Received June 5, 2019, accepted July 15, 2019, date of publication July 25, 2019, date of current version August 9, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2931161

An Efficient Method of Parallel Multiplication on a Single DSP Slice for Embedded FPGAs

ZHANGQIN HUANG, SHUO ZHANG, AND WEIDONG WANG

Beijing Engineering Research Center for IoT Software and Systems, Beijing University of Technology, Beijing 100124, China
Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China

Corresponding authors: Zhangqin Huang (zhuang@bjut.edu.cn), Shuo Zhang (zhangshuo2013@emails.bjut.edu.cn), and Weidong Wang (wangweidong@bjut.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61502018, in part by the Beijing Municipal Science and Technology Project under Grant KM201910005031, and in part by the International Research Cooperation Seed Fund of the Beijing University of Technology under Grant 2018B07.

ABSTRACT Field-programmable gate arrays (FPGAs) can efficiently implement custom applications via their embedded digital signal processor (DSP) slices, including binary multipliers. An increasing number of binary multipliers belonging to a DSP slice usually demonstrate that it has the capacity to process as many multiplication operations as possible in one clock cycle. In order to fully utilize the DSP resource, in this paper, we propose a novel DSP slice optimization method to achieve parallel multiplication on single DSP slice, namely PMSDS. First, the PMSDS splits multiplicators into two separate parts, i.e., valid bits and vacant bits, using a customized polynomial algebra method. Then, the PMSDS pre-calculates the maximum number of overflow bits combining the above-mentioned polynomial algebra method. Finally, it computes the total multiplicators' bit numbers and parallel the final multiplicators. We also propose an optimization model to find the best parallel solution according to the performance and precision of a single DSP slice. Moreover, we implement a PMSDS-based matrix multiplication algorithm supporting the computing precision dynamically changing. The experiments based on a large-scale and real-world matrix multiplication show that the PMSDS has better performance in latency and resource utilization than the traditional, add-tree, and full-unroll methods and is more outstanding in frequency and dynamic power consumption comparing with the state-of-the-art methods.

INDEX TERMS DSP slice, FPGAs, multiplication, performance optimization, compute resource.

I. INTRODUCTION

In embedded computing systems, DSP (Digital Signal Processor) slices inside FPGAs (Field Programmable Gate Arrays), are well-known as one of the most precious and limited resources [1]–[3]. Designers always expect to use the limited DSP resource to accomplish as much work as possible in a given time [4]. For example, a DSP slice configured as multiplier can do a multi-digit multiplication in one clock cycle. In most cases, to accomplish massive multiplication calculations in the given time, we have to use high-performance FPGAs including as many DSP multipliers as possible, which can cause dramatic increase in development cost [5], [6]. Due to the cost of high-performance FPGAs, these multiple DSP multipliers are usually only employed for critical systems [7]–[10]. In the embedded computing design, however, it is possible to build a high-performance embedded system without paying the cost of FPGAs integrating multiple DSP multipliers. This is because that we can optimize the structure of a multiplier and obtain more multipliers available during the clock cycle without changing the hardware design. In practice, many previous studies have presented various optimization approaches for DSP multipliers [2], [6]–[16]. Here we discuss some of the reputed work from two aspects, application-level optimization and structure-level optimization, respectively.

Application-level optimization approaches assign DSP multiplier resource to the executable program specified in priority order. In that case, the increasing utilization rate of DSP multiplier resource directly leads to the fast speed of program execution. Ronak and Fahmy, [2], [11], [12] presented an automated tool for mapping arbitrary multiply expressions from the applications to FPGA DSP resource at maximum throughput. Cheah *et al.* [6] presented a lean DSP deployed method to minimize additional logic utilization. Lucas *et al.* [13] proposed a methodology to infer arithmetic

The associate editor coordinating the review of this manuscript and approving it for publication was Stavros Souravlas.

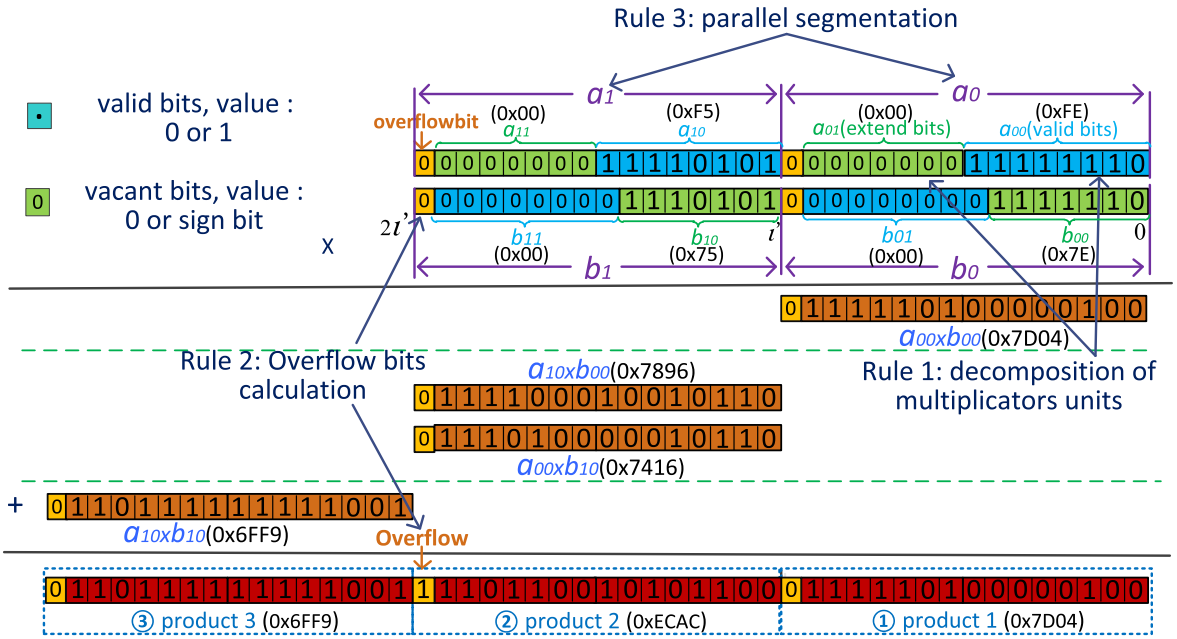


FIGURE 1. The basic of PMSDS method.

operations on DSP slices by supporting DSP resource reuse. However, such application-level optimization approaches do not take into account the configurable structure inside the DSP slice.

These above optimization approaches are always tendentious to improve the dispatching velocity and most of innate functions in DSP resource are executed in a sequential manner.

Structure-level optimization approaches analyze, design, and reconfigure the architecture of DSP multipliers to improve computational efficiency. Zhang *et al.* [8] presented a merged floating-point multiplier architecture. Sano and Yamamoto [10] proposed a data-flow based fine-grain parallelism architecture for scalable fluid simulation. Fu *et al.* [14] proposed an architecture to process two multipliers concurrent on the same DSP slice. In our previous work [17], we have done some optimization research of the DSP slice in parallel and unroll loops. However, such structure-level optimization approaches are absent from the scale cost analysis of the candidate algorithms.

Being complementary to the above methods, this paper proposes a parallel approach for the DSP slice and focuses on the DSP slice input and output structure to extend the full potential of a single DSP slice.

The contributions of this paper are listed as follows,

• we present a method of parallel multiplication on a single DSP slice at the same cycle, i.e. PMSDS method. The parallel method employs three separate steps to realize parallel multiplication. Firstly, this approach extends the multiplicators into two separate parts, original valid bits and extended vacant bits, by using a customized polynomial algebra method. Secondly, we propose a method to search the maximum number of overflow bits combining the above polynomial algebra method based on the enumeration method. Finally, it computes the total bit numbers of the multiplicators and parallel the final multiplier.

• we propose an optimization model to search the best parallel solution according to the acceleration, precision, and throughput of the DSP multiplier based on the Integer Programming Optimization Theory.

• we implement a precision dynamically changing PMSDS method, which is able to further extend the flexibility and efficiency of the DSP slice on FPGA.

The rest of this paper is organized as follows. Section II shows the preliminary of the DSP slice optimization method. Section III describes the proposed optimization algorithm in detail. Section IV presents the experimental results. And Section V concludes this paper and outlines the future work.

II. PRELIMINARY

Our parallel approach is based on the analysis of the polynomial algebra multiplication [17]–[19], and we summarize three rules for the parallel multiplier design, as shown in Figure 1. The first rule is for the decomposition of multiplicator units. we propose a customized polynomial algebra method by splitting the multiplicator units into two separate parts, valid bits and vacant bits, respectively. The valid bits are of the actual multiplicator bits and the vacant bits are of the extended bits for the higher bits of the product. The second rule is for the calculation of the maximum number of overflow bits. We pre-calculate the maximum overflow bits' number and reserve them in the bit number of the multiplicator unit to keep the parallel products no contamination. The third rule is for the multiplier parallel segmentation. we calculate the final bit numbers and parallel numbers of

the multiplicators and parallel products to implement parallel multiplier. The details of the above three rules are as follows.

Rule1 Decomposition of Multiplicator Units: The maximum valid bits' number of the product from a multiplier, is the sum of two input multiplicators' valid bits' numbers [18]. So, if we extend the bit number of two multiplicators from a multiplier with vacant value *0*, and guarantee that the total bit number of each multiplicators equals the sum of their original valid bit numbers, the product of these multiple multiplicators in parallel can be separable.

In this rule, we use Formula (1) to express the multiplicators units as follows,

$$\begin{aligned} A &= \sum_{i=0}^{n} (a_{i1}2^{v_A} + a_{i0})2^{i\iota}, \\ B &= \sum_{j=0}^{k} (b_{j1}2^{v_B} + b_{j0})2^{j\iota} \end{aligned} \tag{1}$$

where a_{i0}, b_{i0} are the multiplicators' original valid bits, a_{i1} and b_{i1} are the extended bits of the multiplicators, v_A and v_B are the bit numbers of a_{i0} and b_{i0}, n and k are the maximum decomposition values of A and B, ι is the counting unit of the multiplier.

We use N_a and N_b to present the numbers of multiplicators a_i and b_i, and the values of N_a and N_b are $(n+1)$ and $(k+1)$ as defined in Formula (1). And N_a and N_b are to be the parallel numbers of the multiplicator units.

Based on Formula (1), the product of A and B can be expressed as Formula (2),

$$AB = \sum_{\tau=0}^{n+k} c_\tau 2^{\tau\iota}, \quad c_\tau = \sum_{i+j=\tau} a_{i0}b_{j0}, \tag{2}$$

where τ is the coefficient of counting unit ι, c_τ is the product sum (or product, when the solution number of $i+j=\tau$ is *1*, i.e. $i=0, j=0$ or $i=n, j=k$) of multiplicators' original valid bits, and i is in range from 0 to n, j is in $[0, k]$, n, k and ι are the same as Formula (1).

As shown in Formula (2), the values of c_τ are the parallel products that we want to calculate inparallel multiplication. We use N_p to present the number of c_τ, and the value of N_p is $(n+k+1)$, as defined in Formula (2), or calculated by N_a and N_b, as (N_a+N_b-1). And N_p is to be the segmented number of parallel products.

As shown in Figure 1, we define a multiplier with the variables of unsigned *8* valid bits and unsigned *7* valid bits, and the product's valid bit number is *15* $(8+7=15)$. It is worth noting that, the valid bit here is for the variable type, not the actual value, e.g. the valid bit number of the *short* type variable is *16*, the valid bit number of a *char* type variable is *8*, and the valid bit number of an arbitrary precision type, *(u)int< n >*, is *n*. To show more clearly of the multiplicator units decomposition and parallelization, we present two multiplicator units in parallel, i.e., $0xFE(a_{00})$*$0x7E(b_{00})$ and $0xF5(a_{10})$*$0x75(b_{10})$, and the values of *0xFE* and *0xF5* represent *8* valid bits multiplicators (*uint<8>*), and the values of *0x7E* and *0x75* represent *7* valid bits multiplicators (*uint<7>*). We extend the above two multiplicators' bit numbers to 15 with the vacant value *0*, and the valid bits of their product can be stored within the extended part.

Forth type variables signed, there are some differences. The vacant bit value of signed variables is the value of their sign bit. For the highest multiplicator unit, the vacant bit needs to be extended to the leftmost bit of final multiplicator. In this way, the sign info of lower multiplicators can be well preserved, and the sign info of the final multiplicator is taken in the highest multiplicator unit. Moreover, as shown in the sum operation in Formula (1), it needs to subtract *1* to the multiplicator itself, if the lower multiplicator unit is negative. Except the lowest multiplicator unit, there is no lower multiplicator unit for the sum operation. It is noteworthy that, when the negative multiplicator is in the highest, the subtraction operation of the multiplicator may lead to the final multiplicator sign bit converting, if the value of the multiplicator is the minimum negative value and the vacant part is absent when the input port width of the multiplier is limited, that we will see in Section III. To avoid this instance, it needs to reserve one bit for the sign bit of the highest multiplicator, in the final multiplicator, to keep the signed multiplication correct.

Rule 2 Calculation of Overflow Bits Number: In the parallel multiplication of multiplicator units decomposed in Rule 1, the sum operations of c_τ in Formula (2), may generate overflow bits, which may lead to the contamination in final parallel products. In this rule, we pre-calculate the number of overflow bits and reserve them in the bit number of a multiplicator unit, to make the product of the parallel multiplier separable.

In binary systems, the number of overflow bits can be calculated with a log_2 function with the partial product number from a multiplication operation [19]. Moreover, to prevent the contamination of parallel products, the reserved bit number of the overflow bits needs to be long enough to contain the longest overflow bits generated by the partial products' sum operation in Formula (2). So, we use the maximum bit number of all the overflow bits generated by each sum operation in Formula (2) as the final reserved bit number of overflow bits. And the problem of finding the maximum partial product number of each sum operation in Formula (2), can be described as follows,

Given $i \in [0, n], j \in [0, k], \tau \in [0, n+k], i, j, \tau, n, k \in N$, for each τ, find the solution number of i and j, meeting $(i+j=\tau)$, and obtain the maximum solution number.

We divide the problem into two parts as follows.

• When n equals k, the maximum solution number is the number of values in $[0, n(\text{or } k)]$, i.e. $(n+1)$.

It can be proven with Set method as follows.

We analyze i first. As i is in $[0, n]$, the value number of i is $n+1$. As τ is in $[0, n+k]$,

(1) If τ is in $[0, n)$, $(\tau - i)$ is in $[-n, n)$. As j is in $[0, k]$ and k equals n, the solutions of j meeting $(i+j=\tau)$, are in $[0, n)$. As the value number in $[0, n)$ is less than $n+1$, the possible

solution number of j, is to be less than $n+1$. Consequently, the solution number of i, is less than $n+1$.

(2) If τ is equal to n, $(\tau - i)$ is in $[0, n]$. That is to say, for any i in $[0, n]$, we can find the value that meets $(i + j = \tau)$ in $[0, n]$. As j is in $[0, k]$ and k equals n, the solution number of i and j, meeting $(i + j = \tau)$, is the value number in $[0, n]$, i.e. $(n + 1)$.

(3) If τ is in $(n, n+k]$, $(\tau - i)$ is in $(0, n+k]$. As j is in $[0, k]$ and k is less than $n+k$, the solutions of j meeting $(i+j = \tau)$, are just in $(0, k]$. As k equals n, the value number in $(0, k]$ is less than $n+1$, i.e., the possible solution number of j, is to be less than $n+1$. Consequently, the solution number of i, is less than $n+1$.

And the analysis for j, it is the same. So, when n is equal to k, the maximum solution number is $n+1$.

• When n is not equal to k, the maximum solution number is the less one of the value numbers of i and j.

It can also be proven with Set method, as follows,

Suppose n is less than k. We analyze i first. As i is in $[0, n]$, the value number of i is $n+1$. As τ is in $[0, n+k]$,

(1) If τ is in $[0, n)$, $(\tau - i)$ is in $[-n, n)$. As j is in $[0, k]$ and n is less than k, the solutions of j meeting $(i+j = \tau)$, are just in $[0, n)$. As the value number in $[0, n)$ is less than $n+1$, the possible solution number of j is to be less than $n+1$. Consequently, the solution number of i, is less than $n+1$.

(2) If τ is in $[n, k]$, $(\tau - i)$ is in $[0, k]$. As j is in $[0, k]$, for any i in $[0, n]$, we can find the value that meets $(i + j = \tau)$. So, the solution number of i meeting $(i+j = \tau)$, is the value number of i, i.e. $(n+1)$.

(3) If τ is in $(k, n+k]$, $(\tau - i)$ is in $(k-n, n+k]$. As k is greater than n, $(k-n)$ is greater than 0. Meanwhile, as j is in $[0, k]$ and k is less than $(n+k)$, the solutions of j meeting $(i + j = \tau)$, are just in $(k-n, k]$. As the value number in $(k-n, k]$ is less than $n+1$, the possible solution number of j is to be less than $n+1$. Consequently, the solution number of i, is less than $n+1$.

So, for i, the maximum solution number of i and j meeting $(i+j = \tau)$, is the value number of i, i.e., $(n+1)$.

And for j, as j is in $[0, k]$, i is in $[0, n]$, and n is less than k, the maximum value number of i is less than j. So, the possible solution number of i and j meeting $(i+j = \tau)$, is no more than the value number of i, i.e., $(n+1)$.

Based on the analyses for both i and j, when n is less than k, the maximum solution number of i and j meeting $(i+j = \tau)$, is $(n+1)$.

Similarly, it can be proven that when n is greater than k, the maximum solution number is $(k+1)$.

So, when n is not equal to k, the maximum solution number is the less value number of $[0, n]$ and $[0, k]$.

As the proofs of the two parts above, we obtain the conclusion to the above problem, that the maximum solution number of i and j meeting $(i+j = \tau)$, is the less (including equal) one of the value numbers of i and j.

Therefore, the maximum partial product number of sum operations in Formula (2) is to be the less value number of a_i and b_i, i.e., N_a or N_b. If we use O_{max} to present the maximum number of overflow bits, and use N_{max} to present the maximum partial product number, the maximum number of overflow bits can be calculated as Formula (3).

$$O_{max} = \log_2(N_{max}), \quad N_{max} = \min(N_a, N_b), \tag{3}$$

where N_a and N_b are the maximum parallel numbers of multiplicators a_i and b_i.

As shown in Figure 1, in the parallel multiplication of $0xFE(a_{00})$*$0x7E(b_{00})$ and $0xF5(a_{10})$*$0x75(b_{10})$, the parallel numbers of multiplicator A and B are both 2. And there are four partial products, $0x7D04$, $0x7896$, $0x7416$ and $0x6FF9$. The partial product numbers with the coefficient τ values of 0, 1, 2, are 1, 2, 1, and the maximum is 2, which is the same with the parallel numbers of A and B. With the maximum partial product number 2, we use the log_2 function to calculate the final bit number of overflow bits, and the result is 1. Actually, there are only two partial products, $0x7896$ and $0x7416$, to be calculated with a sum operation and will generate an extra overflow bit of 1-bit.

Rule 3 Parallel Segmentation: Up to now, the multiplicator unit consists of three parts, original valid bits, extended vacant bits, and overflow bits. To parallel the multiplication, we recombine the three parts as one parallel multiplicator unit, and join multiple parallel multiplicator units head and tail, to implement parallel multiplication.

If present the extended vacant bits as S_{ext}, and mark the sign info of the lower multiplicator unit as S_{lower}, the parallel multiplicator units can be calculated as follows.

$$\begin{aligned} a_i &= (S_{ext\_a} a_{i0} - S_{lower\_a}) 2^{i(O_{max}+v_B+v_A)} \\ b_j &= (S_{ext\_b} b_{j0} - S_{lower\_b}) 2^{j(O_{max}+v_A+v_B)}, \end{aligned} \tag{4}$$

where a_i, b_j are parallel multiplicator units, a_{i0}, b_{j0} are the original valid bits, O_{max} is the maximum number of overflow bits, v_A, v_B are the bit numbers of a_{i0} and b_{i0}, i is in $[0, n]$, j is in $[0, k]$, S_{lower} is 0 for the lowest parallel multiplicator unit, and for unsigned variables, S_{ext} and S_{lower} can be ignored.

We use L_p to present the bit number of a parallel multiplicator unit, and the value of L_p is to be $(O_{max} + v_A + v_B)$, which also is the bit number of a parallel product.

We join these parallel multiplicator units head and tail for both A and B, and implement multiplication on a multiplier. Based on the analysis of Formula (2), the output product of the multiplier, is the sum of multiple partial products and/or partial product sums of the multiplicator original valid bits. And we can separate these partial products (sums) without contamination as the bit number has been reserved in the parallel multiplicator unit. That means, we can achieve multiple multiplication operations or multiple groups of multiplication and summation operations on one multiplier in one clock cycle. We call this multiplier as parallel multiplier.

For the parallel multiplier, the parameters of parallel multiplicator include the valid bit numbers, i.e., v_A and v_B, the overflow bits' number, i.e., O_{max}, and the parallel numbers, i.e., N_a and N_b. And the parameters of parallel product are, the parallel product bit number, which is the length of a

parallel multiplicator unit, i.e., L_p, and the parallel number, i.e., N_p. The parameters of the parallel product can be calculated with the parameters of parallel multiplicators, as ($L_p = O_{max} + v_A + v_B$) and ($N_p = N_a + N_b - 1$). With the parallel parameters above, we can implement a parallel multiplier made up of multiplicator paralleling, parallel multiplying and product splitting three steps. And the parallel parameters are employed by the multiplicator paralleling and product splitting steps, to parallel the input multiplicators and split out the parallel products.

As shown in Figure 1, after the parallel segmentation, the bit number of the parallel multiplicator unit is *16*, *16* of $(1 + 7 + 8)$ for *8* valid bits multiplicators (*0xFE* and *0xF5*), and *16* of $(1 + 8 + 7)$ for *7* valid bits multiplicators (*0x7E* and *0x75*). And the parallel numbers of parallel multiplicator units *A* and *B* are both *2*. After the parallel multiplication, the parallel products' number is *3*. As shown in Figure 1, product 1 is the product of (*0xFE*0x7E*), and product 3 is the product of (*0xF5*0x75*). For product 2, it is the result of (*0xFE*0x75*+*0xF5*0x7E*). And we can choose these results according to the application requirements. In the parallel multiplication above, we achieve *2* multiplication operations and a group of *2* multiplication operations and *1* summation operations on one parallel multiplier in one clock cycle. The values of parallel parameters in the example parallel multiplier are, v_A and v_B, are *8* and *7*, O_{max}, is *1*, N_a and N_b are both*2*. And the values of the parallel product parameters are, L_p is *16*, and N_p is *3*.

In particular, for the signed format variable, in the product splitting step, if the lower parallel product is negative, it needs to add *1* to the result of the parallel product split. It is because of the sign bit extension of negative values, and the value of the higher split product actually is the sum with $2^{Lp} - 1$ [20], not 2^{Lp}.

Hence, if we implement parallel multiplier on one DSP slice, the single DSP slice is able to achieve the parallelism of multiple multiplication operations or multiple groups of multiplication and summation operations in one clock cycle. And if we make the parameters of the parallel multiplicators configurable, the precision of DSP slice may support being configured dynamically during run-time. However, as the bit numbers of DSP slice input ports are limited, it is not possible to implement any parallel multiplier on one DSP slice. Moreover, for the limited bit numbers of the DSP slice input ports, there will be multiple groups of the parallel parameters available, and in some of them, the numbers of the multiplicator valid bits are very small, that will make the DSP slice inefficient. So, in next section, we will discuss the implementation approach for the optimal parallel multiplier on a single DSP slice.

III. PARALLEL MULTIPLIER ON A SINGLE DSP SLICE

A typical function of DSP slice in FPGA, is the multiplication function (multiplier), and the function is mainly improved by the precision extending, e.g. DSP48A in Xilinx 6 Series FPGAs contains an 18 × 18-bit multiplier, and in 7 Series FPGAs, DSP48E is extended to 25 × 18-bit. Generally, one FPGA architecture only has one type of the DSP slice resource, once we know the target FPGA, we can get the DSP slice specifications (precision) from the supplier. Usually in the acceleration design of a function or algorithm on embedded FPGAs, the DSP slice resource is employed for the acceleration of massive multiplication operations, and the accelerated function or algorithm is implemented as a functionally independent IP core, through detailed parallel analyses and resource optimization. In the integrated application systems, once the IP core is loaded on FPGA, it is hard to be reconfigured during run-time.

Algorithm 1: PMPSA $f(x_1, x_2)$

Input: valid bit numbers of the two multiplicators x_i
Output: optimal results (x_i, N_i, O)
Begin
Init: $O = 0$, $N_i = 1$;
$Label_s$ = signed ? *1*: *0*;
Search:
 # step 1: Decomposition of multiplicators units
Compute multiplicator unit bit number $L_u = x_1 + x_2$;
for all K_i **meet** $L_u * K_i < P_i$ ***do***
  Compute higher remainder bits number
  $highbits_i = P_i - L_u^* K_i$;
  if $highbits_i >= x_i$ **then**
    Mark N_i with a label $Label_i = 1$;
    Set $N_i = K_i + 1$;
  else Set $N_i = K_i$;
  # step 2: Calculation of overflow bits number
  Set $N_{max} =$ **Min**(N_i);
  Compute overflow bits number $O = log_2(N_{max})$ and **Ceil** to an integer
  # step 3: Parallel Segmentation
  Compute total parallel multiplicator units bit number
  Set $L_p = L_u + O$;
  $T_i = L_p^* (N_i - Label_i) + Label_i^* bx_i + Label_s$;
  # step 4: Results validation and optimization
  if $T_i \leq P_i$ **then**
    Optimize: **Output** (x_i, N_i, O) to PMPOM
Output: **Output** the optimal (x_i, N_i, O) from PMPOM
end

In this section, we propose a searching algorithm for optimizing parallel multiplier parameters, i.e. PMPSA, shown as Algorithm 1, to search the optimal parallel method for a given type DSP slice, based on three rules proposed in Section II, to improve the performance of DSP slice. Furthermore, the proposed parallel method for DSP multiplier can be reconfigured during run-time. If the PMPSA is run on general CPUs, the searching results of PMPSA are to be hardened as predefined parameters over a DSP multiplier for the parallelization of massive multiplication operations on FPGA. And for multiple searching results of one or more applications

with the same function, it is able to integrate multiple these parallel parameters over the same DSP multiplier as tuning logic. For the tuning work, it is possible to provide a select parameter to choose corresponding parallel parameters when call the DSP multiplier. We call this configuration method as the partially dynamical configuration method (PD-PMSDS). One advantage of PD-PMSDS is that, there is no time consumption of the PMPSA algorithm in the running system.

The PMPSA can also be implemented on embedded systems independently, serving as a configuring module, to generate parallel parameters dynamically and configure the parallel DSP multiplier in other IP cores. In this case, the parallel parameters in the tuning logic over the DSP multiplier need to be set as variables. With an adjustable bound control of the parallel parameters and an appropriate interface design in the target IP core, the precision of parallel DSP multiplier can also be reconfigured during run-time. We call this configuration method as a totally dynamic configuration method (TD-PMSDS).

For the details of PMPSA, there are two main stages in PMPSA, one is parallel parameters searching, which is made up of three steps based on the three rules defined in section II, the other is parameters optimizing. Besides, the algorithm initialization and output, are in the start and end of the algorithm. And we discuss PMPSA in two separate subsections referring to the main stages as follows.

The major variables used in PMPSA are denoted in Table 1.

TABLE 1. Notations in the PMSDS approach.

Notations	Specification	Notations	Specification
P_i	bit numbers of the input ports of the DSP slice, $i=1,2$	Z	objective function value
L_u	bit number of the multiplicator unit	ψ	acceleration of PMSDS
N_i	parallel numbers of the multiplicators, $i=1,2$	θ	valid bits throughput of PMSDS
O	the number of overflow bits	ω	precision of PMSDS
L_p	bit number of the parallel multiplicator unit	C_i	valid bit rates of PMSDS, $i=1,2$
T_i	total bit numbers of parallel multiplicator units, $i=1,2$	x_i	valid bit number of input multiplicators, $i=1,2$

A. PARALLEL PARAMETERS SEARCHING

As discussed in Section II, to implement a parallel multiplier, we need to obtain the parameters of multiplicators' valid bit numbers, parallel numbers, and the maximum overflow bits' number. And the parallel products' parameters can be calculated with the above three parameters. Generally, in the acceleration of an algorithm on FPGA, the variable types are pre-defined in the algorithm, therefore the multiplicators' valid bit numbers are known. So, to implement a parallel multiplier on a single DSP slice for an accelerated algorithm, we just need to obtain the parameters of the parallel numbers and the maximum number of overflow bits. And in PMPSA, we define the parameters of multiplicators' valid bit numbers as input variables (x_i). Moreover, we define the bit numbers of the DSP slice input ports (P_i) as known values, as the target FPGA is usually clearly specified in the algorithm accelerating design. For a pair of inputs x_i, the searching results may be multi-groups, and we provide a parallel multiplier parameters optimizing model, i.e., PMPOM, that we will discuss in next section, as a part of PMPSA to find the optimal combination of the parallel parameters, on one of the specified DSP multiplier. And the output of PMPSA is the optimal combination of multiplicators valid bit numbers (x_i), parallel numbers (N_i), and the overflow bits' number (O), i.e., PMSDS parameters, which are all essential information for the multiplicator paralleling and product splitting of the parallel DSP multiplier [21].

In the beginning, PMPSA needs to initialize some variables, including O and N_i. And the multiplicator variable format, signed or unsigned, also need be specified. We define a variable ($Label_s$) to save the multiplicator variable format, and set $Label_s = 1$ for the signed format variable, as shown in Algorithm 1, which is to reserve one bit for the highest multiplicator's sign bit.

The core part of PMPSA is a double-loop for the parameters searching of the DSP multiplier two input ports. Before searching work, we use Rule 1 in Section II, to calculate the multiplicator unit's bit number (L_u), i.e., $L_u = x_1 + x_2$. And we define K_1 and K_2 as the loop variables of the double-loop, and the values of K_1 and K_2 are the rough values of the parallel numbers. With the values of K_1 and K_2, we parallel L_u for each input ports of the DSP multiplier, and the total multiplicator units' bit number of each port can be calculated as $L_u^* K_i$. The total multiplicator units' bit numbers here, are the searching bound equations of the double-loop, for each loop, the upper bound is P_i, and the lower bound is the value when K_i is 1.

With the rough values of K_1 and K_2, in the loop body, we begin to calculate the exact values of the parallel numbers. In some cases, after the parallelization of a multiplicator unit with K_i, there may be some remainder bits left in the higher bits of each DSP input port, and those bits are not long enough to contain all the multiplicator unit bits, yet, are possible to be stuffed into a valid bits' part. As this happens just in the highest bits of the final multiplicator, i.e. there is no valid bit contamination ahead, it is still ok for another valid bits' part to be stuffed into the final multiplicator without the vacant bits. To avoid losing these possible parallel multiplicators, we calculate the numbers of the leftmost remaining bits with L_u, K_i and P_i, as shown in Algorithm 1, and mark these particular situations with a pair of labels, i.e., $Label_{i=1,2}$. If an extra valid bits' part of x_i is able to be stuffed into the head of the final multiplicator, we set $Label_i$ as true, and set N_i as $K_i + 1$, that will increase one multiplicator in the parallel multiplication. And in other cases, we set N_i as K_i. These are the first step of PMPSA, and we have obtained a parameters group of the valid bit numbers (x_i), and the parallel numbers (N_i).

In Step 2, we go on to calculate the maximum overflow bits' number (O) with the parameters got in step 1. As discussed in Rule 2 of Section II, the maximum partial product number is the less one of the two multiplicators' par-

allel numbers. As N_1 and N_2 are the parallel numbers of the two multiplicators, the less (or equal) one of N_1 and N_2 is to be the maximum partial product number (N_{max}). With N_{max}, we use the log_2 function to calculate the value of O, as shown in Algorithm 1. In particular, the result of O may not be an integer value, and it needs to be ceiled to the infinity to avoid the parallel products in the final product overflowing. These are Step 2, and we have obtained the maximum number of overflow bits (O).

With the multiplicator unit bit number (L_u), and the maximum overflow bits' number (O) obtained above, in the step 3, we add them together as the parallel multiplicator unit bit number (L_p, here we just obtain the bit number, and the implementation step of Formula (4) in Section II, we will discuss in Section IV), then calculate the total bit number of the parallel multiplicator units (T_i). T_i is the sum of three parts, as shown in Algorithm 1. The first part, is the total bit number of parallel multiplicator units, which is the product of L_p and its parallel number, here, the additional valid x_i, is not included in the parallel number, as it is absent with the vacant bits. We subtract *1* to the parallel number of L_i, if the value of $Label_i$ is *1*, and deal with the additional x_i in the following part. The parallel multiplicator unit part is an essential part of T_i, which means the parallel multiplicator unit parallelization. The following two parts are complements for special situations. The second part is the additional x_i, and it is the product of $Label_i$ and x_i, that maybe is not existing in the final multiplicator, referring to $Label_i$. The third part is the reserved one sign bit for signed format variables, and it is *0* for unsigned variables.

After the parallelization of parallel multiplicator units in the step 3, T_i may be larger than P_i, as the inserted bit number, O. And in step 4, we first verify T_i with P_i. If T_i is less than or equal to P_i, for both $i = 1, 2$, the parameters combination of x_i, N_i, and O, is a feasible PMSDS solution of the given type DSP slice, and we output the feasible parameters combination of (x_i, N_i, O) to PMPOM for optimization. Then, go on searching with $N_i + 1$.

B. PMSDS PARAMETERS OPTIMIZING

In last section, we have got the PMSDS solutions of the given type DSP slice. However, the solutions are multi-groups. And PMPOM is to find the optimal solution, according to the acceleration (ψ), valid bits throughput (θ), and precision (ω) of the given DSP multiplier, based on the Integer Programming Optimization Theory [22] as follows.

$$\text{Maximize } Z = \psi^*\theta^*\omega$$

$$\begin{cases} \psi = 1.5^* \prod_{i=1}^{2} N_i - 0.5^*(\sum_{i=1}^{2} N_i - 1) \\ \theta = \sum_{i=1}^{2}(x_i^* N_i)/(P_1 + P_2) + 1 \end{cases}$$

$$\omega = \prod_{i=1}^{2}(L_p/P_i)$$

Subject to

$$\begin{aligned} & x_i/L_p \geq C_i, \quad i = 1, 2; \\ & x_1 = 1, 2, \ldots, \quad P_1; \\ & x_2 = 1, 2, \ldots, \quad P_2., \end{aligned} \tag{5}$$

where the objective function value Z presents the final trade-off value of the parallel DSP multiplier, and we want to make the value of Z to the maximum. Z is the product of three subitems, the acceleration of parallel DSP multiplier (ψ), the valid bits throughput of parallel DSP multiplier (θ), and the precision of parallel DSP multiplier (ω). In details, for ψ, we define the acceleration of a parallel multiplier with the parallel numbers of multiplication and addition operations. One multiplication operation contributes the multiplier acceleration of *1*, and one addition operation contributes *0.5*. These are because for a pipelined DSP slice, one multiplication operation may need one clock cycle, and a followed addition operation may need half clock cycle totally [22]. The number of multiplication operations can be computed as $N_1^*N_2$, and the number of the addition operations can be computed with the numbers of parallel multiplicators, as $N_1^*N_2 - (N_1 + N_2) + 1$, just excluding the partial product numbers of the last one in each row and the last row of the partial product array, where there is no another available addend. Then Add these two operation numbers together, and merge similar items, the result is the expression of ψ. For θ, it is got from the quotient of the total valid bit number of all parallel multiplicators and the total bit number of the two input ports of DSP multiplier, in one parallel multiplication operation. To avoid the value of θ disappearing, we define the base rate of θ as *1*. For ω, it is the product of a pair of ratios, i.e., the bit number of the parallel multiplicator unit to the bit number of each port of the DSP multiplier.

Moreover, C_i is predefined searching coefficient, which is the minimum valid bit rate to the parallel multiplicator unit, to throw out the parameter combinations, of which the multiplicators' valid bit numbers are too small. And for different i, the value of C_i can be different. For x_1 and x_2, one of them can be specified as a predefined value, e.g. *8*-bit for Byte, for special application requirements. The bit numbers of the DSP slice two input ports, P_1 and P_2, can be different, according to the actual DSP slice type in target FPGA.

PMPOM acts as a part of PMPSA, and it is employed by PMPSA to optimize output PMSDS parameters. For each PMSDS parameter combination output from PMPSA, it needs to be sent to PMPOM for comparison, and PMPOM saves the optimal PMSDS parameters combination. Finally, PMPSA outputs the optimal PMSDS parameters combination, as the final searching result.

Above all, we have proposed the algorithm of PMPSA and the optimization model of PMPOM. An analysis of the operations in both PMPSA and PMPOM shows that the computational complexity is $O(N_1^*N_2)$, where N_1 and N_2 are the variables related with the predefined bit numbers of the DSP

slice two input ports (P_i), and the input multiplicators valid bit numbers (x_i), i.e. $N_i \in [1, P_i/(x_1 + x_2)], i = 1, 2$.

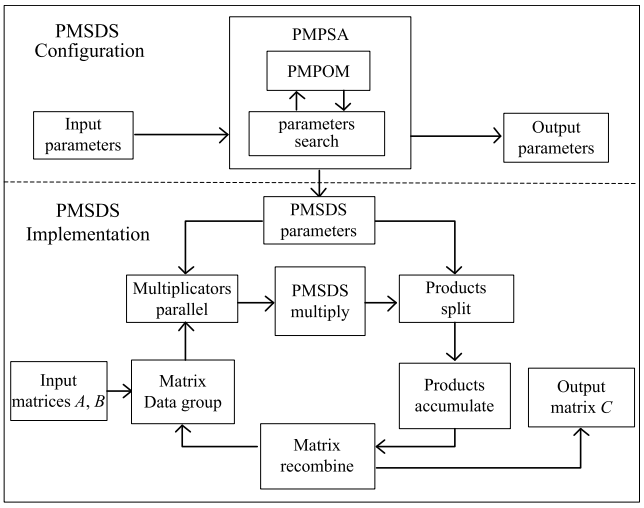
IV. EXPERIMENTS

In this section, we present the experiments of the proposed PMSDS method. Firstly, the effectiveness of PMPSA is evaluated. We choose three FPGA platforms from the product families of Xilinx, i.e. 6 Series FPGAs, 7 Series FPGAs, and Ultrascale FPGAs, as the target platforms, and search PMSDS solutions for the DSP slices in target FPGA platforms, under different searching configuration. And to test the practicability of PMPSA, mainly for the practicability analyses of PMSDS dynamic configuration methods, we implement PMPSA on different environments, e.g., different CPUs and FPGA, and analyze the time and resource consumption of PMPSA. we also design a prototype of the parallel DSP multiplier in the application of matrix multiplication. Based on the prototype above, we implement multiple PMSDS based matrix multiplication algorithms with different PMSDS parameters, and use the Xilinx SDSoc design tool [23] to transform the C/C++ language based PMSDS matrix multiplication algorithms to embedded systems, to test the performance of PMSDS method on actual FPGAs. Further, we compare the results with the typical methods and the state-of-the-art in FPGA resource optimization and matrix multiplication optimization fields to demonstrate the efficiency and performance improvements of the proposed method. Moreover, to test the flexibility of the proposed PMSDS method, we implement the PMSDS dynamic configuration methods of both PD-PMSDS and TD-PMSDS on embedded FPGA systems, referring to the practicability analyses mentioned above, and we will discuss the test results of the dynamic configuration methods in the last subsection.

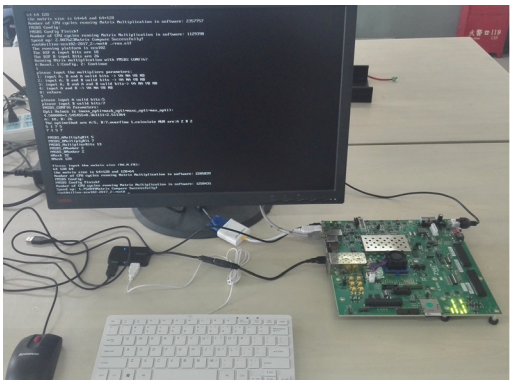
TABLE 2. DSP slice types.

DSP Type	Maximum multiplier bits	FPGA family
DSP48A	18*18	6 Series
DSP48E1	18*25	7 Series
DSP48E2	18*27	Ultrascale

The precision of the DSP multipliers in all selected FPGA platforms is listed in Table 2. The prototype of the PMSDS based parallel DSP multiplier is designed based on Xilinx Zynq serial FPGA platforms [24], and the prototype is shown as Figure 2. The prototype in Figure 2, is made up of two parts, PMSDS configuration and PMSDS implementation. The PMSDS configuration part is the implementation of PMPSA. The input parameters of PMPSA are the multiplicators valid bit numbers (x_i). For the bit numbers of the DSP multiplier input ports (P_i) and multiplicator variable format, they can be predefined in a configuration file. PMPOM is integrated in PMPSA to compare the searching results of PMSDS parameters and output the optimal parameters combination. The PMSDS configuration part can be implemented



a) the prototype of PMSDS.



b) the platform of PMSDS.

FIGURE 2. The prototype and platform of PMSDS. The platform of PMSDS is based on Xilinx Zynq FPGA platform. And Xilinx SDSoc tool is used for the algorithm integration.

on both CPUs and FPGAs. If run on CPUs, PMPSA is to provide PMSDS parameters for parallel DSP multiplier design on FPGAs. And if integrated on embedded FPGA systems, PMPSA is an independent configuration module, which is called by other applications to generate PMSDS parameters dynamically.

The PMSDS implementation part in the bottom of Figure 2, is to use the parallel DSP multiplier implementing the matrix multiplication algorithm. The implementation of the PMSDS based matrix multiplication algorithm is partitioned as six steps, i.e., matrix data grouping, multiplicator paralleling, PMSDS multiplication, product splitting, products accumulating, and matrix recombining. The data input of the matrix multiplication algorithm is in matrix data grouping step. And the PMSDS parameters are predefined (PD-PMSDS) or input from the PMSDS configuration part above (TD-PMSDS). Moreover, the bit numbers of the DSP slice input ports (P_i) are also predefined. The matrix data grouping step obtains the matrix data from the input interfaces and stores them locally as the same matrix arrays. And the three following steps of multiplicator paralleling, PMSDS multiplication,

and product splitting, are the implementation steps of the parallel DSP multiplier. The multiplicator paralleling step employs the parallel numbers (N_i) in the PMSDS parameters, to read N_i elements that to be calculated referring to the calculation rule of matrix multiplication, from the two candidate matrix arrays. Then recombines them as parallel multiplicator units with the parameters of x_i, and O, referring to Formula (4) in Section II. Finally, parallel the parallel multiplicator units to the length of P_i, and output them to the PMSDS multiplication step. The PMSDS multiplication step is the implementation step of the parallel multiplication on the specified DSP multiplier. The bit number of each input multiplicator in PMSDS multiplication step, is fixed as P_i, and the bit number of the output product is fixed as (P_1+P_2). The product in PMSDS multiplication step, is sent to the product splitting step. The product splitting step precomputes the parallel product bit number and parallel number with the PMSDS parameters, based on Rule 3 in Section II, and employs them to split out the parallel products from the product of PMSDS multiplication step. The following step of products accumulating, implements the accumulation of each split parallel products referring to the matrix multiplication rule, and the matrix recombining step stores the accumulation results to corresponding positions of the output matrix array. Then go on calculating with the next input data groups.

The PMSDS implementation part in Figure 2, is implemented as an independent IP core on FPGA, and for PD-PMSDS, the PMSDS configuration part is absent in the integration system. The input and output interfaces of the IP core are implemented with the axis protocol [25], and connected with the DMA. The DMA is used for directly large size data access between the DDR and the IP core, to increase the data transferring speed.

A. BENCHMARK AND COMPETING METHODS

For clearer comparison of our PMSDS method, we collect some typical competing methods, including the traditional method [26], additional memory method [27], add-tree method [28], and full unroll method [29], the details are specified as below.

- **Traditional method** [26] which is implemented in the original matrix multiplication rule of one row element multiplies one column element.
- **Additional memory method** [27] which employs additional memory in FPGAs to improve the data access speed and the matrix multiplication throughput. This method requires larger internal memory.
- **Add-tree method** [28] which is to use addition, subtraction and bit shift operations to implement the multiplication operations of the matrix multiplication. This method does not consume DSP slice resource in FPGA, however, the logic resources, e.g. LUTs and REGs, may cost more.
- **Full unroll method** [29] which employs the logic resources in FPGA to fully unroll the inner loop of the matrix multiplication, and improve the matrix multiplication throughput. However, the resource utilization and power consumption may be greatly increased.

B. OPTIMIZATION RESULTS AND ALGORITHM COSTS

The goal of the proposed PMSDS method is to find the best benefit between the multiplicator valid bit number and the parallel number for a given type DSP multiplier. We cannot implement long valid bit multiplicators with larger parallel number on one DSP multiplier at the same time, as the limited bit numbers of the DSP slice input ports, i.e. if the parallel number is larger, the multiplicators' valid bit number will be smaller, and vice versa. We employ PMPSA to search the best tradeoff between the valid bit number and the parallel number for some commonly used DSP slice types, and the searching results are shown in Table 3.

TABLE 3. Results of PMPSA.

DSP Type	Model	Bits and numbers			Optimization variables			
		$x_1 * x_2$	O	$N_1 * N_2$	ψ	θ	ω	Z
DSP48A	DMax	18*18	0	1*1(1)	1	2	1	2
	uMax	5*6	1	2*2(4)	4.5	1.61	0.44	3.22
	sMax	4*6	1	2*2(4)	4.5	1.58	0.37	2.66
	DByte	8*8	0	1*1(1)	1	1.44	0.20	0.28
	uByte1	5*8	0	1*2(2)	2	1.50	0.32	0.96
	sByte1	4*8	0	1*2(2)	2	1.47	0.30	0.87
	uByte2	8*8	0	1*1(1)	1	1.44	0.20	0.28
	sByte2	8*8	0	1*1(1)	1	1.47	0.20	0.29
DSP48E1	DMax	18*25	0	1*1(1)	1	2	1	2
	uMax	5*7	1	2*2(4)	4.5	1.56	0.38	2.63
	sMax	5*4	1	2*3(6)	7	1.53	0.22	2.38
	DByte	8*8	0	1*1(1)	1	1.37	0.14	0.19
	uByte1	9*8	0	1*2(2)	2	1.58	0.34	1.08
	sByte1	8*8	0	1*2(2)	2	1.58	0.28	0.90
	uByte2	8*8	0	1*2(2)	2	1.56	0.28	0.89
	sByte2	8*8	0	1*2(2)	2	1.58	0.28	0.90
DSP48E2	DMax	18*27	0	1*1(1)	1	2	1	2
	uMax	5*5	1	2*3(6)	7	1.56	0.25	2.71
	sMax	6*4	1	2*3(6)	7	1.56	0.25	2.71
	DByte	8*8	0	1*1(1)	1	1.35	0.13	0.17
	uByte1	11*8	0	1*2(2)	2	1.60	0.43	1.38
	sByte1	10*8	0	1*2(2)	2	1.60	0.37	1.19
	uByte2	8*8	0	1*2(2)	2	1.53	0.26	0.81
	sByte2	8*8	0	1*2(2)	2	1.55	0.26	0.82

There are three DSP slice types in Table 3, which are chosen from the FPGA product families of Xilinx, the same as Table 2. And in each DSP slice types, there are three test models, one is the optimal model, named Max. The second is the model of one multiplicator valid bit number specified as *8*, called Byte1. The last one is the model of both two multiplicator valid bit numbers are specified as *8*, called Byte2. And in each test model, the start letter of '*D*', means the default type, i.e., without the PMSDS optimization (called non-PMSDS, below). And the following lines are the results of PMPSA under unsigned (u) and signed (s) configuration types. The following columns named Bits and numbers, are the outputs of PMPSA, where x_1 and x_2 are the multiplicators'

valid bit numbers, O is the overflow bits' number, N_1 and N_2 are the multiplicators' parallel numbers. The columns named Optimization variables, are the variables of PMPOM, where ψ is the acceleration of the parallel DSP multiplier, θ is the valid bits throughput of the parallel DSP multiplier, ω is the multiplier' precision, and Z is the final tradeoff value, the same as Section III. The coefficient values of C_i in PMPOM are set as 1/3 for all DSP multipliers.

As seen in table 3, the value of ψ in the DMax model is *1*. It is because there is only one multiplication operation in the non-PMSDS based DSP multiplier. And the tradeoff value in the DMax model is *2*. It is because in DMax, we set the two multiplicators' valid bit numbers to the maximum, i.e., P_i (as it is the Max model), and the calculated values of θ and ω in DMax are both *1,* referring to Formula (5) in Section III. Meanwhile, as we set the base rate of θ as*1*, the final value of θ is $2(1+1)$. And the final tradeoff value Z is *2(1*2*1)*. The DMax model is the baseline for the PMSDS based Max model. As seen in table 3, in PMSDS based Max models, uMax and sMax, the acceleration and tradeoff values are greater than the baseline for all the three DSP slice types. Among them, the maximum acceleration is *7*, and the minimum is *4.5*, and the tradeoff values are from *2.38* to *3.22*. E.g., in uMax model, one DSP48E1 slice via the PMSDS method is able to implement a parallel multiplication of four multiplicators with the valid bit numbers of *5* and *7*, in one clock cycle. And the tradeoff value of the parallel multiplier is *2.63*, based on the three optimization values, i.e., the value of acceleration ψ, which is*4.5* of *4* multiplication operations and *1* addition operation, the value of valid bits throughput θ, which is *1.56*, and the value of multipliers' precision ω, which is *0.38*. Combining the values of ψ and Z, it can be known that the performance and efficiency of the DSP slice are improved in the Max model with the proposed PMSDS method.

For the Byte model, the baseline is DByte, and it is the result of both two multiplicators' valid bit numbers are set as *8*. And we can see in table 3, the tradeoff value of the PMSDS based Byte1 method is obvious greater than the non-PMSDS based method. It is because the multiplicator valid bit-width (8-bit), is at least 10-bit less than the minimum DSP slice port, that is even longer than the multiplicator valid bit-width itself. Correspondingly, in the DByte model, the values of θ and ω are small, especially for ω, whose values are less than 0.15 for all the three DSP slice types. Subsequently, the tradeoff value Z is smaller. Actually, for the signed multiplication operation of the DByte model, all the higher bits of the DSP input ports are filled with value 0s or 1s, which carry only 1-bit of the sign info. That may be inefficient. However, the proposed PMSDS method is able to make up that in some extend. E.g., the DSP48E1 slice is able to deploy a parallel multiplication of three multiplicators with one signed 8-bit and two signed 8-bit at the same clock cycle. In this way, the higher bits of one input port of the DSP multiplier are able to achieve another 8-bit multiplication operation, and it is the reason that the value of ψ in the PMSDS based Byte1 model, is 2 times than the DByte model. And for unsigned variables, the one 8-bit multiplicator is able to extend to 9-bit, and the tradeoff value is about 0.9 greater than the non-PMSDS based DByte model. And for the Byte2 model, the result is similar with the Byte1 model, except for DSP48A, there is no optimization with the PMSDS method. It is because the bit numbers of the input ports of DSP48A are smaller than DSP48E1 and DSP48E2, and they are not long enough to parallel two Byte multiplicators at the same time.

And for the differences between unsigned and signed configuration types, the reserved one sign bit for the highest multiplicator lead to the DSP multiplier precision lose one bit, that can be seen through the value of θ, and it influences the searching results of the parameters of multiplicators' valid bit numbers and parallel numbers. E.g. for the PMSDS based uMax model of DSP48E1, the searching results are 5*7-bit (x_1*x_2) of 2*2 ($N_1^*N_2$). However, for sMax model, the searching results are 5*4-bit of 2*3, and the value of θ is 0.03 less than uMax. And for the sByte1 model of DSP48E2, the precision of the multiplier is 10*8-bit, which is one bit less than the uByte1 model, i.e. 11*8-bit. Moreover, in the Byte2 model, the value of θ under signed configuration is higher than the unsigned. It is because the higher extended bits of the sign bit are treated as one valid bit for the throughput, as they influence the multiplication type of the DSP multiplier.

For the time consumption test of the PMSDS configuration part in Figure 2, we implement PMPSA on Intel CPU (i5-3230M), ARM CPU (Cortex A9), and FPGA (7 Series), respectively. We use DSP48E1 as the tested DSP slice type, and the run times under different test models are shown in table 4.

TABLE 4. Time consumption of PMPSA.

CPU type	Time/us		
	Max	x_1 or x_2	x_1 and x_2
Intel	295	48	30
ARM	1618	202	121
FPGA	1466	162	110

As shown in table 4, the Max column is the searching time of the global optimum solution with no multiplicator valid bit number predefined, the column named x_1 or x_2 is the searching time of one multiplicator valid bit number, x_1 or x_2, predefined, and the column named x_1 and x_2 is the run time of both x_1 and x_2 are predefined. The results in table 4 show that the fastest running environment is Intel CPU, the run times of which are all less than 300us. And with two known variables' valid bit numbers of an algorithm, the run times are just 30us, shown as x_1 and x_2 of Intel in table 4. As the demand for real-time in this applied method is not very strong, the run time of PMPSA on PC (Intel CPU), is not to be a burden.

The run times of PMPSA on both ARM CPU and FPGA, may be a little more, if PMPSA is integrated as a real-time configuration module on embedded systems. The run times of Max on ARM CPU and FPGA are about 1.6ms and 1.5ms, which are about the same with the run times

of a 25*25 size matrix multiplication on ARM CPU. The run times of x_1 and x_2 on both ARM CPU and FPGA are much faster, i.e., 121us and 110us, which are much more practical. Maybe those are acceptable for some low real-time systems. However, on FPGA, the resource consumption is very high. It costs 42 DSP slices (19%), as the float operations in PMPOM, and more than 50% LUT resource of the Zynq platform, that may lead to the DSP and logic resources lose more than gain. On ARM CPU, the FPGA resources need not be cared about.

Through the analyses above, we know that the optimization results of PMPSA are able to improve the DSP multiplier's performance and efficiency. And for the running environment, PMPSA is suitable to be run on PC to search parallel methods for the DSP multiplier in FPGA. Meanwhile, PMPSA is also possible to be employed as a configuration module on the hardcore CPU in FPGAs to provide PMSDS parameters and configure the PMSDS based hardware IP cores dynamically, in some low real-time systems. However, PMPSA is not economic to be integrated on FPGA as a real-time configuration module, as the high resource consumption.

C. MULTIPLIER THROUGHPUT AND PERFORMANCE

We have analyzed the PMPSA optimization results in last section, however, there may be more factors that affect the performance of an accelerated algorithm on FPGAs. In following sections, we will show the performance and efficiency of the optimization results on actual embedded FPGA systems.

To show the multiplier throughput and performance of the PMSDS method, we implement the matrix multiplication algorithm with different methods on embedded FPGA. We choose the traditional method as the baseline, and compare our PMSDS based methods with other notable methods, i.e. additional memory method, add-tree method, and full unroll method, under the same system configuration. And we choose three PMSDS parameters from the searching results of PMPSA in Table 3, and implement them with Xilinx HLS tool respectively, based on the prototype in Figure 2. It is noteworthy that we integrate only one combination of the PMSDS parameters over the DSP multiplier here, and there is no tuning cost. In the implementation of the parallel DSP multipliers, there may be some changes to the searching results in Table 3, as the calculation rule of matrix multiplication. The changed PMSDS parameters are shown in table 5.

The Name column in table 5, lists the methods tested in the experiments. And the following two columns are the valid bit numbers and operation numbers in one clock cycle of each method. The fourth column is the valid parallel product numbers for the matrix multiplication application. The last column is the throughput of the multiplier, and we define the throughput as the valid bit number sum of all multiplicators in one multiplication operation on one multiplier.

In table 5, the Double method, is to implement the parallel multiplication of one element of one row in the first matrix

TABLE 5. Throughput of the multipliers.

Name	Valid bit number	operation number	valid product number	throughput
Double	8*8	2	2	24
Quad	5*6	2(1)*	1	22
Triple	5*5	3	3	20
tradition	8*8	1	1	16
Add-memory	8*8	1	1	16
Add-tree	8*8	1	1	16
Full-unroll	8*8	1	1	16

* 1 is for addition operation

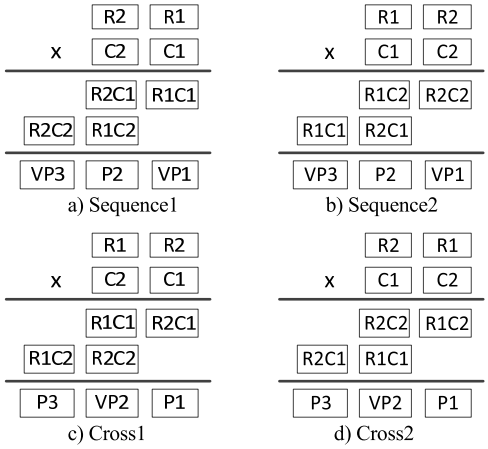


FIGURE 3. Placement ways of multiplicators in Quad.

and two elements in two cols of the second matrix, in one clock cycle. The multiplication operation number is *2*, so is the parallel product number. And both the *2* parallel products are valid, which are to be stored to the two corresponding cols of the output matrix, after the accumulation step. The Quad method is to implement the multiplication of two elements of one row in the first matrix and two elements of one col in the second matrix. In Quad, the number of valid operations and valid parallel products are different in different placing ways of the elements. There are four placing ways for the four elements as shown in Figure 3, and they can be divided into two groups, one is to place the elements in sequence order, named Sequence, and the other is to place in cross order, named Cross. E.g. in Figure 3, *R1* and *R2*, are the elements from the same row of the first matrix, and the col number of *R1* is less than *R2*. *C1* and *C2* are the elements from the same col of the second matrix, and the row number of *C1* is less than *C2*. In traditional matrix multiplication, we calculate *R1*C1* first, then calculate *R2*C2*. Therefore, we define the calculation order numbers of *R1* and *C1* are the same, so are *R2* and *C2*, and the order numbers of *R1* and *C1* are less than *R2* and *C2*. And Sequence is to put the same order elements in the same positions of the final multiplicators, as shown in Figure 3 a) and b). And Cross is to put the same order elements in the cross positions of the final multiplicators as shown in Figure 3 c) and d). The valid parallel product numbers and positions for the matrix multiplication are different

in Sequence and Cross. The valid parallel product numbers in Sequence1 and Sequence2 are *2*, and the positions of them are *VP1* and *VP3,* which are the same in Sequence1 and Sequence2, as shown in Figure 3 a) and b). The parallel products in Sequence1 and Sequence2 are partial products, we need to add the parallel products together, before or in the following accumulation step, for matrix multiplication. And in Cross1 and Cross2, the valid parallel product numbers are both *1*, and the positions of the valid parallel products are the same, i.e. *VP2*. In Cross1 and Cross2, the parallel products are partial product sums, and there need not be an addition operation for the parallel products themselves, as it is included in the parallel multiplication. And we use Cross1 as the implementation method of Quad, which is able to achieve *2* multiplication operations and *1* addition operation in one clock cycle, as shown in table 5. For the parallel numbers of 2*3 in Table 3, we change it to 1*3, as it is hard to find another multiplicator whose calculation order number is the same with the multiplicators in Quad, and it is to achieve the parallel multiplication of one row element and three cols' elements in one clock cycle, called as Triple. The parallel product number in Triple is *3*, and all of them are valid for the matrix multiplication application, and to be stored to the three corresponding cols of the output matrix, after the following accumulation operation.

We set the base valid bit number of the matrix data is *8*, except Quad and Triple methods, which are changed to 5*6-bit and 5*5-bit. For the non-PMSDS based method, the multiplier valid bits throughput is 16-bit. And for Double, it is 24-bit, which is 8-bit more than the non-PMSDS based method. For Quad and Triple, they are 22-bit and 20-bit, which are 6-bit and 4-bit more than the non-PMSDS based method. Moreover, all matrix data here are signed format.

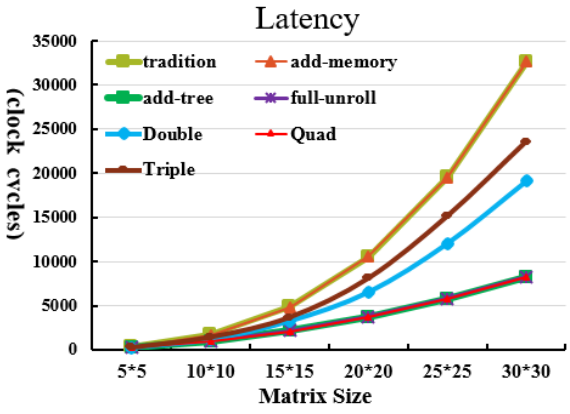


FIGURE 4. Performance of seven tested methods on FPGA. In particular, as the newly version HLS tool does not support DSP48A of 6 series FPGA family, we use DSP48E1 in 7 series FPGA with 7 higher bits disable instead. And the results are the synthesis results of HLS tool.

For the performance comparison of PMSDS based methods, we tested the matrix multiplication algorithms with multiple fixed size matrix arrays from 5*5 to 30*30. The latency of all tested methods are shown as Figure 4.

As it is shown in Figure 4, the latency of our PMSDS based methods, i.e., Double, Quad, and Triple, is obvious less than the traditional and add-memory methods. The Triple method is slower than Double, that is because the limitation of the BRAM ports, which are just two, and the access of triple data at the same BRAM may need more clock cycles. The add-tree and full-unroll methods are faster than Double and Quad. It is because they employ more calculating resources to make the multiplication operations in the matrix multiplication algorithm compute totally in parallel, which we will see in next subsection. However, the latency of our Quad method is still faster than the add-tree and full-unroll methods (one clock cycle faster), that is because of the acceleration of the parallel DSP multiplier. These are the performance improvements of the PMSDS method.

D. RESOURCE AND POWER CONSUMPTION

Resource utilization and power consumption are very important in embedded systems. In this subsection, we display the resource and power consumption of our PMSDS based matrix multiplication algorithms on specified FPGAs, and compare them with other methods. The results are shown as Figure 5.

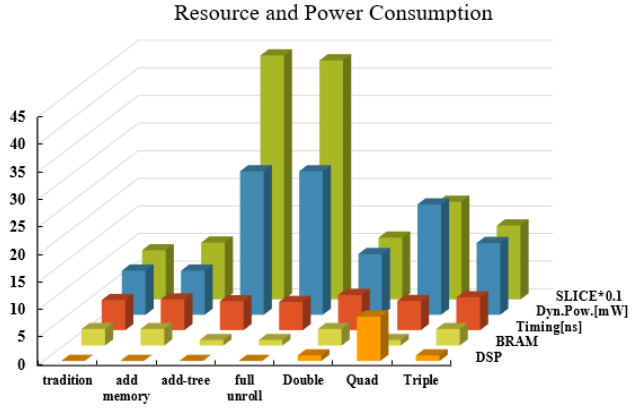


FIGURE 5. Resource utilization and power consumption of seven matrix multiplication methods.

As shown in Figure 5, it is obvious that the add-tree and full-unroll methods consume more FPGA logic resources and the power consumption is higher than the PMSDS based methods. The resource utilization and power consumption of the Double method are about the same with the traditional and additional memory methods. The Triple and Quad methods cost more logic and DSP resources than Double, as the limited BRAM ports and the loop parallelism.

Combining the results in Figure 4 and Figure 5, we can say that the PMSDS based Double method is of better tradeoff than other methods in the computation of massive multiplication operations on FPGA, considering the factors of valid bits throughput, latency, resource utilization, and power consumption.

In particular, the resource utilization and power consumption of the thee PMSDS based matrix multiplication algorithms are the results for the signed type. And there will

be less resource utilization and power consumption for the unsigned type, as there is no subtraction or addition operation in multiplicator paralleling and product splitting steps. And the total bit numbers of final multiplicators will be one bit more than the signed type, as the reserved one sign bit in the highest, which may increase the valid bits throughput of the DSP multiplier, as discussed in Table 3.

E. EFFICIENCY

We also compare our work with the state-of-the-art in the fields of resource optimization and matrix multiplication optimization on FPGAs, to show the efficiency of the proposed PMSDS method. We compare our work with the work of Lucas *et al.* [13], who optimize the DSP slice assignment approach with the configuration ports of the DSP slice in application-level, and the work of Kumm *et al.* [30], who optimize the constant matrix multiplication without using the DSP slice resource on FPGA. The comparison results are shown in Figure 6.

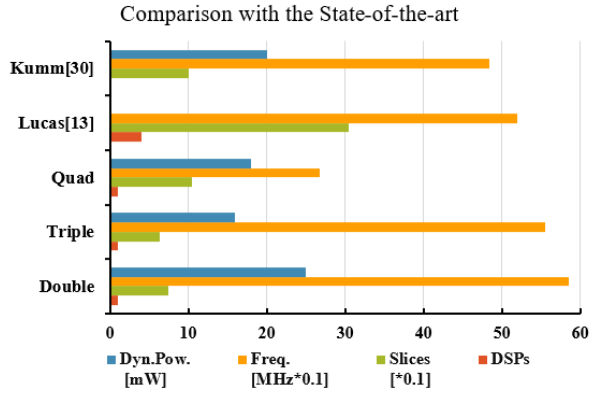


FIGURE 6. Comparison with the work of the state-of-the-art.

We change the matrix size of the PMSDS based matrix multiplication algorithms to the same as the work of Lucas *et al.* [13] and Kumm *et al.* [30], respectively. Although the actual algorithm and the FPGA version are different, the results may still show the efficiency and performance of our PMSDS method. As seen in Figure 6, the utilization of DSP and logic slices in our PMSDS based algorithms is obvious less than the work of Lucas *et al*. For the performance, the maximum frequency of our Triple and Double methods is higher than the work of Kumm *et al.*, i.e., our PMSDS based algorithms can run in faster FPGA clocks. And the dynamic power of our Quad and Triple methods is less than the work of Kumm *et al.* Moreover, in Kumm *et al.*'s work, the elements of one matrix are all constant values, which may lower the complexity of the algorithm.

F. DYNAMIC-PRECISION PMSDS MATRIX MULTIPLICATION

To further test the flexibility of the proposed PMSDS method, we try to make the parallel DSP multiplier precision dynamically configuring. We implement the two dynamic configuration methods, PD-PMSDS and TD-PMSDS, based on the prototype of Figure 2. Here, the PD-PMSDS method, is to integrate multiple predefined PMSDS parameters, including 1*1 of 16*16-bit (Single), 1*2 of 8*8-bit (Double), 1*3 of 5*5-bit (Triple), and 2*2 of 5*6-bit (Quad), over one DSP multiplier in the matrix multiplication algorithm, and provide a parameter to select corresponding parallel methods in run-time. And the TD-PMSDS method is to integrate the PMSDS parameters as variables over the DSP multiplier in the matrix multiplication algorithm IP core, and implement PMPSA as an independent module on ARM CPU of Zynq platform, to generate the PMSDS parameters and configure the DSP multiplier dynamically. In particular, as the calculation rule of the matrix multiplication, in TD-PMSDS, we just implement the following parallel number groups. One group is 1^*N, which is to implement one element of one row in the first matrix, multiplies N elements of N cols in the second matrix, and the number of valid parallel products is N, which are to be stored to N cols of the output matrix. Another group is 2*2, which is to implement two elements of one row, multiply two elements of one col, the same as subsection *C* in this section. And for other parallel number groups, we tune the parallel numbers to 2*2.

As the DMA data transfer size is limited on Zynq-7000 FPGA platform, we set the maximum matrix size to 64 in the integration systems, and make it downward adjustable. For the maximum matrix size, as the DSP multiplier precision can be configured lower with the proposed PMSDS method, the computed matrix size is able to be extended as the bit-width of the input port is fixed. E.g. in a 64*64 matrix multiplication, the maximum matrix size is 64*64 for Single method, and for Double method, the second matrix size is able to be extended to 2 times, as we compute two multiplicators of the second matrix once. And for Quad, both the two matrices can be extended to 2 times.

We define two reference baselines, namely ARM_C and Xilinx_C. The first baseline is ARM_C, in which the matrix multiplication algorithm is run on ARM CPU of Zynq platform with the traditional method, and it is to show the algorithm acceleration on FPGA. And the second is the Xilinx pipeline matrix multiplication [23], in which the algorithm is run on the FPGA part of Zynq platform, and the multiplication loop in the matrix multiplication algorithm is pipelined. The Xilinx_C method is to compare the performance and the extended function of the PMSDS dynamic configuration methods. And we implement four dynamic configuration test systems, i.e., P3, PM3, PM4, and PMD. The P3 system supports three PMSDS parameters dynamically configured, including Single, Double, and Triple. The PM3 system improves P3 with the function of matrix size extending. The PM4 system supports one more PMSDS parameters configuration than PM3, i.e., Quad method. The above three systems are PD-PMSDS based systems. And PMD is a TD-PMSDS based system, with the function of matrix size extending.

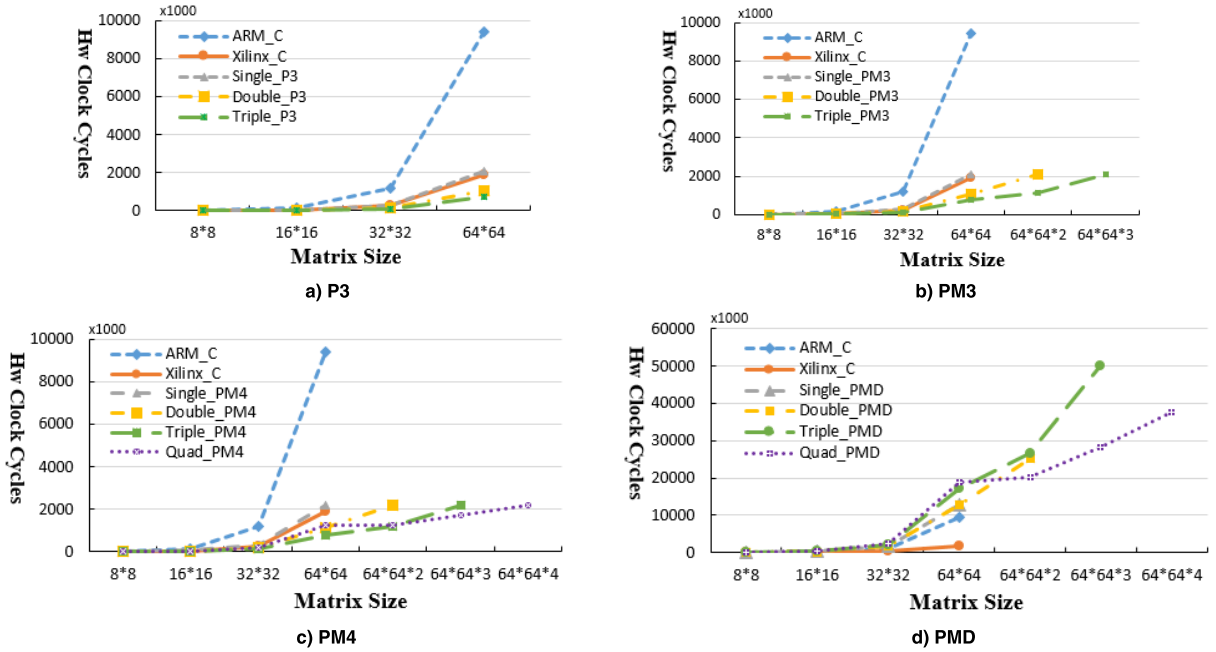


FIGURE 7. Performance comparison of different PMSDS dynamic-precision methods.

The performance test results are shown in Figure 7. As shown in Figure 7 a), b), and c), the run times of Single, Double, Triple, and Quad in P3, PM3, and PM4 systems, are much less than the ARM_C baseline. The Single methods in P3, P4 and PM4 systems, are about the same with the Xilinx_C baseline. These show that the extra logic of PMSDS dynamic configuration does not bring many bad effects to the performance of the same size matrix multiplication. Furthermore, the run times of the Double, Triple and Quad methods are all less than the Xilinx_C baseline. These show the performance improvements of the PD-PMSDS method. Moreover, in PM3 and PM4 systems, the faster methods of Double, Triple and Quad, support matrix size extending. More importantly, the run times of the extended size matrix multiplication, are no more than the Xilinx_C baseline. These show that the PD-PMSDS method is more efficient and flexible in the memory utilization than the Xilinx_C pipeline method.

For the TD-PMSDS method, as shown in Figure 7 d), the run times of Single, Double, Triple and Quad in PMD, are all more than the ARM_C and Xilinx_C baselines. These are because the PMSDS parameters in multiplicator paralleling and product splitting steps, are all variables, and the variable bounds of the inner loops, prevent the outer loop of the matrix multiplication from being pipelined on FPGA.

In particular, the run times of PMD do not include the run time of PMPSA. If considering them, the run time will be about 20000 clock cycles more, under the configuration type of x_1 and x_2. However, the PD-PMSDS method does not need to consider those run times, as the PMSDS parameters have been hardened in the algorithm IP core.

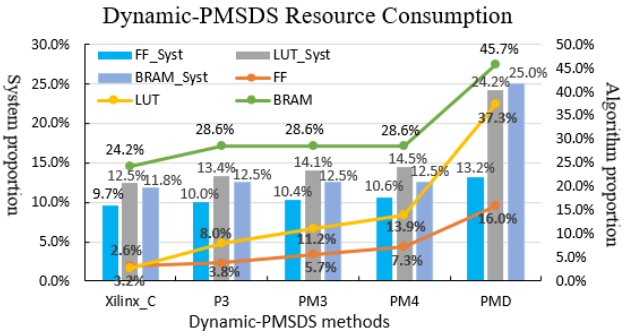


FIGURE 8. Resource consumption of dynamic-precision PMSDS.

The resource consumption of the matrix multiplication algorithms and systems, are shown in Figure 8. The line chart in Figure 8 is the percentage of the matrix multiplication algorithm in the system. And the bar chart in Figure 8 is the percentage of the system in total Zynq-7000 platform.

As seen in Figure 8, the resource percentages of the four matrix multiplication algorithms i.e., Xilinx_C, P3, PM3 and PM4, in the systems, are less than 10%, 15%, and 30% (line charts), for the resources of FF, LUT and BRAM, respectively. These show that the matrix multiplication algorithm uses far less resources than the data transferring module in the integration system. And if we improve the performance or extend the function of the matrix multiplication algorithm, the resources used in the data transferring module will gain more benefits. And that is what PMSDS method does.

The percentages of the three PD-PMSDS based systems, i.e., P3, PM3 and PM4, are all less than 15% of the Zynq

platform, which are about the same with Xilinx_C. The growing of the BRAM resource between P3 and Xilinx_C systems, is 0.7%, that is because the valid bit numbers of two input matrix arrays in P3, are 18 and 25, which are the same with the DSP slice input ports, and in Xilinx_C, both of them are 16 for short type. And the percentages of the BRAM resource in PM3 and PM4 are not increased than P3, as the DSP slice type is the same. The increases of the LUT and FF resources in P3, PM3 and PM4 than Xilinx_C, are mainly because the dynamic configuration logic in multiplicator paralleling and product splitting steps, and they are 0.3%, 0.7%, 0.9% for FF, and 0.9%, 1.6%, 2.0% for LUT, respectively. We can see the resource increase in the PD-PMSDS based method is small.

In detail, the system resources increased in P3 than Xilinx_C, are the dynamic configuration logic consumption for three configuration methods, i.e., Single, Double, Triple, which are about 0.3% for FF, and 0.9% for LUT. The increased resources between P3 and PM3 are the memory extending logic costs, i.e., 0.4% for FF, and 0.7% for LUT. The increased resources between PM3 and PM4, show the costs of integrating one more PMSDS configuration method (Quad) with memory size extending, i.e., 0.2% for FF, and 0.4% for LUT. It can be seen the cost of the function memory size extending, is also very small, in the PD-PMSDS based method.

Through the analyses above, we can say that, the PD-PMSDS method is efficient in the calculation of variable-scale based massive multiplication operations on FPGA, as it is able to dynamically configure the DSP multiplier precision and extend the memory size with little resource increase and faster computing speed, especially for lower precision.

However, for the TD-PMSDS method, PMD costs far more resources than Xilinx_C. It is because the unknown bound loops in the multiplicator paralleling and product splitting steps, increase the algorithm complexity and cost more resources for the loop bound comparison. The results of the performance and resource utilization of PMD, show that the TD-PMSDS method is not suitable to be applied as a multiplication accelerating method for the DSP slice on FPGA.

In particular, all the methods tested above cost 1 DSP slice, and we do not show that in Figure 8.

V. CONCLUSION

In this paper, we have proposed the parallel multiplication on a single DSP slice method, i.e. PMSDS method. The proposed method is able to improve the multiplier performance and valid bits throughput of a single DSP slice obviously, and downgrade the resource and power consumption of embedded FPGA systems. Moreover, the proposed PMSDS method supports the DSP multiplier precision dynamically changing in real-time when running on FPGA. That increases the performance and memory efficiency of massive multiplication operations accelerated on FPGA. For future applications, e.g. CNN algorithm, the proposed method may be able to implement multiply CNN algorithms of lower accuracy in parallel without extra DSP slices, like the pedestrian detection and license plate recognition, in the same vehicle smart device on FPGA at the same time, or dynamically switch the CNN algorithm precision for different applications, like different precision pedestrian detection, *etc*. The experiments of PMSDS based matrix multiplication algorithms show that the PMSDS method has better performance and lower resource utilization for massive multiplication operations accelerated on FPGA than other traditional methods, e.g. add-tree method, and full-unroll method *et. al*, and also has advantage in frequency and power consumption, comparing with the state-of-the-art. And the PMSDS based partially dynamic configuration method is more efficient and flexible in variable-scale based massive multiplication operations on the DSP slice resource, than traditional pipelined methods.

REFERENCES

[1] R. Warrier, S. Shreejith, W. Zhang, C. H. Vun, and S. A. Fahmy, "Fracturable DSP block for multi-context reconfigurable architectures," *Circuits, Syst. Signal Process.*, vol. 36, no. 7, pp. 3020–3033, Jul. 2017.

[2] B. Ronak and S. A. Fahmy, "Multipumping flexible DSP blocks for resource reduction on Xilinx FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 9, pp. 1471–1482, Sep. 2017.

[3] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Throughput oriented FPGA overlays using DSP blocks," in *Proc. Design, Automat. Test Eur. Conf. Exhib.*, Mar. 2016, pp. 1628–1633.

[4] T.-J. Lin, W. Zhang, and N. K. Jha, "FDR 2.0: A low-power dynamically reconfigurable architecture and its FinFET implementation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 10, pp. 1987–2000, Oct. 2015.

[5] M. Garrido, M. Á. Sánchez, M. L. López-Vallejo, and J. Grajal, "A 4096-point radix-4 memory-based FFT using DSP slices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 1, pp. 375–379, Jan. 2017.

[6] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, "The iDEA DSP block-based soft processor for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, Aug. 2014, Art. no. 19.

[7] M. K. Jaiswal and R. C. C. Cheung, "Area-efficient architectures for double precision multiplier on FPGA, with run-time-reconfigurable dual single precision support," *Microelectron. J.*, vol. 44, no. 5, pp. 421–430, May 2013.

[8] H. Zhang, D. Chen, and S.-B. Ko, "Area- and power-efficient iterative single/double-precision merged floating-point multiplier on FPGA," *IET Comput. Digit. Tech.*, vol. 11, no. 4, pp. 149–158, Jul. 2017.

[9] K. Sano, S. Abiko, and T. Ueno, "FPGA-based stream computing for high-performance N-Body simulation using floating-point DSP blocks," in *Proc. 8th Int. Symp. Highly Effic. Accel. Reconfigurable Technol.*, Jun. 2017, pp. 1–6.

[10] K. Sano and S. Yamamoto, "FPGA-based scalable and power-efficient fluid simulation using floating-point DSP blocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2823–2837, Mar. 2017.

[11] B. Ronak and S. A. Fahmy, "Minimizing DSP block usage through multi-pumping," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Dec. 2015, pp. 184–187.

[12] B. Ronak and S. A. Fahmy, "Mapping for maximum performance on FPGA DSP blocks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 4, pp. 573–585, Apr. 2016.

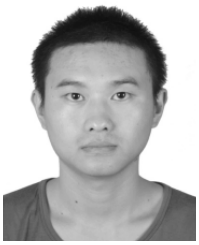
[13] E. De Lucas, M. Sanchez-Elez, and I. Pardines, "DSPONE48: A methodology for automatically synthesize HDL focus on the reuse of DSP slices," *J. Parallel Distrib. Comput.*, vol. 106, pp. 132–142, Aug. 2017.

[14] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with INT8 optimization on Xilinx devices," White Paper WP485, Nov. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[15] S. Zhang, Z. Huang, W. Wang, R. Tian, and J. He, "HACO-F: An accelerating HLS-based floating-point ant colony optimization algorithm on FPGA," *Int. J. Performability Eng.*, vol. 13, no. 6, pp. 854–863, Oct. 2017.

[16] X. Zhou, Y. Ito, and K. Nakano, "An efficient implementation of the gradient-based Hough transform using DSP slices and block RAMs on the FPGA," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2014, pp. 762–770.
[17] M. Fürer, "Faster integer multiplication," *SIAM J. Comput.*, vol. 39, no. 3, pp. 979–1005, Sep. 2009.
[18] A. Schönhage and V. Strassen, "Schnelle multiplikation großer Zahlen," *Computing*, vol. 7, nos. 3–4, pp. 281–292, Sep. 1971.
[19] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Dokl. Akad. Nauk SSSR* vol. 145, no. 2, pp. 293–294, Feb. 1962.
[20] M. Véstias, R. P. Duarte, J. T. De Sousa, and H. Neto, "Parallel dot-products for deep learning on FPGA," in *Proc. 27th Int. Conf. Field-Program. Logic Appl.*, Sep. 2017, pp. 1–4.
[21] W. Yang, K. Li, and K. Li, "A parallel computing method using blocked format with optimal partitioning for SpMV on GPU," *J. Comput. Syst. Sci.*, vol. 92, pp. 152–170, Mar. 2018.
[22] Xilinx, *Virtex-5 FPGA XtremeDSP Design Considerations User Guide UG193 (V3.6)*. Accessed: Jul. 27, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides /ug193.pdf
[23] Xilinx, *SDSoC Development Environment*. Accessed: Jul. 2017. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html
[24] S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "A 16-nm multiprocessing system-on-chip field-programmable gate array platform," *IEEE Micro*, vol. 36, no. 2, pp. 48–62, Mar./Apr. 2016.
[25] M. Ramirez, M. Daneshtalab, J. Plosila, and P. Liljeberg, "NoC-AXI interface for FPGA-based MPSoC platforms," in *Proc. 22nd Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2012, pp. 479–480.
[26] D. Bagni, A. Di Fresco, J. Noguera, and F. M. Vallina. A Zynq accelerator for floating point matrix multiplication designed with Vivado HLS, version 2.0. Xilinx. Accessed: Jan. 21, 2016. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf
[27] S. G. Singapura, A. Panangadan, and V. K. Prasanna, "Performance modeling of matrix multiplication on 3D memory integrated FPGA," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, May 2015, pp. 154–162.
[28] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377–1392, Oct. 2007.
[29] P. Zhou, H. Park, Z. Fang, J. Cong, and A. DeHon, "Energy efficiency of full pipelining: A case study for matrix multiplication," in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 172–175.
[30] M. Kumm, M. Hardieck, and P. Zipf, "Optimization of constant matrix multiplication with low power and high throughput," *IEEE Trans. Comput.*, vol. 66, no. 12, pp. 2072–2080, Dec. 2017.

ZHANGQIN HUANG received the B.S., M.S., and Ph.D. degrees in computer science from Xi'an Jiaotong University, Xi'an, China, in 1986, 1989, and 2000, respectively. From 2001 to 2003, he was a Postdoctoral Researcher with the Technische Universiteit Eindhoven (TU/e), Eindhoven, The Netherlands. He is currently a Professor and a Doctoral Supervisor of the Faculty of Information Technology, Beijing University of Technology. His current research interests include network communication, codesign for embedded software and hardware, human–computer interaction based on the Internet, and mass data storage.



SHUO ZHANG was born in Beijing, China, in 1991. He received the B.S. degree in software engineering, and the M.B.A. and D.B.A. degrees in software engineering from the School of Software, Beijing University of Technology, Beijing, in 2013 and 2014, respectively, where he is currently pursuing the Ph.D. degree with the Faculty of Information Technology. His current research interests include codesign for embedded software and hardware, embedded system architecture, and FPGA hardware acceleration.

WEIDONG WANG received the Ph.D. degree in computer science from Beijing Jiaotong University. From January 2013 to January 2015, he was a Visiting Scholar with the Department of Computer Science, University of Wyoming. He is currently a Lecturer with the Faculty of Information Technology, Beijing University of Technology. His research interest includes the IoT hardware/software optimization. For the aspect of design, he focuses on using hardware/software design to improve the performance of the IoT systems. For implementation, he has been working on optimizing performance, resilience, and service-oriented architecture.