Vol.2018-HPC-166 No.9
2018/9/28

Level-3 BLASに基づく高精度行列積計算法による高精度かつ再現性のあるBLASルーチンの実装とその最適化

椋木 大地[1,a)]　荻田 武史[2,b)]　尾崎 克久[3,c)]

概要：浮動小数点演算による丸め誤差は，計算結果の正しさ（accuracy）と再現性（reproducibility）に影響を与え，数値計算の信頼性を失わせる原因となりうる．そのため高精度かつ再現可能な演算をサポートした数値計算ライブラリの実現が求められている．本研究では基本線形代数演算を提供する Basic Linear Algebra Subprograms（BLAS）の内積（DOT），行列ベクトル積（GEMV），行列積ルーチン（GEMM）を，尾崎らが提案した level-3 BLAS に基づく高精度行列積計算法（尾崎スキーム）によって高精度かつ再現可能にする際の高性能実装手法を検討した．省メモリ化と高速化のためのいくつかの既提案手法に加え，本稿ではメモリ律速な DOT・GEMV を高速化するための新手法を検討し，NVIDIA GPU において実装を行った．Volta アーキテクチャの Titan V における性能評価では，最適化手法を適用した実装が期待される理論性能の 8 割程度を達成できることを確認した．

1. はじめに

有限桁で計算を行う浮動小数点演算には丸め誤差が存在し，科学技術計算のスタンダードとして用いられている IEEE754 の binary64 倍精度であっても，時に計算結果の精度に深刻な影響をもたらすことがある．加えて，浮動小数点演算においては数学的結合法則が成り立たず，計算順序が異なると計算結果が丸め誤差のレベルで異なる可能性がある．この問題は例えば総和を並列に計算するときに，プログラムの並列度によって加算順序が異なったり，アトミック加算を用いる場合などに起こりうる．このような計算結果の再現性（reproducibility）の欠如は，プログラムのデバッグや数値実験の検証・再現を困難にしている．丸め誤差に起因する精度低下と再現性の欠如の問題は，計算スケールの大規模化にともない，丸め誤差の蓄積によって今後さらに深刻な問題となる可能性があり，エクサスケールコンピューティング時代に向けて解決すべき課題の一つであると考えられる．

このような背景から，計算結果の高精度化や再現性を保証するための計算手法，そして数値計算ライブラリにおける実装手法が研究されている．線形代数演算の基本ライブラリの一つである Basic Linear Algebra Subprograms（BLAS）[1] については，その次世代規格において高精度演算および再現性のある演算のサポートが盛り込まれている [2]．また，これまでには XBLAS[3] や MBLAS[4] といった高精度演算をサポートする BLAS の実装が行われているほか，一部ライブラリにおいて再現性を考慮した実装がサポートされている．さらに近年では，高精度と再現性の両方をサポートした ExBLAS[5] などの実装も提案されている．しかし通常の浮動小数点演算に対する計算速度のオーバーヘッドや実装の難しさなど，解決すべき実装上の課題が数多く存在する．

本研究では，尾崎らが提案した level-3 BLAS に基づく高精度行列積計算手法（尾崎スキーム）[6] による，高精度かつ再現可能な DOT（内積），GEMV（行列ベクトル積），GEMM（行列積）の高性能実装手法を検討する．本研究で実装する BLAS ルーチンは，インタフェースは既存の BLAS ルーチンと同様に入出力が倍精度浮動小数点型であるが，計算結果の精度をある程度の粒度で調整可能で，correct-rounding も達成可能である．また同一のアルゴリズムによる実装であれば，たとえ異なるアーキテクチャや異なるスレッド数，プロセス数の並列実行であっても，必ずビットレベルで一致した演算結果を返すことができる．

尾崎スキームを用いる最大の利点はその開発コストにある．似た機能を実現できる既存の実装方式は，行列積であれば行列積の計算コードをフルスクラッチで実装しなければならないが，尾崎スキームでは計算の中核部分に既存の

[1] 東京女子大学 現代教養学部
[2] 東京女子大学 理学研究科
[3] 芝浦工業大学 システム理工学部
[a)] mukunoki@lab.twcu.ac.jp
[b)] ogita@lab.twcu.ac.jp
[c)] ozaki@sic.shibaura-it.ac.jp

BLAS ルーチン（例えば Intel MKL などの最適化された BLAS 実装）を活用できる．一方で尾崎スキームのナイーブな実装ではメモリ使用量が大きく，メモリ使用量の削減が課題となる．また，尾崎スキームは内積に基づく処理に適用でき，level-1 から 3 までの BLAS ルーチンのうち，内積，行列ベクトル積，行列積系統のルーチンを実装できるが，性能がメモリ律速となる処理においてはメモリアクセスの削減が性能向上の鍵となる．本稿では尾崎スキームを用いて level-1 から 3 までのルーチンを実装することを目的として，省メモリ化および省メモリアクセス化を実現するためのいくつかの最適化手法を検討し，GPU における実装において，その性能を議論する．

2. 関連研究

浮動小数点演算の高精度化は，浮動小数点型が表現できる桁数をソフトウェア的に拡張する，いわゆる多倍長演算による方法が古くから行われている．この方法による BLAS 実装としては，例えば Dekker[7] による二倍長の浮動小数点演算（いわゆる double-double 演算）のアイディアに基づく XBLAS[3] や，double-double や MPFR[8] 等の多倍長演算ライブラリを用いて BLAS/LAPACK を高精度化した MPACK[4] が存在する．また，演算結果の再現性を保証するもっとも古典的な方法は計算順序を固定する方法であり，Intel は Conditional Numerical Reproducible[9] と呼ばれるオプションを，自社の BLAS を含む数値計算ライブラリである MKL でサポートしているほか，NVIDIA の GPU 向け BLAS である cuBLAS[10] においては，一部のルーチンについて atomicAdd 命令の使用を避けることにより計算順序を固定したルーチンを提供している[*1]．

一方で，浮動小数点演算における演算精度と再現性の問題はともに丸め誤差に起因する問題であるため，この二つを同時に対処可能な実装方法も提案されている．ExBLAS[5] は correct-rounding となるまで高精度に演算を行うことにより，丸め誤差の影響を完全に排除して演算結果の再現性を保証している．OpenCL で実装されており OpenCL に対応した GPU においても動作する．RARE-BLAS[11] も同様に correct-rounding を達成するが，level-1/2 の一部の BLAS のみを実装している．ReproBLAS[12] は精度のコントロールが可能な level1–3 の実装を提供しているが，現状では CPU 向けのシングルコア実装のみである．これらの実装に共通する問題点は，いずれも複雑なコーディングを必要とし，BLAS 演算そのものをフルスクラッチで実装する必要があるため開発コストが大きく，かつ性能が実装方法に大きく依存してしまう点にある．

[*1] 対称行列ベクトル積（SYMV）ルーチンにおいて，atomicAdd 命令を用いて高速に計算できるが実行毎の結果の再現性を保証しない実装（atomic mode）と，再現性を保証するがそれよりも低速なルーチンの 2 種類を実装している

アルゴリズム 1 尾崎スキームによるベクトル分割

```
function ((x_split[s_x]) ← Split(x, n))
    ρ := ceil((log2(u^{-1}) + log2(n + 1))/2)
    μ := max_{1≤i≤n}(|x_i|)
    j := 0
    while (μ ≠ 0) do
        j := j + 1
        τ := ceil(log2(μ))
        σ := 2^{(ρ+τ)}
        x_split[j] := fl((x + σ) − σ)
        x := fl(x − x_split[j])
        μ := max_{1≤i≤n}(|x_i|)
    end while
    s_x := j
end function
```

本研究で取り上げる尾崎スキームは，計算の中核部分に既存の BLAS ルーチンを活用できるため，実装コストを大幅に削減できることが期待できる．これまでの実装例としては，尾崎らによる論文 [6][13] において，Core 2 Duo の 1 コアにおける MATLAB による実装の性能が示されている．また，その後片桐らのチームとともに高性能実装に関するいくつかの研究が行われている．片桐らによる研究 [14] では T2K オープンスパコン（東大版）においてマルチスレッド化および自動チューニング化が行われている．市村らによる研究 [15] では富士通 FX100 において，分割行列の疎行列化に関していくつかの並列化方式および異なる疎行列フォーマットを用いた場合の性能を評価している．石黒らによる研究 [16][17] では CPU および GPU において batched BLAS を用いた高速化を試みている．

3. 尾崎スキーム

尾崎スキームによって 2 本のベクトル $x \in \mathbb{F}^n$, $y \in \mathbb{F}^n$ の高精度内積 $x^T y$ を求める方法を示す．$\mathbb{F}$ を浮動小数点数の集合, $\mathrm{fl}(\cdots)$ を最近接偶数丸めによる浮動小数点演算, $\mathbf{u}$ を浮動小数点数の丸め単位（the unit round-off, IEEE 754 binary64 倍精度の場合 2^{-53}）とする．x, y をアルゴリズム 1 により，それぞれ $x = x^{(1)} + x^{(2)} + \cdots + x^{(s_x)}, s_x \in \mathbb{N}, x^{(p)} \in \mathbb{F}^n$, $y = y^{(1)} + y^{(2)} + \cdots + y^{(s_y)}, s_y \in \mathbb{N}, y^{(q)} \in \mathbb{F}^n$ と分割して，$x^T y = (x^{(1)} + x^{(2)} + \cdots + x^{(s_x)})^T (y^{(1)} + y^{(2)} + \cdots + y^{(s_y)})$ と計算する．分割したベクトル同士の内積の総和計算には，何らかの高精度総和アルゴリズムが必要となる．ここで correct-rounding な計算手法（例えば NearSum[18] アルゴリズム）で計算することにより，$x^T y$ は correct-rounding が達成され，計算結果は再現可能となる．

アルゴリズム 1 により分割されたベクトルは次のような 2 つの特徴を持つ．まず，(1) $x^{(p)}(i)$ および $y^{(q)}(j)$ がゼロでないときに，$|x^{(p)}(i)| \geq |x^{(p+1)}(i)|$ および $|y^{(q)}(j)| \geq |y^{(q+1)}(j)|$ となる．したがって，下位ビット情報を持つ分割ベクトルからいくつかを切り捨てることで，得られる精度を調節することができる．なおこの場合も再現性は維持

される．次に，(2) $(x^{(p)})^T y^{(q)} = \mathrm{fl}((x^{(p)})^T y^{(q)}), 1 \leq p \leq s_x, 1 \leq q \leq s_y$ となる．これは内積に対するいわゆる無誤差変換であり，$\mathrm{fl}((x^{(p)})^T y^{(q)})$ において丸め誤差が生じないことを意味する．したがって，倍精度演算を想定したときに，分割ベクトル同士の内積は BLAS の DDOT ルーチンで計算することができる．

尾崎スキームにおいて必要なベクトルの分割数は，入力ベクトルに含まれる要素の範囲に依存する．言い換えると一定の分割数で得られる精度は入力ベクトルに依存することになる．また，内積で構成できる行列ベクトル積および行列積についても，同様の方法を適用することができる．行列ベクトル積および行列積の場合，分割したベクトル・行列の計算には DGEMV，DGEMM を用いることができる．

4. 実装最適化手法

尾崎スキームに対して適用できる実装最適化手法を説明する．以下，行列積 $C = AB$ において行列 A, B, C がそれぞれ $m \times k$，$k \times n$，$m \times n$，行列 A, B の分割数（分割回数ではなく生成された分割行列数）を s_A, s_B とする．また行列ベクトル積 $y = Ax$ については行列 A が $m \times n$，ベクトル x, y はそれぞれ長さ n, m であり，行列 A およびベクトル x の分割数はそれぞれ s_A, s_x とする．内積 $r = x^T y$ についてはベクトル x, y が長さ n であり，分割数はそれぞれ s_x, s_y とする．

4.1 行列のブロッキングによる省メモリ化

尾崎スキームによる行列積では分割行列およびその計算結果を格納するために $s_A mk + s_B kn + s_A s_B mn$ のメモリが必要である．m, n についてブロックサイズ $b (0 < b \leq m, 0 < b \leq n)$ でブロッキングすることにより，メモリ消費を $(s_A + s_B)bk + s_A s_B b^2$ に削減できる [19]．ブロッキングを適用した尾崎スキームによる行列積計算の概要を**図 1** に示す．ブロッキングを適用することで，分割行列の格納の他に，行列分割の際に必要となる作業領域も節約できる．また行列ベクトル積においても同様に行列をブロッキングすることでメモリを節約できる．

ブロッキングの副作用としてブロックサイズを小さくしすぎると，行列分割や計算に用いる BLAS ルーチンの実行効率が低下する可能性がある．ブロッキングは m, n 方向に対して適用できるが，メニーコアプロセッサにおける GEMM では m, n が小さい時に十分な性能が得られない実装が多い．これは一般的に GEMM の並列化が m および n 方向に対して 2 次元のデータ並列性を利用して行われるためである．したがって，ブロッキングによる省メモリ化と性能にはある程度のトレードオフがあると言える．なるべく性能が低下しないぎりぎりのブロッキングサイズを自動チューニングなどで特定することが望まれる．一方で，次に示す batched BLAS の活用によって性能低下を抑制できる可能性がある．

4.2 Batched BLAS の適用による高速化

Batched BLAS[20] は複数の同一 BLAS 演算を一つのルーチンで同時に処理するためのインタフェースである．メニーコアプロセッサにおける計算では，問題サイズが物理コア数あるいはスレッド数に対して不足する場合に実行効率が低下してしまう．このようなケースにおいてプロセッサの演算能力を活用するための方策として，batched BLAS が提案された．x86 CPU においては Intel MKL，NVIDIA GPU においては NIVIDIA cuBLAS やテネシー大学の MAGMA[21] などのライブラリが，GEMM 等の一部 BLAS ルーチンのバッチ版を提供している．

尾崎スキームで行列積を計算するときには分割行列に対して最大で $s_A s_B$ 回の GEMM による計算が発生する．これらの処理間にデータ依存性はないため，batched GEMM を用いてまとめて処理することが可能である．Batched BLAS を用いた実装は文献 [16] で報告されており，問題サイズが小規模の時の効果が示されているが，この事例では 4.1 節で示したブロッキングは行われていない．ブロッキングを行うと行列が細長くなるため，ブロッキングを行わない場合と比べて batched BLAS が有効なケースは多くなると予想される．

なおこの手法は行列積以外のケースに対しても利用可能であるが，現時点で主な batched BLAS は DOT や GEMV のバッチ版ルーチンを提供していない．また DOT や GEMV の場合は，次に示す GEMM を用いて計算を行う方法がより効果的である．

4.3 DOT・GEMV における計算の GEMM 化

DOT または GEMV の計算部分において，分割された複数本のベクトルを一つの行列として扱うことにより，GEMM を用いて計算可能となる．DOT において $s_x s_y$ 回の DOT を 1 回の GEMM に，GEMV において $s_A s_x$ 回の GEMV を s_A 回の GEMM に置き換えることが可能である．このとき問題サイズに対してベクトルの分割数が十分に小さければ，GEMM の性能はメモリ律速となることが期待でき，データの再利用性が高められることによる性能向上が期待できる．

なお，4.1 節で述べたように，特にメニーコアプロセッサにおいては，m, n が小さな行列に対して十分な性能が得られない GEMM 実装が多い．DOT や GEMV の計算を GEMM 化すると，通常は m, n が小さな計算となるため，GEMM の実装次第では，GEMM 化を適用してもかえって性能が低下する可能性がある．実際に本研究で GPU において DOT を実装する際に，cuBLAS の GEMM を用いたところこの問題が生じたため，独自にチューニングした

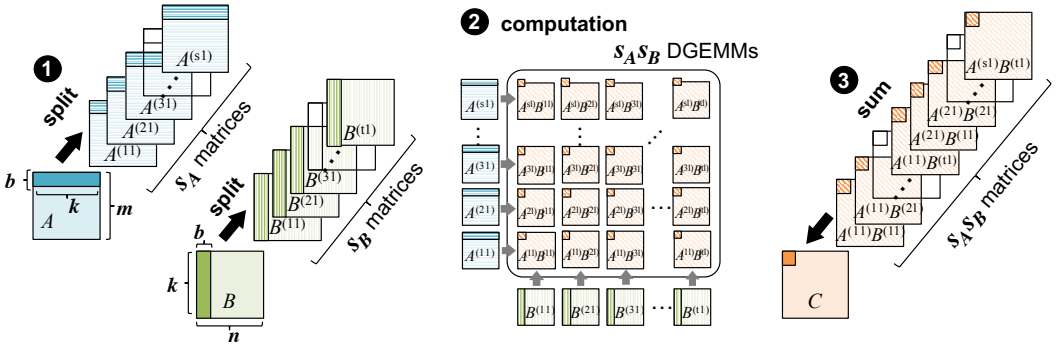

図 1: ブロッキングを行った尾崎スキームによる行列積

GEMM を用いて実装している．

4.4 計算の簡略化（Fast モード）

Correct-rounding の結果が不要で，あらかじめ分割数を指定して計算を行う場合に，分割ベクトル（あるいは行列）同士の計算，$(x^{(p)})^T y^{(q)}$ において，$p+q > \max(s_x, s_y)+1$ の計算は，演算精度への影響がわずかであるため省略することができる [22]．$(x^{(p)})^T y^{(q)}$ の計算回数は $s_x s_y$ から $s_x s_y - \min(s_x, s_y)(\min(s_x, s_y)-1)/2$ に削減できる．これにより若干の精度低下が生じるものの，correct-rounding が不要なケースで，指定した分割数分の厳密な精度を必要とするケースは考えにくく，性能とのトレードオフを考えればこの方法は効果的であると言える．例えば $s_x s_y = s$ とするとき，$s \geq 4$ であれば，省略せずに計算するよりも s を 1 つ大きくして省略して計算した方が，高速かつ高精度に計算できる．

なお，DOT や GEMV においては，4.3 節で述べた計算の GEMM 化を行うと，実装上この方法の適用が難しくなる上，性能がメモリ律速であるため GEMM 化することによりメモリ参照量が最小化されることの方が性能上意義があるため，この方法は不要である．

4.5 分割行列の 32 ビット整数化（INT 化）

この方法はメモリ律速なルーチン向けの手法である．尾崎スキームを倍精度浮動小数点で実装するときに，分割ベクトルあるいは行列（アルゴリズム 1 の 8 行目 x_{split}）は値をシフトすることにより 32 ビット整数型（int32_t）に格納可能である．分割ベクトルのすべての要素は少なくとも下位 26 ビットが 0 であり，符号＋指数部の先頭 12 ビットを除くと，仮数部の有効ビット情報は 26 ビットである．$|x_{split}| \leq \sigma 2^{-\rho}$ であるから，指数部シフト量を $\alpha = \mathrm{floor}(\log2(2^{31}-1)) - \mathrm{ceil}(\log2(\sigma 2^{-\rho}))$（$2^{31}-1$ は 32 ビット整数型が表現できる最大値）として，$x'_i = 2^{\alpha} x_i$ と変換すると 32 ビット整数型に収まる．また α を保持し，$x_i = 2^{-\alpha} x'_i$ で元の値に復元することができる．この方法によって，分割データの書き出しと，計算部分において分割データを読み込む箇所において，メモリアクセス量をほぼ半分にできる．しかし 32 ビット整数型（および指数部シフト量）を読み込んで，レジスタ上で倍精度浮動小数点型に復元して計算するための専用の計算カーネルを実装する必要があり，尾崎スキームの最大の利点である既存の倍精度 BLAS ルーチンの活用が不可能となる．しかしこの方法の効果が期待できる level-1/2 ルーチンは，level-3 ルーチンと比べると高性能実装は比較的容易であるから，もし高速な計算カーネルをフルスクラッチで実装することができれば，一つの最適化手法として検討の余地がある．

4.6 分割行列の疎行列化

本稿では他の最適化手法にフォーカスしたためこの方法は取り上げないが，分割行列は入力行列の要素の分散が大きいほど疎性（ゼロ要素）が現れやすくなるため，ある程度の疎性が現れた場合に Compressed Sparse Row（CSR）形式等の疎行列フォーマットに行列を変換して演算することで，メモリ消費と演算量を削減することができる [6]．ただし計算には行列積の場合には疎行列積ルーチン，行列ベクトル積の場合には疎行列ベクトル積ルーチンが必要となるほか，密行列から疎行列への変換も必要である．

4.7 その他

アルゴリズム 1 において，11 行目の $\max_{1\leq i\leq n}(|x_i|)$ は，10 行目の x の計算と同時にレジスタ上で行うように実装するとメモリアクセスを省くことができる．またあらかじめ分割数を決めて分割を行う場合には，while ループの最終反復において 10 行目と 11 行目の処理を省くことができる．

なお，本稿では以降，図や表などにおいて，ブロッキングの適用：BK，batched BLAS の適用：BB，計算の GEMM 化：MM，計算の簡略化：FS，分割行列の INT 化：IN と表現する．

5. 理論性能

ここでは前節で述べた最適化手法（分割行列の疎行列化を除く）を適用して実装した場合に期待される理論性能を見

表 1: DOT におけるベクトル参照回数（MM：GEMM 化，IN：INT 化，※ 2：本稿では未実装）

	Split	Computation
DDOT	–	2
ExDDOT	$3(s_x + s_y)$	$2s_x s_y$
ExDDOT (MM)	$3(s_x + s_y)$	$s_x + s_y$
ExDDOT (MM+IN ※ 2)	$2.5(s_x + s_y)$	$0.5(s_x + s_y)$

表 2: GEMV における行列参照回数（MM：GEMM 化，IN：INT 化）

	Split	Computation
DGEMV	–	1
ExDGEMV	$3s_A$	$s_A s_x$
ExDGEMV (MM)	$3s_A$	s_A
ExDGEMV (MM+IN)	$2.5s_A$	$0.5s_A$

積もる．以降，本稿では高精度かつ再現性のある演算に対応した DOT，GEMV，GEMM の 3 ルーチンを，ExBLAS[5] に倣ってそれぞれ ExDDOT，ExDGEMV，ExDGEMM と呼ぶ．

5.1 DOT および GEMV

DOT・GEMV はベクトル，行列の分割と最後の高精度総和部分も含めて，すべて $O(n)$ のデータ参照に対して $O(n)$ の演算であるため，ハードウェアの Bytes/Flop 比が十分に小さい近年のプロセッサにおいては性能がメモリ律速となることが期待できる．そのため DOT の場合はベクトルの参照回数，GEMV の場合は行列の参照回数に基づいて性能を見積もる．DOT における各実装のベクトル参照回数を**表 1**，GEMV における各実装の行列参照回数を**表 2** に示す．倍精度 BLAS ルーチンにおける参照回数を 1 とすれば，尾崎スキームによる各実装の相対実行時間を見積もることができる．計算部分のコストは，4.3 節で述べた GEMM 化により，一度アクセスしたデータはすべて再利用される前提（いわばキャッシュが無限にある状態）で議論している．実際に，ベクトルの分割数それほど多くなければ，キャッシュやレジスタの豊富な現代的なプロセッサにおいて，このような前提で性能を見積もることは非現実的なものではないと考えられるが，あくまで GEMM の実装とハードウェア依存であり，理想的な場合の性能見積もりである．

5.2 GEMM

GEMM の場合，BLAS による計算部分において $O(n^3)$ の演算が生じる以外は $O(n^2)$ の演算およびメモリ参照であるため，性能は演算律速となり，性能は BLAS が呼ばれる回数でモデル化できる．行列積 $C = AB$ において行列 A および B の分割数がそれぞれ s_A および s_B であるとき，通常の実装では $s_A s_B$ 回の GEMM が呼ばれるが，4.4 節の計算の簡略化を行った場合は，$s_A s_B - \min(s_A, s_B)(\min(s_A, s_B) - 1)/2$ 回の GEMM が呼ばれる．

6. 実装

本稿では尾崎スキームによる BLAS を OzBLAS（仮称）と呼び，DOT，GEMV，GEMM の 3 ルーチンを実装した．実装プラットフォームとして NVIDIA の Volta アーキテクチャ GPU を用い，CUDA により実装したが，本稿で示した最適化手法はいずれも Intel x86 CPU においても実装できる．なお，現時点での実装上の制約として，GEMV と GEMM の引数に指定できるスカラー値（α, β）は $\alpha = 1.0$, $\beta = 0.0$ に限定されているほか，転置モードや incx, incy 等の対応はまだ実装されていない．

ベクトルおよび行列の分割ルーチンはアルゴリズム 1 に基づいて実装したが，アルゴリズム 1 のままでは入力の x が破壊されるため，これを防ぐために一度メモリ上の作業領域に配置してから分割を行っている．そのほか，分割データの格納などに必要なメモリ上の作業領域は，BLAS ルーチンが呼ばれる度に確保・解放するとオーバーヘッドとなるため，OzBLAS の初期化ルーチンを用意し，その中で一つの大きな作業用メモリを確保して，個々のルーチン内でその先頭アドレスから必要な作業用メモリ領域を切り出して使用するようにした．計算部分には基本的には cuBLAS を使用しているが，後述するように GEMM 化および INT 化を行うために一部のカーネルは独自実装している．cuBLAS の初期化（cublasCreate）は OzBLAS の初期化ルーチン内で行う．また，最後の高精度総和計算には，correct-rounding な結果を返す総和アルゴリズムである NearSum[18] を用いた．

4 節に示した最適化方法（分割行列の疎行列化を除く）については以下のように実装を行った．

- 行列のブロッキングによる省メモリ化：ブロッキングのブロックサイズは理論性能のおおむね 80%程度を達成できる最小サイズを実験的に決定した．GEMV の場合 $b_{max} = 5120$，GEMM の場合 $b_{max} = 1024$ をブロックサイズの最大値として用い，$n \times n$ の行列に対してブロッキング後のサイズは $b \times n$，ただし $b = \lceil m/\lceil m/b_{max} \rceil \rceil$ とした．
- Batched BLAS の適用による高速化（BB）：cuBLAS の cublasDgemmBatched を用いた．
- DOT・GEMV における計算の GEMM 化（MM）：cuBLAS の cublasDgemm は 4.3 節に述べたように m, n が小さな行列に対して性能が遅く，DOT では GEMM 化せず DDOT で計算した方が高速であった．そこで m, n が小さな行列に最適化した DGEMM を独自に実装して用いた（実装自体は DOT の実装を複数本ベクトルの計算に拡張した形である）．なお GEMV の場合は cublasDgemm で十分な性能が得られたため，

表 3: 最近接偶数丸めの結果と比較した最大相対誤差（GEMM, $n = m = k = 1000$，FS：計算の簡略化（Fast モード））

ϕ	cuBLAS	OzBLAS					
	(double prec.)	$s = 2$	$s = 2$ (FS)	$s = 3$	$s = 3$ (FS)	$s = 4$	$s = 4$ (FS)
0	3.27E-09	0	9.39E-08	0	0	0	0
1	4.19E-10	3.37E-05	2.23E-04	1.28E-11	6.47E-10	0	2.07E-16
2	6.99E-11	2.02E-01	5.42E-01	1.60E-07	5.12E-07	9.76E-14	8.34E-13
4	7.33E-11	9.62E+04	1.99E+04	4.82E+02	3.14E+02	4.95E-04	2.09E-03
8	5.84E-12	1.33E+04	1.19E+04	2.76E+04	2.64E+04	1.78E+03	2.81E+03

cublasDgemm を用いた．

- 計算の簡略化（Fast モード：FS）：DOT・GEMV は GEMM 化により不要であるため GEMM のみに適用した．
- 分割行列の 32 ビット整数化（INT 化：IN）：本研究では GEMV の行列のみに適用した．32 ビット整数型の行列を計算する GEMM カーネルは，椋木らによる高性能 GEMV の実装 [23] を複数本のベクトルの同時計算ができるように拡張する方法で実装した．演算は 32 ビット整数型をレジスタ上で binary64 倍精度型に変換してから計算している．
- その他：4.7 節に述べたようにメモリアクセスが最小となるように実装した．

7. 評価実験

7.1 評価方法

NVIDIA Titan V（Volta アーキテクチャ，compute capability 7.0）を用いた．Titan V は 5120 個の CUDA Core（80 マルチプロセッサ，マルチプロセッサあたり 64 コア），12288MB の HBM2 メモリを搭載し，倍精度理論ピーク演算性能 7449.6 GFlops，理論メモリバンド幅 652.8GB/s である．使用した CUDA バージョンは 9.1，ドライババージョンは 390.67，ホストマシンは CPU：Intel Xeon W-2123 (3.6GHz, 4 コア), OS：CentOS Linux release 7.4.1708 (3.10.0-693.2.2.el7.x86_64) である．コンパイラは gcc 4.8.5 20150623, nvcc release 9.1, V9.1.85 で，nvcc のコンパイルオプションは “-arch=sm_70 -O3” とした．

入力ベクトル・行列の初期化は尾崎らの論文 [6] に示されている方法と同様に，$(\texttt{rand} - 0.5) \times \texttt{exp}(\phi \times \texttt{ceil}(\texttt{randn})$（rand は (0,1) の一様乱数，randn は標準正規分布の乱数）として，ϕ が大きいほど浮動小数点数の絶対値のレンジが大きくなるように初期化した．データはすべてグローバルメモリ上に配置されている状態で実行し，DOT では内積結果がグローバルメモリ上に返されるモード（cuBLAS の CUBLAS_POINTER_MODE_DEVICE に相当）とした．

7.2 演算精度の評価

GEMM において $n = m = k = 1000$ の行列に対して演算精度を評価した．比較対象として，MPFR[8] を用いて実装されている MPACK/MBLAS[4] のルーチンを使用し，2048 ビットで計算を行った．OzBLAS の計算結果は倍精度浮動小数点数で得られるため，MBLAS の計算結果を MPFR の mpfr_get_d で MPFR_RNDN（最近接偶数丸め）を指定して倍精度に丸めて比較している．比較対象の計算結果を $C = (c_{ij})$，MBLAS（MPFR）による結果を $R = (r_{ij})$ としたときに，最大相対誤差 $e = \max_{1 \le i,j \le n}(|(c_{ij} - r_{ij})/r_{ij}|)$ を求めた．**表 3** に結果を示す．最大相対誤差が 0 となったケースは correct-rounding が達成されていることを意味する．

7.3 性能の評価

分割数が $s = 2 - -4$ の場合の OzBLAS の性能と，比較対象として cuBLAS の倍精度ルーチンの性能を評価した．ルーチンの実行時間を測定し，性能を内積，行列ベクトル積，行列積のそれぞれの数学的な四則演算回数を実行時間で割った値で評価した．例えば $n \times n$ の行列積の場合，実行時間が t[sec] であるとき，$2n^3/t$ である．単位は通常の倍精度演算の場合 Flops で表されるが，尾崎スキームの場合は実浮動小数点演算回数とは異なるため，本稿では Ops（operations per second）と表記する．なお入力データは $\phi = 4$ で初期化し，最低でも必要分割回数が 4 回以上となる条件で測定している[*2]．

Flops・Ops による性能測定結果を図 2 に，**図 3** には図 2 の結果の $s = 2$ の場合（GEMV は INT 化非適用）の性能内訳を示す．DOT・GEMV はメモリ律速であるため行列分割コストが多くを占めているが，GEMM は演算量が大きく行列分割コストはほぼ見えない．GEMM において行列 A と B の分割性能が異なるのは実装上の問題である．GEMV において $n = 6144$ で性能が低下するのはブロッキングにより計算サイズが小さくなるためである．

表 4 には測定した最大問題サイズにおける性能について，cuBLAS の倍精度ルーチンの性能を 1 としたときの実行時間および 5 節での議論に基づく期待される理論性能との比較を示す．cuBLAS の倍精度ルーチンの性能は，DGEMM で 7125GFlops（理論ピーク演算性能比約 96%），メモリ律速なルーチンについては DDOT で 82.3GFlops

[*2] 7.2 節に示したように $\phi = 4$ では分割数が小さい時に必ずしも通常の倍精度演算と比べて高精度ではない

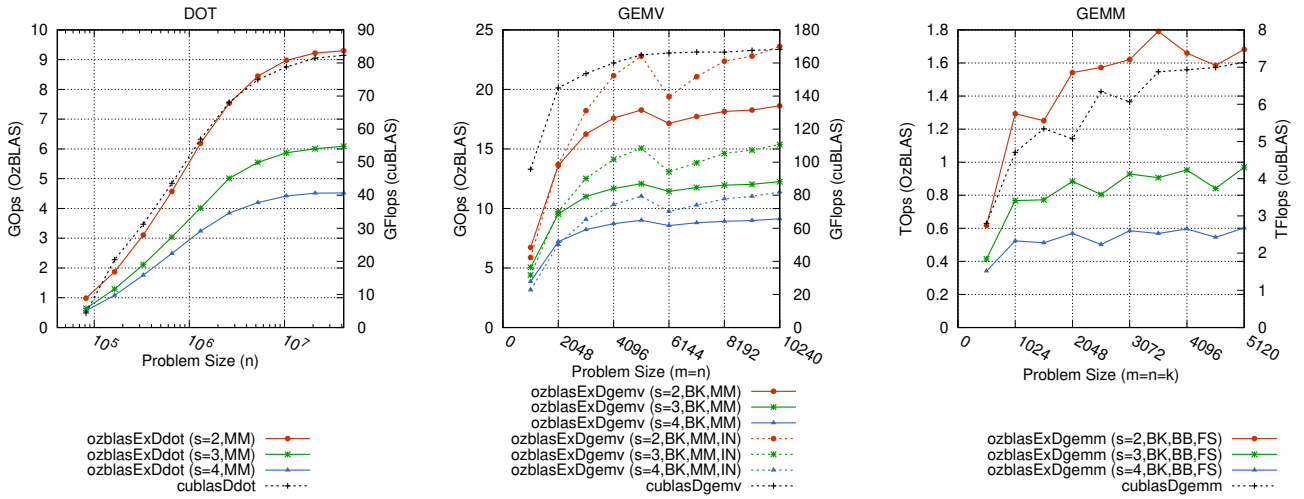

図 2: Titan V における性能（s：分割回数，BK：ブロッキングの適用，BB：batched BLAS の適用，MM：GEMM 化，FS：計算の簡略化，IN：INT 化．注：cuBLAS の性能は右縦軸で示されている）

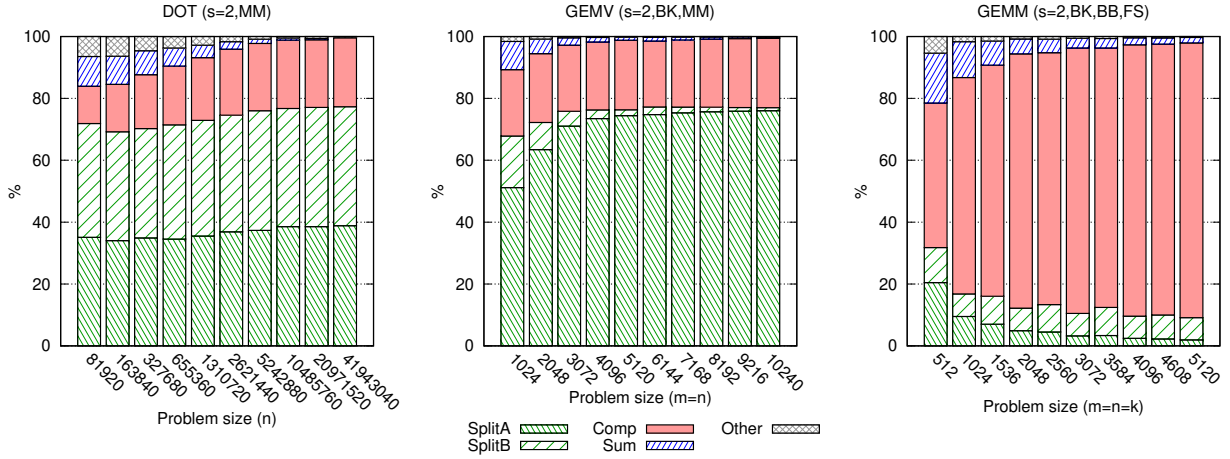

図 3: 分割数 $s = 2$ のときの性能内訳（図 2 の結果に対応，ただし GEMV は INT 化非適用．注：凡例の SplitA，SplitB は，DOT では SplitX，SplitY，GEMV では SplitA，SplitX を意味する）

(658.6GB/s，理論ピークメモリバンド幅比約 101%[*3])，DGEMV で 168.2GFlops（672.8GB/s，理論ピークメモリバンド幅比約 103%）であり，ハードウェアの理論ピーク性能に近い性能が得られている．これらの性能を基準として我々の実装は，GEMM の $s = 2$ で 71%，それ以外では 81%以上の性能を達成している．GEMM の $s = 2$ についても，$n = 3584$ のときに最大 1.79TOps を達成しており，この性能は理論性能の 84%であった．したがって，全体として概ね期待される性能の 8 割程度の性能が得られることが確認でき，我々が実装した各種最適化手法もおおむね期待通りの効果が得られたと判断できる．

8. まとめ

本稿では尾崎スキームを用いて高精度かつ再現可能な DOT，GEMV，GEMM の実装を行った．また省メモリ化および高速化のためのいくつかの最適化手法を検討した．本稿では特に性能がメモリ律速となる level-1/2 ルーチンの性能改善に有効となる，メモリアクセス量の削減手法を検討し，DOT や GEMV においても GEMM と同程度か数倍程度のオーバーヘッドで高精度かつ再現可能な演算が可能となった．Titan V GPU における実装および性能評価では，期待される理論性能に対して，実際に少なくとも 8 割程度の性能が得られることを確認した．本稿で GEMV に適用した最適化手法は疎行列ベクトル積（SpMV）においても有効であることが期待される．今後の展望として，本研究では実装を見送った疎行列化などの他の最適化手法の実装，また既存の高精度・再現可能な BLAS 実装との性能比較を行う予定である．

謝辞 本研究は，文部科学省 ポスト「京」萌芽的課題「極限の探究に資する精度保証付き数値計算学の展開と超高性能計算環境の創成」の助成，および科研費（16K16062）

[*3] なぜ 100%を超えるのか明らかではないが，周波数の動的制御によりメーカー公表値より高いクロックで動作している可能性がある．なお，理論メモリバンド幅はメーカー公表値より 0.85[GHz]×2（double data rate）×3072[bit]/8=652.8[GB/s] と算出できる．

表 4: 理論性能および実性能（s：分割回数，MM：GEMM 化，IN：32 ビット整数表現．理論値は 5 節の議論に基づく）

	DOT ($n = 2^{22}$)			GEMV ($m = n = 10240$)			GEMM ($m = n = k = 5120$)		
	実行時間	cuBLAS 比実行時間		実行時間	cuBLAS 比実行時間		実行時間	cuBLAS 比実行時間	
	[sec]	理論	実測 (理論比)	[sec]	理論	実測 (理論比)	[sec]	理論	実測 (理論比)
cuBLAS	1.02E-03	1.0	1.0 (100%)	1.25E-03	1.0	1.0 (100%)	3.77E-02	1.0	1.0 (100%)
OzBLAS($s = 2$)	9.02E-03	8.0	8.9 (90%)	1.13E-02	8.0	9.0 (89%)	1.60E-01	3.0	4.2 (71%)
OzBLAS($s = 3$)	1.38E-02	12.0	13.5 (89%)	1.71E-02	12.0	13.7 (87%)	2.77E-01	6.0	7.3 (82%)
OzBLAS($s = 4$)	1.86E-02	16.0	18.2 (88%)	2.30E-02	16.0	18.4 (87%)	4.46E-01	10.0	11.8 (84%)
OzBLAS($s = 2$,IN)	–	–	–	8.88E-03	6.0	7.1 (84%)	–	–	–
OzBLAS($s = 3$,IN)	–	–	–	1.37E-02	9.0	10.9 (82%)	–	–	–
OzBLAS($s = 4$,IN)	–	–	–	1.84E-02	12.0	14.8 (81%)	–	–	–

の助成を受けたものである．

参考文献

[1] Lawson, C., Hanson, R., Kincaid, D. and Krogh, F.: Basic Linear Algebra Subprograms for Fortran Usage, *ACM Trans. Math. Softw.*, Vol. 5, No. 3, pp. 308–323 (1979).

[2] Hammarling, S.: Workshop on Batched, Reproducible, and Reduced Precision BLAS., `http://eprints.maths.manchester.ac.uk/2494/1/Workshop.pdf` (2016).

[3] Li, X. S., Demmel, J. W., Bailey, D. H., Hida, Y., Iskandar, J., Kapur, A., Martin, M. C., Thompson, B., Tung, T. and Yoo, D. J.: XBLAS – Extra Precise Basic Linear Algebra Subroutines, `http://www.netlib.org/xblas`.

[4] Nakata, M.: The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK), `http://mplapack.sourceforge.net`.

[5] Iakymchuk, R., Collange, S., Defour, D. and Graillat, S.: ExBLAS – Exact BLAS, `https://exblas.lip6.fr`.

[6] Ozaki, K., Ogita, T., Oishi, S. and Rump, S. M.: Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications, *Numer. Algorithms*, Vol. 59, No. 1, pp. 95–118 (2012).

[7] Dekker, T. J.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol. 18, pp. 224–242 (1971).

[8] Hanrot, G., Lefèvre, V., Pélissier, P., Théveny, P. and Zimmermann, P.: MPFR : GNU MPFR Library, `http://www.mpfr.org`.

[9] Todd, R.: Introduction to Conditional Numerical Reproducibility (CNR), https://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr (2012).

[10] NVIDIA Corporation: The NVIDIA CUDA Basic Linear Algebra Subroutines, `https://developer.nvidia.com/cublas`.

[11] Chohra, C., Langlois, P. and Parello, D.: Reproducible, Accurately Rounded and Efficient BLAS, *22nd International European Conference on Parallel and Distributed Computing (Euro-Par 2016)*, pp. 609–620 (2016).

[12] Ahrens, P., Nguyen, H. D. and Demmel, J.: ReproBLAS - Reproducible Basic Linear Algebra Sub-programs, `https://bebop.cs.berkeley.edu/reproblas`.

[13] Ozaki, K., Ogita, T. and Oishi, S.: Matrix multiplication with guaranteed accuracy by level 3 BLAS, *International Conference of Computational Methods in Sciences and Engineering 2009 (ICCMSE 2009)*, Vol. 1504, pp. 1128–1133 (2012).

[14] 片桐孝洋，尾崎克久，荻田武史，大石進一：高精度行列 - 行列積アルゴリズムのスレッド並列化と ABCLibScript への機能実装，情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2012-HPC-133, No. 26, pp. 1–8 (2012).

[15] 市村駿太郎，片桐孝洋，尾崎克久，荻田武史，　永井亨，荻野正雄：マルチコア計算機による高精度行列 - 行列積アルゴリズムの性能評価，情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2017-HPC-160, No. 16, pp. 1–8 (2017).

[16] 石黒史也，片桐孝洋，大島聡史，　永井亨，荻野正雄：高精度行列–行列積アルゴリズムにおける batched BLAS の適用，第 80 回全国大会講演論文集，Vol. 2018, No. 1, pp. 49–50 (2018).

[17] 石黒史也，片桐孝洋，大島聡史，　永井亨，荻野正雄：GPGPU による高精度行列-行列積アルゴリズムのための Batched BLAS を用いた実装方式の提案，情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2018-HPC-165, No. 32, pp. 1–8 (2018).

[18] Rump, S., Ogita, T. and Oishi, S.: Accurate Floating-Point Summation Part II: Sign, K-Fold Faithful and Rounding to Nearest, *SIAM Journal on Scientific Computing*, Vol. 31, No. 2, pp. 1269–1302 (2009).

[19] 尾崎克久，荻田武史：BLAS を用いた高精度な行列積アルゴリズムの使用メモリ量の削減とその性能について (科学技術計算における理論と応用の新展開)，数理解析研究所講究録，Vol. 1791, pp. 66–75 (2012).

[20] Dongarra, J., Hammarling, S., Higham, N. J., Relton, S. D., Valero-Lara, P. and Zounon, M.: The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems, *International Conference on Computational Science (ICCS 2017)*, Vol. 108, pp. 495–504 (2017).

[21] Innovative Computing Laboratory, University of Tennessee: Matrix Algebra on GPU and Multicore Architectures.

[22] 尾崎克久，荻田武史：行列積に対する再現性のある計算法について，2017 年並列／分散／協調処理に関する『秋田』サマー・ワークショップ (SWoPP2017) (2017).

[23] Mukunoki, D., Imamura, T. and Takahashi, D.: Fast Implementation of General Matrix-Vector Multiplication (GEMV) on Kepler GPUs, *Proc. 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2015)*, pp. 642–650 (2015).