J Supercomput (2013) 64:580–605
DOI 10.1007/s11227-012-0860-0

FPGA implementation of an exact dot product and its application in variable-precision floating-point arithmetic

Yuanwu Lei · Yong Dou · Yazhuo Dong · Jie Zhou · Fei Xia

Published online: 23 January 2013

Abstract The current paper explores the capability and flexibility of field programmable gate-arrays (FPGAs) to implement variable-precision floating-point (VP) arithmetic. First, the VP exact dot product algorithm, which uses exact fixed-point operations to obtain an exact result, is presented. A VP multiplication and accumulation unit (VPMAC) on FPGA is then proposed. In the proposed design, the parallel multipliers generate the partial products of mantissa multiplication in parallel, which is the most time-consuming part in the VP multiplication and accumulation operation. This method fully utilizes DSP performance on FPGAs to enhance the performance of the VPMAC unit. Several other schemes, such as two-level RAM bank, carry-save accumulation, and partial summation, are used to achieve high frequency and pipeline throughput in the product accumulation stage. The typical algorithms in Basic Linear Algorithm Subprograms (i.e., vector dot product, general matrix vector product, and general matrix multiply product), LU decomposition, and Modified Gram–Schmidt QR decomposition, are used to evaluate the performance of the VPMAC unit. Two schemes, called the VPMAC coprocessor and matrix accelerator, are presented to implement these applications. Finally, prototypes of the VPMAC unit and the matrix accelerator based on the VPMAC unit are created on a Xilinx XC6VLX760 FPGA chip.

Y. Lei (✉) · Y. Dou · Y. Dong · J. Zhou · F. Xia
National Laboratory for Parallel & Distributed Processing, NUDT, Changsha, China
e-mail: yuanwulei@nudt.edu.cn

Y. Dou
e-mail: yongdou@nudt.edu.cn

Y. Dong
e-mail: yazhuodong@nudt.edu.cn

J. Zhou
e-mail: zhoujie@nudt.edu.cn

F. Xia
e-mail: xcyphoenix@nudt.edu.cn

Compared with a parallel software implementation based on OpenMP running on an Intel Xeon Quad-core E5620 CPU, the VPMAC coprocessor, equipped with one VPMAC unit, achieves a maximum acceleration factor of 18X. Moreover, the matrix accelerator, which mainly consists of a linear array of eight processing elements, achieves 12X–65X better performance.

Keywords Carry-save accumulation · Exact dot produce · FPGA · MGS-QR decomposition · LU decomposition · Variable-precision floating-point (VP) arithmetic

1 Introduction

A large number of scientific and engineering applications require efficient variable-precision floating-point (VP) arithmetic [1, 2]. These applications range from mathematical computations to large-scale physical simulations, such as computational geometry, climate modeling, and supernova simulation. Providing accurate results for the numerical sensitive calculations in these applications is extremely important.

However, almost all recent high performance general-purpose processors do not provide hardware units for VP arithmetic operations. Such operations are mostly accomplished using software approaches, such as the GNU Multiple-Precision library (GMP) [3], Multiple Precision Floating-Point Reliable library (MPFR) [4], NTL [5], and so on. The primary disadvantage of software approaches is their speed. Compared with 64-bit floating-point arithmetic, software approaches are at least one order of magnitude slower for quadruple precision arithmetic and 40X slower for octuple precision arithmetic [6]. For higher precision arithmetic, the computational cost increases roughly quadratically with the precision.

Numerous hardware designs have attempted to overcome the speed limit of software solutions. Several VP processors were designed to perform basic VP arithmetic operations [7–12]. Other studies focused on the hardware structures for VP elementary functions, such as VP division, square root, logarithm, and triangle function, according to the properties of these functions [13–15]. Hormigo [16] and Saez [17] proposed a CORDIC processor for VP elementary functions, which are evaluated using simple fixed-point add and shift. In [18], various VP elementary functions were implemented on the unified hardware with a Very Large Instruction Word architecture. However, no report is currently available on hardware design for the VP exact dot product (EDP).

The dot product $\sum_{i=1}^{n} x_i \cdot y_i$, as a basic subroutine in Basic Linear Algorithm Subprograms (BLAS) [19], occurs in almost all scientific and engineering applications. However, in most popular architectures, the dot product has to be emulated using scalar floating-point operations. The catastrophic cancellation [20] of the addition operation in the accumulation of products $(x_i \cdot y_i)$ plays a major role in the loss of accuracy or even in the correctness of the results.

Many studies have presented an exact dot product to avoid the accuracy problem. Kulisch [21] suggested importing the EDP into a general-purpose CPU as "the fifth basic floating-point arithmetic operation." Recently, the IEEE-754-2008 standard provides the functions *sum* and *dot* to obtain the exact accumulation of a vector

and the dot product of two vectors [22]. Moreover, the new IEEE arithmetic standard 1788 will specify the EDP operation as an essential tool for computations with reliable and accurate results [23–25].

The designs of an EDP unit have been studied widely. Lopes et al. presented a fused hybrid floating-point and fixed-point dot product unit on field programmable gate-arrays (FPGAs), where the parameterizable fixed-point number system is used internally [26]. Manoukian used the idea of error-free transformations on FPGA to perform accurate floating-point addition, multiplication and dot product [27]. Dinechin [28] presented a parameterizable FPGA-specific approach for floating-point accumulation and dot products. However, the rounding error in the process of accumulation is not avoided completely in these methods. Kulisch [29], Michael [30], and Knofel [31] proposed single- and double-precision floating-point EDP computation units that add the products of the vector components precisely into a long fixed-point register to obtain a desired exact answer. In [32], a similar method was proposed to design High-Precision floating-point Multiplication and Accumulation units on FPGA for the Double-Double and Quad-Double arithmetic.

Recently, FPGA chips, which operate at the bit level and serve as custom hardware for the different computation precisions, could potentially implement high-precision scientific applications and provide significantly higher performance than a general-purpose CPU [11, 32, 33]. The computational capability of FPGAs is increasing rapidly. A top-level FPGA chip from Xilinx Virtex-6 series contains 474240 Slice LUTs, 25920 Kbits of storage, and 864 DSP48E Slices (25×18 MAC). Numerous DSP48E and storage resources help in building more custom arithmetic units for VP arithmetic.

The primary contributions of the current paper are as follows:

- An EDP algorithm for VP arithmetic, which decomposes the VP operations into exact fixed-point operations to obtain an exact result, is presented.
- A VP multiplication and accumulation unit (VPMAC), based on parallel multipliers, two-level RAM bank, carry-save accumulation, and partial summation schemes, is proposed on FPGA platform.
- Two schemes based on VPMAC units, called the VPMAC coprocessor and matrix accelerator, are proposed to accelerate typical VP algorithms in BLAS, as well as in LU and Modified Gram–Schmidt QR (MGS-QR) decompositions.

2 Exact dot product algorithm for VP arithmetic

2.1 Variable-precision arithmetic

The format of variable-precision numbers is similar to that presented in [34]. As shown in Fig. 1, a VP number x consists of a 14-bit exponent field represented with a bias of 8191 (BE_x, the real value of exponent is $E_x = BE_x - 8191$), a sign bit (S_x), mantissa field (M_x), and a 7-bit precision field (P_x) that refers to the number words in the mantissa field. The mantissa field comprises P_x 64-bit words ($M_x[0]$ to $M_x[P_x - 1]$, and $M_x[0]$ is the highest word). The value of the normalized mantissa

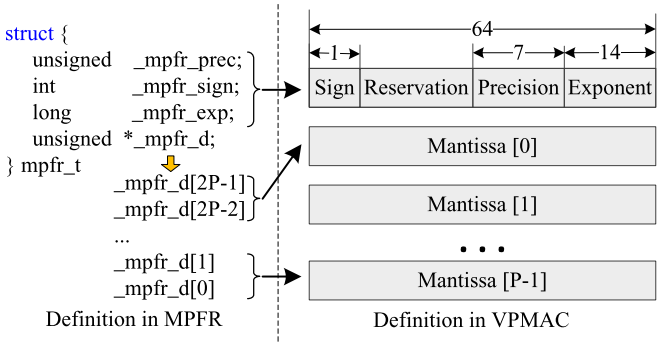


Fig. 1 Definition of the VP format in MPFR and VPMAC

is between 0.5 and 1, i.e., $0.5 \leq M_x < 1$. The precision of x is $64 * P_x$, and the value of x given by

$$x = (-1)^{S_x} \cdot M_x \cdot 2^{BE_x - 8191} = (-1)^{S_x} \cdot M_x \cdot 2^{E_x}$$

Using this definition, the format conversion between the MPFR library and the proposed design can be easily implemented. Thus, the cooperation scheme of hardware and software can be used to implement scientific applications, including numerous VP DOT operations, which will be discussed in Sect. 5.

2.2 Exact dot product

Let $A = (A_i)$ and $B = (B_i)$, $i = 1, \ldots, n$, be two vectors with n components, i.e., A_i and B_i are floating-point numbers. The EDP operation is defined as [29]:

$$c = \diamondsuit \sum_{i=1}^{n} A_i \cdot B_i = \diamondsuit(A_1 \cdot B_1 + \cdots + A_n \cdot B_n) = \diamondsuit s$$

where $\diamondsuit$ is the rounding symbol and all additions and multiplications are operations on real numbers without information loss, and the number of rounding operations is only one.

For VP arithmetic, the multiply operation $(A_i \cdot B_i)$ is emulated with multiple fixed-point operations and the EDP operation is defined as

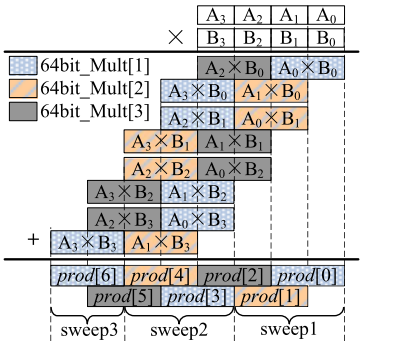
$$c = \diamondsuit \sum_{i=1}^{n} \left\{ (-1)^{S_{A_i} \otimes S_{B_i}} \cdot \sum_{j=0}^{P-1} \sum_{k=0}^{P-1} \left(M_{A_i}[j] * M_{B_i}[k] * 2^{E_{A_i} + E_{B_i} - 64(j+k)} \right) \right\} \quad (1)$$

2.3 Algorithm of the EDP

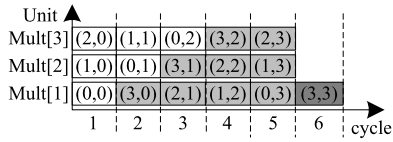
Similar to [29, 32], exact fixed-point addition and multiplication operations are used in the EDP algorithm. Given the maximum support precision is 2048 bits, and the minimum exponent (e_1) is -8190, and maximum exponent (e_2) is 8191. Therefore, all the products $(P_i = A_i \cdot B_i)$ can be stored into a long fixed-point register (called *sum*) of length $L = 2|e_1| + 2r + 2e_2 = 2(8190 + 2048 + 8191) = 36858$ without information loss.

Input: A_i, B_i (i=1,⋯,n)
Output: *Result*
Initialize: *sum*[L:1] = 0;
1: **for** i=1 **to** n **do**
2: Load(A_i, B_i); Set a=A_i, b=B_i;
3: s=S_a^S_b; exp=E_a+E_b;
4: **for** t=0 **to** 2P-2 **do**
5: $prod[t]$=$\sum_{j+k=t}(M_a[j]*M_b[k])$;
6: e_l=2^{exp}+64(2P-t-2); e_h=2^{exp}+64(2P-t)+7;
7: $n\_sum[e_l$-1:1]= $sum[e_l$-1:1];
8: **if**(s==0) **then**
9: {c, $n\_sum[e_h$-1:e_l]}= $sum[e_h$-1:e_l]+$prod[t]$;
10: $n\_sum[L$: e_h]= $sum[L$: e_h]+c;
11: **else**
12: {c, $n\_sum[e_h$-1:e_l]}= $sum[e_h$-1:e_l] -$prod[t]$;
13: $n\_sum[L$: e_h]= $sum[L$: e_h] - c;
14: **end if**(s==0)
15: $sum[L$:1]=$new\_sum[L$:1];
16: **end for** t=0 **to** 2P-2 **do**
17: **end for** i=1 **to** n **do**
18: *Result* = normalize (*sum*[L:1])

(A) EDP Algorithm for VP Arithmetic



(B) Example of VP mantissa multiplication



(C) Spacetime diagram of example

Fig. 2 EDP algorithm for VP arithmetic, given $P = 4$ and $m = 3$ in (**B**) and (**C**). A_i and B_i in (**B**) refer to $M_a[i]$ and $M_b[i]$, respectively. (i, j) in (**C**) stands for the partial product $M_a[i] \times M_b[j]$

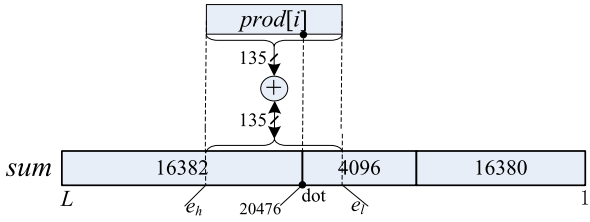


Fig. 3 Alignment of the partial product with *sum*

As shown in Fig. 2(A), the EDP algorithm for VP arithmetic works in a loop after initializing the sum to zero. First, A_i and B_i are loaded successively. Fixed-point multiplication operations are then performed on the mantissas of A_i and B_i, which consists of P 64-bit words. Hence, P^2 partial products are generated and accumulated, which is the most time-consuming part in a high-precision dot product operation. The partial products with the same exponent are first accumulated to derive $prod[t]$ in Line 5.

Steps 7 to 15 are performed to accumulate the partial products ($prod[t]$) into *sum*. Usually, an addition unit with length L is required. Note that the exponent of each partial product can divide the addition into three parts, reducing the addition length to 135 bits, as illustrated in Fig. 3. The *sum* is an L-bit register with a dot in the position $(2r + 2|e_1|)$, where the most significant bit is in the position L. The partial product of $prod[t]$ is located the position from $e_l = 2^{exp} + 64(2P - t - 2)$ to $e_h - 1 = 2^{exp} + 64(2P - t) + 6$, where a 7-bit guard word is added. Thus, as illustrated in line 7 of the algorithm in Fig. 2(A), the first part of *sum* on the right of e_1 is copied directly to *new_sum*. Only the 135-bit length of the addition/subtraction is required in lines 9

and 12, and is subsequently inserted into the second part of the *sum*. The third part only depends on the carry bit generated from the second part.

The final step is the normalization of the L-bit fixed-point number to a VP floating-point number, where the leading zeros are counted to obtain the exponent first, before obtaining the mantissa.

2.4 Analysis of the EDP algorithm

2.4.1 Precision analysis

As the VP EDP algorithm shown in Fig. 2(A), we apply exact fixed-point operations in the process of dot product, instead of usual floating-point operations, to obtain an exact result. First, exact fixed-point multiplications and additions are employed to generate partial products of VP mantissa multiplication and accumulate these with the same exponent, as Line 5. Then the long fixed-point register *sum* is used to accumulate these partial products without information loss and the guard bits are used to prevent overflow of *sum*, as analyzed in Sect. 3. We just introduce one rounding operation in the last normalization step in Line 18, which may lead to one bit error of final result. Therefore, this algorithm meets the definition of EDP operation.

2.4.2 Performance analysis

As shown in Eq. (1), the evaluation of VP EDP for each vector element is composed of the generation and accumulation of P^2 partial products, where P is the precision. This process will consume a large number of cycles for high-precision dot product. However, the evaluation of partial products are independent of each other and can be executed in parallel, so parallel multipliers are designed in VPMAC unit to accelerate the generation of partial products and $\lceil P^2/m \rceil$ cycles are needed, where m is the number of multipliers.

In our EDP algorithm, the two-level add scheme is proposed to accumulate the P^2 partial products into the long register *sum*. In the first level, the partial products are divided into $2P-1$ groups and the partial products within each group have the same exponent. So, a simple short fixed-point adder can be used to accumulate the partial products within each group, as Line 5 in Fig. 2(A). In VPMAC unit, a 135-bit fixed-point accumulator is followed with each multipliers and the latency of this level is hidden in multiplication. In the second level, pipeline techniques, such as segmented addition in Lines 7 to 15 in Fig. 2(A) and several optimization methods in Sect. 3, are explored to accumulate the results of $2P-1$ group into the long register *sum*. Thus, $2P-1$ cycles are needed to accumulate the partial products.

In the implementation of EDP algorithm, the performance bottleneck will be one of the above two stages and is analyzed in Sect. 3.6 in detail.

3 FPGA implementation of VPMAC units

In this section, the FPGA implementation of the VPMAC unit is introduced, which closely couples the multiplication and accumulation, and produces exact result of

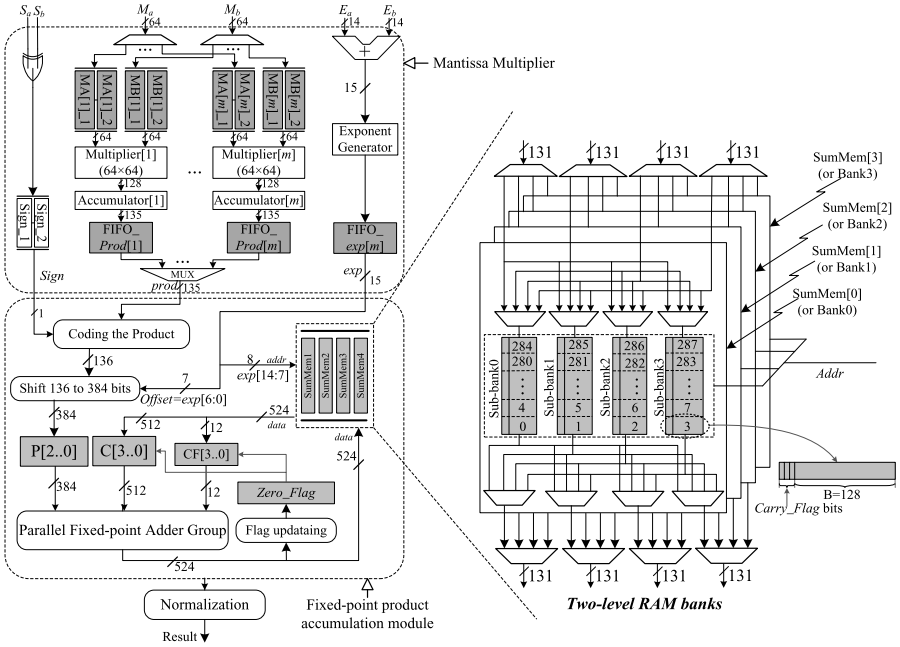


Fig. 4 Structure of VPMAC

dot product by reducing the number of rounding operation to only one. As shown in Fig. 4, the VPMAC unit is mainly composed of the mantissa multiplier, fixed-point accumulation, and normalization modules.

3.1 Parallel multipliers

As shown in Line 5 of Fig. 2(A), P^2 partial products are generated and accumulated, which is the most time-consuming part in a high-precision dot product operation. These partial products can be calculated in parallel, so multiple 64×64 fixed-point multipliers can be used to reduce the latency. Thus, m groups of fixed-point multipliers are equipped in the mantissa multiplier module, as shown in Fig. 4. Each group is composed of a 64×64 fixed-point multiplier, a 135-bit fixed-point accumulator, a FIFO used to store the result of accumulator, and four on-chip memories, which are used to store the mantissa of A_i and B_i. DSP48E on FPGAs is used as the basic building blocks to build the fixed-point multiplier module, which reads the mantissa of M_a and M_b from the corresponding RAMs (MA and MB).

Each group of fixed-point multipliers runs $\lceil (2P+1)/m \rceil$ sweeps. In one sweep, each group is responsible for calculating Line 5 once. As shown in the example in Fig. 2(B), in the first sweep, $M_a[0] \times M_b[0]$ in multiplier [1]; $M_a[1] \times M_b[0]$ and $M_a[0] \times M_b[1]$ in multiplier [2]; and $M_a[2] \times M_b[0]$, $M_a[1] \times M_b[1]$, and $M_a[0] \times M_b[2]$ in multiplier [3] are calculated simultaneously. As shown in the space-time diagram in Fig. 2(C), the P^2 partial products are generated by m 64-bit pipeline multipliers in parallel without stalling, and the load of each multiplier

is balanced. Thus, $\lceil P^2/m \rceil$ cycles, instead of P^2 cycles, can complete the generation of partial products. Moreover, a Ping-Pong mantissa memory structure, with MA[i]_1&MB[i]_1 and MA[i]_2&MB[i]_2, is equipped. Thus, the initialization of mantissa RAMs and the computation of mantissa multiplication can be executed in parallel, thus further improving the performance of the VPMAC unit.

3.2 Two-level RAM bank structure

The second stage of VPMAC is to add the product $A_i \times B_i$ into long register (*sum*). As shown in the right part of Fig. 4, the *sum* is organized as a two-level RAM bank structure, with the first level called *SumMem* (or *bank*) and second called *sub-bank*. In the current paper, the terms *SumMem* and *bank* are used interchangeably. According to the analysis in Sect. 2.3, the length of *SumMem* should not be less than $L = 36858$ bits. The length is set at 36864, including a 6-bit guard digit. Each *SumMem* is organized in a multiple *sub-bank* structure, using distributed RAMs on FPGAs, to fetch multiple words from *SumMem* (at least 135 bits) simultaneously, which correspond to the position of partial product *prod*[i]. Several bits are added to store the carry for each word, called *carry_flag*, which will later be demonstrated in detail.

Another flag is built for each word in *sub-bank*, called *zero_flag*. The *zero_flag* is set to 0 if all bits in the corresponding word and *carry_flag* are 0; otherwise, it is set to 1. A major role of the *zero_flag* is to initialize the two-level RAM banks. At the start of the fixed-point product accumulation, the *zero_flag* register is set to 0 rather than initializing all words in the *sub-bank* sequentially. The cycles required in the latter method are the same as the depth of the *sub-bank*. The other role of the *zero_flag* is to speed up the process of normalization. Generally, the first step in normalizing a fixed-point number into a floating-point number is to count the leading zeros of the fixed-point number, subsequently shifting it left according to the number of leading zeros. With the help of the *zero_flag*, the leading words where the *zero_flag* is 0 can be skipped, and the first significant word containing the most significant nonzero bit can be quickly fixed. Thus, the count of the leading zeros of the L bit summation in *SumMem* is translated to count the leading zeros of the first significant word.

Similar to the optimal choice in [32], in balancing the design of a VPMAC unit, we divide the *SumMem* into four *sub-banks* with 128-bit bandwidth port. The *SumMem*, organized in a four *sub-banks* structure, can provide four consecutive 128-bit words in one cycle. The address is cross decoding, as shown in Fig. 4.

3.3 Carry-save accumulation scheme

In the present work, the carry-save accumulation scheme is employed in the fixed-point product accumulation module. Similar to the conventional carry-save-adder, this scheme computes the sum 128-bit word by 128-bit word. The carry bits will be added in the following accumulation of summand. The result can be calculated in one cycle, because each 128-bit result does not depend on any of the others. Unlike the n-bit carry-save-adder, which consists of n full adders, four 128-bit adders are used to build the fixed-point accumulation.

Fig. 5 Partial product accumulation with carry-save scheme

Input: $Prod_i[t]$, $exp_i[t](i=1,\cdots,n;\ t=0,\cdots,2P\text{-}2)$
Output: $C[287:0]$, $CF[287:0]$, $Zero\_Flag[287:0](ZF[287:0])$
Initialize: $ZF[287:0]= 0$;
1: **for** i=1 **to** n **do**
2: **for** t=0 **to** 2P-2 **do**
3: $\{P[2],P[1],P[0]\}=Prod_i[t]<<exp_i[t][6:0]$; Set $k=exp_i[t][14:7]$.
4: Load $C[k]$~$C[k+3]$ and $CF[k]$~$CF[k+3]$ from *SumMem*;
5: **if** $(ZF[k+j]==0)$ **then** $\{CF[k+j],C[k+j]\}=0;(j=0..3)$
6: Parallel Do
$\{CF[k],C[k]\}=C[k]+P[0]$;
$\{CF[k+1],C[k+1]\}=C[k+1]+P[1]+CF[k]$;
$\{CF[k+2],C[k+2]\}=C[k+2]+P[2]+CF[k+1]$;
$\{CF[k+3],C[k+3]\}=\{CF[k+3],C[k+3]\}+CF[k+2]$;
7: Update ZF, **if** $(\{CF[k+j],C[k+j]\}==0)$ **then** $ZF[k+j]=0;(j=0..3)$
8: Store $C[k]$~$C[k+3]$ and $CF[k]$~$CF[k+3]$ to *SumMem*;
9: **end for** t=0 **to** 2P-2 **do**
10: **end for** i=1 **to** n **do**

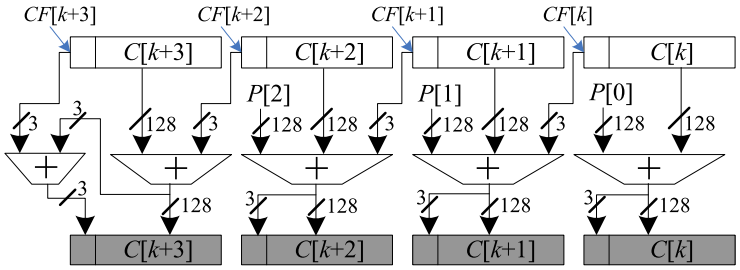


Fig. 6 Parallel adders in the carry-save accumulation scheme

As shown in Fig. 5, the first step in adding the partial product (*prod*) into *SumMem* is to align this product with *SumMem*. We use a two-level alignment scheme to remove the necessity for a very long shifter (L-bit). In the first level, a 384-bit shifter is used to align the product with a 128-bit word in *SumMem*, according to the least seven significant bits of *exp*. In the second level, the highest eight significant bits of *exp* is used to access the four aligned 128-bit words ($C[k] \sim C[k+3]$) and the corresponding carry-flag bits ($CF[k] \sim CF[k+3]$) from *SumMem*. Four 128-bit adders in the group of the fixed-point module add three partial products into *SumMem* in parallel, as shown in Fig. 6 and in step 6 in Fig. 5.

In the conventional carry-save-adder, all the carry bits are passed from each digit to the one on its left in one cycle. However, in the proposed scheme, only three *carry_flags*, aligned with three partial products, will propagate to the left 128-bit word, whereas the other *carry_flags* hold. This condition may result in an overflow in some words that accumulate many carriers in the front iterations if only one-bit *carry_flag* is used as the conventional carry-save adder. The optimal number bits for *carry_flag* is discussed below.

Theorem 1 *The product accumulation is considered based on the proposed carry-save accumulation scheme. The value of carry flag is less than* 3, *and* $W[i] < 2^{129} + 2 * j$ *for* $j < 2^{127}$, *where* $0 \le i \le 287$ *and* $W[i] = \{CF[i], C[i]\}$.

Proof Mathematical induction is used to prove the above statement.

Basic: At the initial stage, set $W[i] = 0$ through *zero_flag*, i.e., the statement holds for $j = 0$.

Inductive step: The statement is assumed to hold for j, given that $C[k] \sim C[k+3]$ and $CF[k] \sim CF[k+3]$ are calculated in the $(j+1)$th iteration. $P[i] \leq 2^{128} - 1$, $C[k+i] \leq 2^{128} - 1$, and $CF[k+i] < 3$ $(i = 0, 1, 2)$. Thus, using the induction hypothesis, the following inequations hold:

$$
\begin{aligned}
W[k] &= P[0] + C[k] < 2^{129} - 1 < 2^{129} + 2 * (j+1) \\
W[k+1] &= P[1] + C[k+1] + CF[k] < 2^{129} + 2 * (j+1) \\
W[k+2] &= P[2] + C[k+2] + CF[k+1] < 2^{129} + 2 * (j+1) \\
W[k+3] &= W[k+3] + CF[k+2] < 2^{129} + 2 * (j+1) \\
W[t] &< 2^{129} + 2 * j < 2^{129} + 2 * (j+1), \quad \text{for } t \neq k \cdots (k+3)
\end{aligned}
$$

Therefore, the statement holds for $(j+1)$. If the carry flag $CF[i]$ is equal to or greater than 3, then $W[k] \geq 3 * 2^{128}$ and $j > 2^{127}$, for some k. Indeed, the theorem statement holds. Thus, the 3-bit *carry_flag* is used for each 128-bit word. The highest bit is the sign, and the other bits are carry bits.

Compared with the fast carry resolution [32], the carry-save accumulation scheme has several advantages. First, four 128-bit adders, instead of the 384-bit adder, work in parallel to reduce the delay in the fixed-point product accumulation module. Second, this scheme can simplify the logic of the carry resolution. In the fast carry resolution, fixing the closest carry terminate factor and carry skip factor according to the 287-bit flag registers are difficult. In the proposed carry-save accumulation scheme, a 3-bit adder and an additional storage resource, which is used to store the *carry_flag*, are employed to deal with the carry resolution. □

3.4 Full pipeline accumulation design

In the fixed-point partial product accumulation module, a 4-cycle pipeline partitioning is proposed to accumulate one partial product with a high running frequency, as shown in Fig. 7(A). The stages are given in detail as follows:

Stage 1, RdC: The partial product is shifted, and the aligned 128-bit words and carry flags are read from *SumMem*, i.e., $C[k] \sim C[k+3]$ and $CF[k] \sim CF[k+3]$, according to the highest eight significant bits of *exp* the corresponding value of *zero_flag*.

Stage 2, CalC: The partial product is added to the summation using four 128-bit adders in parallel.

Stage 3, WrC: The new summation and carry flags are written to *SumMem*. The corresponding bits of the *zero_flag* are calculated according to the new summation and carry flags.

Stage 4, UpF: The *zero_flag* register is updated according to the new corresponding bits of the *zero_flag*.

A scheme of splitting the summation into four partial summations is introduced to build a high-throughput pipelined accumulation design. The four partial summations are stored in four banks, *SumMem*[0] ∼ *SumMem*[3], which are used in different phases of the accumulation in the same cycle. As shown in Fig. 7(B), the product

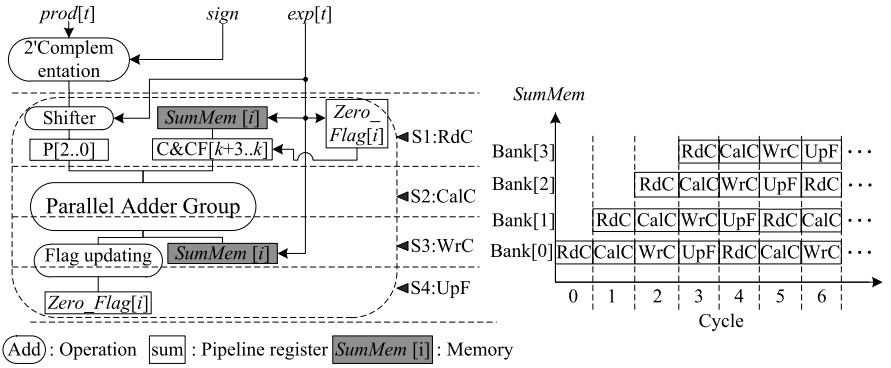


Fig. 7 Pipeline of partial product accumulation and space/time diagram of 4 banks in *SumMem*

Fig. 8 Algorithm of normalization with four partial sums

Algorithm : Normalization with four partial sums

Input: *SumMem*[*i*], *zero_flag*[*i*], *carry_flag*[*i*] (*i*=0..3)
Output: *Result*
Initialize: $c\_f = 0$, $c\_n = 0$, $k = 0$;
1: Calculate the minimum valid address (*j*=*mva*) according to *zero_flag*.
2: **while** (*zero_flag*[*i*]!=0) **do** \\i=0..3
3: $\{c\_n, sum\} = c\_f + \sum_{i=0..3} SumMem[i][j] + \sum_{i=0..3} Carry\_flag[i][j]$;
4: *r_ram*[*k*]=*sum*; *c_f*=*c_n*;
5: *k*++; *j*++;
6: Set *zero_flag*[*i*][*j*]=0 for *i*=0..3;
7: **end while**
8: Normalization *r_ram* to obtain variable-precision *Result*

$A_i \cdot B_i$ entering in the fixed-point product accumulation will be added to the partial sum *SumMem*[i%4] in the ith cycle.

In summary, the first level in the two-level RAM bank scheme comprises four banks to achieve pipeline throughput in the product accumulation. In the second level, a bank comprises four *sub-banks* to provide multiple ports to gain multiple words per cycle.

3.5 Normalization

Unlike [32], the four partial sums in *SumMem* should be added before normalization into the VP format, as shown in Fig. 8. However, it is not necessary to add all 128-bit words in *SumMem*[0–3] together to obtain one L-bit result before normalization. Only the significant words indicated by the *zero_ flag* should be evaluated. First, the addresses of the word containing the least significant nonzero bit for each partial sum are evaluated immediately, and the minimum address is selected among these addresses. The partial sums and corresponding *carry_ flags* in *SumMem*[0–3] are then read and added from the least significant to the most significant word, and the result is stored into memory *r_ram* until all the valid 128-bit words are added. Finally, the

normalization result is obtained by counting the leading zeros in the most significant word of *r_ram*, subsequently shifting the words to gain mantissa of result.

3.6 Performance analysis

In the proposed design, the evaluation of VP EDP consists of four phases, namely, initialization, mantissa multiplication, partial product accumulation, and normalization. In the present implementation, the first three phases may be overlapped. The time for partial product accumulation (T_a) is given by

$$T_a = n \cdot (2P - 1)$$

where n is the input vector size, and P is the precision because P^2 partial products can be divided into $2P - 1$ groups, with each group having the same exponent, as shown in Fig. 2(A) and (B).

The time for mantissa multiplication (T_m) is $T_m = n \cdot (\lceil P^2/m \rceil + 1)$, where m is the number of fixed-point multiplier groups. One additional pipeline stage is used to switch the mantissa ram and reset the registers in each group of 64-bit fixed-point multipliers.

The total execution time (T) can be calculated as the time for the front phases plus the time for normalization (T_n), which is given by

$$T = \begin{cases} n(\lceil \frac{P^2}{m} \rceil + 1) + T_n + T_p, & m < P^2/(2P-2) \\ n(2P-1) + T_n + T_p, & m \geq P^2/(2P-2) \end{cases} \quad (2)$$

where T_p is the overall pipeline latency in front three phases. In the present implementation, T_p is 9, and T_n is approximately $2P + 15$. In the first case of Eq. (2), the time for mantissa multiplication is greater than that for partial product accumulation. The performance is limited by the large number of partial products to be generated for high-precision arithmetic. The number of fixed-point multipliers can be increased to reduce the total execution time. However, the scalability of fixed-point multipliers is limited by the DSP48E block resource, and the performance is no longer improved when $m \geq P^2/(2P - 2)$, as shown in the second case of Eq. (2). To develop fully the capability of FPGAs in accelerating VP scientific applications based on a VPMAC unit, the DSP48E blocks, Block RAMs, and slice resources should be considered, and the optimum number of multipliers must be selected to balance resource utilization and achieve optimum performance.

4 Applications based on VPMAC unit

In this section, LU and MGS-QR decomposition, as well as typical algorithms in BLAS, such as DOT, GEMV, and GEMM, are taken as examples to illustrate the manner by which to use the VPMAC unit in accelerating the high-precision scientific applications on a FPGA platform. Two implementation schemes, the VPMAC coprocessor and matrix accelerator, are proposed.

4.1 The VPMAC coprocessor

The VPMAC coprocessor scheme is a CPU-FPGA hybrid accelerating system. A VPMAC unit is instantiated on FPGA as the coprocessor to implement the evaluation of a VP DOT operation, and the other operations are evaluated by the general-purpose CPU. The execution time for this scheme includes five parts, namely, T_t, T_s, T_r, T_f, and T_c. T_t is the time used in transferring the data format from MPFR to the current definition, as shown in Fig. 1. T_s and T_r are the times for sending the data into the FPGA platform and receiving results from the FPGA platform, respectively. T_f and T_c are the execution times on the FPGA and CPU, respectively. These times may be overlapped to improve performance. In the current study, the followings are given:

- The maximum precision is 1024-bit, i.e., $P \leq 16$.
- The number of multipliers in the VPMAC unit is 8, i.e., $m = 8$.
- The running frequency (F) of the VPMAC unit is 160 MHz.

4.1.1 Typical algorithms in BLAS

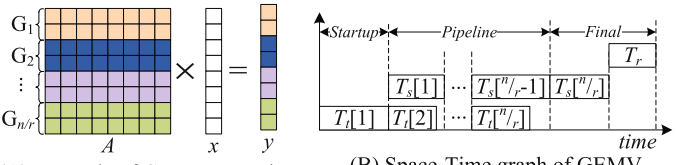
From Eq. (2), we know that the execute time of one EDP operation is $n(2P-1)+2P+24$, and the computation bandwidth is bounded by

$$\big(16n \cdot (P+1) \cdot F\big) \big/ \big(n(2P-1)+2P+24\big) > 800 \text{ (MB/s)}$$

when $3n > 2P+24$. The I/O bandwidth between CPU and FPGA in the current study is smaller than 560 MB/s, as illustrated in Sect. 5. Thus, the execution overhead on FPGA is fully hidden by the communication overhead. For DOT operation in BLAS, the total time is $T_t + T_s + T_r$.

The GEMV and GEMM operations in BLAS are composed of DOT operations and no data dependence exists among such operations. To reduce the total latency, these DOT operations can be divided into multiple groups. The format conversion and data communication between adjacent groups can be executed in parallel, as shown in the space-time graph in Fig. 9(B). Each group includes r DOT operations, thereby $\lceil n/r \rceil$ and $\lceil n^2/r \rceil$ groups in GEMV and GEMM operations, respectively. The total times are as follows:

$$\begin{cases} T_{\text{GEMV}} = T_t + (\lceil n/r \rceil - 1)\max\{T_t, T_s\} + T_s + T_r \\ T_{\text{GEMM}} = T_t + (\lceil n^2/r \rceil - 1)\max\{T_t, T_s\} + T_s + T_r \end{cases} \tag{3}$$



(A) Example of GEMV operation (B) Space-Time graph of GEMV

Fig. 9 Example of GEMV operation

The optimal grouping scheme in the proposed experimental platform will be introduced in the following section. The goal is to select an optimal value of r to minimize the total times for GEMV and GEMM operations.

4.1.2 LU decomposition

LU decomposition, the kernel of the Linpack benchmark that is commonly used to evaluate top-level supercomputers, comprises DOT and division operations. As shown in Fig. 10(A), Doolittle's method [35] computes a lower triangular matrix L and an upper triangular matrix U from a nonsingular matrix A, costing approximately $2n^3/3$ operations.

The evaluation of elements in the row of matrix U or in the column of matrix L can be executed in parallel. However, data dependence exists between DOT operations in different iterations. As shown in Fig. 10(B), in the kth iteration, the evaluation of $U(k, k:N)$ is associated with the elements of matrix U, calculated in the previous iterations ($U(1:k-1, k:N)$), and the elements of the kth row of matrix $L(L(k, 1:k-1))$. Similarly, $L(k+1:N, k)$ are associated with the results in the previous iterations. The evaluation of each column of L is divided into two steps. First, the $(N-k)$ DOT operations are calculated on FPGA. The results of DOT operations are then received from FPGA, and subsequently divided by $U(k, k)$ to yield $L(k+1:N, k)$.

Figure 10(C) shows the space-time diagram of LU decomposition on the VPMAC coprocessor. The evaluation sequence of DOT and division operations is carefully scheduled to reduce the execution time. For the reason that $T_t < T_s$, as analyzed in following section, the overheads of format conversion are almost fully overlapped with communication overheads. The performance is thus limited by the I/O bandwidth of PCI-Express. $L(k, 1:k-1)$ are used in the evaluation of elements of matrix U, so the evaluation of $L(k:N, k-1)$ in the $(k-1)$th are separated into two parts. The first element $(L(k, k-1))$ is calculated first, and the evaluation of other elements is parallel with the data communication. Hence, the evaluation overhead of L

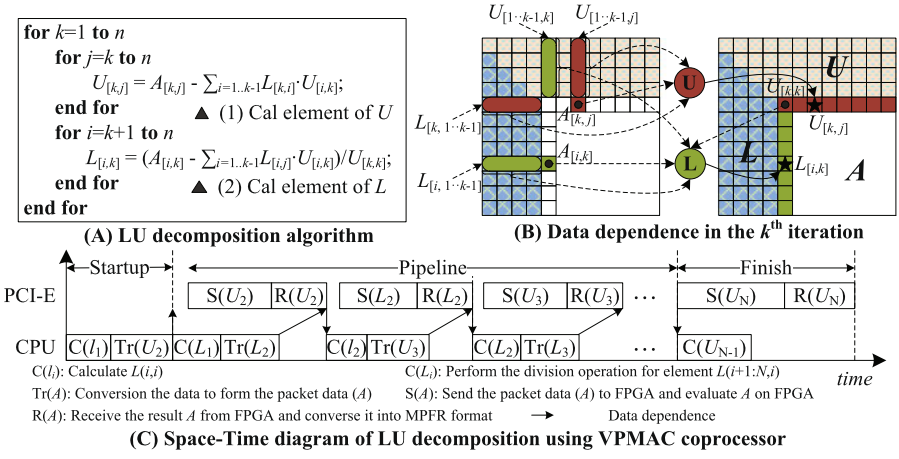


Fig. 10 LU decomposition and the data dependence

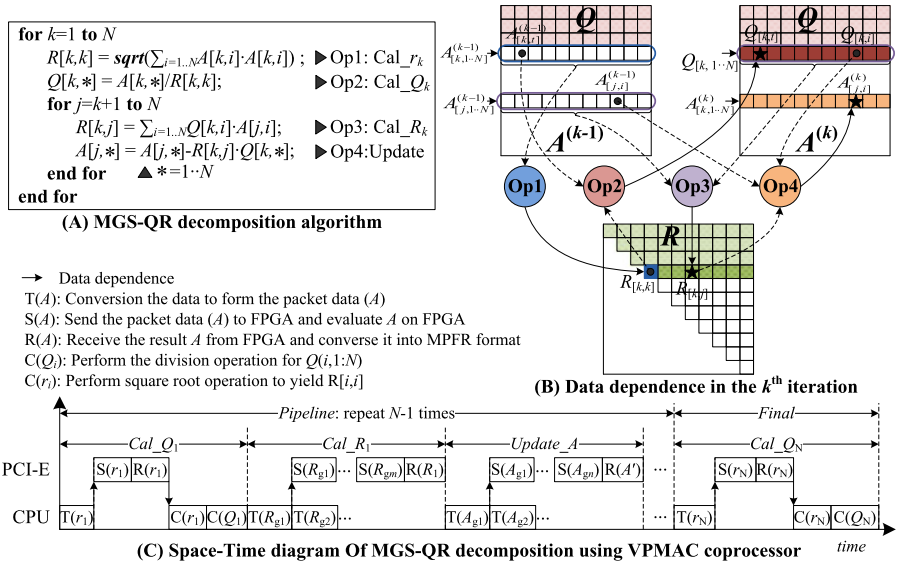


Fig. 11 MGS-QR decomposition and the data dependence

is overlapped, and total time is given by

$$T_{\mathrm{LU}} = T_c(l_1) + T_t(U_2) + T_s(U_N) + T_r(U_N) + \sum_{i=2}^{N-1} \left\{ T_s(U_i) + T_r(U_i) + T_s(L_i) + T_r(L_i) \right\}$$

4.1.3 MGS-QR decomposition

As shown in Fig. 11(A), the MGS-QR decomposition algorithm [36] decomposes the nonsingular matrix A into two matrices, namely, Q and R, costing approximately $2n^3$ operations, where Q is a matrix with orthonormal columns, and R is an upper triangular matrix with positive diagonal elements.

Figure 11(B) shows the data dependence existing in the different iterations of MGS-QR decomposition. Four operation blocks (Op1–Op4) must be executed serially in each iteration. In the kth iteration, after the elements of one row of matrix Q are calculated (i.e., the Op2 is finish), the $N-k$ DOT operations in Op3 can be executed in parallel, and the matrix updating operation in Op4 comprises DOT operations with a length of 2, which can also be executed in parallel. In the present implementation, the optimal grouping scheme in GEMV is used to reduce the execution time for Op3 and Op4. The total time is given by

$$T_{\text{MGS-QR}} = \sum_{i=1}^{N-1} \left\{ T_t(R_{g1}) + m_i \cdot T_s(R_{g1}) + T_r(R_1) + T_t(A_{g1}) + n_i \cdot T_s(A_{g1}) + T_r(A') \right\} + N \cdot \left(T_t(r_1) + T_s(r_1) + T_r(r_1) + T_c(r_1) + T_c(Q_1) \right)$$

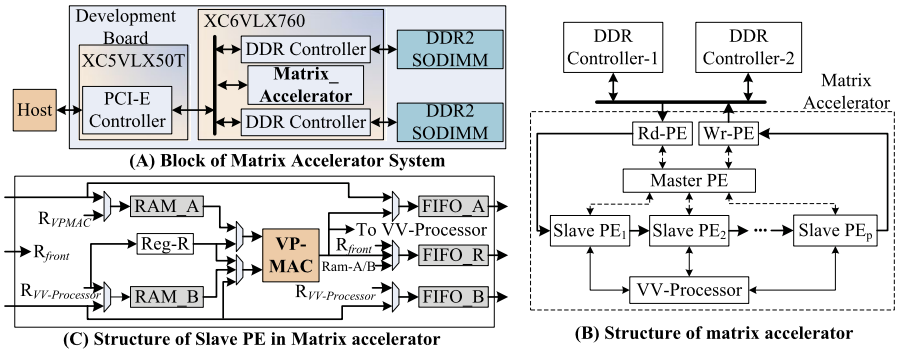


Fig. 12 Structure of matrix accelerator

The VPMAC coprocessor scheme has sufficient flexibility in implementing applications, including numerous DOT operations. However, the performance of this scheme is limited by the communication bandwidth, and the low ratio of data reuse and the overhead in format conversion will reduce the performance further. In the following subsection, the matrix accelerator scheme will fully utilize the computational and storage resources of FPGA chips to achieve high performance.

4.2 Matrix accelerator

Similar to the unified coprocessor architecture for matrix decomposition in [36], in this subsection, a unified hardware framework (called matrix accelerator) for VP matrix operations, such as GEMV, GEMM ($A \times B$), LU decomposition ($A = L \times U$), and MGS-QR decomposition ($A = Q \times R$), is proposed. The matrix operations exhibit the similar two-level loop structure and data dependency. Thus, integrating these operations into a unified structure with a linear computation array is possible.

4.2.1 Structure of matrix accelerator

As shown in Fig. 12(B), the proposed hardware design for matrix operations is primarily composed of a Master processing element (PE), a Read PE (Rd-PE), a Write PE (Wr-PE), a linear array of Slave PEs, and a VV-Processor unit. The Master PE controls the operation of the other PEs. The Rd-PE reads the initial data from two DDR2 memories, and the Wr-PE writes the results into two DDR2 memories. The VV-Processor [18] is used to evaluate the VP elementary function, such as VP division and VP square root. The combination of the VV-Processor unit and the array of Slave PEs can implement multiple matrix operations. The structure of the accelerator and the connections between PEs are similar for different matrix operations because of the similar loop structure and data dependency.

Figure 12(C) illustrates the structure of a Slave PE. Each Slave PE consists of two RAMs, three FIFOs (FIFO_A, FIFO_B, and FIFO_R), a VPMAC module, seven multiplexes, and some control logic. The two RAMs are RAM_A and RAM_B. RAM_A is used to store one row of matrix A for GEMM and MGS-QR decomposition, as well as one row of matrix L for LU decomposition. RAM_B is used to

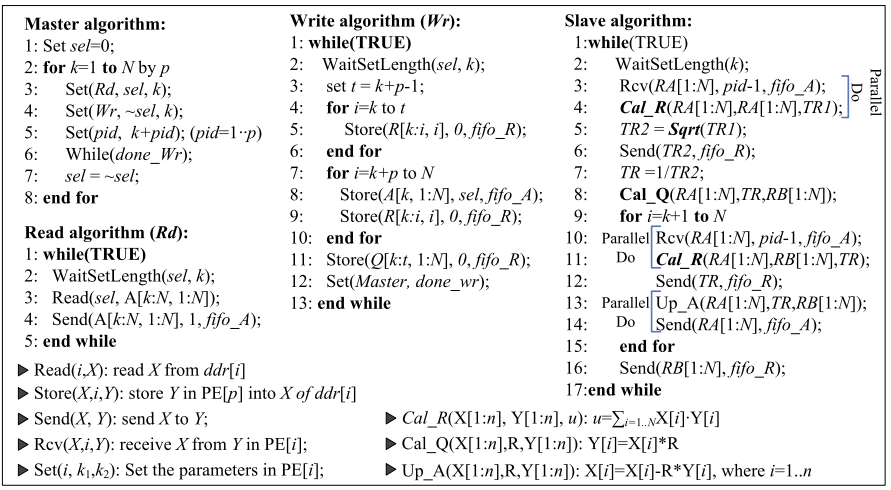


Fig. 13 Pipelined MGS-QR decomposition

store one column of matrix U for LU decomposition, as well as one row of matrix Q for MGS-QR decomposition. Furthermore, RAM_B is not used in GEMM. FIFO_A and FIFO_B are used to buffer the data transferred to the next PE, and FIFO_R is used to buffer the result calculated in the previous and current PEs. The VPMAC module is used to evaluate the DOT operation in the matrix operation. The VP multiplication operation ($x = a \times b$), such as *Cal_Q* in Fig. 13, and the VP FMA (fused multiply-add) operation ($x = a \times b + c$), such as *Up_A* in Fig. 13, can be viewed as special cases of DOT operation. These operations are evaluated in the VPMAC module. The data path of each Slave PE differs in different matrix operations. Thus, seven multiplexes are used to build the corresponding data path for each matrix operation, as shown in Fig. 12(C).

4.2.2 Parallel algorithms of matrix operations

To exploit more parallelism, pipelined parallel algorithms for these matrix operations are proposed, which are described in the SPMD model with message passing primitives. Figure 13 shows that the sequential MGS-QR decomposition algorithm is separated into multiple parallel algorithms, called Master, Read, Write, and Slave algorithms, which run on the Master PE, Rd-PE, Wr-PE, and Slave PEs in Fig. 12(B), respectively. The parallel algorithms for GEMM and LU decomposition using Doolittle's method are the similar to those shown in Fig. 2 of [37] and in Fig. 9 of [32]. Let N and p be the size of matrix A and the total number of Slave PEs executing the Slave algorithm, respectively, assuming that p is less than N.

In the following, MGS-QR decomposition is taken as an example to illustrate the implementation of matrix operations on the proposed matrix accelerator. The Master, Read, Write, and Slave algorithms run $\lceil N/P \rceil$ sweeps. In each sweep, the problem size processed of each PEs is configured by the Master. The Rd-PE reads the initial data from the assigned address of one DDR2 and feeds these data into the first Slave

PE. The Wr-PE writes the updated matrix $A[k+p:N, 1:N]$ from the last Slave PE to the other DDR2, which will be the initial data for the next sweep. Meanwhile, the Wr-PE writes p rows of matrix Q and R, calculated in p Slave PEs, into the assigned address. The *sel* is used to implement a Ping-Pong memory structure for Rd-PE and Wr-PE. After each sweep, the problem size will be reduced by p. In each sweep, each Slave PE is responsible for completing one iteration of k in Fig. 11(A). As shown in Fig. 13, those Slave PEs working as follows:

Step 1: The configuration of the problem size is first processed and then stored in the variable k.
Step 2: The kth row of matrix A, updated by the previous PE, is received and stored in *RAM_A*, whereas *Cal_R* in Line 4 is executed in parallel. The VV-Processor is then used to evaluate the VP square root operation in Line 5 and the VP division operation in Line 7. The value of $R[k,k]$ and $\frac{1}{R[k,k]}$ are then obtained.
Step 3: The VP multiplication in Line 8 is executed on the VPMAC module N times to obtain the kth row of matrix Q. The results are stored in *RAM_B*.

Repeat the Step 4 *and Step* 5 $(N-k)$ *times*

Step 4: Similar to Step 2, Line 10 and Line 11 are executed to obtain the value of $R[i,k]$, which is then seeded to *FIFO_R*.
Step 5: The update operation (i.e., the VP FMA) in Line 13 is executed, and the row of the updated matrix A is sent to the next PE.

From the above analysis, the receive primitive in Line 3 and the DOT operation in Line 4 (or Lines 10 and 11) can be executed in parallel. The send primitive in Line 14 and the update operation in Line 13 can also be executed in parallel. Thus, the overhead of data transferring between PEs can be hidden by the calculation. The ratio of data reuse increases with the data flow though the array of Slave PEs. Moreover, multiple VPMAC units in the array of Slave PEs are executed in parallel to improve the performance of the matrix accelerator further.

However, the performance of the matrix accelerator for MGS-QR decomposition is less than that for GEMM and LU decomposition because most DOT operations in MGS-QR decomposition are VP multiplication and VP FMA, and the lengths of such DOT operations are one and two, respectively. Thus, fully developing the execution efficiency of VPMAC units for MGS-QR decomposition is difficult.

5 Experiments

5.1 Experimental setup

The experiments are performed on a self-designed development board containing one FPGA chip and two 4 GB DDR2 DRAM modules. As shown in Fig. 12(A), the proposed hardware system mainly consists of two FPGAs (Xilinx Virtex-5 XC5VLX50T-1FF1136 and Virtex-6 XC6VLX760-2FF1760) and two 4 GB DDR2 DRAM modules. Each DDR Controller runs at 200 MHz with 128-bit data width. The peak I/O bandwidth can reach 6.4 GB/s. The FPGA development board is linked

Table 1 Synthesis results

Resource	VPMAC			VV-Processor [18]	Matrix-Accelerator
	$m = 2$	$m = 4$	$m = 8$		
Slice Reg	9485 (1 %)	13359 (1 %)	17818 (1 %)	8258 (1 %)	121715 (12 %)
Slice LUT	19855 (4 %)	31641 (7 %)	38880 (8 %)	16235 (3 %)	246664 (52 %)
DSP48E	24 (3 %)	48 (6 %)	96 (11 %)	96 (11 %)	864 (100 %)
BRAM (18 K)	–	–	–	43 (3 %)	1378 (96 %)
Freq (MHz)	214.1	213.5	205.1	253.5	156.6

to the host PC via a PCI-Express 8 line interface. The proposed hardware design is implemented using Verilog HDL, ModelSim 6.5, and ISE 12.3 tools. As a base for performance comparison, the MPFR library, MPFR 3.0.0 [38], is applied to measure the result accuracy and delay time. The software environment includes a host PC with Intel Xeon E5620 CPU at 2.40 GHz and 32 GB DDR3 1333 MHz Memory, Microsoft Windows 7, and Microsoft Visual Studio 2010. The running time of the FPGA accelerator includes computation time and the time for sending the initial data to the device, as well as the time for receiving the results back to the host PC.

5.2 Resource utilization

Table 1 shows the details of the FPGA synthesis data for different modules including the VPMAC module with different numbers of multipliers (m), VP-Processor module, and matrix accelerator with eight Slave PEs. In the present implementation, the resource usage is carefully planned to avoid unbalanced resource utilization.

5.2.1 DSP resource

DSP48E blocks, used to build 64-bit fixed-point multiplication units in parallel multiplier modules, become the most constrained resource. One VPMAC unit consumes approximately 11 % of the DSP48E available in the Virtex-6 XC6VLX760 FPGA chip. Therefore, 9 VPMAC units can be integrated into the chip. In the matrix accelerator in the current study, 8 VPMAC units are equipped, and the other DSP48E resources are used in the VV-Processor.

5.2.2 Local memory resource

The local memory modules (on-chip memory), classified into distributed RAM and embedded 18 Kbit block RAM, play an important role in the proposed design. The distributed RAM is used to store fixed-point summation in the VPMAC unit to prevent long routing delays. The distributed RAM elements are mapped into LUT Slices, which are of great advantage to the FPGA placement and layout phase compared with the fix-positioned embedded block RAMs. However, distributed RAMs consume more LUT resources. A VPMAC unit requires approximately 8 % of the available Slice LUT resources. The 18 Kbit block RAMs are used in the Slave PE of the

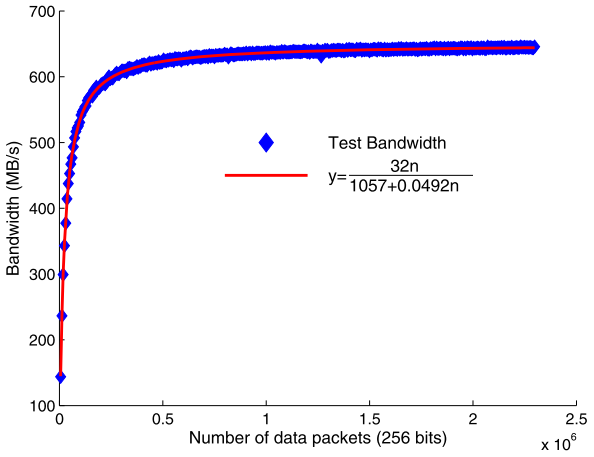


Fig. 14 Bandwidth of PCI-Express interface

matrix accelerator to implement the data reuse and data buffer between different PEs. The current study uses 116 and 24 18 Kbit block RAMs to implement two RAMs and three FIFOs in each Slave PE.

5.2.3 Frequency

The achievable maximum frequency of a VPMAC unit exceeds 200 MHz in XC6VLX760. And the synthesis frequency of the matrix accelerator with 8 Slave PEs is 156 MHz. Thus, this accelerator can correctly run at 133 MHz on the self-designed development board.

5.2.4 FPGA vs. ASIC

In future, the ASIC implementation of VPMAC unit will be taken into consideration. The modified booth encoding method will be used to implement the 64×64 multiplier and custom RAM structure will be explored to implement the two-level RAM banks and other RAMs efficiently. Moreover, the pipeline structure of VPMAC unit will be adjusted appropriately for the higher frequency of ASIC implementation. Since FPGA implementation is emulated using a large number of configurable elementary blocks and a network of wires, many silicon area is used for wire routing and switching transistor to obtain flexibility. Thus, the resource utilization of FPGA silicon is lower than ASIC implementation. However, the performance of FPGAs is improved through custom hardware for applications equipped with multiple VPMAC units working in parallel.

5.3 PCI-Express bandwidth

The bandwidth of the PCI-Express interface is shown in Fig. 14. The time for one communication between the CPU and FPGA involves two parts, namely, the start-up time (A) and data packet transferring time with the DMA scheme. The first part,

used to build the link between the CPU and FPGA and to start up the communication, is always the same for each communication. The second part is proportional to the number of data packets (n_p), as well as to the size of the data. Each data packet comprises 256 bits of data. Therefore, the bandwidth is approximately given by

$$(256 \cdot n_p)/\left(8 \cdot 10^6 \cdot (A + B \cdot n_p)\right) \text{ (MB/s)}$$

Using the CFTOOL, a curve fitting toolbox in Matlab, the coefficients above equation and the curves for the bandwidth of PCI-Express are obtained, as shown in Fig. 14. Hence, the peak bandwidth of PCI-E is 650 MB/s, with the communication time given by $T_s = 1.057 * 10^{-3} + 4.92 * 10^{-8} * n_p$.

5.4 Performance of the VPMAC coprocessor scheme

5.4.1 DOT operation

The format conversion time T_t is also proportional to the number of data packets (n_p). In the adopted Intel Xeon CPU platform, these values satisfy $T_t \approx 3.25 * 10^{-8} * n_p$ and $T_s > T_t$. For DOT operation, the total time is approximately given by

$$T_{\text{DOT}} = 2.114 * 10^{-3} + 4.085 * (P + 1) * N * 10^{-8}$$

where N and P are the length and precision of vectors, respectively. As shown in Fig. 15(C), the speedup factor of the VPMAC coprocessor scheme increases with the precision of the vector, reaching 18X for 1024-bit precision. This condition is attributable to the fact that the computation overhead increases faster than the conversion overhead and the transferring overhead for the VPMAC coprocessor scheme. The ratio of T_s to T_t is between 1.7 and 2.7, as shown in Fig. 15(D). Thus, the parallel execution scheme of data conversion and data transferring is used in matrix operations (GEMV, GEMM, LU, and MGS-QR) to hide the conversion overhead.

5.4.2 GEMV and GEMM operations

According to Eq. (3), the total time of the GEMV and GEMM are given by

$$T_{\text{GEMV}} \geq 8.28 \cdot 10^{-6} \cdot \sqrt{P'} \cdot n + 2.46 \cdot 10^{-8} \cdot P' \cdot N^2 + T_r$$
$$T_{\text{GEMM}} \geq 8.28 \cdot 10^{-6} \cdot \sqrt{n \cdot P'} \cdot n + 2.46 \cdot 10^{-8} \cdot P' \cdot n^3 + T_r$$

where $P' = P + 1$. The optimal grouping is $r = 255/\sqrt{P+1}$ for GEMV and $r = 255\sqrt{n}/\sqrt{P+1}$ for GEMM. For GEMV operation, the optimal grouping is only dependent on the computation precision. Using this optimal grouping scheme, the performance of the VPMAC coprocessor is compared with the Intel Xeon CPU using four threads, achieving a speedup factor of 1.4X–8.6X, as shown in Fig. 16. For GEMM operation, the optimal grouping scheme is relies on the computation precision and the matrix size and a speedup factor of 3.4X–9.0X can be achieved, as shown in Table 2.

For both GEMV and GEMM operations, similar performances can be obtained for the same precision with different matrix sizes. However, the performance improves with the computation precision because the computational cost increases roughly quadratically with the precision, and the conversion and transferring costs increase linearly with the precision.

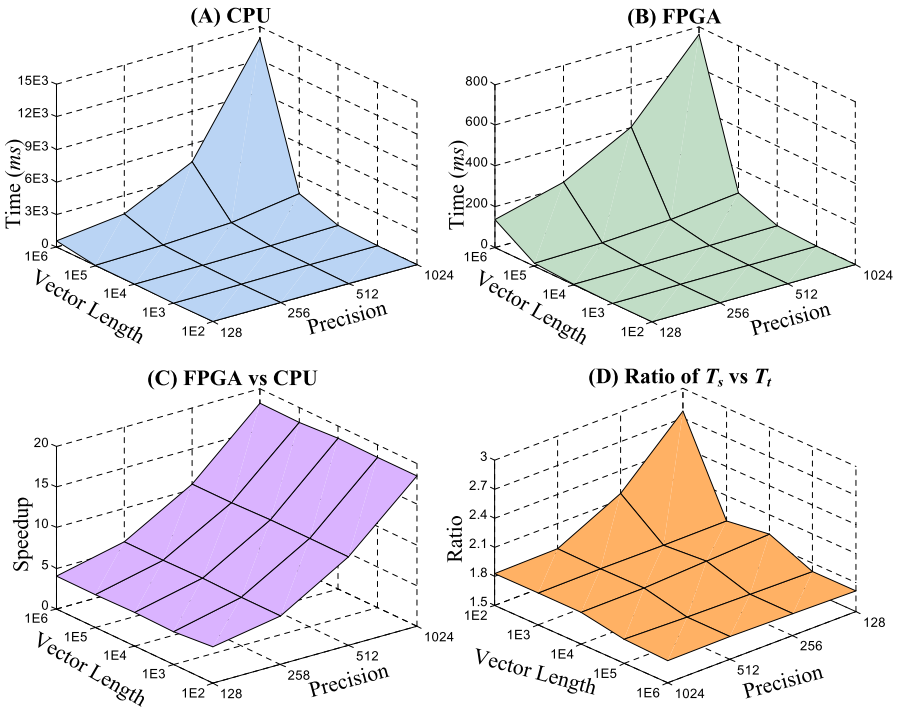


Fig. 15 Performance comparison of DOT operation

5.4.3 LU and MGS-QR decomposition

According to the analysis in Sect. 4(A), the average speedup factor of the VPMAC coprocessor scheme is obtained as 5.4X and 2.5X for LU and MGS-QR decomposition, respectively, as shown in Table 2. Similar to the DOT, GEMV, and GEMM operations, the performance of this scheme increases with computation precision. Moreover, for the same precision, the performance improves with the matrix size because the average length of DOT operations in LU and MGS-QR decomposition, as well as the transfer efficiency of PCI-Express increases with the matrix size.

5.5 Performance of matrix accelerator

Table 2 also summarizes the performance of matrix accelerator scheme for GEMM, LU, and MGS-QR decomposition. Compared with the parallel software implementation based on OpenMP running on a quad-core Intel Xeon E5620 CPU, an average speedup factor of 51.6X, 42.5X, and 21.5X can be achieved for GEMM, LU, and MGS-QR decomposition, respectively, because the data reuse and parallel techniques among the array of Slave PEs and the performance of matrix accelerator, which is equipped with 8 VPMAC units and 1 VV-Processor unit, are significantly higher than that of the VPMAC coprocessor scheme.

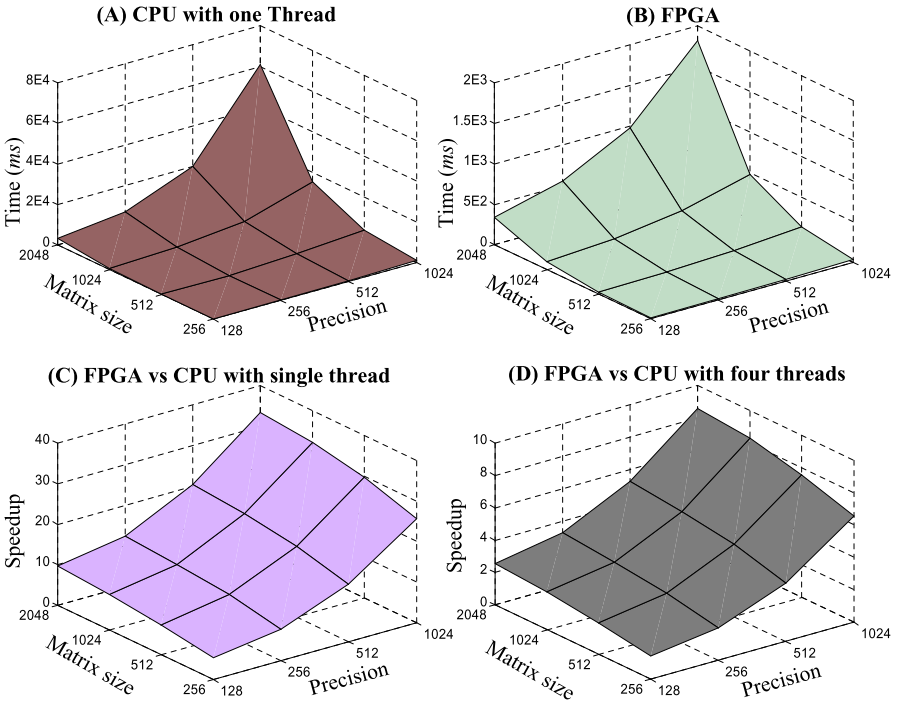


Fig. 16 Performance comparison of GEMV operation

The large number of short-length DOT operations in MGS-QR decomposition, as discussed in Sect. 4, makes the full development of the efficiency of multiple VP-MAC units in the array of Slave PEs difficult. Thus, the performance for MGS-QR decomposition (12X–30X) is only half of that for GEMM operation (40X–63X) and LU decomposition (16X–65X).

6 Conclusion and future work

In this paper, a VP EDP algorithm and VPMAC unit on FPGA were presented. Exact result was obtained by exact fixed-point operations, and high frequency and throughput were achieved by parallel multipliers, two-level RAM bank, carry-save accumulation, and partial summation schemes. Moreover, the VPMAC coprocessor and matrix accelerator based on VPMAC unit were proposed on XC6VLX760 FPGAs to implement DOT, GEMV, GEMM, LU, and MGS-QR decomposition. Compared with the parallel software version based on OpenMP running on Intel Xeon Quad-core E5620 CPU, the proposed matrix accelerator achieved significantly performance improvement.

Due to the poor efficiency of software approaches for variable-precision arithmetic on general-purpose CPU, FPGA chips can custom high-performance functional unit

Table 2 Performance comparison of GEMM, LU decomposition, and MGS-QR decomposition

Func	Platform	Precision	256 bits				512 bits				1024 bits			
		Size	256	512	1024	2048	256	512	1024	2048	256	512	1024	2048
GEMM	CPU[a]	Single(s)[c]	27.7	221.8	1783.7	14199	80.9	646.6	5177.8	41134	247.7	1968.9	15827	126674
		Four(s)[c]	7.3	58.2	467.6	3760.8	20.6	166.3	1329.7	10685	62.8	503.6	4032.6	32327
	FPGA[b]	Scheme1(s)	2.1	16.7	132.7	1058.5	3.8	30.0	238.7	1904.6	7.2	56.6	450.4	3596.4
		Speedup[d]	3.42	3.48	3.52	3.55	5.39	5.54	5.57	5.61	8.75	8.90	8.95	8.99
		Scheme2(s)	0.18	1.24	9.18	70.6	0.40	2.89	21.8	169.7	1.38	10.4	81.0	639.1
		Speedup[d]	40.8	46.9	50.9	53.2	51.2	57.6	60.8	62.9	45.5	48.3	49.8	50.5
LU	CPU[a]	Single(s)	9.35	74.0	592.1	4787.8	26.8	214.1	1711.5	13728	81.8	657.5	5265.2	42020
		Four(s)	1.87	12.9	167.4	1325.5	8.1	37.2	503.7	4048.8	23.7	180.5	1543.3	12155
	FPGA[b]	Scheme1(s)	1.77	7.69	48.5	361.4	2.33	12.1	83.8	643.5	3.44	20.9	154.4	1207.8
		Speedup[d]	1.05	1.67	3.45	3.66	3.49	3.07	6.01	6.29	6.87	8.60	9.99	10.1
		Scheme2(s)	0.11	0.63	3.91	26.9	0.25	1.28	8.53	61.5	0.58	3.96	28.9	220.8
		Speedup[d]	16.3	20.5	42.8	49.2	32.0	29.1	59.0	65.7	40.6	45.5	53.2	55.0
MGS-QR	CPU[a]	Single(s)	27.2	216.5	1773.5	14112	79.5	634.0	5154.3	41563	243.9	1972.6	15753	126229
		Four(s)	6.76	52.6	430.6	3413.8	20.6	37.2	1309.3	10425	62.8	502.5	3982.1	31719
	FPGA[b]	Scheme1(s)	8.57	44.2	295.6	2211.8	11.8	70.5	506.6	3900.1	18.4	123.3	928.6	7276.5
		Speedup[d]	0.78	1.19	1.45	1.54	1.73	2.31	2.58	2.67	3.41	4.08	4.29	4.36
		Scheme2(s)	0.54	3.99	30.6	240.2	0.92	6.85	52.6	412.5	2.47	18.5	143.9	1133.1
		Speedup[d]	12.5	13.2	14.0	14.2	22.3	23.9	24.8	25.3	25.4	27.0	27.6	27.9

[a] Intel Xeon Quad E5620 2.40 GHz (32 nm), 12 MB Intel Smart Cache, 32.0 GB DDR3 memory

[b] Xilinx Virtex-6 XC6VLX760-2FF1760 (40 nm), 8 VPMAC units, running at 133 MHz

[c] *Single* and *Four* refer to the single thread implementation and the parallel implementation based-on OpenMP

[d] The speedup factor for both schemes compared with the parallel software implementation

for variable-precision arithmetic. So, FPGAs exhibit the potential capability to implement variable-precision arithmetic and provide significant higher performance than a general-purpose CPU.

In the future, the implementation of VPMAC on ASIC chips will be investigated, and the performance of the architecture that closely couples this ASIC chip with the CPU will be analyzed. Furthermore, the VPMAC unit will be applied to accelerate large-scale scientific applications, such as the Krylov subspace method, to exhibit the potential capability of FPGA chips.

Acknowledgements This work is partially supported by NSFC (61125201, 61202127, and 60903057).

References

1. Yun H, Chris D (2001) Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. J Supercomput 18(3):259–277
2. Bailey DH (2005) High-precision floating-point arithmetic in scientific computation. Comput Sci Eng 7(3):54–61
3. GNU Multiple-Precision Arithmetic Library (2011) Available from: http://www.swox.com/gmp
4. Fousse L, Hanrot G, Lefevre V, Pelissier P, Zimmermann P (2007) MPFR: a multiple-precision binary floating-point library with correct rounding. Trans Math Softw 33(2):1–15
5. NTL: A Library for Doing Number Theory (2011) Available from: http://www.shoup.net/ntl/
6. Fujimoto J, Ishikawa T, Perret-Gallix D (2005) High precision numerical computations—a case for an happy design. ACPP IRG note, ACPP-N-1: KEK-CP-164
7. Cohen MS, Hull TE, Hamarcher VC (1983) A controlled-precision decimal arithmetic unit. IEEE Trans Comput C-32:370–377
8. Chiarulli DM, Ruaa WG, Buell DA (1985) DRAFT: a dynamically reconfigurable processor for integer arithmetic. In: Proceedings of the 7th symposium on computer arithmetic, pp 309–318
9. Carter TM (1989) Cascade: hardware for high/variable precision arithmetic. In: Proceedings of the 9th symposium on computer arithmetic, pp 184–191
10. Schulte MJ, Swartzlander EE Jr (2000) A family of variable-precision, interval arithmetic processors. IEEE Trans Comput 49(5):387–397
11. El-Araby E, Gonzalez I, El-Ghazawi T (2007) Bringing high-performance reconfigurable computing to exact computations. In: Proceedings of FPL 2007, pp 79–85
12. Alexandre FT, Milos DE (1998) A variable long-precision arithmetic unit design for reconfigurable coprocessor architectures. In: Proceedings of FCCM 1998
13. Hormigo J, Villalba J (2000) A hardware algorithm for variable-precision division. In: Proceedings of the 4th conference on real numbers and computers, pp 1–7
14. Hormigo J, Villalba J, Schulte M (2000) A hardware algorithm for variable-precision logarithm. In: Proceedings of ASAP2000, pp 215–224
15. Hormigo J, Villalba J, Zapata EL (1999) Interval sine and cosine functions computation based on variable-precision cordic algorithm. In: Proceedings of Arith 1999, pp 186–193
16. Hormigo J, Villalba J, Zapata EL (2004) CORDIC processor for variable-precision interval arithmetic. J VLSI Signal Process 37:21–39
17. Saez E, Villalba J, Hormigo J, Quiles FJ, Benavides JI, Zapata EL (1998) FPGA implementation of a variable precision CORDIC processor. In: Proceedings of 13th conf on design of circuits and integrated systems (DCIS'98), pp 604–609
18. Lei Y, Dou Y, Zhou J (2011) FPGA-specific custom VLIW architecture for arbitrary precision floating-point arithmetic. IEICE Trans Inf Syst E94-D(11):2173–2183
19. Li XS, Demmel JW, Bailey DH, Henry G (2002) Design, implementation and testing of extended and mixed precision blas. ACM Trans Math Softw 18(2):152–205
20. Rump SM (1988) Algorithms for verified inclusions-theory and practice. In: Moore RE (ed) Reliability in computing. Academic Press, San Diego, pp C109–C126
21. Kulisch U (1997) The fifth floating-point operation for top-performance computers. Universitat Karlsruhe

22. IEEE (2008) Standard for binary floating point arithmetic ansi/ieee standard 754-2008. The Institute of Electrical and Electronic Engineers, Inc. Revised version of original 754-1985 Standard
23. Edmonson W, Melquiond G (2009) IEEE interval standard working group—p1788: current status. In: Proceedings of Arith 2009, pp 183–190
24. Kulisch U, Snyder V (2011) The exact dot product as basic tool for long interval arithmetic. Computing 91(3):307–313
25. Kulisch U (2011) Very fast and exact accumulation of products. Computing 91(4):397–405
26. Lopes AR, Constantinides GA (2010) A fused hybrid floating-point and fixed-point dot-product for FPGAs. In: Proceedings of ARC 2010, vol 5992, pp 157–168
27. Manoukian MV, Constantinides GA (2011) Accurate floating point arithmetic through hardware error-free transformations. In: Proceedings of ARC 2011, vol 6578, pp 94–101
28. Dinechin FD, Pasca B, Cret O, Tudoran R (2008) An fpga-specific approach to floating-point accumulation and sum-of-products. In: Proceedings of FPT 2008, pp 33–40
29. Kulisch U (2008) Computer arithmetic and validity: theory, implementation, and applications. de Gruyter, Berlin
30. Muller M, Rub C, Rulling W (1991) Exact accumulation of floating-point numbers. In: Proceedings of Arith 1991, pp 64–69
31. Knofel A (1991) A fast hardware units for the computation of accurate dot products. In: Proceedings of Arith 1991, pp 70–74
32. Dou Y, Lei Y, Wu G (2010) FPGA accelerating double/quad-double high precision floating-point application for exascale computing. In: Proceedings of ICS 2010, pp 325–336
33. Underwood K (2004) FPGAs vs. CPUs: trends in peak floating-point performance. In: Proceedings of FPGA 2004, pp 171–180
34. Schulte MJ, Swartzlander EE Jr (1995) Hardware design and arithmetic algorithms for a variable-precision, interval arithmetic coprocessor. In: Proceedings of the 12th symposium on computer arithmetic, pp 222–228
35. Higham NJ (2002) Accuracy and stability of numerical algorithms, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia
36. Dou Y, Zhou J, Wu G, Jiang J, Lei Y (2010) A unified co-processor architecture for matrix decomposition. J Comput Sci Technol 25(4):874–885
37. Dou Y, Vassiliadis S, Kuzmanov GK, Gaydadjiev GN (2005) 64-bit floating-point FPGA matrix multiplication. In: Proceedings of FPGA 2005, pp 86–95
38. Fousse L, Hanrot G, Lefevre V, Pelissier P, Zimmermann P (2007) MPFR: a multiple-precision binary floating-point library with correct rounding. Trans Math Softw 33(2):1–15