Numer Algor (2012) 59:95–118
DOI 10.1007/s11075-011-9478-1

ORIGINAL PAPER

Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications

Katsuhisa Ozaki · Takeshi Ogita · Shin'ichi Oishi · Siegfried M. Rump

Received: 8 February 2010 / Accepted: 26 May 2011 /
Published online: 14 June 2011

Abstract This paper is concerned with accurate matrix multiplication in floating-point arithmetic. Recently, an accurate summation algorithm was developed by Rump et al. (SIAM J Sci Comput 31(1):189–224, 2008). The key technique of their method is a fast error-free splitting of floating-point numbers. Using this technique, we first develop an error-free transformation of a product of two floating-point matrices into a sum of floating-point matrices. Next, we partially apply this error-free transformation and develop an algorithm which aims to output an accurate approximation of the matrix product. In addition, an a priori error estimate is given. It is a characteristic of the proposed method that in terms of computation as well as in terms of memory consumption, the dominant part of our algorithm is constituted by ordinary floating-point matrix multiplications. The routine for matrix multiplication is

K. Ozaki (✉)
Department of Mathematical Sciences, Shibaura Institute of Technology,
307 Fukasaku, Minuma-ku, Saitama-shi, Saitama 337-8570, Japan
e-mail: ozaki@sic.shibaura-it.ac.jp, k_ozaki@aoni.waseda.jp

K. Ozaki · T. Ogita · S. Oishi
Japan Science and Technology Agency (JST)/CREST, Tokyo, Japan

T. Ogita
Division of Mathematical Sciences, Tokyo Woman's Christian University,
2-6-1 Zempukuji, Suginami-ku, Tokyo 167-8585, Japan

S. Oishi · S. M. Rump
Faculty and Science and Engineering, Waseda University, 3-4-1, Okubo,
Shinjyuku-ku, Tokyo 169-0072, Japan

S. M. Rump
Institute for Reliable Computing, Hamburg University of Technology,
Schwarzenbergstr. 95, 21071 Hamburg, Germany

highly optimized using BLAS, so that our algorithms show a good computational performance. Although our algorithms require a significant amount of working memory, they are significantly faster than 'gemmx' in XBLAS when all sizes of matrices are large enough to realize nearly peak performance of 'gemm'. Numerical examples illustrate the efficiency of the proposed method.

Keywords Matrix multiplication · Accurate computations · Floating-point arithmetic · Error-free transformation

1 Introduction

This paper is concerned with accurate matrix multiplication in floating-point arithmetic. To obtain an accurate result, there are several possibilities:

- an accurate algorithm for sum and dot product (for example, [4, 9, 11, 12])
- a multiple-precision library (for example, [2, 7, 14, 15])

We suggest other algorithms which mainly exploit standard matrix multiplications in pure floating-point arithmetic rather than individual dot products.

Recently, an accurate summation algorithm was developed by Rump et al. [11], which outputs a faithfully rounded result of the summation of floating-point numbers. Their method avoids sorting of input data, branches in a main loop and direct access to the significand or exponent. Each of those would slow down the performance of computation on present-day architecture significantly. By avoiding all of those, their method is not only fast in terms of the number of floating-point operations but also in terms of measured computing time. Moreover, only basic floating-point operations [1] are required for this method, so that it can be implemented on all computational environments following the IEEE 754 standard.

We suggest an error-free transformation of the product of two floating-point matrices into an unevaluated sum of floating-point matrices by applying the strategy of the accurate summation algorithm in [11]. Applying our error-free transformation partially, we can suggest algorithms which deliver an accurate matrix product by mainly using floating-point matrix multiplications, which in turn are performed very fast by BLAS (Basic Linear Algebra Subprograms) routines. As to optimized BLAS, there are Intel Math Kernel Library, Goto BLAS [6], ATLAS [13] and others available. These codes are highly optimized for particular architectures. Some BLAS routines in such libraries automatically use multi-threads. Moreover, there are subprograms for sparse matrix computations, for example, Sparse BLAS [17] and UMFPACK [18]. Our algorithms mainly use such routines for matrix multiplication so that they are not only very effective in terms of computational performance but also easy and often automatically parallelizable.

This paper is organized as follows. In the following section, we introduce our notation and present the key technique of splitting a floating-point number. Next, we establish an error-free transformation of the product of two

floating-point matrices into an unevaluated sum of floating-point matrices. In Section 3, we investigate algorithms which output an accurate result of matrix multiplication. In the final section, we give error estimations and the level 3 fraction for the algorithms described in Section 3. Computational results confirm the accuracy and speed of the method.

Algorithms are represented by MATLAB-like notation [19] for simplicity.

2 Error-free transformation of a matrix product

In this section, we investigate the error-free transformation of the product of two floating-point matrices into an unevaluated sum of floating-point matrices.

2.1 Notation

We assume that all computations are performed in binary floating-point arithmetic as defined by the IEEE 754 standard [1]. Let $\mathbb{F}$ be the set of floating-point numbers and $\mathbf{u}$ be the unit roundoff.[1] Then, the significand of a floating-point number has $-\log_2 \mathbf{u}$ bits including the implicit 1. Let $\mathrm{fl}(\cdots)$ be denote that an expression inside the parentheses is evaluated in floating-point arithmetic with rounding to nearest (round to even tie-breaking). Note that double roundings are not allowed in $\mathrm{fl}(\cdots)$. Throughout this paper, assume that neither overflow nor underflow occurs in $\mathrm{fl}(\cdots)$. Inequalities for vectors are interpreted element-wise. Say, for $x, y \in \mathbb{F}^n$, $x > y$ means $x_i > y_i$ for $1 \leq i \leq n$.

2.2 Error-free splitting

We explain the technique used in the accurate summation algorithm developed by Rump et al. [11]. For $p \in \mathbb{F}$ and $M \in \mathbb{N}$, the following is a definition of σ:

$$\sigma = 2^M 2^{\lceil \log_2 |p| \rceil}. \tag{1}$$

Assume that $\sigma \in \mathbb{F}$, i.e. no overflow occurs. The following algorithm is presented in [11].

Algorithm 1 [11] *Assume $p \in \mathbb{F}$ and (1) is satisfied (note that $|p| \leq \sigma$). The following algorithm transforms p into $p', q \in \mathbb{F}$ by floating-point arithmetic such that $p = q + p'$.*

$$\begin{aligned} &\text{function } [q, p'] = \texttt{ExtractScalar}(p, \sigma) \\ &\quad q = \mathrm{fl}\left((\sigma + p) - \sigma\right); \\ &\quad p' = \mathrm{fl}\left(p - q\right); \end{aligned}$$

[1] For IEEE 754 binary64 (double precision), $\mathbf{u} = 2^{-53}$. For the binary32 (single precision), $\mathbf{u} = 2^{-24}$.

The values q and p' computed by Algorithm 1 satisfy

$$|p| \le \sigma 2^{-M} \tag{2}$$

and

$$|p'| \le \mathbf{u}\sigma, \tag{3}$$

$$|q| \le \sigma 2^{-M}, \tag{4}$$

$$q \in \mathbf{u}\sigma\mathbb{Z}. \tag{5}$$

Note that (4) and (5) imply that there are at most $-\log_2 \mathbf{u} - M$ nonzero leading bits in the binary representation of q. Here a floating-point number has "at most α nonzero leading bit" means that f has a binary expansion $f = \sum_{i=1}^{\alpha'} m_i \cdot 2^{e-i}$ with $\alpha' \le \alpha$. Hence some of the leading α bits may be zero.

Next, we introduce a splitting algorithm for a vector. For $x \in \mathbb{F}^n$, the following algorithm divides x into $x^{(1)}$ and $x^{(2)}$.

Algorithm 2 [11] *Assume $x \in \mathbb{F}^n$ and σ is a power of two satisfying*

$$\sigma \ge \max_{1\le i\le n} |x_i|.$$

The following algorithm transforms x into $x^{(1)}$, $x^{(2)}$ such that $x = x^{(1)} + x^{(2)}$.

function $[x^{(1)}, x^{(2)}] =$ `ExtractVector`(x, σ)
  `for` $i = 1 : n$
    $x_i^{(1)} = \mathrm{fl}\left((\sigma + x_i) - \sigma\right)$;
    $x_i^{(2)} = \mathrm{fl}\left(x_i - x_i^{(1)}\right)$;
  `end`

The properties of Algorithms 1 and 2 are different from that of Dekker's splitting algorithm [3]. Dekker's algorithm splits a floating-point number x into two parts $x^{(1)}, x^{(2)}$ satisfying $x = x^{(1)} + x^{(2)}$ with $|x^{(1)}| \ge |x^{(2)}|$, where both parts have at most $s - 1$ nonzero bits with $s = \lceil \frac{-\log_2 \mathbf{u}}{2} \rceil$ (see [11] for details), `ExtractVector` may produce $x_i^{(1)} = 0$ and $x_i^{(1)} = x_i$ (see Fig. 1).

Remark 1 Properties (2)–(4) are still valid for σ not being a power of two (see [10]). In our approach, it is advantageous to assume σ to be a power of two.

2.3 Image of error-free transformation

To understand the error-free transformation of a matrix product described in the next subsection, we first consider a dot product.

Let $x, y \in \mathbb{F}^n$, namely, $x = (x_1, x_2, \ldots, x_n)^T$, $y = (y_1, y_2, \ldots, y_n)^T$ and assume $n \ll \mathbf{u}^{-1}$. In this subsection, we transform a dot product $x^T y$ into an

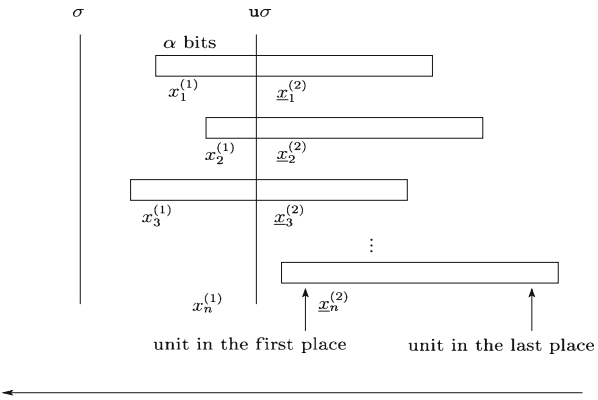

Fig. 1 Each *rectangle* depicts a floating-point number. Left and right end points of a *rectangle* show the unit in the first place and in the last place, respectively. Algorithm 2 divides floating-point numbers into two floating-point numbers

unevaluated sum of floating-point numbers. First, we define a constant α by

$$\alpha = \left\lfloor -\frac{\log_2 \mathbf{u} + \lceil \log_2 n \rceil}{2} \right\rfloor. \tag{6}$$

We aim to split x into two floating-point n-vectors $x^{(1)}, \underline{x}^{(2)}$ such that

$$x = x^{(1)} + \underline{x}^{(2)},$$

where the binary representation of each element in $x^{(1)}$ has at most α nonzero leading bits. The situation is depicted in Fig. 1. If we use Algorithm 2, then it is possible to obtain two such vectors by pure floating-point arithmetic. Similarly, we split y into $y^{(1)}, \underline{y}^{(2)} \in \mathbb{F}^n$ such that $y = y^{(1)} + \underline{y}^{(2)}$. After these splittings, the dot product $x^T y$ is calculated as

$$x^T y = \left(x^{(1)}\right)^T y^{(1)} + \left(x^{(1)}\right)^T \underline{y}^{(2)} + \left(\underline{x}^{(2)}\right)^T y. \tag{7}$$

A similar splitting is used in 'lssresidual' in INTLAB [20]. Note that there is no rounding error in $\mathrm{fl}\left(\left(x^{(1)}\right)^T y^{(1)}\right)$ (see Fig. 2). The reason is as follows: We execute

$$[x^{(1)}, \underline{x}^{(2)}] = \texttt{ExtractVector}(x, \sigma), \quad [y^{(1)}, \underline{y}^{(2)}] = \texttt{ExtractVector}(y, \tau),$$

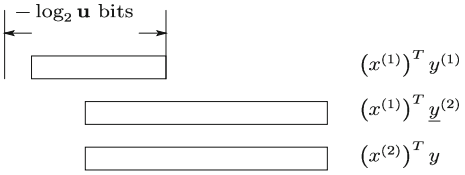

Fig. 2 Assume that vectors x and y are divided into $x^{(1)} + x^{(2)}$ and $y^{(1)} + y^{(2)}$, respectively. A floating-point evaluation of $\left(x^{(1)}\right)^T y^{(1)}$ does not cause rounding errors. It is expected that the magnitude of $\left(x^{(1)}\right)^T y^{(1)}$ is greater than the magnitude of $\left(x^{(2)}\right)^T y^{(1)}$ and $\left(x^{(2)}\right)^T y$

where the constant τ is a power of two and $\tau \geq \max_{1\leq i\leq n} |y_i|$. For suitable σ and τ, we obtain

$$\left|x_i^{(1)}\right| \leq 2^{\alpha}\mathbf{u}\sigma, \quad x_i^{(1)} \in \mathbf{u}\sigma\mathbb{Z} \quad \text{and} \quad \left|y_i^{(1)}\right| \leq 2^{\alpha}\mathbf{u}\tau, \quad y_i^{(1)} \in \mathbf{u}\tau\mathbb{Z} \quad \text{for } 1 \leq i \leq n.$$

A detailed discussion follows in next subsection. Now we have $x_i^{(1)} y_i^{(1)} \in \mathbf{u}^2\sigma\tau\mathbb{Z}$, and the dot product $(x^{(1)})^T y^{(1)}$ is bounded by

$$\left|\left(x^{(1)}\right)^T y^{(1)}\right| \leq \sum_{i=1}^{n} \left|x_i^{(1)} y_i^{(1)}\right| \leq n2^{2\alpha}\mathbf{u}^2\sigma\tau < \mathbf{u}\sigma\tau, \tag{8}$$

where the last inequality is derived from $2^{2\alpha} < (n\mathbf{u})^{-1}$.

Remark 2 If a real number $|f|$ is an integer multiple of 2^m and $f < 2^m\mathbf{u}^{-1}$ is satisfied for $m \in \mathbb{N}$, then $f \in \mathbb{F}$.

This remark and (8) imply that $\mathrm{fl}\left(\left(x^{(1)}\right)^T y^{(1)}\right) = \left(x^{(1)}\right)^T y^{(1)}$, see also Fig. 3.

Exploiting (7) to calculate $x^T y$, it is expected that the accuracy of the computed result of $x^T y$ increases, in particular, if cancellation occurs in the computation. The reason is as follows: $\mathrm{fl}\left(\left(x^{(1)}\right)^T y^{(1)}\right)$ does not cause rounding errors, so that the accuracy of the floating-point result by (7) depends on the errors in $\mathrm{fl}\left(\left(x^{(1)}\right)^T y^{(2)}\right)$ and $\mathrm{fl}\left(\left(x^{(2)}\right)^T y\right)$. Since $|x^{(1)}|$ and $|y^{(1)}|$ are greater than $|x^{(2)}|$ and $|y^{(2)}|$, respectively, except $x^{(1)} = y^{(1)} = 0$, the order of the magnitude of the rounding errors in $\mathrm{fl}\left(\left(x^{(2)}\right)^T y^{(1)}\right)$ and $\mathrm{fl}\left(\left(x^{(2)}\right)^T y\right)$ are usually smaller than in $\mathrm{fl}\left(x^T y\right)$.

Next we split $\underline{x}^{(2)}$ into two floating-point numbers such that $\underline{x}^{(2)} = x^{(2)} + \underline{x}^{(3)}$, where $x^{(2)}$ has at most α nonzero leading bits. Then $x = x^{(1)} + x^{(2)} + \underline{x}^{(3)}$. By repeating this splitting $k-1$ times, we have

$$x = x^{(1)} + x^{(2)} + \cdots + \underline{x}^{(k)}, \quad y = y^{(1)} + y^{(2)} + \cdots + \underline{y}^{(k)},$$

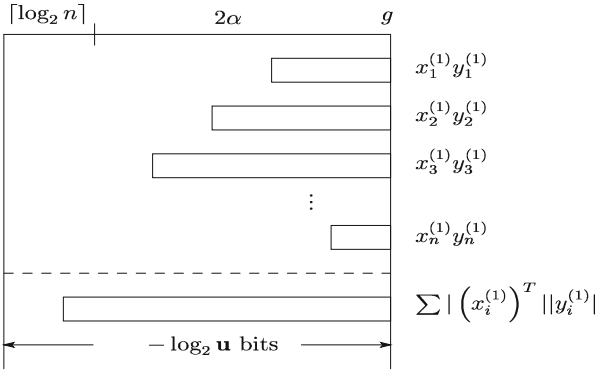

Fig. 3 This visualizes the dot product. Each of $x_1^{(1)} y_1^{(1)}, \ldots, x_1^{(n)} y_1^{(n)}$ has at most 2α nonzero leading bits, so that $|\left(x^{(1)}\right)^T y^{(1)}|$ is representable in at most $-\log_2 \mathbf{u}$ bit

where $x^{(i)}$, $y^{(i)}$ $(i < k)$ have at most α nonzero leading bits. After that, we expand

$$x^T y = \left(x^{(1)} + x^{(2)} + \cdots + \underline{x}^{(k)}\right)^T \left(y^{(1)} + y^{(2)} + \cdots + \underline{y}^{(k)}\right). \tag{9}$$

By factorizing some parts as follows:

$$x^T y = \sum_{i+j\le k} \left(x^{(i)}\right)^T y^{(j)} + \sum_{i=1}^{k-1} \left(\left(x^{(i)}\right)^T \underline{y}^{(k-i+1)}\right) + \left(\underline{x}^{(k)}\right)^T y \tag{10}$$

because

$$\underline{y}^{(k-i+1)} = \sum_{j=k-i+1}^{k-1} y^{(j)} + \underline{y}^{(k)}, \quad \underline{x}^{(k-1)} = x^{(k-1)} + \underline{x}^{(k)}.$$

Note that utilizing the recursive definition of $\underline{y}^{(k-i+1)}$ reduces the number of terms in (10) to about one half compared to a full expansion of (9). We stress that there is no rounding error in each dot product in the summation of the first term in (10) (see Fig. 4 for illustration), such that

$$\left(x^{(i)}\right)^T y^{(j)} = \mathrm{fl}\left(\left(x^{(i)}\right)^T y^{(j)}\right) \text{ for } i + j \le k.$$

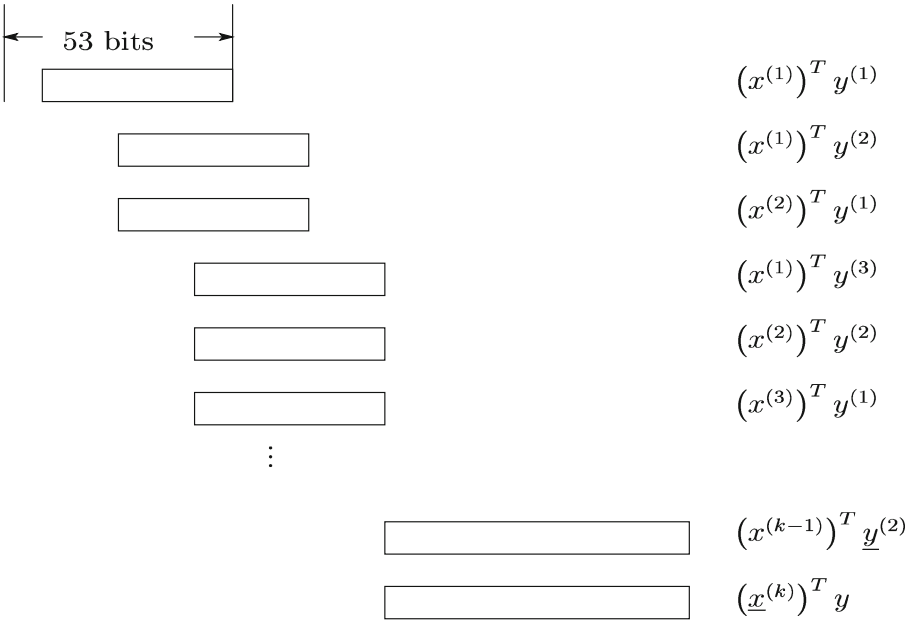

Fig. 4 In this figure the number of nonzero leading bits in $\left(x^{(i)}\right)^T y^{(j)} (i + j \le k)$ does not exceed $-\log_2 \mathbf{u}$. The computation of $\mathrm{fl}\left(\left(x^{(i)}\right)^T y^{(j)}\right)$ may cause rounding errors only for $i + j = k + 1$

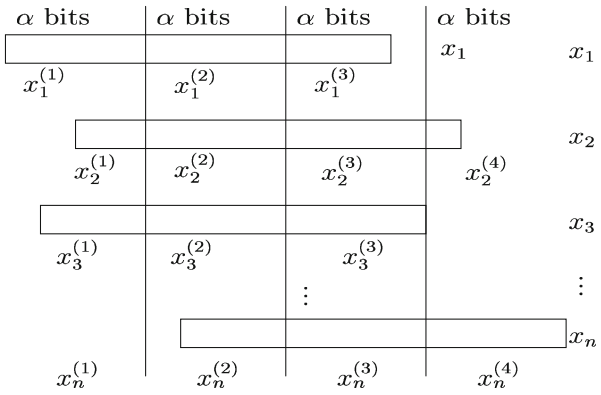

Fig. 5 This figure illustrates the splittings. Each floating-point number has at most α nonzero leading bits

Using similar arguments, it can be expected that the result of the floating-point evaluation of (10) is more accurate than that of (7). By repeating the splitting, there exists some $p \in \mathbb{N}$ such that $x = \sum_{i=1}^{p} x^{(i)}$ where all elements in $x^{(i)}$ have at most α nonzero leading bits. Figure 5 is the image of the splitting of x. Similarly,

$$y = \sum_{j=1}^{q} y^{(j)}, \quad q \in \mathbb{N}$$

so that all elements of $x^{(i)}$, $y^{(j)}$ have at most α nonzero leading bits. The dot product $x^T y$ is

$$x^T y = \left(x^{(1)} + x^{(2)} + \cdots + x^{(p)}\right)^T \left(y^{(1)} + y^{(2)} + \cdots + y^{(q)}\right).$$

If we expand the above expression straightforwardly, we can compute each $\mathrm{fl}\left(\left(x^{(i)}\right)^T y^{(j)}\right)$ without rounding error, i.e.

$$\left(x^{(i)}\right)^T y^{(j)} = \mathrm{fl}\left(\left(x^{(i)}\right)^T y^{(j)}\right) \text{ for } (1 \le i \le p,\ 1 \le j \le q).$$

If each $\mathrm{fl}\left(\left(x^{(i)}\right)^T y^{(j)}\right)$ is stored into $s_l \in \mathbb{F}$, the dot product can be transformed into the unevaluated sum of floating-point numbers s_l such that $x^T y = \sum_{l=1}^{pq} s_l$. This is an error-free transformation of the dot product.

2.4 Proposed method

We extend the error-free transformation of the dot product to that of a matrix product. For $A = (a_{ij}) \in \mathbb{F}^{m \times n}$ and $B = (b_{ij}) \in \mathbb{F}^{n \times p}$, we develop an error-free transformation for the multiplication AB. We assume $n \ll \mathbf{u}^{-1}$. First, we define a constant β by[2]

$$\beta = \left\lceil \frac{\lceil \log_2 n \rceil - \log_2 \mathbf{u}}{2} \right\rceil = \left\lceil \frac{\log_2 n - \log_2 \mathbf{u}}{2} \right\rceil, \tag{11}$$

[2]From (6) and (11), we could confirm that $\alpha + \beta = -\log_2 \mathbf{u}$.

where the second equality is not difficult to prove. We also define two vectors $\sigma^{(1)} \in \mathbb{F}^m$, $\tau^{(1)} \in \mathbb{F}^p$ by

$$\sigma_i^{(1)} = 2^\beta \cdot 2^{P_i^{(1)}}, \quad \tau_j^{(1)} = 2^\beta \cdot 2^{Q_j^{(1)}},$$

where two vectors $P^{(1)}$ and $Q^{(1)}$ are defined by

$$P_i^{(1)} = \lceil \log_2 \max_{1\le j\le n} |a_{ij}| \rceil, \quad Q_j^{(1)} = \lceil \log_2 \max_{1\le i\le n} |b_{ij}| \rceil. \tag{12}$$

Remark 3 Purpose of computing $2^{P_i^{(1)}}$ and $2^{Q_j^{(1)}}$ is to find $f, g \in \mathbb{F}^n$ which are powers of 2 such that

$$\max_{1\le j\le n} |a_{ij}| \le f, \quad \max_{1\le i\le n} |b_{ij}| \le g.$$

We need not to assume in (12) that all row vectors in A and column vectors in B are not zero vectors if we use the function `NextPowerTwo` in [11].

We split A and B as follows:

$$A^{(1)} = \mathrm{fl}\left(\left(A + \sigma^{(1)} \cdot e^T\right) - \sigma^{(1)} \cdot e^T\right), \quad \underline{A}^{(2)} = \mathrm{fl}\left(A - A^{(1)}\right)$$
$$B^{(1)} = \mathrm{fl}\left(\left(B + e \cdot (\tau^{(1)})^T\right) - e \cdot (\tau^{(1)})^T\right), \quad \underline{B}^{(2)} = \mathrm{fl}\left(B - B^{(1)}\right), \tag{13}$$

where $e = (1, 1, \ldots, 1)^T$. Then, as in Algorithm 1,

$$A = A^{(1)} + \underline{A}^{(2)}, \quad B = B^{(1)} + \underline{B}^{(2)}.$$

Next, we define $\sigma^{(2)}$ and $\tau^{(2)}$ from $\underline{A}^{(2)}$ and $\underline{B}^{(2)}$ by

$$\sigma_i^{(2)} = 2^\beta \cdot 2^{P_i^{(2)}}, \quad \tau_j^{(2)} = 2^\beta \cdot 2^{Q_j^{(2)}},$$

where $P^{(2)}$ and $Q^{(2)}$ are defined by

$$P_i^{(2)} = \lceil \log_2 \max_{1\le j\le n} |\underline{a}_{ij}^{(2)}| \rceil, \quad Q_j^{(2)} = \lceil \log_2 \max_{1\le i\le n} |\underline{b}_{ij}^{(2)}| \rceil.$$

Using these vectors, we compute

$$A^{(2)} = \mathrm{fl}\left(\left(\underline{A}^{(2)} + \sigma^{(2)} \cdot e^T\right) - \sigma^{(2)} \cdot e^T\right), \quad \underline{A}^{(3)} = \mathrm{fl}\left(\underline{A}^{(2)} - A^{(2)}\right),$$
$$B^{(2)} = \mathrm{fl}\left(\left(\underline{B}^{(2)} + e \cdot (\tau^{(2)})^T\right) - e \cdot (\tau^{(2)})^T\right), \quad \underline{B}^{(3)} = \mathrm{fl}\left(\underline{B}^{(2)} - B^{(2)}\right).$$

The above computations are also the extended variants of Algorithm 1, so that we have

$$\underline{A}^{(2)} = A^{(2)} + \underline{A}^{(3)}, \quad A = A^{(1)} + A^{(2)} + \underline{A}^{(3)},$$
$$\underline{B}^{(2)} = B^{(2)} + \underline{B}^{(3)}, \quad B = B^{(1)} + B^{(2)} + \underline{B}^{(3)}.$$

Generally, let $\sigma^{(w)}$, $\tau^{(w)}$ be

$$\sigma_i^{(w)} = 2^\beta \cdot 2^{P_i^{(w)}}, \quad \tau_j^{(w)} = 2^\beta \cdot 2^{Q_j^{(w)}}, \tag{14}$$

where $P^{(w)}$ and $Q^{(w)}$ are defined by

$$P_i^{(w)} = \lceil \log_2 \max_{1\le j\le n} |\underline{a}_{ij}^{(w)}| \rceil, \quad Q_j^{(w)} = \lceil \log_2 \max_{1\le i\le n} |\underline{b}_{ij}^{(w)}| \rceil.$$

Then

$$|\underline{a}_{ij}^{(w)}| \le 2^{P_i^{(w)}}, \quad |\underline{b}_{ij}^{(w)}| \le 2^{Q_j^{(w)}}.$$

We obtain $A^{(w)}$, $B^{(w)}$, $\underline{A}^{(w+1)}$ and $\underline{B}^{(w+1)}$ by

$$\begin{aligned} A^{(w)} &= \mathrm{fl}\left(\left(\underline{A}^{(w)} + \sigma^{(w)} \cdot e^T\right) - \sigma^{(w)} \cdot e^T\right), \quad \underline{A}^{(w+1)} = \mathrm{fl}\left(\underline{A}^{(w)} - A^{(w)}\right), \\ B^{(w)} &= \mathrm{fl}\left(\left(\underline{B}^{(w)} + e \cdot (\tau^{(w)})^T\right) - e \cdot (\tau^{(w)})^T\right), \quad \underline{B}^{(w+1)} = \mathrm{fl}\left(\underline{B}^{(w)} - B^{(w)}\right). \end{aligned} \tag{15}$$

Repeating the calculations (14) and (15), there exist some constants $n_A, n_B \in \mathbb{N}$ such that

$$A = \sum_{r=1}^{n_A} A^{(r)}, \quad B = \sum_{s=1}^{n_B} B^{(s)}, \quad \underline{A}^{(n_A+1)} = O_{mn}, \quad \underline{B}^{(n_B+1)} = O_{np}, \tag{16}$$

where O_{mn} denotes the m-by-n zero matrix. If there is a large difference in the order of magnitude among the elements in the rows of A or the columns of B, then n_A or n_B in (16) becomes large, respectively, and the amount of working space increases. In this case, many matrix products need to be computed. However, it is expected that several matrices are sparse in this case. By utilizing the sparse representation and sparse matrix products, the amount of working space decreases and the algorithm works faster.

Remark 4 An upper bound of n_A and n_B can be obtained as follows: If $\max_{1\le k\le n} |\underline{a}_{ik}^{(n_A+1)}|$ is less than the last bit of $\min_{1\le k\le n,\ a_{ij}\ne 0} |a_{ik}|$ for all i, that is

$$\max_{1\le k\le n} |\underline{a}_{ik}|^{(n_A+1)} < 2\mathbf{u} \min_{1\le k\le n,\ a_{ij}\ne 0} |a_{ik}|, \quad 1 \le i \le m,$$

then the splitting is finished. We take an upper bound on the left-side in the above-mentioned inequality as follows:

$$\begin{aligned} \max_{1\le k\le n} |\underline{a}_{ik}|^{(n_A+1)} &\le \mathbf{u}\sigma_i^{(n_A)} = \mathbf{u}2^{\beta} 2^{P_i^{(n_A)}} \le \mathbf{u}2^{\beta+1} \cdot \max_{1\le k\le n} |\underline{a}_{ik}^{(n_A)}| \\ &\le (\mathbf{u}2^{\beta+1})^2 \cdot \max_{1\le k\le n} |\underline{a}_{ik}^{(n_A-1)}| \\ &\le \cdots \le (\mathbf{u}2^{\beta+1})^{n_A} \max_{1\le k\le n} |a_{ik}| \end{aligned}$$

Therefore, we aim to find n_A satisfying the following:

$$(\mathbf{u}2^{\beta+1})^{n_A} \max_{1\le k\le n} |a_{ik}| < 2\mathbf{u} \min_{1\le k\le n,\ a_{ik}\ne 0} |a_{ik}|, \quad 1 \le i \le m.$$

Let m_A be

$$m_A = \min_{1\le i\le m} \frac{2\min_{1\le k\le n, a_{ik}\neq 0} |a_{ik}|}{\max_{1\le k\le n} |a_{ik}|},$$

then

$$\left(\mathbf{u}2^{\beta+1}\right)^{n_A} < \mathbf{u}m_A.$$

Finally, we obtain n_A as

$$n_A > \frac{\log_2 \mathbf{u} + \log_2 m_A}{\log_2 \mathbf{u} + \beta + 1}.$$

Therefore, n_A should be the smallest integer satisfying the above-mentioned inequality. Using similar arguments, it is possible to obtain the upper bound of n_B. Let m_B be

$$m_B = \min_{1\le j\le p} \frac{2\min_{1\le k\le n, b_{kj}\neq 0} |b_{kj}|}{\max_{1\le k\le n} |b_{kj}|}.$$

Then we have

$$n_B > \frac{\log_2 \mathbf{u} + \log_2 m_B}{\log_2 \mathbf{u} + \beta + 1}.$$

Therefore, the matrix product AB can be calculated as

$$AB = \left(\sum_{r=1}^{n_A} A^{(r)}\right)\left(\sum_{s=1}^{n_B} B^{(s)}\right).$$

Now we present the following theorem stating that there is no roundoff error in $\mathrm{fl}\left(A^{(r)}B^{(s)}\right), 1 \le r \le n_A, 1 \le s \le n_B$.

Theorem 1 *Let $A \in \mathbb{F}^{m\times n}$ and $B \in \mathbb{F}^{n\times p}$. Applying (14) and (15) repeatedly, (16) is satisfied for A and B, and there is no roundoff error in $\mathrm{fl}\left(A^{(r)}B^{(s)}\right)$, $1 \le r \le n_A$, $1 \le s \le n_B$, i.e.*

$$\mathrm{fl}\left(A^{(r)}B^{(s)}\right) = A^{(r)}B^{(s)}.$$

Proof For all (i, j) elements in AB, we aim to show

$$\mathrm{fl}\left(\sum_{k=1}^{n} a_{ik}^{(r)} b_{kj}^{(s)}\right) = \sum_{k=1}^{n} a_{ik}^{(r)} b_{kj}^{(s)}, \quad 1 \le i \le m, \quad 1 \le j \le p.$$

From (15), $A^{(r)}$ and $B^{(s)}$ are generated using $\sigma^{(r)}$ and $\tau^{(s)}$, respectively. From (14), we can use the relations (3)–(5). From (2) we have

$$|\underline{a}_{ik}^{(r)}| \le 2^{-\beta}\cdot\sigma_i^{(r)}, \quad |\underline{b}_{kj}^{(s)}| \le 2^{-\beta}\cdot\tau_j^{(s)}, \quad 1 \le i \le m, \quad 1 \le j \le p, \quad 1 \le k \le n,$$

and by (5) it follows that

$$a_{ij}^{(r)} \in \mathbf{u}\sigma_i^{(r)}\mathbb{Z}, \quad b_{ij}^{(s)} \in \mathbf{u}\tau_j^{(s)}\mathbb{Z}. \tag{17}$$

From (17) we obtain

$$a_{ik}^{(r)} b_{kj}^{(s)} \in \mathbf{u}^2 \sigma_i^{(r)} \tau_j^{(s)} \mathbb{Z}, \tag{18}$$

which implies

$$\sum_{k=1}^{n} a_{ik}^{(r)} b_{kj}^{(s)} \in \mathbf{u}^2 \sigma_i^{(r)} \tau_j^{(s)} \mathbb{Z}. \tag{19}$$

It follows by (4) that

$$|a_{ik}^{(r)}| \le 2^{-\beta} \cdot \sigma_i^{(r)}, \quad |b_{kj}^{(s)}| \le 2^{-\beta} \cdot \tau_j^{(r)}. \tag{20}$$

From (20) we calculate upper bounds of the dot products

$$\begin{aligned} \left| \sum_{k=1}^{n} a_{ik}^{(r)} b_{kj}^{(s)} \right| &\le \sum_{k=1}^{n} |a_{ik}^{(r)}||b_{kj}^{(s)}| \le \sum_{k=1}^{n} \left(2^{-\beta} \sigma_i^{(r)} \cdot 2^{-\beta} \tau_j^{(s)} \right) \\ &= n 2^{-\beta} \sigma_i^{(r)} \cdot 2^{-\beta} \tau_j^{(s)} = n 2^{-2\beta} \cdot \sigma_i^{(r)} \cdot \tau_j^{(s)}, \end{aligned} \tag{21}$$

and by (11) we obtain

$$n 2^{-2\beta} \le n 2^{-2\lceil \frac{\log_2 n - \log_2 \mathbf{u}}{2} \rceil} \le n 2^{-\log_2 n + \log_2 \mathbf{u}} = \mathbf{u}. \tag{22}$$

From (21) and (22) we have

$$\left| \sum_{k=1}^{n} a_{ik}^{(r)} b_{kj}^{(s)} \right| \le \mathbf{u} \sigma_i^{(r)} \tau_j^{(s)}, \tag{23}$$

so that the relations (18), (19) and (23) show

$$\begin{cases} \mathbf{u}^2 \sigma_i^{(r)} \cdot \tau_j^{(s)} \le \left| \sum_{k=1}^{n} a_{ik}^{(r)} b_{kj}^{(s)} \right| \le \mathbf{u} \sigma_i^{(r)} \cdot \tau_j^{(s)} \text{ if } \sum a_{ik}^{(r)} b_{kj}^{(s)} \ne 0 \\ \mathrm{fl}\left(\sum a_{ik}^{(r)} b_{kj}^{(s)} \right) = 0 \text{ if } \sum a_{ik}^{(r)} b_{kj}^{(s)} = 0 \end{cases}. \tag{24}$$

From Remark 2, this means that there is no roundoff error in $\mathrm{fl}\left(A^{(r)} B^{(s)}\right)$. This completes the proof. □

Theorem 1 implies an error-free transformation of a matrix product by

$$AB = \sum_{1 \le i \le n_A, 1 \le j \le n_B} \mathrm{fl}\left(A^{(i)} B^{(j)}\right).$$

It means that the product AB of two floating-point matrices can be transformed into an unevaluated sum of $n_A n_B$ floating-point matrices, provided no overflow and underflow occurs.

Next we give an algorithm for error-free transformation of the matrix A into an unevaluated sum of floating-point matrices.

Algorithm 3 Let δ with $0 \le \delta < 1$ be the criterion for using the sparse formula. Concretely, if the number of nonzero elements in an m-by-n matrix is less than

δmn, a sparse representation is used. Algorithm `Split_A`(A, ℓ) transforms A into $\sum_{r=1}^{\underline{\ell}} D^{(r)}$ $(\underline{\ell} \le \ell)$ without rounding errors by (14) and (15).

```
function D = Split_Mat(A, ℓ, δ)
    q = size(A, 2);
    k = 1;
    β = fl(⌈(−log2(u) + log2(q))/2)⌉);
    D{1} = zeros(size(A));
    while (k < ℓ)
        μ = max(abs(A), [], 2);    % μ(i) = max_{1≤j≤q} |a_ij|
        if (max(μ) == 0)    % check a_ij = 0 for all (i, j)
            return;
        end
        w = fl(2.^(ceil(log2(μ)) + β));
        S = repmat(w, 1, q);    % S = w · e^T
        D{k} = fl((A + S) − S);
        A = fl(A − D{k});
        % Checking sparsity of D{k}
        if (nnz(D{k}) < δ ∗ m ∗ n), D{k} = sparse(D{k}); end
        k = k + 1;
    end
    if (k == ℓ)
        D{k} = A;
    end
end
```

If condition (16) is required, set $\ell = \infty$. In that case, we can find n_A such that $A = \sum_{r=1}^{n_A} D^{(r)}$ ($A = \sum_{r=1}^{n_A} D\{r\}$ in MATLAB notation).

Similarly, the matrix B is split into an unevaluated sum of floating-point matrices $B = \sum_{s=1}^{\ell} E^{(s)}$ by $E = \left(\texttt{Split\_Mat}(B^T, \ell)\right)^T$.

This suggests the following algorithm:

Algorithm 4 For two matrices $A \in \mathbb{F}^{m\times n}$, $B \in \mathbb{F}^{n\times p}$, this algorithm transforms the matrix product AB into an unevaluated sum of floating-point matrices without rounding errors such that

$$AB = \sum_{i=1}^{n_A n_B} C^{(i)}, \quad C^{(i)} \in \mathbb{F}^{m\times n}.$$

```
function C = EFT_Mul(A, B, δ)
    [m, n] = size(A); [n, p] = size(B);
    D = Split_Mat(A, inf, δ); n_A = length(D);
    E = Split_Mat(B^T, inf, δ); n_B = length(E);
    for r = 1 : n_B, E{r} = E{r}^T; end
    t = 1;
    for r = 1 : n_A
        for s = 1 : n_B
            C{t} = fl(D{r} * E{s});
            t = t + 1;
        end
    end
end
```

After this transformation we can apply accurate summation algorithms for floating-point numbers. Exploiting the algorithms in [11, 12] we can guarantee

- element-wise signs,
- a faithful result,
- the nearest result, or
- a result with K-fold accuracy

of the matrix product. If we apply such accurate summation algorithms, then the additional cost for using an accurate summation algorithm becomes $\mathcal{O}(n_A n_B mp)$ flops.[3] If there is no sparse computation in Algorithm 4, then the costs of Algorithm 4 becomes $2n_A n_B mnp$ flops. Therefore, it is expected that the computing time for the usage of accurate summation algorithms is not too expensive compared to that of Algorithm 4 for large n.

2.5 Numerical examples

In this section we show numerical examples to illustrate the efficiency of the proposed algorithm. First, we generate matrices A, B by

$$(\texttt{rand}(n) - 0.5).* \texttt{exp}(\phi * \texttt{randn}(n)). \tag{25}$$

Here, all elements of A and B are binary64 floating-point numbers. The function $\texttt{randn}(n)$ returns an n-by-n matrix containing pseudo-random values drawn from the standard normal distribution. The function $\texttt{rand}(n)$ returns an n-by-n matrix containing pseudo-random values drawn from the standard uniform distribution on the open interval $(0, 1)$. The function $\texttt{exp}(X)$ returns the componentwise exponential of X. The items n_A and n_B labeled in Tables 1 and 2 show that the matrices A and B are split into an unevaluated sum of n_A

[3] Here flops means a number of floating-point operations as defined in [5]. Note that it does not mean 'floating-point number operations per second'.

Table 1 The ratio of computing time of Algorithm 4 to the built-in matrix product for various problems with $n = 1{,}000$ (Intel Core 2 Duo)

ϕ	n_A	n_B	Ratio (no sparse)	Ratio (using sparse)
1	4	4	18.7	13.0
5	6	6	40.9	29.8
10	9	9	85.0	42.1
15	12	12	151	66.1

and n_B floating-point matrices, respectively. Let t_1 be computing time for dense matrix multiplication AB by pure floating-point arithmetic, t_2 be computing time for Algorithm 4. The examples in Table 1 are tested using Intel Core 2 Duo 1.2 GHz and MATLAB 2009a, $n = 1{,}000$. We set δ in Algorithm 4 as 0.1. The examples in Table 2 use Intel Core 2 Extreme 3.0 GHz and MATLAB 2007b, $n = 2{,}000$. Again, δ in Algorithm 4 is taken as 0.1. The item 'ratio' in Tables 1 and 2 denotes t_2/t_1. When ϕ in (25) increases, n_A and n_B become larger. When $\phi = 1$, both A and B are divided into an unevaluated sum of four floating-point matrices. This means 16 matrix products in Algorithm 4. Since we use the MATLAB built-in sparse format, the ratio becomes 13.0, which is less than 16. When ϕ becomes much larger, applying sparse routine can reduce the computing time significantly.

Remark 5 Sparse matrix multiplication by MATLAB is executed with a single thread. If this point is improved in the future, the performance of our method is also improved.

3 Application of the error-free transformation of a matrix product

The algorithm in the previous section first transforms a matrix product into an unevaluated sum of floating-point matrices. Applying accurate summation algorithms, for example [10–12], produces an accurate approximation of the product independent of the condition number. In this section, we apply only a partial error-free transformation in order to compute an approximation of the product with improved accuracy (compared to the ordinary product). We also present numerical results. Mathematical properties of the proposed algorithm will be discussed in the next section.

Table 2 The ratio of computing time of Algorithm 4 to the built-in matrix product for various problems with $n = 2{,}000$ (Intel Core 2 Extreme)

ϕ	n_A	n_B	Ratio (no sparse)	Ratio (using sparse)
1	4	4	19.7	20.8
5	6	6	41.5	32.8
10	9	9	89.0	74.9
15	12	12	154	108

3.1 Accurate matrix multiplication

First, we divide the matrices A and B into unevaluated sums of two floating-point matrices by (13). Then the matrix multiplication AB can be calculated by

$$AB = \left(A^{(1)} + \underline{A}^{(2)}\right)\left(B^{(1)} + \underline{B}^{(2)}\right). \tag{26}$$

As in 'lssresidual' in INTLAB [20] we expand the above-mentioned expression into

$$A^{(1)}B^{(1)} + A^{(1)}\underline{B}^{(2)} + \underline{A}^{(2)}B^{(1)} + \underline{A}^{(2)}\underline{B}^{(2)} = A^{(1)}B^{(1)} + \left(A^{(1)}\underline{B}^{(2)} + \underline{A}^{(2)}B\right). \tag{27}$$

Recall from Theorem 1 that there is no roundoff error in the computation $\mathrm{fl}\left(A^{(1)}B^{(1)}\right)$. Here, roundoff errors may occur only in the computations in $\mathrm{fl}\left(A^{(1)}\underline{B}^{(2)}\right)$ and $\mathrm{fl}\left(\underline{A}^{(2)}B\right)$. However, it is expected that the order of magnitude in $\underline{A}^{(2)}$ and $\underline{B}^{(2)}$ is smaller than that of A and B, respectively. Thus, roundoff errors in $\mathrm{fl}\left(A^{(2)}B\right)$ and $\mathrm{fl}\left(A^{(1)}B^{(2)}\right)$ are usually less than that in $\mathrm{fl}(AB)$. Therefore, compared to the result of $\mathrm{fl}(AB)$, the accuracy of the result is often improved by using the formula (27).

Moreover, it is possible to extend this strategy into the case of using an unevaluated sum of three floating-point matrices. In this case, the matrix product AB is expanded into

$$AB = \left(A^{(1)} + A^{(2)} + \underline{A}^{(3)}\right)\left(B^{(1)} + B^{(2)} + \underline{B}^{(3)}\right). \tag{28}$$

Collecting some parts in the above expression we obtain

$$AB = A^{(1)}B^{(1)} + \left(\left(A^{(2)}B^{(1)} + A^{(1)}B^{(2)}\right) + \left(A^{(1)}\underline{B}^{(3)} + A^{(2)}\underline{B}^{(2)} + A^{(3)}B\right)\right)$$

where there is no roundoff error in $\mathrm{fl}\left(A^{(1)}B^{(1)}\right)$, $\mathrm{fl}\left(A^{(2)}B^{(1)}\right)$ and $\mathrm{fl}\left(A^{(1)}B^{(2)}\right)$ as by Theorem 1. It can be expected that the accuracy is improved compared to (27). If we split the matrices A and B into unevaluated sum of k floating-point matrices respectively by (14) and (15) such that

$$A = \sum_{i=1}^{k-1} A^{(i)} + \underline{A}^{(k)}, \quad B = \sum_{j=1}^{k-1} B^{(j)} + \underline{B}^{(k)},$$

where

$$\underline{A}^{(l-1)} = A^{(l-1)} + \underline{A}^{(l)}, \quad \underline{B}^{(l-1)} = B^{(l-1)} + \underline{B}^{(l)}, \quad 2 \le l \le k,$$
$$\underline{A}^{(1)} = A, \quad \underline{B}^{(1)} = B,$$

then we compute the matrix product AB by

$$\sum_{i+j\le k} A^{(i)}B^{(j)} + \sum_{i=1}^{k-1} A^{(i)}\underline{B}^{(k-i+1)} + \underline{A}^{(k)}B. \tag{29}$$

This involves $k(k-1)/2+k$ matrix products. From Theorem 1, it was proved that

$$\mathrm{fl}\left(A^{(i)}B^{(j)}\right) = A^{(i)}B^{(j)},\ i+j \le k.$$

Next, we present an algorithm computing the matrix product AB by (29):

Algorithm 5 For matrices $A \in \mathbb{F}^{m\times n}$ and $B \in \mathbb{F}^{n\times p}$, this algorithm outputs an accurate approximation of AB using (29).

```
function C = Acc_Mul(A, B, k, δ)
    [m, n] = size(A); [n, p] = size(B);
    D = Split_Mat(A, k, δ); h_A = length(D);
    E = Split_Mat(B^T, k, δ); h_B = length(E);
    for i = 1 : h_B, E{i} = E{i}^T; end
    l = 0;
    for r = 1 : min(h_A, k − 1)
        for s = 1 : min(h_B, k − 1)
            if (r + s <= k)
                l = l + 1;
                G{l} = fl(D{r} * E{s});          % error-free
            end
        end
    end
    for r = 1 : h_A
        F = zeros(n, p);
        for s = k − r + 1 : h_B, F = fl(F + E{s}); end
        l = l + 1;
        G{l} = fl(D{r} * F);
    end
    C = G{1};
    for i = 2 : l
        C = fl(C + G{i});
    end
end
```

Remark 6 We recommend using an accurate summation algorithm for the final summation of l floating-point matrices $\sum_{i=1}^{l} G\{i\}$ in the last loop of Algorithm 5. The additional cost for the use of an accurate summation algorithm is relatively small except when the inner dimension n is very small. Otherwise, we apply the pure floating-point summation in ascending order of indices i. The reason is that the magnitude of $G\{i\}$ is almost known in advance.

3.2 Numerical results

We show some numerical examples to illustrate the efficiency of Algorithm 5. First, we generate matrices A and $B \in \mathbb{F}^{n\times n}$ by (25) with $n = 1000$,

Table 3 Comparison of $\texttt{RelErr}(AB, C)$

ϕ	M_1	M_2	M_3	M_4	M_5	M_6
1	5.64e–010	8.12e–011	7.95e–015	2.20e–016	3.27e–016	1.11e–016
5	3.48e–011	2.39e–011	7.28e–012	2.19e–016	3.24e–016	1.11e–016
10	2.90e–011	2.68e–011	8.88e–011	1.59e–012	2.21e–014	1.11e–016
15	6.81e–012	5.29e–012	5.39e–012	5.60e–012	4.18e–012	1.11e–016

respectively. We compare the accuracy of results and computing times of the following six methods:

- M_1: Pure floating-point matrix multiplication by $A * B$ in MATLAB ($2n^3$ flops)
- M_2: lssredisual in INTLAB [20], ($6n^3 + \mathcal{O}(n^2)$ flops, $-\texttt{lssresidual}$ $(A, B, \texttt{zeros}(n))$)
- M_3: Algorithm 5, $k = 2$ ($6n^3 + \mathcal{O}(n^2)$ flops, $\texttt{Acc\_Mul}(A, B, 2, 0.2)$)
- M_4: Algorithm 5, $k = 3$ ($12n^3 + \mathcal{O}(n^2)$ flops, $\texttt{Acc\_Mul}(A, B, 3, 0.2)$)
- M_5: Algorithm 5, $k = 4$ ($20n^3 + \mathcal{O}(n^2)$ flops, $\texttt{Acc\_Mul}(A, B, 4, 0.2)$)
- M_6: XBLAS [7] ($37n^3 + \mathcal{O}(n^2)$ flops, $\texttt{gemmx}(\texttt{1}, \texttt{A}, \texttt{B}, \texttt{'x'})$)

Let C be a computed result by method M_1, M_2, M_3, M_4, M_5 or M_6. The maximum relative error to the exact matrix product AB as displayed in Table 3 is defined by

$$\texttt{RelErr}(AB, C) := \max_{1 \le i, j \le n} |(AB)_{ij} - C_{ij}|/|(AB)_{ij}|, \ (AB)_{ij} \neq 0.$$

The exact matrix product AB is computed by multi-precision library with enough precision. Table 3 shows the above-mentioned $\texttt{RelErr}(AB, C)$. In Table 4, the computing times for M_1, M_2, M_3, M_4, M_5 and M_6 are displayed with that for M_1 being normalized to 1. These examples are tested on Intel Core 2 Duo 1.2 GHz and MATLAB 2009a.[4] We downloaded the MATLAB mex[5] code of M_6 for the implementation from [16].

From Table 3 we see that the accuracy of the results of M_2 and the proposed methods are improved when ϕ is small. When we look at $\phi = 10$ in Table 4, the ratio of the computing time of M_3 is 2.40. If the computations are performed by the routine of dense matrix multiplication, the ratio should be larger than 3 since M_3 involves three matrix multiplications. This is the effect of using sparse matrix computations. Actually, $D^{(1)}$ and $E^{(1)}$ in Algorithm 5 become sparse matrices. From Table 4, the ratio of computing time for XBLAS is much larger than expected by the ratio ($37n^3/2n^3$) of the flop count. This is because M_1, M_2, M_3, M_4 and M_5 receive much benefit from the optimized BLAS in terms of computational performance while M_6 does not.

[4]The CPU has two cores, however, the examples are tested with a single core since gemmx by [16] works with single thread.

[5]MATLAB provides interfaces to external routines written in other programming languages.

Table 4 Comparison of the computing times

ϕ	M_1	M_2	M_3	M_4	M_5	M_6
1	1	3.30	3.34	7.41	10.4	99.1
5	1	3.29	3.35	7.62	11.7	95.9
10	1	3.30	2.40	3.41	8.60	84.9
15	1	3.30	2.74	3.81	7.13	97.1

As a drawback of our algorithm, the required amount of working space is significantly larger than that for `gemmx`. In addition, if ϕ is large, the proposed method could not improve the accuracy of the result efficiently, whereas XBLAS could output more accurate results than our algorithm.

Next, we generate matrices A and B by

$$A = \texttt{gallery}('randsvd', 1000, cnd, 3, 1000, 1000, 1), \quad B = A \setminus \texttt{randn}(n),$$

and compare the maximum of relative errors by M_1 to M_6 (Table 5). For the detail of the function `gallery`, see [8]. Here, *cnd* is the matrix condition number. If the condition number is large, then heavy cancellation occurs in the product of AB. Therefore, the more the condition number increases, the more difficult it is to obtain an accurate result. From Table 5 M_4 and M_5 are comparable to M_6 in terms of the accuracy for condition number 10^4. For condition number 10^{12}, M_5 is also comparable to M_6. We stress that the computing times in this example are similar to those displayed in Table 4, so that M_5 is much faster than M_6.

4 The characteristic of the proposed method

In this section, we describe properties of our method. First, we will provide an a priori error estimate of Algorithm 5. Next, we will give the level 3 fraction of the algorithm.

4.1 A priori error estimate for the proposed method

We assume that each matrix product in (29) is stored into $G^{(i)} \in \mathbb{F}^{m\times p}$. As for details, each matrix product in the first term in (29) is stored from $G^{(1)}$ to $G^{(k(k-1)/2)}$. Each matrix product in the second term in (29) is stored from $G^{(k(k-1)/2+1)}$ to $G^{(k(k-1)/2+k)}$. Namely, we have

$$G^{(1)} = A^{(1)}B^{(1)},\ G^{(2)} = A^{(1)}B^{(2)},\ G^{(3)} = A^{(2)}B^{(1)}, \ldots,\ G^{(k(k-1)/2)} = A^{(k-1)}B,$$

Table 5 Comparison of each $\texttt{RelErr}(AB, C)$

$\texttt{cond}(A)$	M_1	M_2	M_3	M_4	M_5	M_6
10^4	1.59e–007	6.15e–009	1.21e–013	2.21e–016	3.26e–016	1.11e–016
10^8	1.74e–003	2.33e–006	4.45e–010	1.37e–015	2.17e–016	1.11e–016
10^{12}	1.42e+001	1.25e–002	5.32e–005	8.95e–012	2.13e–016	1.48e–015

$G^{(k(k-1)/2+1)} = A^{(1)}\underline{B}^{(k)}$, $G^{(k(k-1)/2+2)} = A^{(2)}\underline{B}^{(k-1)}, \ldots,$ $G^{(k(k-1)/2+k)} = A^{(k)}B$.

Moreover, we define $\tilde{G}^{(v)} := \mathrm{fl}(G^{(v)})$ for all v.

Theorem 2 *Assume that*

$$C = \sum_{i+j\le k} \mathrm{fl}\left(A^{(i)}B^{(j)}\right) + \sum_{i=1}^{k-1} \mathrm{fl}\left(A^{(i)}\underline{B}^{(k-i+1)}\right) + \mathrm{fl}\left(\underline{A}^{(k)}B\right) = \sum_{k=1}^{k(k-1)/2+k} \tilde{G}^{(k)}.$$

Then,

$$|C - AB| \le nk\gamma_n \cdot 2^{(\beta-h)(k-1)} \cdot \left(2^{P^{(1)}}\right)\left(2^{Q^{(1)}}\right)^T =: E. \tag{30}$$

If $D^{(1)}$ is the result of $\mathrm{fl}(\sum_{k=1}^{k(k-1)/2+k} \tilde{G}^{(k)})$*, then*

$$|D^{(1)} - C| \le \gamma_{k(k-1)/2+k-1} \sum_{i=1}^{k(k-1)/2+k} |\tilde{G}^{(i)}|$$

If $D^{(2)}$ is the result of a summation algorithm for $\sum_{k=1}^{k(k-1)/2+k} \tilde{G}^{(k)}$ with faithful rounding as in [11]*, then*

$$|D^{(2)} - C| \le 2\mathbf{u}\left|\sum_{i=1}^{k(k-1)/2+k} \tilde{G}^{(i)}\right|.$$

Therefore, if we apply ordinary floating-point summation in final summation in Algorithm 5*, then*

$$|D^{(1)} - AB| \le |C - AB| + |D^{(1)} - C| = E + \gamma_{k(k-1)/2+k-1} \sum_{i=1}^{k(k-1)/2+k} |\tilde{G}^{(i)}|. \tag{31}$$

If we apply a summation algorithm with faithfully rounded result, then

$$|D^{(2)} - AB| \le |C - AB| + |D^{(2)} - C| = E + 2\mathbf{u}\left|\sum_{i=1}^{k(k-1)/2+k} \tilde{G}^{(i)}\right|. \tag{32}$$

Proof The error bounds of $|D^{(1)} - C|$ and $|D^{(2)} - C|$ are easily obtained by an a priori error analysis and definition of the faithful rounding. Therefore, we show only the error bound of $|C - AB|$. Let γ_n be

$$\gamma_n = \frac{n\mathbf{u}}{1 - n\mathbf{u}} \text{ for } n \in \mathbb{N}. \tag{33}$$

Then from [8], an upper bound of the rounding error of the floating-point matrix product $\mathrm{fl}(AB)$ is

$$|AB - \mathrm{fl}(AB)| \le \gamma_n |A||B|. \tag{34}$$

From (29) we obtain

$$|C - AB| = \left| \sum_{i=1}^{k(k-1)/2+k} \tilde{G}^{(i)} - \sum_{i+j\le k} A^{(i)} B^{(j)} - \sum_{i=1}^{k-1} A^{(i)} \underline{B}^{(k-i+1)} - \underline{A}^{(k)} B \right|$$
$$= \left| \sum_{i=1}^{k(k-1)/2+k} \tilde{G}^{(i)} - \sum_{i=1}^{k(k-1)/2+k} G^{(i)} \right|$$
$$\le \sum_{i=1}^{k(k-1)/2+k} |G^{(i)} - \tilde{G}^{(i)}|.$$

Since there is no rounding error in $\mathrm{fl}\left(A^{(i)} B^{(j)}\right)$ for $i + j \le k$ (see Theorem 1), we obtain an upper bound of $|C - AB|$ using (34) as

$$|C - AB| \le \sum_{i=k(k-1)/2+1}^{k(k-1)/2+k} |G^{(i)} - \tilde{G}^{(i)}| \le \gamma_n \sum_{i+j=k+1} |A^{(i)}||\underline{B}^{(j)}| =: F,$$

where we use $\underline{B}^{(1)} = B$. By extending (3) and (4) to matrix operation, we obtain

$$|A^{(i)}| \le 2^{-\beta} \cdot \sigma^{(i)} \cdot e^T, \quad |B^{(j)}| \le 2^{-\beta} \cdot e \cdot \left(\tau^{(j)}\right)^T, \quad i, j < k, \tag{35}$$

$$|\underline{A}^{(i)}| \le \mathbf{u} \cdot \sigma^{(i-1)} \cdot e^T, \quad |\underline{B}^{(j)}| \le \mathbf{u} \cdot e \cdot \left(\tau^{(j-1)}\right)^T, \quad i, j < k, \quad i, j \ne 1. \tag{36}$$

From the definition of $\sigma^{(s)}$ in (14) and $P^{(s)}$, we obtain

$$\sigma^{(i)} = 2^{\beta} \cdot 2^{P_i^{(s)}} = 2^{\beta} \cdot 2^{\lceil \log_2 \max_{1\le j\le n} |\underline{a}_{ij}^{(s)}| \rceil}.$$

We take an upper bound for $\sigma^{(i)}$ by using (36) as follows:

$$\sigma^{(i)} \le 2^{\beta} \cdot 2^{\lceil \log_2 \max_{1\le j\le n} (\mathbf{u}\cdot\sigma^{(i-1)}\cdot e^T)_{ij} \rceil} = 2^{\beta} \cdot 2^{\log_2 \max_{1\le j\le n} (\mathbf{u}\cdot\sigma^{(i-1)}\cdot e^T)_{ij}}$$
$$= 2^{\beta} \cdot \max_{1\le j\le n} (\mathbf{u} \cdot \sigma^{(i-1)} \cdot e^T)_{ij} = 2^{\beta} \cdot \mathbf{u} \cdot \sigma^{(i-1)} = 2^{\beta-h} \cdot \sigma^{(i-1)}, \tag{37}$$

where $h := -\log_2 \mathbf{u}$. Here for $X_{ij} \in \mathbb{R}^{m\times n}$, we define

$$\max_{1\le j\le n} X_{ij} = \left(\max_{1\le j\le n} X_{1j}, \max_{1\le j\le n} X_{2j}, \ldots, \max_{1\le j\le n} X_{mj} \right)^T \in \mathbb{R}^m.$$

Note that all elements in $\mathbf{u} \cdot \sigma^{(i-1)}$ are powers of 2. Applying (37) recursively yields

$$\sigma^{(i)} \le 2^{(\beta-h)} \sigma^{(i-1)} \le 2^{2(\beta-h)} \sigma^{(i-2)} \le \cdots \le 2^{(\beta-h)(i-1)} \sigma^{(1)} = 2^{(\beta-h)(i-1)} \cdot 2^{\beta} \cdot 2^{P^{(1)}}. \tag{38}$$

By substituting (38) to (35), we have

$$|A^{(i)}| \le 2^{(\beta-h)(i-1)} \cdot 2^{P^{(1)}} \cdot e^T. \tag{39}$$

Substituting (38) to (36) yields

$$|\underline{A}^{(i)}| \le \mathbf{u} \cdot 2^{\beta} \cdot 2^{(\beta-h)(i-2)} \cdot 2^{P^{(1)}} \cdot e^T \le 2^{(\beta-h)(i-1)} \cdot 2^{P^{(1)}} \cdot e^T.$$

Using similar arguments for B, we obtain

$$|B^{(j)}| \le 2^{(\beta-h)(j-1)} \cdot e \cdot \left(2^{Q^{(1)}}\right)^T, \quad |\underline{B}^{(j)}| \le 2^{(\beta-h)(j-1)} \cdot e \cdot \left(2^{Q^{(1)}}\right)^T. \tag{40}$$

Note that (40) is also satisfied for $j = 1$. From (39) and (40), $\mathtt{err}_1$ is bounded by

$$\begin{aligned} |C - AB| \le F &\le \gamma_n \sum_{i+j=k+1} 2^{(\beta-h)(i-1)} \cdot \left(2^{P^{(1)}}\right) \cdot e^T \cdot 2^{(\beta-h)(j-1)} \cdot e \cdot \left(2^{Q^{(1)}}\right)^T \\ &= \gamma_n \sum_{i+j=k+1} 2^{(\beta-h)(i+j-2)} \cdot \left(2^{P^{(1)}}\right) \cdot e^T \cdot e \cdot \left(2^{Q^{(1)}}\right)^T \\ &= nk \cdot \gamma_n \cdot 2^{(\beta-h)(k-1)} \cdot \left(2^{P^{(1)}}\right)\left(2^{Q^{(1)}}\right)^T. \end{aligned} \tag{41}$$

□

In the estimate E is dominant. Since the upper bound $|C - AB|$ depends on row-wise or column-wise maximum elements in A and B, we cannot compare (34) with (32) straightforwardly. However, when $2^{\beta-h} \ll 1$ and k increases, the order of magnitude of E_1 decreases with the factor around $2^{\beta-h}$. For example, in binary64, $n = 1{,}000$ yields $2^{\beta-h} = 2^{-21}$. Also, $n = 10{,}000$ yields $2^{\beta-h} = 2^{-19}$. This is the reason why our method can produce the accurate approximation of a matrix product. We give the following corollary.

Corollary 1 *Assume that both the matrices A and B are divided into k floating-point matrices and we apply the accurate summation algorithm for the final summation in Algorithm* 5. *If*

$$2^{-\omega} \le \left|\frac{a_{rs}}{a_{ij}}\right|, \left|\frac{b_{rs}}{b_{i'j'}}\right| \le 2^{\omega} \text{ for all combinations of indices and } \omega > 0 \tag{42}$$

is satisfied, then we obtain

$$|D^{(2)} - AB| \le \left(2^{(\beta-h)(k-1)+2\omega+1} + \frac{2(1-n\mathbf{u})}{n}\right)\gamma_n |A||B|. \tag{43}$$

Proof From the definition of $P^{(1)}$ and $Q^{(1)}$, and (42), we have

$$n\left(2^{P^{(1)}}\right)\left(2^{Q^{(1)}}\right)^T \le 2^{2\omega}|A||B|. \tag{44}$$

From (30), we obtain

$$|C| \le |AB| + E \le |A||B| + E. \tag{45}$$

By substituting (44) and (45) into (32), and a little consideration, (43) can be obtained. □

If there is not much difference in the order of the magnitude in the elements in the same row in A and the elements in the same column in B, $k = 2$ is enough to obtain a better bound than (33) except for huge n. Suppose, as an example, $10^{-2} \le a_{ij}, b_{ij} \le 10^2$ for all i, j and $n = 1{,}000$. Then

$$|D^{(2)} - AB| \le 2^{-6}\gamma_n |A||B|.$$

4.2 Level 3 fraction

In order to show the performance of the proposed method, we derive a level 3 fraction for Algorithm 5:

$$\texttt{Acc\_Mul}(A, B, k, \delta), \quad 0 \le \delta \le 1$$

Here, the level 3 fraction shows the amount of matrix multiplication in the given algorithm [5]. In this paper, we consider the ratio:

$$\frac{\text{computational costs for dense matrix products}}{\text{computational costs for Algorithm 5}}$$

The computational costs for Algorithm 5 are counted by 'flops'. To simplify the discussion simple, let $A, B \in \mathbb{F}^{n\times n}$ be square matrices. Then the number of matrix products becomes $k(k-1)/2 + k =: t$. Let s be the number of sparse matrix products. It implies that the number of dense matrix products is $t - s$. Computational costs for dense matrix products are $2n^3(t-s)$ flops. For sparse matrix multiplication, we count the maximal computational costs as $2s\delta n^3$ flops. Note that the costs for splitting matrices and others are $\mathcal{O}(n^2)$ flops. If n is large to extent, in total, the minimal ratio of the amount of dense matrix products in whole computations becomes

$$\frac{2n^3(t-s)}{2n^3(t-s) + s \cdot 2\delta n^3 + \mathcal{O}(n^2)} \approx \frac{t-s}{t + s(\delta - 1)}. \tag{46}$$

This ratio confirms that for few sparse matrix products the level 3 fraction for our algorithm is quite high. Namely, our method derives full benefit of the optimization by BLAS.

5 Conclusion

In this paper, we have investigated an error-free transformation of a matrix product. We transformed the product of two floating-point matrices into an unevaluated sum of floating-point matrices without roundoff errors. Using this error-free transformation, we presented a method which outputs an accurate approximation of the matrix product. Our method uses mainly level 3 routines by optimized BLAS resulting a good performance. Finally, we showed why our method produces an accurate approximation of the matrix product. The accuracy of the computed result increases with the number of summands by the error-free transformation of the matrix product.

References

1. IEEE Standard for Floating-Point Arithmetic, Std 754–2008 (2008)
2. Bailey, D.H.: A Fortran-90 based multiprecision system. ACM Trans. Math. Softw. **21**(4), 379–387 (1995)
3. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18**(3), 224–242 (1971)
4. Demmel, J., Hida, Y.: Accurate and efficient floating point summation. SIAM J. Sci. Comput. **25**(4), 1214–1248 (2003)
5. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore (1996)
6. Goto, K., Geijn, R.V.D.: High-performance implementation of the level-3 BLAS. ACM Trans. Math. Softw. **35**(1), Article no. 4 (2008)
7. Li, X., Demmel, J., Bailey, D., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S., Kapur, A., Martin, M., Thompson, B., Tung, T., Yoo, D.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2), 152–205 (2002)
8. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, 2nd edn. SIAM Publications, Philadelphia (2002)
9. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**, 1955–1988 (2005)
10. Rump, S.M.: Ultimately fast accurate summation. SIAM J. Sci. Comput. **31**(5), 3466–3502 (2009)
11. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part I: faithful rounding. SIAM J. Sci. Comput. **31**(1), 189–224 (2008)
12. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part II: sign, K-fold faithful and rounding to nearest. SIAM J. Sci. Comput. **31**(2), 1269–1302 (2008)
13. Whaley, C.R., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Comput. **27**, 3–35 (2001)
14. The MPFR Library: http://www.mpfr.org/. Accessed 7 Feb 2010
15. exflib—extend precision floating-point arithmetic library: http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/exflib-index.html. Accessed 7 Feb 2010
16. http://www.eecs.berkeley.edu/~yozo/. Accessed 25 Dec 2009
17. The NIST Sparse BLAS: http://math.nist.gov/spblas/original.html. Accessed 7 Feb 2010
18. http://www.cise.ufl.edu/research/sparse/umfpack/. Accessed 7 Feb 2010
19. MATLAB Programming Version 7, The MathWorks (2005)
20. Rump, S.M.: INTLAB—INTerval LABoratory. In: Csendes, T. (ed.) Developments in Reliable Computing, pp. 77–104. Kluwer Academic, Dordrecht (1999)