МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2004
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2004
Proceedings of the Thirty Third Spring Conference of
the Union of Bulgarian Mathematicians
Borovets, April 1–4, 2004

EXACT SCALAR PRODUCT ON HIGH-PERFORMANCE COMPUTING*

Hassan El-Owny

Algorithms for summation and dot product of floating point numbers are presented which are fast in terms of measured computing time. The algorithms are widely applicable because they require only addition, subtraction and multiplication of floating point numbers in the same working precision as the given data, no other data format (higher precision) is necessary. The computed results are shown to be as accurate as if computed in doubled or k-fold working precision.

1. Introduction. In this paper fast algorithms presented to compute the dot product of floating point numbers in a specified precision. This algorithms are based on so called error-free transformations [10]. It is for long known that the approximation error of a floating point operation is itself a floating point number. That means, a pair of floating point numbers can be transformed into another pair of floating point numbers, one being equal to the result of the floating point operation between the input numbers and the other representing the (precise) approximation error. Fortunately, very fast algorithms are known to compute such pairs for all operations we need, that is for addition, subtraction and multiplication. Amazingly, those algorithms consist only of the same basic floating point operations [2], [7], [9]. Throughout the paper assume a floating point arithmetic adhering to IEEE 754 floating point standard [5], and that no overflow occurs, but we allow underflow. We will use only one working precision for floating point computations. If this working precision is IEEE 754 double precision, then this corresponds to 53 bits precision including implicit one. The set of floating point numbers in this working precision is denoted by $\mathbb{F}$, the relative rounding error unit – by eps, and the underflow unit — by eta. For IEEE 754 double it is eps $= 2^{-53}$ and eta $= 2^{-1073}$. We denote by fl(.) the result of a floating point computation, where all operations inside parentheses are executed (in floating point) in working precision. It is well known that for each of the four basic operations the approximation error of the floating point operation can be expressed by a floating point number:

$$\begin{aligned} x = fl(a \pm b) &\Rightarrow a \pm b = x + y \quad for \quad y \in \mathbb{F}, \\ x = fl(a \cdot b) &\Rightarrow a \cdot b = x + y \quad for \quad y \in \mathbb{F}, \\ x = fl(a/b) &\Rightarrow a = x \cdot b + y \quad for \quad y \in \mathbb{F}, \end{aligned} \tag{1}$$

*2000 Mathematics Subject Classification: 65G99, 65Y99, 68M99.
Key words: High-performance computing, exact scalar product, fast algorithms, verified error bounds.

where in the case of multiplication and division it is assumed that no underflow occurs. This is true for all floating point numbers $a, b \in \mathbb{F}$. These are error-free transformations of the pair (a, b) into (x, y), where x is the result of the corresponding floating point operation. Note that no information is lost; the equalities in (1) are mathematical identities.

2. Error-free transformations. Our goal is to extend the error-free transformations for the sum and product of two floating point numbers to sums of vector elements and to dot products, respectively. Therefore, the first two of the identities in (1) will play a fundamental role in the following. Fortunately, the quantities x, y are effectively computable, without any if-statements, only using ordinary floating point addition, subtraction and multiplication. For addition (and subtraction by adding $-b$) the following algorithm by Knuth [6] can be used.

$$\begin{aligned} \text{function}[x, y] &= \text{TwoSum}(a, b) \\ x &= fl(a + b) \\ z &= fl(x - a) \\ y &= fl((a - (x - z) + (b - z)) \end{aligned}$$

ALGORITHM(1). Error-free transformation of the sum of two floating point numbers.

The multiplication routine needs to split the input arguments into two parts, respectively. The number p is given by $eps = 2^{-p}$, and we define $s := \lceil p/2 \rceil$; in IEEE 754 double precision it is $p = 53$ and $s = 27$. The following algorithm by Dekker [3] splits a floating point number $a \in \mathbb{F}$ into two parts x,y, where both parts have at most $s - 1$ nonzero bits. In a practical implementation the variable "factor" can, of course, be replaced by a constant.

$$\begin{aligned} \text{function}[x, y] &= \text{Split}(a) \\ \text{factor} &= 2^s + 1 \\ c &= fl(\text{factor}.a) \\ x &= fl(c - (c - a)) \\ y &= fl(a - x) \end{aligned}$$

ALGORITHM(2): Error-free split of a floating point number into two parts.

The multiplication to calculate "c" in Algorithm 2 cannot cause underflow except when the input "a" is deep in the gradual underflow range. Since addition and subtraction is exact in case of underflow, the analysis [3] of Split is still valid and we obtain

$$a = x + y \quad \text{and} \quad x \quad \text{and} \quad y \quad \text{non-overlapping with} \quad |y| \leq |x|$$

With this the following multiplication routine by G. W. Veltkamp (see [3]) can be formulated.

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

ALGORITHM(3): Error-free transformation of the product of two floating point numbers.

Note that no branches, only basic and good optimizable floating point operations are necessary. In case no underflow occurs, we know [3] that

$$a\cdot b = x + y \quad \text{and} \quad x = fl(a\cdot b).$$

We summarize the properties of the algorithms TwoSum and twoProduct as follows (flops denotes the number of floating point operations counting additions, subtractions and multiplications separately).

Theorem 1. *Let $a, b \in \mathbb{F}$ and denote the results of Algorithm 1 (TwoSum) by x, y. Then, also in the presence of underflow,*

$$a + b = x + y,\ x = fl(a + b),\ |y| \le eps|x|,\ |y| \le eps|a + b|.$$

The algorithm TwoSum requires 6 flops.
Let $a, b \in \mathbb{F}$ and denote the results of Algorithm 3 (TwoProduct) by x, y. Then, if no underflow occurs,

$$a\cdot b = x + y,\ x = fl(a\cdot b),\ |y| \le eps|x|,\ |y| \le eps|a\cdot b|.$$

and in the presence of underflow,

$$a\cdot b = x + y + 5\eta,\ x = fl(a\cdot b),\ |y| \le eps|x| + 5eta,\ |y| \le eps|a\cdot b| + 5eta \text{ with } |\eta| \le eta.$$

The algorithm TwoProduct requires 17 flops.

3. Summation. Let floating point numbers $p_i \in \mathbb{F}$, $0 \le i \le n$, be given. In this section we aim to compute a good approximation of the sum $s = \sum p_i$. With Algorithm 1 we have a possibility to add two floating point numbers with exact error term. So we may try to cascade Algorithm 1 and sum up the error in order to improve on the result of the ordinary floating point summation $fl(\sum p_i)$.

$$\begin{gathered}\text{function res} = \text{Sum}K(p,K)\\ \text{for } k = 1:K\\ p = \text{VecSum}(p)\\ \text{res} = \text{fl}((\textstyle\sum_{i=0}^{n-2} p_i) + p_{n-1})\end{gathered}$$

ALGORITHM(4): K-fold error-free vector transformation for summation and approximate sum.

Where VecSum(p) is transforms Algorithm for the vector p without changing the sum and replace p_n by $\text{fl}(\sum p_i)$.

$$\begin{gathered}\text{function } p = \text{VecSum(p)}\\ \text{for } i = 1:n\\ [p_i, p_{i-1}] = \text{TwoSum}(p_i, p_{i-1})\end{gathered}$$

ALGORITHM(5): Error-free vector transformation for summation.

Proposition 1. *Let floating point numbers $p_i \in \mathbb{F}$, $0 \le i \le n$, be given and $K \ge 1$. Then, also in the presence of underflow, the result "res" of Algorithm 4 (SumK) satisfies*

$$\begin{aligned} |res - s| &\le (eps + \gamma_{n-1}\gamma_{4n-4})|s| + \gamma_{2n-2}^{K+1} S; \\ &\le (eps + \gamma_{2n}^2)|s| + \gamma_{2n}^{K+1} S; \end{aligned}$$

where $s := \sum p_i$ and $S := \sum |p_i|$. Algorithm 4 requires $(6K+1)(n-1)$ flops.

4. Dot product. With Algorithm 3 we already have an error-free transformation of the product of two floating point numbers into the sum of two floating point numbers. The algorithm for $K = 1$ corresponding to doubled working precision is as follows.

$$\begin{gathered} \text{function res} = \text{Dot1}(x, y) \\ [p, s] = \text{TwoProduct}(x_0, y_0) \\ \text{for } i = 1 : n \\ [h, r] = \text{TwoProduct}(x_i, y_i) \\ [p, q] = \text{TwoSum}(p, h) \\ s = \text{fl}(s + (q + r)) \\ \text{res} = \text{fl}(p + s) \end{gathered}$$

ALGORITHM(6): Dot product in doubled working precision.

Proposition 2. *Let $x_i, y_i \in \mathbb{F}$, $0 \le i \le n$, be given and denote by $res \in \mathbb{F}$ the result computed by Algorithm 6 (Dot1). Then, if no underflow occurs,*

$$|res - x^T y| \le eps|x^T y| + \gamma_n^2 |x^T||y|,$$

and in the presence of underflow,

$$|res - x^T y| \le eps|x^T y| + \gamma_n^2 |x^T||y| + 5neta.$$

Algorithm 6 requires $25n - 7$ flops.

$$\begin{gathered} \text{function res} = \text{Dot}K(x, y, K) \\ [p, r_0] = \text{TwoProduct}(x_0, y_0) \\ \text{for } i = 1 : n \\ [h_i, r_i] = \text{TwoProduct}(x_i, y_i) \\ [p, r_{n+i-1}] = \text{TwoSum}(p, h) \\ r_{2n} = p \\ \text{res} = \text{SumK}(r, K - 1) \end{gathered}$$

ALGORITHM(7): Dot product with K-fold summation, $K \ge 2$.

Proposition 3. *Let $x_i, y_i \in \mathbb{F}$, $0 \le i \le n$, be given and denote by $res \in \mathbb{F}$ the result computed by Algorithm 7 (DotK). Then, if no underflow occurs,*

$$|\text{res} - x^T y| \le (\text{eps} + 2\gamma_{4n-1}^2)|x^T y| + \gamma_{4n-2}^{K+1}|x^T||y|,$$

and in the presence of underflow,

$$|\mathrm{res} - x^T y| \quad \leq \quad (\mathrm{eps} + 2\gamma_{4n-1}^2)|x^T y| + \gamma_{4n-2}^{K+1}|x^T||y| + 5n\mathrm{eta}.$$

Algorithm 7 requires $13n + 6K(2n-1) - 1$ *flops.*

4. Numerical results. In this section we present computational results for the dot product in doubled working precision (Algorithm 6, Dot1) and for the dot product in K-fold working precision (Algorithm 7, DotK). This also tests the summation algorithms presented in Section 3. We will not say about the many practical applications of algorithms because *i*) this is widely known, and *ii*) there are excellent treatment in the recent literature [4], [8].

For the measurement of the actual computing time compared to BLAS, the algorithms were implemented in C++ and tested on the following high-Performance computing environment (ALiCE) [1]:

- **Nodes:** 128 Compaq DS10
- **Processors:** 616MHz Alpha 21264 EV6/7
- **Cache:** 2Mbyte($2^n d$), 64k/64k($1^s t$)
- **Memory:** 256MB/node → 32 Gbytes
- **Disks:** 10GB/node → 1.3 TB
- **Connectivity:** 64bit/33MHz Myrinet.
- **Power:** 15 kW

We used Compaq C++ compiler and GNU g++ compiler, and BLAS routines were taken from Compaq Extended Math Library.

We tested Algorithm 6 (Dot1, corresponding to quadruple precision), and Algorithm 7 (DotK) for the dot product in K-fold working precision for dimension 100 to 5000 in steps of 100. This also tests summation algorithms.

A more practical application are programs or higher-level operations involving dot products. A typical such operation is matrix-vector multiplication. Therefore we wrote a rather straightforward code for the computation of $A * y - b$ for a given $n \times n$ matrix A and n-vectors y and b using Dot1 and tested against the corresponding BLAS routine DGEMV. For dimensions 100, 200, ..., 3000. We display the ratio of the measured computing time of this routine compared to the measured computing time of the BLAS routine DGEMV.

Example: Suppose

$$X = \begin{vmatrix} 1 \\ w \\ -w \\ w \\ -w \\ \cdots \\ \cdots \\ w \\ -w \\ n \end{vmatrix}, \; Y = \begin{vmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \cdots \\ \cdots \\ \cdots \\ \cdots \\ 1 \end{vmatrix}, \quad \begin{vmatrix} 1 & w & -w & \cdots & \cdots & \cdots & n \\ 2 & w & -w & \cdots & \cdots & \cdots & n-1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ n & w & -w & \cdots & \cdots & \cdots & 1 \end{vmatrix}$$

where $w = e^{20}$ and n is even. We obtained the exact result by using Dot1 and DotK algorithms.

minimum	median	maximum
9.654	7.98204	19.5484

Table 1. Measured ratio of computing time for matrix-vector residual

K	minimum	median	maximum
1	0.33	0.5	1
2	16	22.43	38
3	23	38.33	47.15
4	29	45.2	54.81
5	34	47	69.36
6	39.5	59.2	79.36
7	44	67.5	87.94

Table 2. Measured ratio of computing time for dot products

5. Conclusion. The algorithms use only basic floating point operations addition, subtraction and multiplication, and they use only the same working precision as the data are given in. This offers the possibility to put them into numerical library algorithms since no special computer architecture is required. We stress again that the algorithms are based on the error-free transformations TwoSum and twoProduct. If those would be available directly from the processor, precise dot product evaluation in double working precision by Dot1 would cost only twice as much as the ordinary dot product.

REFERENCES

[1] ALiCE: http://alice.iai.uni-wuppertal.de/

[2] J. H. Davenport, Y. Siret, G. Tournier. Computer algebra: Systems and algorithms for algebraic computation. Academic Press, London, 2nd edition, 1993. Transl. from the French by A. Davenport a. J. H. Davenport (eds. English).

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

[4] N. J. Higham. Accuracy and Stability of Numerical Algorithms. SIAM Publications, Philadelphia, 2nd edition, 2002.
[5] ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic, 1985.
[6] D. E. Knuth. The Art of Computer programming: Seminumerical Algorithms, volume 2. Addison Wesley, Reading, Massachusetts, second edition 1981.
[7] H. Leuprecht, W. Oberaigner. Parallel algorithms for the rounding exact summation of floating point numbers. *Computing*, **28** (1982), 89–104.
[8] X. Li. et al. Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Trans. Math. softw.*, **28(2)** (2002), 152–205.
[9] M. Malcolm. On accurate floating-point summation. *Comm. ACM*, **14(11)** (1971), 731–736.
[10] Takeshi Ogita, Siegfried M. Rump, Shin'ichi Oishi. *Accutare Sum and Dot Product*, submitted for publication, 2003.

University of Wuppertal
Faculty of Mathematics and Natural Sciences (Faculty C)
Scientific Computing / Software Engineering
Gaustrae 20
42097 Wuppertal
Germany

ТОЧНО СКАЛАРНО ПРОИЗВЕДЕНИЕ ЗА ВИСОКО-ПРОИЗВОДИТЕЛНИ ИЗЧИСЛЕНИЯ

Хасан Ел-Оуни

Представени са алгоритми за сумиране и скаларно произведение на числа с плаваща точка, които са бързи в термините на машинно време. Алгоритмите са широко приложими защото изискват само събиране, изваждани е умножение на числа с плаваща точка във формата на входните данни. Демонстрира се, че получените резултати имат същата точност, както изчисления в двоен или четворен формат.