大値を B とすると，GPU が同時に起動できるブロック数は式 (7) のように表せる．

$$M \times min(\lfloor \frac{T}{t} \rfloor, \lfloor \frac{S}{s} \rfloor, \lfloor \frac{R}{rt} \rfloor, B) \qquad (7)$$

なお，式 (7) は GPU のハードウェア資源を最大限に利用するのに必要最低限のブロック数を計算する式である．実際には，メモリアクセス遅延の隠ぺいを行うためにブロック数をさらに多くすると性能が上昇する場合がある．ここで，最低限必要な小行列への分割数を確認するため，Tesla K 20c の仕様のうち，SM あたりの性能の一部を表 1 に示し，提案手法を実装したカーネルの仕様の一部を表 2 に示す．表 1 と表 2 から，式 (7) を用いて提案手法を実装したカーネルの性能を最大化できると考えられる最低限必要なブロック数 B_l を求める．

$$\begin{aligned} B_l &= 13 \times min(\lfloor \frac{2048}{64} \rfloor, \lfloor \frac{49152}{4096} \rfloor, \lfloor \frac{65536}{55 \times 64} \rfloor, 16) \\ &= 156 \end{aligned} \qquad (8)$$

今回用いた実装では，B_l を超えるブロック数を用意すれば，一度に利用可能なハードウェアが持つ資源を使い切れると考えられる．なお，ただ使い切ればよいのではなく，できる限り同時起動できるブロックの数を多くする必要がある．今回の実装の場合，各スレッド，及びブロックには均等に計算が割り当てられている．つまり，各ブロックの実行が同時に開始されたとき，終了するのもほぼ同時になると考えられる．よって，原則最も効率の良いブロック数の理論値は B_l の倍数と考えられる．

表 1 Tesla K20c の仕様

	数値
SM あたりの共有メモリの容量	49152 バイト
SM あたりのレジスタ数	65536
SM で同時に起動できるスレッド数	2048
SM で同時に実行できるブロック数の最大値	16
SM の数	13

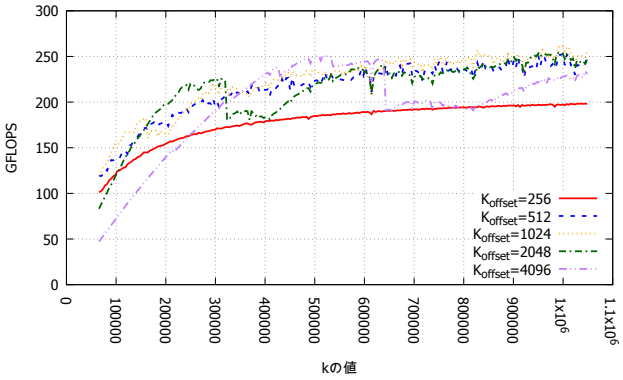

図 1 提案手法の性能例 ($m = 16, n = 16$)

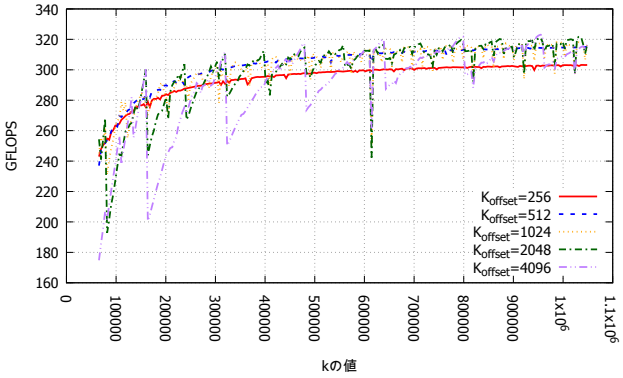

図 2 提案手法の性能例 ($m = 32, n = 32$)

4.3 提案手法の性能

まず，$m = n = 16$ の場合と，$m = n = 32$ の場合に，いくつかの異なるサイズの小行列への分割を行った場合における提案手法の性能の評価を行った．小行列への分割は，行列 A と B の k 方向をサイズ K_{offset} で分割した．また，k の値を $65536(= 2^{16})$ から 4096 ずつ増加させ，$1048576(= 2^{20})$ までの範囲で，それぞれの K, K_{offset} について実験を行った．実験結果を図 1，2 に示す．

表 2 から，提案手法による分割を行わない場合，$m = n = 16$ のときブロック数は 1，$m = n = 32$ のときブロック数は 4 となる．この値に分割数 p をかけると実際に提案手法が起動しているブロック数となる．今回の場合，k の方向に K_{offset} ごとに分割を行っているので，$p = \frac{k}{K_{offset}}$ となる．図 1，2 から，一般的な傾向として，K_{offset} の値が小さいほど性能のぶれが小さくなるが，最大性能は低下し，K_{offset} が大きいほど性能のぶれが大きくなるが，最大性能は高くなる傾向があることが分かった．特に，k の値が小さく，K_{offset} が大きい値である場合は，

表 2 提案手法のカーネルの仕様

	数値
ブロックあたりのスレッド数	64
ブロックあたりの共有メモリ容量	4096 バイト
スレッドあたりのレジスタ利用数	55
ブロックあたりの計算領域	16×16

© 2016 Information Processing Society of Japan

表 3　$m = n = 16$ の実験結果の一部

K の値	K_{offset} の値	p の値	性能 [GFLOPS]
638976	1024	624	236.5
643072	1024	628	246.2
638976	2048	312	238.4
643072	2048	314	230.5
638976	4096	156	243.9
643072	4096	157	191.1

分割数 p の値が式 (8) の値を下回り，ハードウェアが持つ性能を十分発揮できなくなる．K_{offset} が大きい値になるほど，性能のぶれが大きくなるのは，ハードウェアの性質による性能への影響が考えられる．同時に実行できないブロックが存在すると，そのブロックは他のブロックのいずれかが終了してから起動される．このとき，残ったブロックの数が少ないと，ハードウェア資源を使い切れていない期間が生じ，性能が悪化する．特に，K_{offset} の値が大きくなるとブロックが担当する計算が多くなるため，性能の悪化がより顕著になる．その傾向を示す $m = n = 16$ の場合における実験結果の一部を表 3 に示す．表 3 の結果より，K_{offset} の値が大きいとき，p の値を B_l で割った際に余りが生じる場合，他のブロックと同時に実行できないブロックが存在することになり，性能が低下することがわかる．しかし，$K_{offset} = 1024$ の場合はむしろ性能が上昇している．これは，ブロック数が B_l と比較して十分大きく，ハードウェア資源が余っている状態になる期間が全体実行時間に占める割合が少なかったと考えられる．なお，全体として，k の値が大きくなるほど，性能は上昇傾向にあり，K_{offset} の値を大きくしたり，入力サイズが極端に小さくならない限り，性能のぶれは十分小さいと考えられる．

4.4　提案手法と cuBLAS の比較

次に, 提案手法と cuBLAS の倍精度浮動小数点型の行列積を計算する関数である cublasDgemm 関数の性能比較を行った．実験結果のうち例として，$m = n = 16, m = n = 32$ の場合における性能比較の結果を図 3，4 に示す．なお，提案手法の性能は，4.2 節の実験のうち，ブロック数による性能のぶれが小さく，最大性能が比較的高い $K_{offset} = 1024$ を用いた．

図 3，4 の結果より，提案手法を用いることで，入力サイズより出力サイズが十分小さい場合の GEMM 演算を cuBLAS よりも高速に計算できる場合があることが分かる．特に，$m = n = 16$ のとき，最大で 2 倍以上 cuBLAS よりも高速である．しかし，$m = n = 32$ の場合は全ての設定で cuBLAS より高速であるものの，$m = n = 16$ の場合と比較すると大幅な高速化はされていない．この理由として，まず，十分なブロック数さえ確保できれば，出力サイズがより小さい GEMM 演算に対しても，十分な性能上昇が期待できる提案手法では，$m = n = 16$ の場合の性能上昇が相対的に大きくなったと考えられる．もう一つの理由として，cublasDgemm 関数は多数の内部ルーチンをもち，入力サイズに応じて異なる計算方法を用いていることが挙げられる．出力サイズが小さい GEMM 演算の中でも，比較的大きい出力サイズの計算に最適化された内部ルーチンが用いられた結果，$m = n = 16$ の場合と比較して cublasDgemm 関数は高速になったと考えられる．これらの理由から，$m = n = 32$ のとき，$m = n = 16$ のときと比較して，cublasDgemm 関数に対する性能上昇比が低くなったと考えられる．

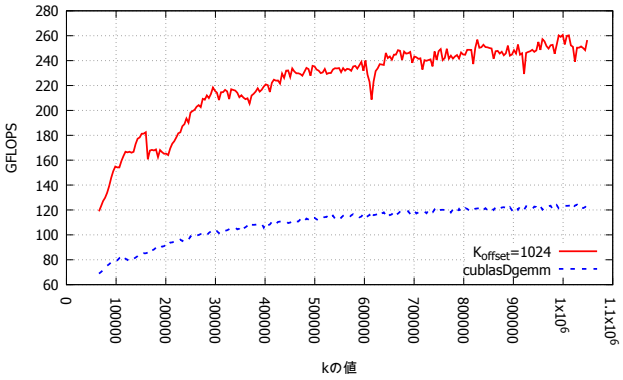

図 3　提案手法と cuBLAS の性能比較 $m = 16, n = 16$

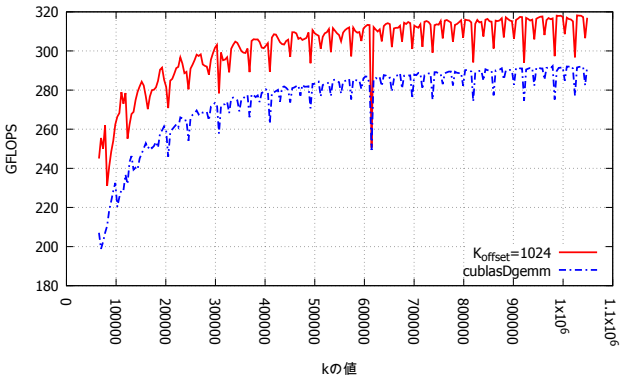

図 4　提案手法と cuBLAS の性能比較 $m = 32, n = 32$

このように，提案手法は cuBLAS と比較して，入力サイズより出力サイズが十分に小さい GEMM 演算に対してより高速な計算を可能にすることが分かる．また，提案手法の最大性能は小行列を計算するカーネルの最大性能に依存する．つまり，今回実装したカーネルよりも高速なカーネルを用いることによって，入力サイズより出力サイズが十分に小さく，転置を伴わない GEMM 演算を高速化できると考えられる．実際に，提案手法の小行列同士の行列積を計算する際に，cuBLAS を用いて行うとハードウェアが持つ限界性能に近い性能を得られる場合があることを確認している．しかし，cuBLAS を用いると，ソースコードが公開されていないため，atomicAdd() 関数を用いて自ら提案手法を実装できなくなるので，各小行列積に対して領域を確保する必要があるなどの好ましくない変更が必要にな

© 2016 Information Processing Society of Japan

る．さらに，cuBLAS の内部ルーチンの中には，並列性の確保を行うために，スレッドブロックを多数起動するように設計されている内部ルーチンが存在し，それが用いられた場合は提案手法による分割を行っても性能はほぼ変化しない，または低速になる．これらの理由から，提案手法と，自ら実装を変更するのが困難なライブラリを組み合わせるのは難しい．よって，提案手法の性能を最大化するためには，より高速かつ atomicAdd() 関数による実装の変更を行えるカーネルの実装が必要である．

5. まとめ

本論文では，GPU において入力サイズより出力サイズが十分小さい GEMM 演算を，高速な GPU 向け線形代数演算ライブラリである cuBLAS と提案手法によって計算した場合の性能を比較，検証し，提案手法が従来の手法より高速に解ける場合があること，提案手法と既存手法を組み合わせることでより高速な行列積計算を行うことができる場合があること，最後に，より高速で，自ら実装を変更できるカーネルを用いることで提案手法の性能を最大化できる可能性があることを示した．

今後の課題は 3 点挙げられる．一つ目は，今回は倍精度浮動小数点による実験のみを行ったため，単精度浮動小数点における提案手法の性能調査を行うことである．二つ目は，今回の実装したカーネル関数は最大性能が低いため，やや大きい出力サイズの問題に対しては十分な高速化が得られなかった．そのため，より高速な実装を行い，提案手法を適用できると判断できる範囲を拡大することが必要であると考える．最後に，今回の実装では行列サイズがブロックサイズで割り切れなければならない制約が存在する．そのため，どのようなサイズの GEMM 演算に対しても適用可能なカーネルを実装し，その性能を調査することが挙げられる．

参考文献

[1] Abdelfattah, A., Haidar, A., Tomov, S. and Dongarra, J.: Performance, Design, and Autotuning of Batched GEMM for GPUs, *International Supercomputing Conference (ISC), LNCS*, Vol. 9697, pp. 21–38 (2016).
[2] Choi, J.: A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers, *International Conference High Performance Computing on the Information Superhighway (HPC Asia)*, pp. 224–229 (1997).
[3] Choi, J., Walker, D. W. and Dongarra, J.: PUMMA : Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers, *Concurrency and Computation : Practice and Experience*, Vol. 6, No. 7, pp. 543–570 (1994).
[4] Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O. and Spillinger, O.: Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication, IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 261–272 (2013).
[5] Fox, G. C. and Otto, S. W.: Matrix Algorithms on a Hypercube I: Matrix Multiplication, *Parallel Computing*, Vol. 4, No. 1, pp. 17–31 (1987).
[6] J. Dongarra, J. Croz, S. H. and Duff, I.: A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, pp. 1–17 (1990).
[7] Jhurani, C. and Mullowney, P.: A GEMM Interface and Implementation on NVIDIA GPUs for Multiple Small Matrices, *Journal of Parallel and Distributed Computing*, Vol. 75, pp. 133–140 (2013).
[8] Lawson, C. L., Hanson, R. J., Kincaid, D. and Krogh, F. T.: Basic Linear Algebra Subprograms for FORTRAN Usage, *ACM Transactions on Mathematical Software*, Vol. 5, No. 3, pp. 308–323 (1979).
[9] Nath, R., Tomov, S. and Dongarra, J.: Accelerating GPU Kernels for Dense Linear Algebra, *International Meeting on High Performance Computing for Computational Science (VECPAR'10), LNCS*, Vol. 6449, pp. 83–92 (2011).
[10] NVIDIA Corp.: cuBLAS, , available from ⟨http://docs.nvidia.com/cuda/cublas/⟩ (accessed 2016-7-5).
[11] NVIDIA Corp.: CUDA C Programming Guide, , available from ⟨http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html⟩ (accessed 2016-7-5).
[12] Strassen, V.: Gaussian Elimination Is Not Optimal, *Numerische Mathematik*, Vol. 13, No. 4, pp. 354–356 (1969).
[13] van de Geijn, R. A. and Watts, J.: SUMMA: Scalable Universal Matrix Multiplication Algorithm, Technical Report 96, LAPACK Working Note (1995).
[14] Volkov, V.: Better Performance at Lower Occupancy, the GPU Technology Conference (GTC) (2010).

© 2016 Information Processing Society of Japan