A Scalable Architecture for Accelerating Multi-Operation and Continuous Floating-Point Matrix Computing on FPGAs

LONGLONG ZHANG, YUANXI PENG, AHUI HUANG, AND XIAO HU
College of Computer, National University of Defense Technology, Changsha 410073, China
Corresponding author: Xiao Hu (xiaohu@nudt.edu.cn)

This work was supported by the National Science and Technology Major Project under Grant 2017-V-0014-0066.

ABSTRACT Matrix computing is a basic operational model that was broadly used in science and engineering applications. In this study, we first propose a novel optimization method to obtain a high-performance and scalable architecture for matrix multiplication, including reducing data transmission, optimizing data flow, improving resource utilization, and dynamically changing the length of the linear array. Based on the optimized architecture, we present a multi-operation floating-point matrix computing unit (design-I), which extends the function of matrix computing from single matrix multiplication operation to matrix addition, matrix subtraction, matrix-vector multiplication, matrix-scalar multiplication. With low storage demand and computing efficiency, design-I can be used in computing dense matrices of arbitrary sizes. Moreover, we propose a continuous floating-point matrix computing unit (design-II), which not only has the same function of multi-operation but also meets the requirement of continuous matrix computing in practical engineering and avoids a large amount of intermediate data transfer. Finally, the authors adopt the above-mentioned unit cores to build a matrix computing acceleration system according to different engineering requirements. The experiments implemented in the Xilinx 585T FPGA device show that the accelerator achieves a maximum frequency of 195Mhz with 256 processing elements (PEs) and performs 99.8GFLOPS. The architecture is more outstanding in application scope and prospects compared with state-of-the-art methods.

INDEX TERMS Matrix computing, accelerator, floating-point, FPGAs, scalability

I. INTRODUCTION
Floating-point matrix computing is applied widely in the fields of system control, digital signal processing [1], image processing [2], and deep learning [3]. Its computing efficiency directly affects the performance of the entire system. In recent years, there are various platforms for accelerating matrix calculation, such as CPU, GPGPU, FPGA, and software libraries.

Field-programmable gate arrays (FPGAs) are suitable for accelerating matrix calculation as a co-processing platform. Many studies showed that FPGAs are superior to general-purpose processors and GPGPU platforms in sustained performance [4]–[6]. Its flexible programmability and rich logic resources, especially the large number of embedded DSPs and BRAMs (Block RAMs), lay the foundation for the performance improvement of matrix computing acceleration.

Various algorithms and corresponding hardware structures mainly focus on a trade-off between resource requirements and performance. For example, algorithms for fixed-point matrix multiplication were proposed [7], [8]. Designers proposed a complete coprocessor of FPGA architecture for matrix multiplication [9]. Similar systolic array structures were presented due to high data throughput and computing speed [10]–[13]. Besides, some scholars have considered the structure of matrix-vector multiplication [14]–[16].

Although many kinds of research have been done, matrix computing is still faced with the following challenges. Firstly, previous architectures were designed with a limited capacity which can only process fixed-point data, small or medium-sized matrices. We expect to design a matrix computing acceleration (MCA) unit, which can handle large matrices computing and meet the demand of high data precision.
Secondly, if we use a fixed structure, it is difficult for us to deal with the matrix calculation of arbitrary sizes. Therefore, considering the flexibility of the structure is necessary and meaningful. Moreover, most of the previous work can only deal with a single matrix operation. But in fact, a certain matrix operation is often not sufficient in many engineering applications now. It requires a variety of matrices operation and even needs to support a sequence of matrix operations based on the same structure.

This study addresses the above challenges by building floating-point matrix computing acceleration unit cores with a unified architecture. The major contributions of this work are as follows:

- We deeply optimize the performance of the parallel block matrix multiplication algorithm and linear array structure from multiple forms. The optimized architecture combines the schemes of directly initializing matrix values to reduce data transfer, adopting two-stage FIFO to optimize data flow, and considering the balance between DSPs and logic resources to improve resource utilization. By combining the idea of block matrix and variable length of the linear array, the architecture is scalable.
- The proposed multi-operation floating-point matrix computing unit performs matrices of arbitrary sizes with low storage demand and computing efficiency. This unified architecture can compute matrix multiplication, matrix addition, matrix subtraction, matrix-vector multiplication, and matrix-scalar multiplication.
- By increasing the shared-memory mechanism appropriately, the proposed continuous floating-point matrix computing unit significantly reduces delay and data access overhead for a sequence of matrix operations.
- We use matrix computing unit cores mentioned above to build a matrix computing acceleration system and compare it to some state-of-the-art accelerators. The results show that the proposed system meets different actual needs in engineering and achieves better performance than other accelerators.

The following sections are organized as follows. In Section II, background and related work are introduced. The improved matrix multiplication algorithm and structure are discussed in Section III. Matrix computing processes for other operations based on the same architecture are described in Section IV. The details of two designs and accelerating system are discussed in Section V. The implementation results and comparisons with previous work are shown in Section VI. Section VII concludes the work.

II. BACKGROUND

A. RELATED WORK

Here we discuss some of the related work from matrix multiplication and multi-operation matrix computing, respectively.

To meet the requirement of matrix multiplication for large sizes and floating-point precision, similar linear array structures have been adopted [17]–[19]. A more significant design was a linear array based on a parallel block matrix multiplication algorithm by Dou et al. [20], which processed matrix multiplication of any sizes with a certain amount of on-chip storage.

This design method is worthy of our study, but for now, it is too complicated. Based on the architecture, others have increased the number of PEs to 256 and reduced the storage requirement [21]. They used double buffering and memory scheduling to improve performance [22]. This parallel structure has been also used in neural network acceleration [23]–[25] and achieved a better effect.

This design method has become the optimal solution for large-scale matrix multiplication problems. After deep learning, we found that there still exists the space of improvement and better application prospects based on the advantages of parallel block matrix multiplication algorithm and linear array structure.

Aiming at the fusion problem of multiple matrix operations, a matrix computing system which integrated multiple accelerators have been developed for the mobile system [26]. They improved the communication between accelerators by setting shared-matrix cache, which could reduce the external memory bandwidth requirement. However, they need independent computing accelerators for different matrix operations. Owing to the structure of each accelerator can not be reusable, this method requires more resources and area. In addition to the PC, a separate soft core processor is required to allocate data blocks and generate instructions, so the data access scheduling mechanism is also more complicated.

Another recent matrix computing structure based on circulant matrix have been proposed [27], which can handle multiple operations of matrix computing. But it only used for fixed-point data, which can not meet the needs of most engineering calculations. For instance, the nonlinear Kalman filter algorithm and the extended Kalman filter algorithm have a requirement of multi-operation for floating-point matrix computing.

B. PARALLEL BLOCK MATRIX MULTIPLICATION ALGORITHM AND LINEAR ARRAY STRUCTURE

The most important function of matrix computing is matrix multiplication, as shown in Equation (1), where matrix A and matrix B are arbitrary sizes.

\[ C_{M \times N} = A_{M \times K} \times B_{K \times N} \]  

(1)

We first introduce an algorithm for parallel block matrix multiplication [21]. As shown in Algorithm 1, it includes three outer loops and two innermost loops. The outer loops with loop variable \( T_p \), \( T_{t1} \), and \( T_2 \) are used for data transfer of subblocks and are implemented as “Data Transfer Controller”. The inner loops with loop variable \( p \) and \( t1 \) are used for actual calculation in processor array. After the block processing, the size of the submatrix blocks \( A \), \( B \), and \( C \) are \( S_p \times K \), \( K \times S_{t1} \), and \( S_p \times S_{t1} \), respectively. Parameters \( S_p \) and \( S_{t1} \) are also represent the number of PE and the depth of local memory in each PE, respectively.

\[
\begin{align*}
\text{A} & \rightarrow \text{PC} \\
\text{PC} & \rightarrow \text{PE}_i \\
\text{PE}_i & \rightarrow \text{PC} \\
\text{PC} & \rightarrow \text{PE}_j \\
\text{PE}_j & \rightarrow \text{PC} \\
\text{PC} & \rightarrow \text{PE}_k \\
\text{PE}_k & \rightarrow \text{PC} \\
\text{PC} & \rightarrow \text{PE}_m \\
\text{PE}_m & \rightarrow \text{PC} \\
\text{PC} & \rightarrow \text{PE}_n \\
\text{PE}_n & \rightarrow \text{PC} \\
\end{align*}
\]

We first introduce an algorithm for parallel block matrix multiplication [21]. As shown in Algorithm 1, it includes three outer loops and two innermost loops. The outer loops with loop variable \( T_p \), \( T_{t1} \), and \( T_2 \) are used for data transfer of subblocks and are implemented as “Data Transfer Controller”. The inner loops with loop variable \( p \) and \( t1 \) are used for actual calculation in processor array. After the block processing, the size of the submatrix blocks \( A \), \( B \), and \( C \) are \( S_p \times K \), \( K \times S_{t1} \), and \( S_p \times S_{t1} \), respectively. Parameters \( S_p \) and \( S_{t1} \) are also represent the number of PE and the depth of local memory in each PE, respectively.
As shown in Figure 1, the linear array structure corresponding to Algorithm 1 generally consists of a group of PEs and a data transfer controller. Each PE includes two registers for storing a data element from submatrix A and B respectively, a local memory c for storing a row data elements from submatrix C and a multiply-accumulate functional unit for the matrix multiplication operation.

All data elements of submatrix C and one column data elements of submatrix A are preloaded into local memory c and registers a in each PE, respectively. While the elements of submatrix B flow into the first PE in rows, the linear array starts computing. Then the elements of submatrix B pass between adjacent PEs in a pipelined manner. Each PE completes the operation of $c^k = a \times b + c^{k-1}$ through the write-back mechanism and saves the intermediate result in local memory. After K iterations, the data transfer controller moves the results to external memory, and then proceed to the next submatrix process.

Algorithm 1 Block Matrix Multiplication

```plaintext
for $T_p = 0$ to $M - 1$, $S_p$ do
  for $T_{t1} = 0$ to $N - 1$, $S_{t1}$ do
    Load data block
    $C[T_p: T_p + S_p - 1, T_{t1}: T_{t1} + S_{t1} - 1]$ for $t2 = 0$ to $K - 1$ do
      Load data block $A[T_p: T_p + S_p - 1, t2]$
      Load data block $B[t2, T_{t1}: T_{t1} + S_{t1} - 1]$ for $p = T_p$ to $T_p + S_p - 1$ do
        for $t1 = T_{t1}$ to $T_{t1} + S_{t1} - 1$ do
          $C[p, t1] = C[p, t1] + A[p, t2] \times B[t2, t1]$
        Store data block $C[T_p: T_p + S_p - 1, T_{t1}: T_{t1} + S_{t1} - 1]$
    End Loop
End Loop
```

III. OPTIMIZATION FOR PARALLEL BLOCK MATRIX MULTIPLICATION

In this section, we mainly focus on the inner loop of Algorithm 1, that is, how to efficiently handle the matrix multiplication with submatrix blocks. To this end, we optimize the original algorithm and structure from four aspects.

A. REDUCING DATA TRANSMISSION

For submatrix C in Algorithm 1, its initial value is zero for each element. The original method is preloading the data from external memory, which may consume memory bandwidth and increases the overall running time.

The better way is to determine the size of submatrix block $(S_p \times S_{t1})$, and then directly initialize submatrix C as a zero matrix on the chip. After the calculation of submatrix block is completed, the final result is moved to external memory, and submatrix C is re-initialized to a zero matrix. A simplified matrix multiplication algorithm for the submatrix is shown in Algorithm 2. In the computing process of each subblock, our method reduces the size of $S_p \times S_{t1}$ floating-point data transmission, so we can use the limited memory bandwidth for other data transmission.

Algorithm 2 Simplified Matrix Multiplication

```plaintext
Initialize data block $C[0:S_{p} - 1:0:S_{t1} - 1]$ to zero for $t2 = 0$ to $K - 1$ do
  load data block $A[0:S_{p} - 1, t2]$
  load data block $B[t2:0:S_{t1} - 1]$
  for $p = 0$ to $S_{p} - 1$ do
    for $q = 0$ to $S_{t1} - 1$ do
      $C[p, q] = C[p, q] + A[p, t2] \times B[t2, q]$
  Store data block $C[0: S_{p} - 1:0: S_{t1} - 1]$
```

B. OPTIMIZING DATA FLOW

The data sources participating in the matrix multiplication are divided into data flow A and data flow B. The optimized structure for matrix multiplication is shown in Figure 2.

- Data flow A is stored in registers, which greatly reduces storage requirement. Considering that it is difficult to ensure the continuity of data flow A by a single register, it may happen that the computing unit waiting for computing data. So we add a small FIFO in front of the register to prepare some computing data in advance. In the implementation, the data width of the FIFO is 32 or 64 bit. Considering the packet transmission, the depth of the FIFO is set to 32, 64, or 128.

- Data flow B transfers between adjacent PEs, which can minimize I/O. However, when the number of PEs increases, this way generates additional transmission and calculation delays. Moreover, separate counters and address generators are used to control data flow A and data flow B in each PE, respectively. Other structures use the broadcast method [19], but they only use a first-stage FIFO. In this case, because of
the restriction of PEs placement and the critical path is too long, it will reduce the frequency of the system.

In our design, we adopt the form of two-stage FIFO. The first-stage FIFO drives all the second-stage FIFOs, and each second-stage FIFO drives several PEs. One principle is to control the size of average fan-out from 4 to 8 according to the number of PEs. On one hand, the average fan-out is reduced and the logic is optimized, so we can guarantee the overall performance of the design. On the other hand, it is easy to control the data flow of all PEs and achieve a compromise between delay and complexity. When the number of PEs becomes larger, the form of three-stage FIFO can also be considered.

Through careful analysis of Algorithm 2 and Figure 2, we find two types of time hiding, which effectively avoids the waiting time and data risk of the computing process.

- The time of data moving is hidden in computing time. Submatrix B enters one data per clock cycle, and submatrix A enters a column of data \(S_{\text{t1}}\) every \(S_{\text{t1}}\) clock cycles. It means that the data of submatrix A will be reused \(S_{\text{t1}}\) times. While reading data flow B, the remaining data bandwidth can be used to load data flow A required for the next calculation in advance. Therefore, by properly selecting the depth of the submatrix B \(S_{\text{t1}}\), the computing time and memory access time can be overlapped.

- The latency of floating-point multiply-accumulate functional unit is hidden in the computing time. The floating-point multiply-accumulate functional unit we used complies with the IEEE 754 standard [28]. There is a certain delay when doing a multiply-accumulate operation. As shown in Table 1, this parameter can be set according to our requirements. As long as the depth of submatrix B \(S_{\text{t1}}\) is larger than the parameter, this delay can also be hidden in the calculation without any pause.

### C. BALANCING RESOURCE UTILIZATION

With advanced technology, FPGAs contain more and more DSPs, which are exactly the important part of the floating-point multiply-accumulate functional unit. These configurable DSPs can replace logic resources to implement multiply, addition, and other functions. Therefore, how to allocate and mine DSPs resources seems particularly meaningful. For instance, a method of parallel multiplication for single DSPs was developed to fully exploit the computing potential of DSPs [29].

As shown in Table 1, the number of DSPs can be flexibly set by medium usage or maximum usage. We use more DSPs to perform floating-point operations than previous work, which can reduce the consumption of logical resources and reserve more logical resources to other functional modules.

### D. ENHANCING CONFIGURABILITY OF THE ARCHITECTURE

According to our studies, there exist two ways to directly or indirectly enhancing configurability of the architecture. One is to use the block idea to decompose large-scale matrix multiplication into multiple submatrix multiplication. The other is to flexibly configure the length of linear arrays, such as constructing two-dimensional arrays.

We combine the above two methods. When the number of rows in matrix A is greater than the number of PEs, we select the block idea and one-dimensional array. When the number of rows in matrix A is less than the number of PEs and greater than half of the number of PEs, we reduce the number of PEs involved in the calculation by controlling the enable of the second-stage FIFO. When the number of rows of matrix A is less than half of the number of PEs, our second-stage FIFO can choose to receive the data from first-stage FIFO, forming the effect of two-dimensional arrays working in parallel.

### IV. MULTI-OPERATION OF MATRIX COMPUTING BASED ON UNIFIED STRUCTURE

To solve multiple floating-point matrix operations in a unified hardware structure, in this section, we will discuss how to process other matrix operations based on our optimized solution in section III.

As shown in Figure 3, almost all the logics of matrix multiplication are reused. Data flow A and C are relatively fixed. As long as we provide different data sources (matrix, vector, and scalar) for data flow B, we can implement a variety of matrix operations. The algorithm and process of each operation are explained below.

### A. MATRIX ADDITION AND MATRIX SUBTRACTION

Matrix addition and matrix subtraction use the same structure and data flow method as matrix multiplication. The only
difference is setting the operation mode to add or subtract through the floating-point arithmetic IP core. As shown in Equation (2) and Algorithm 3, we introduce the identity matrix I to achieve the addition or subtraction of submatrix A and submatrix C.

\[ C_{M \times K} = A_{M \times K} \times I_{K \times K} \pm C_{M \times K} \]  

(2)

Algorithm 3 Matrix Addition and Subtraction

| Function | Description |
|----------|-------------|
| Initialization | Initialize data block C[0:S_p-1,0:K-1] to zero |
| Loop | for t_2 = 0 to K-1 do |
| Operations | Load data block A[0:S_p-1,t_2]; |
| Control flow | Load data B[t_2]; |
| Control flow | for p = 0 to S_p-1 do |
| Operations | C[p] = C[p] + A[p,t_2] \times B[t_2]; |
| Termination | Store data block C[0:S_p-1,0:K-1]; |

As we can be seen from the calculation process, the delay of the floating-point multiply-accumulate functional unit can be hidden in the calculation time. There will be some pauses in this calculation process.

C. MATRIX-SCALAR MULTIPLICATION

As shown in Equation (4) and Algorithm 5, when performing matrix-scalar multiplication, the data flow B is a scalar.

\[ C_{M \times K} = A_{M \times K} \times b \]  

(4)

Algorithm 5 Matrix-Scalar Multiplication

| Function | Description |
|----------|-------------|
| Initialization | Initialize data block C[0:S_p-1,0:K-1] to zero |
| Loop | for t_2 = 0 to K-1 do |
| Operations | Load data block A[0:S_p-1,t_2]; |
| Operations | Load data b; |
| Control flow | for p = 0 to S_p-1 do |
| Operations | C[p] = A[p,t_2] \times b; |
| Termination | Store data block C[0:S_p-1,0]; |

As we can be seen from Figure 3(d), matrix-scalar multiplication does not require a rewrite mechanism, so the delay of the floating-point multiply-accumulate functional unit can also be hidden in the calculation.

V. IMPLEMENTATION FOR MCA UNITS AND SYSTEM

Combining the optimization of matrix multiplication in section III with other operations described in section IV, we propose a multi-operation MCA unit (Design-I) and continuous MCA unit (Design-II) for different needs. In this section, we also describe the implementation of the entire system based on our unit cores.

A. DESIGN-I: MULTI-OPERATION MCA UNIT

Design-I supports matrices computing of any sizes and multiple operation modes. Design-I contains a set of PEs, two-stage FIFOs, data channel, and RAM channel. As shown in Figure 4(a), each PE has a floating-point multiply-accumulate calculation unit, two registers, a FIFO, and a RAM block.

Before computing, the MCA unit receives the signal of matrix operation mode provided from outside and decides whether to load or to initialize the subblock C. It is necessary to load subblock C into RAM block when process matrix
addition or matrix subtraction. In other modes, the subblock C is initialized directly on-chip as a zero matrix or vector.

The MCA unit loads both data flow A and B while calculating. Because data flow A and B are supported by FIFO, and the memory bandwidth requirements of two data flows are low, the data supply can be guaranteed.

It is necessary to choose whether to perform the rewrite mechanism according to the operation mode. The matrix-scalar multiplication does not choose a rewrite mechanism and the result is obtained directly after each multiply-accumulate operation. Other modes need to choose rewrite mechanism, that is, after waiting for multiple consecutive multiply-accumulate operations, we get the final result.

After the calculation of submatrix block is completed, the result is transferred to off-chip memory. The calculation of the next submatrix block is performed until all subblocks complete computing.

**B. DESIGN-II: CONTINUOUS MCA UNIT**

For a sequence of matrices computing, the data access process usually has a high delay if the intermediate results of multiple matrices operations are first exported to external memory and then imported into on-chip for the next calculation. Besides, the BRMA resources embedded in most FPGAs are sufficient, so it is necessary to optimize this high time-consuming part individually.

We adjust the structure carefully, replacing PE shown in Figure 4 (a) with Figure 4 (b). We mainly replace the register that originally storing data flow A with a simple dual-port RAM to store a row of matrix A. This method simplifies data transfer control logic and it is easy to access one column of matrix A in the same cycle.

Furthermore, we merge the RAM storing data flow A and data flow C in each PE to form design-II. As shown in Figure 4 (c), we use a shared-memory mechanism to divide the RAM block into the upper half (RAM1) and the lower half (RAM2). Firstly, RAM1 is used to participate in multiplication operation. RAM2 is used to participate in the accumulation operation and save the intermediate results. Then, by switching the address, we use RAM2 to participate in multiplication operation of the next operation, use RAM1 to participate in the accumulation operation and save the intermediate results. Finally, we move the results of a series of operations to off-chip memory.

With the above storage scheme, design-II can handle multi-operation matrix computing as well as design-I. Moreover, it performs better than design-I in processing continuous matrices computing problems. We use Algorithm 6 to process a typical example of continuous matrices operation: the extended Kalman filter time update equation, as shown in Equation (5).

\[ P_k^- = A_k P_{k-1} A_k^T + \Gamma_{k-1} Q_{k-1} \Gamma_{k-1}^T \]

**Algorithm 6 Continuous MCA Unit**

1. Loading \( A_k \) into RAM1 through RAM channel;
2. Sending \( P_{k-1} \) through the data channel, and then computing \( temp1 = A_k \times P_{k-1} \), saving the result to RAM2;
3. Transposing \( \Gamma_{k-1} \) and sending it into the data channel, and then computing \( temp2 = temp1 \times A_k^T \), saving the result to external memory;
4. Loading \( \Gamma_{k-1} \) into RAM1 through RAM channel;
5. Sending \( Q_{k-1} \) through the data channel, and then computing \( temp3 = \Gamma_{k-1} \times Q_{k-1} \), saving the result to RAM2;
6. Transposing \( \Gamma_{k-1} \) and sending into data channel, and then computing \( temp4 = Temp3 \times \Gamma_{k-1}^T \), saving the result to RAM1;
7. Loading \( temp2 \) into RAM2 through RAM channel, and then sending identity matrix I through the data channel, computing result = \( temp4 + temp2 \), saving the final result to external memory.

We effectively reduce delays that critical to real-time requirements by reducing most of the data movement between on-chip and off-chip.

**C. OVERALL ARCHITECTURE OF MCA SYSTEM**

In this section, we built an entire system to verify the function and evaluate the performance of design-I and design-II. As shown in Figure 7, the MCA unit core can be design-I or design-II according to different computing requirements.

The MCA system has two parts: the host processor (such as DSP) and the coprocessor. We use SRI0 as an interconnect interface between the host processor and coprocessor. There is no direct connection between the host processor and the MCA unit. We transfer data from memory of the host processor side to memory (1GB DDR3) of the coprocessor side, and then coprocessor complete computing process by itself, which greatly reduces the burden of the host.

The coprocessor includes MCA unit core and data transfer logic. Data transfer logics can be divided into data transfer controllers (DTC1 and DTC2), memory controller. DTC1 is used to transfer data between the host processor and coprocessor. DTC2 is responsible for internally selecting the computing mode, controlling and transmitting data flows. The memory controller transfers data from DDR to on-chip memory and then returns the result from on-chip memory to DDR.

Besides, the communication between the host processor and coprocessor, the internal data transmission of the coprocessor, and the floating-point multiply-accumulate functional unit all support the AXI-4 protocol, which facilitates communication and control between the various parts.
VI. EXPERIMENTS AND RESULTS

We simulated and synthesized the MCA unit core of two designs using Vivado 2016 and then deployed the MCA system on the 585T development board of the Xilinx.

A. SYNTHESIS RESULTS

As shown in Figure 6 (a), we analyze power consumption and resource consumption of design-I according to different numbers of PEs with single-precision floating-point data. When the number of PEs increases, both the power consumption and resource consumption such as LUTs, Slice Registers, and DSPs are increasing.

After completed the comprehensive implementation, we used the power analysis tool in Vivado 2016 to accurately calculate the power consumption. Our design is synthesized with 28 nm 585T FPGA under typical condition (1V, 25 °C).

The consumption of RAM blocks is not only related to the number of PEs but also related to the size of submatrix block. Because the number of PEs determines the amount of RAM used and the size of submatrix block determines the depth of RAM, both of them affect RAM block consumption.

As shown in Figure 6 (b), when the number of PEs is 256 and the depth of RAM is 512, we get the comparison of used resources and total resources. In general, the longer the linear array, the more data that can be processed at the same time, and the better the computing performance. The main factor that restricts the number of PEs is the number of DSPs, for the floating-point multiply-accumulate functional unit in each PE consumes more DSPs than the fixed-point multiply-accumulate functional unit.

Besides, the number of on-chip RAM blocks consumed is relatively small, which means that there is still a large amount of unused on-chip memory. This is also a prerequisite for setting more on-chip memory in section V. Table 2 shows that we increased the number of RAM blocks used in design-II.

B. PERFORMANCE ANALYSIS

For matrix computing of different sizes and different operation modes, Table 3 shows clock cycles required by the MCA unit.

Taking matrix multiplication as an example, we analyze the rule that the maximum operating frequency of matrix computing unit varies with the number of PEs. As shown in Table 4, the maximum frequency slowly decreases with the increasing number of PEs. When the number of PEs is 256, the maximum frequency is 195 MHz and the performance nearly achieves 99.8 GFLOPS.

Similarly, we process matrices of different sizes to get execution time and compare it to the computing server with an Intel i7-4770 (3.4Ghz). As shown in Figure 7(a), since the parallel structure fully exploits data reuse, our linear array is superior to the general-purpose processor. When the matrix size becomes larger, the acceleration effect of our linear array is more obvious.

As we know, the overall time of matrix calculation includes computing time and external memory access time. We will discuss external memory access time and total time of two designs.
For a single matrix calculation, total time \( T_1 \) is expressed by Equation (6), where \( T_{\text{pre}A} \) represents the time of preloading the first column of submatrix A, and \( T_{\text{compute}} \) represents the time from the start of calculation to the completion of last calculation, \( T_{\text{return}C} \) represents the time returning calculation result.

\[
T_1 = T_{\text{pre}A} + T_{\text{compute}} + T_{\text{return}C} \quad (6)
\]

Compared to total time, the time of prefetching data is too small and the time of other data moving is hidden in the computing time when matrix size becomes larger (such as 512), so design-I and design-II are efficient for large-scale matrix computing.

For continuous matrix operations, if we use single matrix operating method to process in turn, total time \( T_2 \) can be expressed by Equation (7), where \( T_{\text{pre}1} \) represents the time to preload the first column of matrix A, and \( T_{\text{compute}1} \) represents the time of the first matrix operation, \( T_{\text{return}1} \) represents the time to return the result of the first matrix operation, \( T_{\text{pre}2} \) indicates the time to preload the result of first matrix operation, \( T_{\text{compute}2} \) indicates the time to calculate the second matrix operation, \( T_{\text{return}C} \) represents the time to return the final result. At this time, external memory access bandwidth is likely to become a bottleneck for performance improvement.

\[
T_2 = T_{\text{pre}1} + T_{\text{compute}1} + T_{\text{return}1} + T_{\text{pre}2} + T_{\text{compute}2} + \ldots + T_{\text{return}C} \quad (7)
\]

We adopt design-II to handle the above problem. Total time \( T_3 \) can be expressed by Equation (8).

\[
T_3 = T_{\text{pre}1} + T_{\text{compute}1} + T_{\text{compute}2} + \ldots + T_{\text{return}C} \quad (8)
\]

We choose matrices of the same size (such as 256 × 512) and then use these two methods to process a sequence of matrix operations to compare their total time (\( T_2 \) without shared-memory and \( T_3 \) with shared-memory) and total amount of data transferred (\( DT_2 \) without shared-memory and \( DT_3 \) with shared-memory). As shown in Figure 7(b), design-II greatly reduces total time and total amount of data transferred compared to design-I.

**TABLE 5.** Performance and hardware overhead compared to related work.

| Data type        | [21] | [26] | [27] |
|------------------|------|------|------|
| Matrix size      | \( M \times N \) | \( M \times N \) | \( M \times N \) |
| No. of PEs       | \( P \) | \( P \) | N.A. |
| Data preloading  | \( S_p \) | \( S_p + S_p S_{11} \) | N.A. |
| Storage          | \( 2S_p S_{11} \) | \( 2S_p S_{11} \) | N.A. |
| Latency          | \( N^3 \) / \( P \) + 7 | \( N^3 \) / \( P \) + \( P \) | N.A. |
| \( f (\text{MHz}) \) | 195 | 181 | 150 | 346 |
| Performance      | 99.8 | N.A. | 76.8 | 173 |
| Power(W)         | 5.24 | N.A. | 4.59 | N.A. |
| Energy efficiency| 19.05 | N.A. | 16.7 | N.A. |

**C. COMPARISON WITH PREVIOUS WORK**

In Table 5, we first compare the performance and hardware overhead of our two designs to the existing design by Wu et al. [21] under the same circumstances. In the calculation of single matrix operation, the total size of data transmission \( (M \times N) \) is reduced both in design-I and design-II, for each submatrix operation saving data transmission \( (S_p \times S_{11}) \). The size of preloading data is also less than Wu et al. [21]. More importantly, design-I and design-II support more matrix computing modes and better structural flexibility than previous work. Besides, design-II supports continuous matrix operations, which further reduces the amount of data transmission and the total computing time.

In terms of integrating different functions of matrix operation in the same system, the two most relevant structures up to date have been designed by Wang et al. [26] and Abbaszadeh et al. [27].

Wang et al. [26] have implemented matrix addition, matrix subtraction, dot multiplication, dot division, and matrix multiplication through integrating multiple independent accelerators. As they obtain a performance of 76.8 GFLOPS on the Stratix V platform for matrix multiplication, our designs achieve a superior performance of 99.8 GFLOPS. Owing to the optimization of architecture, data flow, and memory access, our energy efficiency ratio is better than the work [26].

Abbaszadeh et al. [27] tried to design a matrix calculation unit based on the circulant matrix. This structure uses the DSP48E primitives embedded in FPGAs as a fixed-point multiply-accumulate functional unit, including a 48-bit pre-adder, a 23 × 28 multiplier, and a 48-bit multiply-accumulate unit. It can switch between multiple operations and achieve a performance of 173 GFLOPS for calculating 500 × 500 matrices.
However, considering the scope of application, its data width is limited to fixed-point (18 bits) and can only calculate the square matrix. Its matrix size that can be processed cannot be too large because the size of the matrix is equal to the number of PEs. For example, a $128 \times 1024$ rectangular matrix cannot be processed by their architecture. Besides, it takes a long time to build a circulant matrix before the calculation starts.

As shown in Table 5, our design is superior to the structure designed by Abbassadeh et al. [27] in terms of processing capacity (such as data precision and matrices size). Besides, for matrix computing of the same size, such as $500 \times 500$ matrices, the storage requirement of design-I ($O(n^2)$) is significantly smaller than previous work ($O(n^3)$) [27].

VII. CONCLUSION

This article mainly focuses on the optimization of the matrix multiplication algorithm and linear array on FPGAs, and then we propose two scalable matrix computing hardware structures and build a matrix computing acceleration system. Compared with the existing work, design-I has obvious advantages of low storage demand and computing efficiency in multi-operation matrix computing, design-II features less time consumption and data transfer times. This paper has not discussed the continuous matrix calculation problem with the block idea. How to use the block matrix idea to efficiently handle a sequence of large-scale matrix operation problems will be a direction of future research.

REFERENCES

[1] A. Amira, A. Bouridane, and P. Milligan, “Accelerating matrix product on reconfigurable hardware for signal processing,” in Proc. 11th Int. Conf. Field-Program. Logic Appl., Aug. 2001, pp. 101–111.

[2] F. Bensaali, A. Amira, and A. Bouridane, “Accelerating matrix product on reconfigurable hardware for image processing applications,” IEEE Proc.-Circuits. Devices Syst., vol. 152, no. 3, pp. 236–246, Mar. 2005.

[3] J. V. Frances-Villora, A. Rosado-Muñoz, J. M. Martínez-Villena, M. Bataller-Monpeán, J. F. Guerrero, and M. Wegrzynek, “Hardware implementation of real-time extreme learning machine in FPGA: Analysis of precision, resource occupation and performance,” Comput. Electr. Eng., vol. 51, pp. 139–156, Apr. 2016.

[4] K. D. Underwood and K. S. Hemmert, “Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance,” in Proc. 12th Annu. IEEE Symp. Field-Program. Custom Comput. Mach., Apr. 2004, pp. 219–228.

[5] M. Parker, “High-performance floating-point implementation using FPGAs,” in Proc. IEEE 28th Mil. Commun. Conf. (MICLCOM), Boston, MA, USA, Oct. 2009, pp. 323–327.

[6] Z. Jovanović and V. Milutinović, “FPGA accelerator for floating-point matrix multiplication,” IET Comput. Digit. Techn., vol. 6, no. 4, pp. 249–256, 2012.

[7] J.-W. Jang, S. Choi, and V. K. K. Prasanna, “Area and time efficient implementations of matrix multiplication on FPGAs,” in Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT), Mar. 2003, pp. 93–100.

[8] D. N. Sonawane, M. S. Sutaone, and I. Malek, “Systolic architecture for integer point matrix multiplication using FPGA,” in Proc. 4th IEEE Conf. Ind. Electron. Appl., May 2009, pp. 3822–3825.

[9] P. Corsonello, S. Perri, and P. Zicari, “A matrix product coprocessor for FPGA embedded soft processors,” in Proc. Int. Symp. Signals, Circuits Syst. (ISSCS), Jul. 2005, pp. 489–492.

[10] H. V. Jagadish and T. Kailath, “A family of new efficient arrays for matrix multiplication,” IEEE Trans. Comput., vol. 38, no. 1, pp. 149–155, 1st Quart., 1989.

[11] S. J. Campbell and S. P. Khatri, “Resource and delay efficient matrix multiplication using newer FPGA devices,” in Proc. 16th ACM Great Lakes Symp. VLSI (GLSVLSI), vol. 6, 2006, p. 308.

[12] S. M. Qasmi, S. A. Abbasi, and B. Almarshy, “A proposed FPGA-based parallel architecture for matrix multiplication,” in Proc. IEEE Asia-Pacific Conf. Circuits Syst. (APCCAS), Nov. 2008, pp. 1763–1766.

[13] L. T. Zhou, Y. D. Tao, S. Liu, and L. I. Suo, “Research on systolic multiplication technology based on FPGA,” Comput. Eng. Sci., vol. 37, no. 9, Sep. 2015.

[14] S. Kestur, J. D. Davis, and E. S. Chung, “Towards a universal FPGA matrix-vector multiplication architecture,” in Proc. IEEE 20th Int. Symp. Field-Program. Custom Comput. Mach. New York, NY, USA: IEEE, Apr. 2012, pp. 9–16.

[15] S. Sun, M. Monga, P. H. Jones, and J. Zambreno, “An I/O bandwidth-sensitive sparse matrix-vector multiplication engine on FPGAs,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 59, no. 1, pp. 113–123, Jan. 2012.

[16] S. Jain-Mendon and R. Sass, “A hardware-software co-design approach for implementing sparse matrix vector multiplication on FPGAs,” Microprocessors Microsyst., vol. 38, no. 8, pp. 873–888, Nov. 2014.

[17] J.-W. Jang, S. B. Choi, and V. K. Prasanna, “Energy- and time-efficient matrix multiplication on FPGAs,” Trans. Very Large Scale Integr. (VLSI) Syst., vol. 13, no. 11, pp. 1305–1319, Nov. 2005.

[18] L. Zhuo and V. K. Prasanna, “Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems,” IEEE Trans. Parallel Distrib. Syst., vol. 18, no. 4, pp. 433–446, Apr. 2007.

[19] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan, “FPGA based high performance double-precision matrix multiplication,” Int. J. Parallel Program., vol. 38, nos. 3–4, pp. 322–338, Jun. 2010.

[20] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, “64-bit floating-point FPGA matrix multiplication,” in Proc. ACM/SIGDA 13th Int. Symp. Field-Programmable Gate Arrays (FPGA), Monterey, CA, USA, 2005, pp. 86–95.

[21] G. Wu, Y. Dou, and M. Wang, “High performance and memory efficient implementation of matrix multiplication on FPGAs,” in Proc. Int. Conf. Field-Program. Technol., Beijing, China, Dec. 2010, pp. 134–137.

[22] G. Wu, Y. Dou, and M. Wang, “High performance and memory efficient implementation of matrix multiplication on FPGAs,” in Proc. Int. Conf. Field-Programmable Technol., Beijing, China, Dec. 2010, pp. 134–137.

[23] Z. Liu, Y. Dou, J. Jiang, X. Xu, S. Li, Y. Zhou, and Y. Xu, “Throughput-optimized FPGA accelerator for deep convolutional neural networks,” ACM Trans. Reconfigurable Technol. Syst., vol. 10, no. 3, pp. 1–23, Jul. 2017.

[24] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen, and C. Zhang, “FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency,” Concurrency Comput., Pract. Exper., vol. 29, no. 20, Oct. 2017, Art. no. e3850.

[25] J. Shen, Y. Qiao, Y. Huang, M. Wen, and C. Zhang, “Towards a multi-array architecture for accelerating large-scale matrix multiplication on FPGAs,” in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), May 2018, pp. 1–5.

[26] W. Wang, K. Guo, M. Gu, Y. Ma, and Y. Wang, “A universal FPGA-based floating-point matrix processor for mobile systems,” in Proc. Int. Conf. Field-Programmable Technol. (FPT), New York, NY, USA: IEEE Dec. 2014, pp. 94–106.

[27] A. Abbassadeh, T. Iakymchuk, M. Bataller-Monpeán, J. Francés-Villora, and A. Rosado-Muñoz, “Asynclable matrix computing unit architecture for FPGA, and SCU/MO user design interface,” Electronics, vol. 8, no. 1, p. 94, Jan. 2019.

[28] IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, 2008, pp. 1–70.

[29] Z. Huang, S. Zhang, and W. Wang, “An efficient method of parallel multiplication on a single DSP slice for embedded FPGAs,” IEEE Access, vol. 7, pp. 100993–101008, 2019.

LONGLONG ZHANG was born in 1990. He received the B.S. degree in electronics engineering from the National University of Defense and Technology (NUDT), in 2012, China, where he is currently pursuing the M.S. degree in electronics science and technology. His research interest includes high-performance computer architecture and computing engineering.
**YUANXI PENG** was born in 1966. He received the B.S. degree in computer science from Sichuan University, China, in 1988, and the M.S. and Ph.D. degrees in computer science from the National University of Defense and Technology (NUDT), China, in 1998 and 2001, respectively. He was a Visiting Professor with the Department of Electronics and Computer Engineering, University of Toronto, Canada, from 2010 to 2011. He has been a Professor with the College of Computer, NUDT, since 2011. His research interests are in the areas of high-performance computing, multi- and many-core architectures, on-chip networks, cache coherence protocols, and architectural support for parallel programming.

**AHUI HUANG** was born in 1997. She received the B.S. degree in electronics science and technology in Central South University of Forestry and Technology, China, in 2013. She is currently working toward the M.S. degree in integrated circuit engineering at the National University of Defense and Technology (NUDT), China. Her research interest includes high performance computer architecture and computing engineering.

**XIAO HU** was born in 1977. He received the Ph.D. degree in electronics science and technology from National University of Defense and Technology (NUDT), China. He has been an Associate Professor with College of Computer, NUDT. He has long been engaged in the development and application of digital signal processors. His research interests include DSP design, test, and embedded application technology.

---

**YUANXI PENG** was born in 1966. He received the B.S. degree in computer science from Sichuan University, China, in 1988, and the M.S. and Ph.D. degrees in computer science from the National University of Defense and Technology (NUDT), China, in 1998 and 2001, respectively. He was a Visiting Professor with the Department of Electronics and Computer Engineering, University of Toronto, Canada, from 2010 to 2011. He has been a Professor with the College of Computer, NUDT, since 2011. His research interests are in the areas of high-performance computing, multi- and many-core architectures, on-chip networks, cache coherence protocols, and architectural support for parallel programming.

**AHUI HUANG** was born in 1997. She received the B.S. degree in electronics science and technology in Central South University of Forestry and Technology, China, in 2013. She is currently working toward the M.S. degree in integrated circuit engineering at the National University of Defense and Technology (NUDT), China. Her research interest includes high performance computer architecture and computing engineering.

---

**XIAO HU** was born in 1977. He received the Ph.D. degree in electronics science and technology from National University of Defense and Technology (NUDT), China. He has been an Associate Professor with College of Computer, NUDT. He has long been engaged in the development and application of digital signal processors. His research interests include DSP design, test, and embedded application technology.

---

**YUANXI PENG** was born in 1966. He received the B.S. degree in computer science from Sichuan University, China, in 1988, and the M.S. and Ph.D. degrees in computer science from the National University of Defense and Technology (NUDT), China, in 1998 and 2001, respectively. He was a Visiting Professor with the Department of Electronics and Computer Engineering, University of Toronto, Canada, from 2010 to 2011. He has been a Professor with the College of Computer, NUDT, since 2011. His research interests are in the areas of high-performance computing, multi- and many-core architectures, on-chip networks, cache coherence protocols, and architectural support for parallel programming.

**AHUI HUANG** was born in 1997. She received the B.S. degree in electronics science and technology in Central South University of Forestry and Technology, China, in 2013. She is currently working toward the M.S. degree in integrated circuit engineering at the National University of Defense and Technology (NUDT), China. Her research interest includes high performance computer architecture and computing engineering.

---

**XIAO HU** was born in 1977. He received the Ph.D. degree in electronics science and technology from National University of Defense and Technology (NUDT), China. He has been an Associate Professor with College of Computer, NUDT. He has long been engaged in the development and application of digital signal processors. His research interests include DSP design, test, and embedded application technology.

---

**YUANXI PENG** was born in 1966. He received the B.S. degree in computer science from Sichuan University, China, in 1988, and the M.S. and Ph.D. degrees in computer science from the National University of Defense and Technology (NUDT), China, in 1998 and 2001, respectively. He was a Visiting Professor with the Department of Electronics and Computer Engineering, University of Toronto, Canada, from 2010 to 2011. He has been a Professor with the College of Computer, NUDT, since 2011. His research interests are in the areas of high-performance computing, multi- and many-core architectures, on-chip networks, cache coherence protocols, and architectural support for parallel programming.

**AHUI HUANG** was born in 1997. She received the B.S. degree in electronics science and technology in Central South University of Forestry and Technology, China, in 2013. She is currently working toward the M.S. degree in integrated circuit engineering at the National University of Defense and Technology (NUDT), China. Her research interest includes high performance computer architecture and computing engineering.

---

**XIAO HU** was born in 1977. He received the Ph.D. degree in electronics science and technology from National University of Defense and Technology (NUDT), China. He has been an Associate Professor with College of Computer, NUDT. He has long been engaged in the development and application of digital signal processors. His research interests include DSP design, test, and embedded application technology.