Accurate Matrix Multiplication with Multiple Floating-point Numbers

Katsuhisa Ozaki[†], Takeshi Ogita[‡,†], Siegfried M. Rump[∗,†], Shin’ichi Oishi[†]

† Faculty of Science and Engineering, Waseda University, Tokyo 169-8555, Japan
‡ CREST, Japan Science and Technology Agency (JST)
∗ Institute for Reliable Computing, Hamburg University of Technology
Email: k_ozaki@aoni.waseda.jp

Abstract—This paper is concerned with an accurate computation of matrix multiplication, where components of matrices are represented by summation of floating-point numbers. Recently, an accurate summation algorithm is developed by the latter three of the authors. In this paper, it is specialized to dot product. Using this, a fast implementation of accurate matrix multiplication is discussed. Finally, some numerical results are presented to confirm the effectiveness of the proposed algorithm.

1. Introduction

In this paper, we are concerned with the accurate computation of matrix multiplication

$$C = AB, \quad A \in \mathbb{R}^{m\times p},\ B \in \mathbb{R}^{p\times n}.$$

Fast implementations of multi-precision arithmetic have frequently been discussed and there are many software libraries for that purpose. However, most of such libraries do not guarantee an accuracy of the result because a problem may be arbitrary ill-conditioned i.e. in general, necessary precision for obtaining reliable results can not be known in advance.

Recently, an accurate summation algorithm [5] is developed by Rump, Ogita and Oishi (ROO). ROO algorithm can always give a computed result with faithful rounding by adequate computational cost. Faithful rounding means that the computed result is one of the floating-point neighbors of a true result. Moreover, ROO algorithm has an advantage in terms of measured computing time. In the main computational part, there is neither data dependency nor branch in the inner loop. So, ROO algorithm tends to be affected by compiler’s optimization effectively.

Matrix multiplication consists of dot product. If we use error-free transformation described later section, the computation of dot product can be transformed into that of summation without rounding errors. Therefore, we can immediately apply ROO algorithm and also obtain the result with faithful rounding. However, it is not efficient to apply ROO algorithm straightforwardly in terms of computational cost.

In this paper, we specialize ROO algorithm to dot product. By focusing on the data structure from the transformation of summation into dot product, the computational parts for each loop can be limited. As a result, we can develop a fast and accurate algorithm of calculating dot product. Next, we extend it to a fast implementation for accurate matrix multiplication, whose results achieve faithful rounding. Finally, we present some numerical results to confirm an effectiveness of our algorithm.

2. Multiple floating-point number

In this section, we explain how we represent multi-precision number. First, we state the notation used throughout this paper. Let $\mathbb{F}$ be a set of floating-point numbers. Let $\mathrm{fl}(\cdots)$ be the result of a floating-point computations. Let $\mathbf{u}$ be the roundoff unit (especially, $\mathbf{u} = 2^{-53}$ in double precision defined in IEEE standard 754).

A normalized double precision floating-point number defined by IEEE 754 has 53 mantissa bits. For example, if there is a big difference in the order of magnitude between two positive floating-point numbers a and b with $2^{-40}a \sim b$, the result of $\mathrm{fl}(a + b)$ loses about lower 40 bits of b. To overcome such problems, we represent a multi-precision number as an additive form of floating-point numbers. For example, to express double-double precision number d which has 106 mantissa bits, we represent the number as summation of two floating-point numbers $a^{(1)}, a^{(2)}$ like $d = a^{(1)} + a^{(2)}$. Ideally, $a^{(1)}$ and $a^{(2)}$ express the leading 53 bits of d and the rest $d - a^{(1)}$, respectively. Generally, we represent a number a^* as

$$a^* = a^{(1)} + a^{(2)} + a^{(3)} + \ldots + a^{(k)}, \quad a^{(i)} \in \mathbb{F}. \tag{1}$$

We call this a^* *the multiple floating-point number*. Figure 1 shows the best case called as non-overlapping expansion. We relax the definition of multiple floating-point numbers from non-overlapping expansion to gain the computational speed. For a positive constant c, we assume that the following inequality is satisfied:

$$(c\mathbf{u})^{j-i}|a^{(i)}| \geq |a^{(j)}| \quad \text{for } i < j \tag{2}$$

If c is of $O(1)$, the multiple floating-point number a^* has almost k-fold working precision.

3. Error-free transformation

For $a, b \in \mathbb{F}$, there is a well-known algorithm due to G. W. Veltkamp (see [1]) which transforms the product $a \cdot b$ into $x + y$ with $x, y \in \mathbb{F}$:

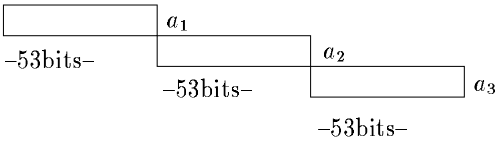

Figure 1: Ideal case for a multiple floating-point number

Algorithm 1 *Error-free transformation of the product of two floating-point numbers.*

```
function $[x, y]$ = TwoProduct($a, b$)
    $x = \mathrm{fl}(a \cdot b)$
    $[a_1, a_2]$ = Split($a$)
    $[b_1, b_2]$ = Split($b$)
    $y = \mathrm{fl}(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))$
```

Then $a \cdot b = x + y$ with $x = \mathrm{fl}(a \cdot b)$ and

$$\mathbf{u}|x| \geq |y|. \qquad (3)$$

Algorithm 1 relies on the splitting by Dekker [1] of a 53-bit floating-point number into two 26-bit parts. Throughout the paper, we denote it as `Split`.

Such algorithms are so-called error-free transformations [2] and very useful for accurate computations by floating-point arithmetic. Moreover, we introduce a new error-free vector transformation [5]. For a vector $p \in \mathbb{F}^n$, the following algorithm provides p' and τ such that $\tau + \sum_{i=1}^{n} p'_i = \sum_{i=1}^{n} p_i$.

Algorithm 2 (Rump-Ogita-Oishi [5]) *For $p \in \mathbb{F}^n$, the following algorithm computes p' and τ such that $\tau + \sum_{i=1}^{n} p'_i = \sum_{i=1}^{n} p_i$ without error.*

```
function $[p', \tau]$ = ExtractVec($p, \sigma$)
    $q = (\sigma + p) - \sigma$;          % $q_i = (\sigma + p_i) - \sigma$
    $\tau$ = sum($q$);                    % $\tau = \sum_{i=1}^{n} q_i$
    $p' = p - q$;                         % $p'_i = p_i - q_i$
```

For simplicity of the latter discussions, we define a general function `err` as

$$\mathtt{err}(f, g) := f - g, \qquad f, g \in \mathbb{R}.$$

For example, let $x = \mathrm{fl}(a \cdot b)$. Then

$$y' = \mathtt{err}(a \cdot b, x),$$

which is identical to the output y of Algorithm 1 (`TwoProduct`).

4. Accurate dot product

In this section, we illustrate how to specialize the accurate summation algorithm to the computation of dot product for multiple floating-point numbers.

First, we introduce the ROO algorithm for accurate summation:

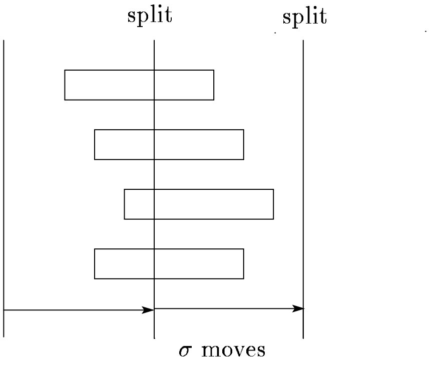

Figure 2: Image of accurate summation algorithm

Algorithm 3 (Rump-Ogita-Oishi [5]) *For $p \in \mathbb{F}^n$, the following algorithm computes* `res` *with faithful rounding of $\sum p$.*

```
function res = AccSum($p$)
    $t = 0$;    $M = \lceil \log_2(n + 2) \rceil$;
    $\phi = \mathbf{u} \cdot 2^M$;    factor $= 2^M \cdot \phi$;
    $\sigma = 2^{M + \lceil \log_2 \max_i(|p_i|) \rceil}$;
    while (true)
        $[p, \tau]$ = ExtractVec($p, \sigma$)
        $\tau_1 = t + \tau$;
        if $|\tau_1| >$ factor $\cdot \sigma$
            $\tau_2 = \tau 1 - (\tau_1 - t)$;
            res $= \tau_1 + (\tau_2 +$ sum($p$));
            return;
        end if
        $t = \tau_1$;
        $\sigma = \sigma \cdot \phi$;
    end
```

Here, Figure 2 explains the image of Algorithm 3.

Next, we explain how to transform the computation of dot product into that of summation without rounding errors. Let $x = (x_1, x_2, \ldots, x_n)^T$ and $y = (y_1, y_2, \ldots, y_n)^T$ with $x_i = x_i^{(1)} + x_i^{(2)} + \ldots + x_i^{(s)}$ and $y_i = y_i^{(1)} + y_i^{(2)} + \ldots + y_i^{(t)}$. Then

$$\begin{aligned} x_i y_i &= (x_i^{(1)} + x_i^{(2)} + \ldots + x_i^{(s)})(y_i^{(1)} + y_i^{(2)} + \ldots + y_i^{(t)}) \\ &= x_i^{(1)} y_i^{(1)} + x_i^{(1)} y_i^{(2)} + \ldots + x_i^{(s)} y_i^{(t)}. \qquad (4) \end{aligned}$$

It can be seen that the number of terms $x_i^{(j)} y_i^{(k)}$ in (4) becomes st. When applying `TwoProduct` to $x_i^{(j)} y_i^{(k)}$ for all (j, k), the computation of $x_i y_i$ can be transformed into summation with $2st$ components. By applying the same discussion to $x_i y_i$ for all i, we obtain an array with $2stn$ components from the dot product $x^T y$.

Here, we remark on the data structure of this array and explain how to apply Algorithm 3. When executing

$$[q_1, q_2] \quad = \quad \mathtt{TwoProduct}(x_i^{(1)}, y_i^{(1)})$$

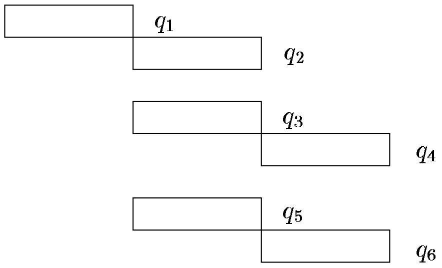

Figure 3: Difference of magnitude among data

$$\begin{aligned}[q_3, q_4] &= \texttt{TwoProduct}(x_i^{(1)}, y_i^{(2)})\\ [q_5, q_6] &= \texttt{TwoProduct}(x_i^{(2)}, y_i^{(1)}),\end{aligned}$$

the following relations hold:

$$\mathbf{u}|q_1| \geq |q_2|, \quad c\mathbf{u}|q_1| \geq |q_3|, \quad c\mathbf{u}|q_1| \geq |q_5|$$
$$(c\mathbf{u})^2|q_1| \geq |q_4|, \quad (c\mathbf{u})^2|q_1| \geq |q_6|,$$

where c is the positive constant of $O(1)$ in (2). These relations show that there is a big difference in the order of magnitude among the data obtained from the transformation.

For any $a, b \in \mathbb{F}$ with $\mathbf{u}|a| \geq |b|$, it holds that

$$\mathrm{fl}(a + b) = a. \tag{5}$$

For $x_i y_i$ for all i, the same discussion can be applied. From the definition of σ in Algorithm 3, the following inequality holds:

$$\sigma > \max_{1 \leq i \leq n} |x_i^{(1)} y_i^{(1)}| \tag{6}$$

Let $t_i := \texttt{err}(x_i^{(1)} y_i^{(1)}, \mathrm{fl}(x_i^{(1)} y_i^{(1)}))$. By (3), it holds that

$$\mathbf{u}|\mathrm{fl}(x_i^{(1)} y_i^{(1)})| \geq |t_i|.$$

This and (6) yields

$$\mathbf{u}\sigma > |t_i|. \tag{7}$$

Then considering (5) and (7), we have

$$\mathrm{fl}(\sigma + t_i) = \sigma \quad \text{for } 1 \leq i \leq n.$$

If c is smaller than $\lfloor \log_2 n \rfloor$, then it holds that

$$\mathrm{fl}(\sigma + x_i^{(j)} y_i^{(k)}) = \sigma \quad \text{for } j + k > 2.$$

In that case, it holds in Algorithm 2 that $q_i = 0$ for all i, $\tau = 0$ and $p' = p$. Therefore, we need not to compute $\texttt{err}(x_i^{(j)} y_i^{(j)}, \mathrm{fl}(x_i^{(j)} y_i^{(j)}))$ for $j \geq 2$ nor $\mathrm{fl}(x_i^{(j)} y_i^{(k)})$ for $j + k > 2$ in the first loop in Algorithm 3. Utilizing this fact, in the first loop, we restrict ourselves to generate a vector p from dot product $x^T y$ by only using $\mathrm{fl}(x_i^{(1)} y_i^{(1)})$, i.e.,

$$p^{(1)} = (\mathrm{fl}(x_1^{(1)} y_1^{(1)}), \mathrm{fl}(x_2^{(1)} y_2^{(1)}), \ldots, \mathrm{fl}(x_n^{(1)} y_n^{(1)}))^T \in \mathbb{F}^n.$$

If the stopping criterion in the first loop in Algorithm 3 is not triggered, then we proceed to the next loop and generate additional data from x and y. Then, we can also restrict ourselves to generate a vector $p^{(2)}$ by the similar discussion to the above one.

For simplicity, we define a general function which generates necessary data from x and y and adds them to the vector p adapting to the k-th loop:

$$p = \texttt{AddElement}(p, x, y, k)$$

For example, when $k = 1$, this function generates

$$\mathrm{fl}(x_i^{(1)} y_i^{(1)}) \quad \text{for } 1 \leq i \leq n$$

and adds them to p. When $k = 2$, this function adds

$$\mathrm{fl}(x_i^{(2)} y_i^{(1)}),\ \mathrm{fl}(x_i^{(1)} y_i^{(2)}),\ \texttt{err}(x_i^{(1)} y_i^{(1)}, \mathrm{fl}(x_i^{(1)} y_i^{(1)}))$$

to p. Moreover, when $k = 3$, this function adds

$$\mathrm{fl}(x_i^{(1)} y_i^{(3)}),\ \mathrm{fl}(x_i^{(2)} y_i^{(2)}),\ \mathrm{fl}(x_i^{(3)} y_i^{(1)}),$$
$$\texttt{err}(x_i^{(2)} y_i^{(1)}, \mathrm{fl}(x_i^{(2)} y_i^{(1)})),\ \texttt{err}(x_i^{(1)} y_i^{(2)}, \mathrm{fl}(x_i^{(1)} y_i^{(2)}))$$

to p. Note that the length of p is changed according to k. When k increases sufficiently and all data are generated, this function does not change the vector p.

Summarizing the above-mentioned discussions, we here present an algorithm of accurate dot product.

Algorithm 4 (Accurate dot product) *For two vectors x, y whose components are multiple floating-point numbers, the following algorithm computes* `res` *where* `res` *is a computed result with faithful rounding of $x^T y$.*

```
function res = AccDot($x, y$)
    $p = []$;
    $p$ = AddElement($p, x, y, 1$);
    $M = \lceil \log_2(n + 2) \rceil$;
    $\phi = \mathbf{u} \cdot 2^M$;    factor $= 2^M \cdot \phi$;
    $\sigma = 2^{M + \lceil \log_2 \max_i(|p_i|) \rceil}$;
    $t = 0$;    $k = 2$;
    while ($true$)
        $[p, \tau]$ = ExtractVec($p, \sigma$);
        $\tau_1 = t + \tau$;
        $p$ = AddElement($p, x, y, k$);
        if $|\tau_1| >$ factor $\cdot\, \sigma$
            $\tau_2 = \tau_1 - (\tau_1 - t)$;
            res $= \tau_1 + (\tau_2 +$ sum($p$));
            return;
        end if
        $t = \tau_1$;
        $\sigma = \sigma \cdot \phi$;
        $k = k + 1$;
    end
```

If the algorithm stops for small k because the given problem is well-conditioned, then the algorithm does not generate all data for p from x and y so that some computational cost can be reduced.

5. Matrix multiplication

In this section, we discuss how to utilize an accurate dot product algorithm (Algorithm 4) for matrix multiplication. A basic algorithm can be written as follows:

Algorithm 5 (Accurate matrix multiplication) *Let A and B be $m \times p$ and $p \times n$ matrices, respectively. Assume that each element of A, B is represented by a multiple floating point number as $A = \sum_{i=1}^{s} A^{(i)}$, $B = \sum_{i=1}^{t} B^{(i)}$ with $A^{(i)} \in \mathbb{F}^{m\times p}$ and $B^{(i)} \in \mathbb{F}^{p\times n}$. Then the following algorithm computes $C \in \mathbb{F}^{m\times n}$, which is a faithful rounding of AB.*

```
function $C$ = AccMatMul($A, B$)
    for $i = 1 : m$
        for $j = 1 : n$
            $C(i, j)$ = AccDot($A(i, :), B(:, j)$)
        end
    end
```

One may consider that `Split` for A and B called in `AccDot` can be moved out of the for-loops, e.g. $A^{(i)} = A_1^{(i)} + A_2^{(i)}$ where $A_1^{(i)}$ corresponds to the leading 26 bits of $A^{(i)}$ and $A_2^{(i)}$ the rest $A^{(i)} - A_1^{(i)}$. However, if all elements of A and B are split into two parts and all the results are stored, much amount of memory is required. Moreover, when a given problem is well-conditioned, splitting all elements of A and B may be wasted because it may be not necessary for the algorithm to generate all data from A and B for calculating the faithful rounding of AB.

Therefore, it seems that there is a tradeoff in the accurate computation of the matrix multiplication AB between the computational speed and memory. The computational speed also depends on how ill-conditioned the problem is. Namely, we should carefully implement the algorithm by considering these aspects.

6. Numerical results

In this section, we present some numerical results. All computations are done in IEEE 754 double precision on Matlab 7.1 using the mex function in C language. Numerical experiments are done on a PC with Pentium IV 2.6GHz CPU.

First, we generate an $n \times n$ matrix A whose components are multiple floating-point numbers such that

$$A = \sum_{i=1}^{7} A^{(i)}, \quad A^{(i)} \in \mathbb{F}^{n\times n}$$

with arbitrary condition number by using a method in [4]. Here, for a square matrix X, condition number of X is defined by

$$\mathrm{cond}_2(X) := \|X\|_2 \|X^{-1}\|_2.$$

Next, we compute an approximate inverse R of A by Rump's method [3] as

$$R = \sum_{i=1}^{7} R^{(i)}, \quad R^{(i)} \in \mathbb{F}^{n\times n}.$$

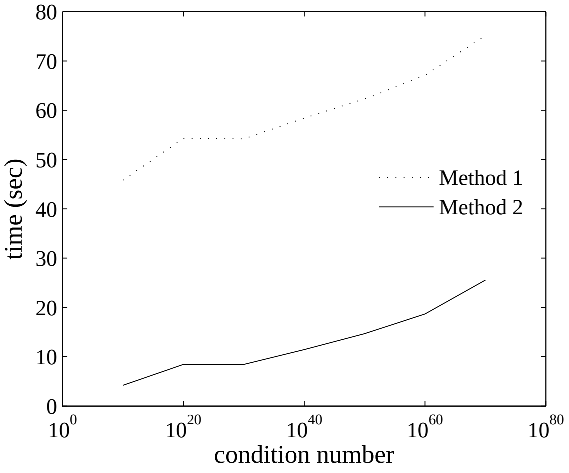

Figure 4: Elapsed time for calculating RA for $n = 300$

Then, we aim on computing a faithful rounding of RA. Note that the higher condition number of A is, the more heavy cancellations occur in the computation of RA.

For $n = 300$, we compare the following two methods:

Method 1 A straightforward method by using Algorithm 1 (`TwoProduct`) for $x^T y$ and Algorithm 3 (`AccSum`)

Method 2 Proposed method (Algorithm 5)

Figure 6 displays the elapsed time for calculating RA by Methods 1 and 2 for various $\mathrm{cond}_2(A)$.

Method 2 works adaptively until obtaining the faithful rounding of RA. As a result, it can be seen that Method 2 is from 3 to 9 times faster than Method 1 in this example.

References

[1] T. J. Dekker, A floating-point technique for extending the available precision, Numer. Math., 18 (1971), 224–242.

[2] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. SIAM Journal on Scientific Computing, 26:6 (2005), 1955–1988.

[3] S. Oishi, K. Tanabe, T. Ogita, and S. M. Rump. Convergence of Rump's method for inverting arbitrarily ill-conditioned matrices, J. Comp. Appl. Math., 205:1 (2007), 533–544.

[4] S. M. Rump. A class of arbitrarily ill-conditioned floating-point matrices, SIAM J. Matrix Anal. Appl, 12:4 (1991), 645–653.

[5] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation, Part I and II, submitted for publication.