Applied Soft Computing 129 (2022) 109639

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Software and hardware co-design and implementation of intelligent optimization algorithms

Zonglin Fu [a], Shu-Chuan Chu [a], Junzo Watada [b], Chia-Cheng Hu [c], Jeng-Shyang Pan [a,d,*]

[a] College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China
[b] Graduate School of Information, Production and Systems, Waseda University, Kitakyushu 808-0135, Japan
[c] College of Artificial Intelligence, Yango University, Fuzhou, 330015, China
[d] Department of Information Management, Chaoyang University of Technology, Taichung, Taiwan

ARTICLE INFO

Article history:
Received 4 May 2022
Received in revised form 15 August 2022
Accepted 8 September 2022
Available online 22 September 2022

Keywords:
Intelligent optimization algorithm
Advanced RISC machines
Field programmable gate array
Advanced eXtensible interface

ABSTRACT

In order to improve the operating efficiency of the algorithm, some intelligent optimization algorithms are considered to be implemented on hardware. However, the existing design scheme has the problem of poor versatility. Therefore, this paper proposes a general software–hardware co-design scheme of intelligent optimization algorithms. In the design scheme, the initialization module and fitness module of the algorithm are deployed on the Advanced RISC Machines (ARM) for execution to increase the flexibility of the program. The update module of the algorithm is deployed on the Field Programmable Gate Array (FPGA) for execution to realize the hardware acceleration. The data between ARM and FPGA is transferred through Advanced eXtensible Interface (AXI) bus. In this paper, the PSO, BA, WOA, GWO, CMAES and EO algorithms are implemented with the proposed design scheme. And the six algorithms are tested on thirteen benchmark functions of different types. The experimental results prove the feasibility of the design scheme. In addition, by comparing with software and other implementation methods in execution time, resource occupancy and convergence, the effectiveness and superiority of the proposed scheme are proved.

1. Introduction

In recent years, intelligent optimization algorithms have developed into an important method for solving optimization problems. Different from traditional calculus-based methods and exhaustive methods, it is a global optimization method with high robustness, self-learning and self-adaptation. Intelligent optimization algorithms can not only solve complex optimization problems in multi-dimensional space, but also solve faster and more efficiently than traditional methods [1,2].

Intelligent optimization algorithms can also be called heuristic algorithms. The inspiration of this kind of algorithm mainly comes from human intelligence, the sociality of biological groups and the laws of natural phenomena [3,4]. For example, inspired by the mechanism of biological evolution, John Holland proposed the genetic algorithm (GA) [5]. Inspired by the foraging behavior of ants, Marco Dorigo proposed the ant colony optimization (ACO) algorithm [6]. In addition, there are the simulated annealing (SA) algorithm [7], sine cosine algorithm (SCA) [8], immune algorithm (IA) [9], cuckoo search optimization (CSO) algorithm [10] and so on. These algorithms provide more solutions for complex optimization problems [11,12]. At present, more and more intelligent optimization algorithms are applied to solve practical application problems, such as wireless sensor network [13,14], path planning [15], image processing [16], etc.

Every intelligent optimization algorithm is not perfect when it first comes out. Although they can achieve the desired effect when solving a certain problem, it is inevitable that they may have the problem of slow convergence speed and easy to fall into the local optimal solution [17,18]. To overcome these shortcomings, researchers use different methods to improve the algorithm. For example, the idea of grouping is added to the algorithm to speed up the convergence speed of the algorithm [19]. The strategy of opposition-based learning can help the algorithm to jump out of the local optimal solution [20]. The compact strategy can reduce the memory footprint of the algorithm at runtime [21]. The binary mechanism can help the algorithm to better solve discrete optimization problems such as feature selection [22,23]. These improved strategies push the optimization algorithm closer to the target solution of the problem.

Intelligent optimization algorithms and their improved versions are usually implemented and verified on software platforms [24–26]. But with the development of Internet of Things

* Corresponding author at: College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China.
E-mail addresses: zonglin@sdust.edu.cn (Z. Fu), scchu0803@gmail.com (S.-C. Chu), junzo.watada@gmail.com (J. Watada), cchu.chiachenghu@gmail.com (C.-C. Hu), jspan@cc.kuas.edu.tw (J.-S. Pan).

https://doi.org/10.1016/j.asoc.2022.109639
1568-4946/

(IoT) technology, optimization problems in different fields (such as power control, autonomous driving, transportation) are gradually turned to "marginal" processing [27]. That is, relevant algorithms are deployed into edge devices for real-time processing. Edge devices typically have lower computing power due to the cost and other constraints. Therefore, when the software-implemented algorithm is processed on edge devices, it usually faces the problems of long solution time and low execution efficiency. These problems cause the algorithm to fail to meet the real-time processing requirements [28]. The main reason for this is that algorithms are usually executed sequentially on software platforms. When dealing with large-scale complex optimization problems, the sequential execution structure severely restricts the running speed of the algorithm. Different from the software platform, the agility and parallelism of the hardware platform can help the algorithm to better overcome the above shortcomings. Therefore, implementing the algorithm on the hardware platform has become the choice of some researchers [29–31].

FPGA device is a standard hardware platform. It can well verify the feasibility and effectiveness of the hardware implementation of the optimization algorithm. The full name of FPGA is Field Programmable Gate Array. It is a semi-custom circuit. And it is a logical array that can be programmed repeatedly. It is characterized by high integration, flexible design and high compatibility [32,33]. The FPGA chip can build a corresponding pipeline structure according to the characteristics of the circuit, so as to realize the parallel calculation of data. Compared with Central Processing Unit (CPU) [34], the parallel computing capability of the FPGA can improve the running speed of the program and reduce the delay. Compared with Graphic Processing Unit (GPU) [35], FPGA has certain advantages in power consumption and flexibility. Compared with Application Specific Integrated Circuit (ASIC) chips [36], FPGA has the advantages of short development cycle and high cost performance.

In this paper, a software–hardware co-design scheme of intelligent optimization algorithms is proposed by using FPGA platform and Advanced RISC Machines (ARM) platform. The intelligent optimization algorithm can usually be divided into three modules: initialization module, fitness module and update module. Among them, the update module includes parameter update, position update and optimal solution update. There is no doubt that the update module is the main structure of the algorithm. It covers the vast majority of calculations of the algorithm. Therefore, in the design scheme, the update module is deployed on the FPGA for execution to realize the hardware acceleration of the algorithm. The fitness module often requires different benchmark functions to evaluate the algorithm. However, the FPGA program is fixed, it is not convenient to change the benchmark function. Therefore, in the design scheme, the fitness module is deployed to the ARM for execution to increase the flexibility of the program. The initialization module is the beginning module of the algorithm. And the ARM controls the execution flow of the algorithm. Therefore, in the design scheme, the initialization module is also deployed to the ARM for execution to reduce data transmission. In addition, executing the initialization module and fitness module on the ARM can also reduce the occupation of hardware resources by the algorithm. The FPGA platform only synthesizes the resources occupied by the main structure of the algorithm, which facilitates the comparison of the resource occupancy rates of different algorithms.

The development device we used is the Xilinx UltraScale MPSOC AXU3EG. The device consists of two parts: processing system (PS) and programmable logic (PL). PS, also known as ARM side, is the software part of the device. PL, also known as FPGA side, is the hardware part of the device. For the convenience of description, the software part is collectively referred to as ARM, and the hardware part is collectively referred to as FPGA. The experimental platform is the Vivado Design Suite [37]. It contains three tools: Vivado High Level Synthesis (HLS), Vivado, and Xilinx Software Development Kit (SDK). Among them, the Vivado HLS can optimize the operation process of the algorithm through instructions and generate the IP core used on the FPGA. The Vivado tool can design the overall architecture and place-and-route diagram of the algorithm. The Xilinx SDK can complete the development of the algorithm on the ARM. ARM and FPGA exchange data through the Advanced eXtensible Interface (AXI) bus. In this paper, six intelligent optimization algorithms are implemented with the above design scheme. They are particle swarm optimization (PSO) algorithm [38,39], bat algorithm (BA) [40], whale optimization algorithm (WOA) [41], grey wolf optimizer (GWO) algorithm [42], covariance matrix adaptation evolution strategy (CMAES) algorithm [43] and equilibrium optimizer (EO) algorithm [44] respectively. The feasibility and effectiveness of the design scheme are verified by the test results of six algorithms on thirteen benchmark functions [45,46] with different characteristics. Compared with the results of software implementation, the algorithm implemented by the design scheme in this paper achieves 3 to 9 times acceleration effect on most functions. Moreover, compared with the implementation method proposed in the recent literature [47], the design scheme in this paper has more advantages in execution time, resource occupancy and convergence. The main contributions of this paper are described as follows:

1. This paper proposes a general software–hardware co-design scheme of intelligent optimization algorithms.
2. The design scheme deploys the update module on the FPGA for execution. The execution speed of the algorithm is accelerated by the parallel computing power of the FPGA platform. The fitness module is deployed on the ARM. It facilitates the modification of the benchmark function and increases the flexibility of the program.
3. The PSO, BA, WOA, GWO, CMAES and EO algorithms are implemented with the design scheme in this paper. Among them, the EO algorithm is implemented in hardware for the first time.
4. This paper evaluates the execution time, resource occupancy and convergence ability of the six algorithms on thirteen benchmark functions.

The rest of the paper is structured as follows: Section 2 is a compilation of some related work. Section 3 describes how the six intelligent optimization algorithms work. Section 4 introduces the software–hardware co-design scheme of intelligent optimization algorithms in detail. Section 5 presents and analyzes the experimental results. Section 6 is a summary of the paper and a discussion of the future work directions.

2. Related works

Although there are a large number of optimization algorithms, they are rarely implemented on the hardware platform [48–50]. In this paper, the PSO, BA, WOA, GWO, CMAES and EO algorithms are implemented with a software–hardware co-design scheme. These six algorithms represent different types of the algorithms such as simple algorithms, complex algorithms, advanced algorithms, and novel algorithms. In this section, we investigate and study the hardware implementation of the above six intelligent optimization algorithms. At present, there are few research materials on the hardware implementation of the CMAES algorithm. Literature [51] implements the CMAES algorithm on the FPGA and uses it to solve numerical optimization problems in embedded systems. In addition, we did not find the hardware implementation information of the EO algorithm. Therefore, this paper will be the first hardware implementation of the EO algorithm.

The PSO algorithm is an earlier meta-heuristic algorithm. Compared with other optimization algorithms, the PSO algorithm has a lot of research materials on hardware. Literature [52] uses the fixed-point method to implement the PSO algorithm on FPGA, and analyzes the impact of different population sizes on the running time and resource occupancy through experiments. Literature [53] uses single precision floating-point numbers when implementing a neural network system trained by the PSO algorithm on the FPGA. This implementation increases the utilization of the device and improves the convergence ability of the algorithm. In general, the representation range and precision of fixed-point numbers in hardware are lower than that of floating-point numbers. At the same time, the operation steps of floating-point numbers are more than those of fixed-point numbers. Therefore, literature [54] proposes a new digital implementation method. It is used to generate the hardware implementation of the PSO algorithm. This hardware architecture based on finite state machine (FSM) is implemented on the FPGA effectively reducing the running time of the algorithm. However, the FSM architecture is a pure hardware architecture. Its state transition process is complex and inconvenient for program changes.

The operator of the BA algorithm is complex. And the randomness of the algorithm is strong. Traditional hardware design techniques cannot provide a good optimization scheme for the BA algorithm. Therefore, people need to adopt advanced design architectures to solve complex computing problems. Literature [55] realizes the BA algorithm on the FPGA platform. The algorithm structure is designed according to the computational complexity. The literature uses the inherent parallelism and powerful computing power of the FPGA to overcome the shortcomings of the algorithm. Literature [56] exploits the FSM to maximize the parallelism of the BA algorithm on FPGA. All states and modules can be executed simultaneously. This implementation greatly reduces the execution time of the algorithm. However, when the module need to be modified or added, the state transition process needs to be readjusted.

The WOA algorithm has a simple structure. However, it has many parameters and has the strong dependence on random functions and absolute value functions. When solving large-scale complex problems, the execution time of the WOA algorithm increases rapidly. However, the parallelism of the FPGA platform can effectively overcome this shortcoming. Literature [57] proposes a framework for the partially parallel WOA algorithm. It uses the Open Computing Language (OpenCL) as the programming language for the SoC. The main part of the algorithm and the fitness module are offloaded to the FPGA for execution. Later, another design of a fully parallel WOA algorithm is proposed in literature [58]. It follows the process of implementing the algorithm in literature [57] on CPU and FPGA. The effectiveness of the two design schemes has been verified on the benchmark function. In both designs, the fitness module is offloaded to the FPGA side for implementation. However, when the benchmark function in the fitness module needs to be changed, the program on the FPGA needs to be completely regenerated.

Literature [47] implements the PSO, BA, GWO, EA, and NM algorithms on FPGA. All five algorithms are implemented on the LabVIEW FPGA platform. And they have the same design scheme. Taking the GWO algorithm as an example, it is divided into three modules for graphical implementation in the literature. The three modules are the initialization module, the update module of the three wolves and the population position update module. The graphical implementation does not require learning hardware description languages (VHDL and Verilog). It is easy to design and implement. However, during the execution of the algorithm, this method will greatly increase the resource consumption.

In the investigation and research, we find that although the pure hardware implementation of the algorithm has higher efficiency, its hardware structure is not easy to change, and it cannot solve different problems. Different from pure hardware implementation, according to the characteristics of different modules, the design scheme in this paper deploys them in software platform and hardware platform respectively. This approach preserves the generality of the algorithm. In addition, different from other software–hardware co-design methods, we put the fitness module on the ARM for execution. On the one hand, this method avoids the occupation of hardware resources by the fitness module. At the same time, it facilitates the modification of the test function in the fitness module and increases the flexibility of the program. On the other hand, this approach reduces the latency of the program in the FPGA.

3. The principles of the six algorithms

In this paper, we design a software–hardware co-design scheme of intelligent optimization algorithms, and select six typical algorithms for implementation. The PSO algorithm is a classic swarm intelligence optimization algorithm. It has the advantages of simple structure, fast convergence and easy implementation. However, it also has the problem of easily falling into local optimum. Compared with the PSO algorithm, the BA algorithm has stronger search ability in a large range. Its structure is simple and its dependence on the initial point is small. But its operator is complex, the convergence is slow, and it is easy to fall into local optimum. Different from the PSO algorithm and BA algorithm, the WOA algorithm has the ability to jump out of the local optimum. But its calculation is complex and the execution time is long. The hardware implementation might be able to overcome this shortcoming. The GWO algorithm has a simple structure and few parameters. The algorithm also contains convergence factors that can be adaptively adjusted. They help the GWO algorithm strike a balance between exploration and exploitation. In addition, the update process of the three wolves in the GWO algorithm is very suitable for parallel processing. The CMAES algorithm is a high-performance optimization algorithm. Its performance is usually simulated in software, and the effect in hardware environment is not clear. The EO algorithm is a novel optimization algorithm proposed recently. It has fast convergence speed and high optimization performance. Currently, the EO algorithm has not been implemented in hardware. In this section, the basic principles of the six algorithms are described.

3.1. Particle swarm optimization

The PSO algorithm is an intelligent optimization algorithm proposed by Eberhart and Kennedy in 1995 based on the foraging behavior of birds. The PSO algorithm uses a massless particle to simulate birds in a flock. Each particle has two properties: velocity and position. The velocity represents the step size of the particle movement. The position indicates the direction in which the particle moves. In the iterative process, each particle independently searches for the optimal solution in the search space, and shares the current individual optimal solution with other particles. The algorithm takes the best individual optimal solution as the current global optimal solution of the population. Each particle updates its velocity and position according to the current individual optimal solution and the current global optimal solution.

$$v_i^{g+1} = \omega \cdot v_i^g + c1 \cdot rand() \cdot (x_{pbest}^g - x_i^g) + c2 \cdot rand() \cdot (x_{gbest}^g - x_i^g) \quad (1)$$

$$x_i^{g+1} = x_i^g + v_i^{g+1} \quad (2)$$

Among them, g is the current iteration number. v_i and x_i represent the velocity and position of the ith particle, respectively, $i = (1, 2, \ldots, NP)$. NP is the population size. ω is the inertia factor. Its value is 0.7. $c1$ and $c2$ are learning factors with the value of 1.4962. x_{pbest} and x_{gbest} represent the current individual optimal solution of the particle and the current global optimal solution of the population, respectively. $rand()$ is a random number in the interval [0,1].

3.2. Bat algorithm

The BA algorithm is an intelligent optimization algorithm proposed by Yang et al. in 2010 based on the predation behavior of bats. In the BA algorithm, in order to simulate the behavior of bats avoiding obstacles and preying on prey, Yang et al. propose three assumptions:

1. All bats in the population sense distance using echolocation.
2. The bat flies randomly at position x with velocity v. Meanwhile, bats have different wavelength λ, loudness A and pulse emission rate r.
3. The loudness varies in the range between the maximum value A_0 and the minimum value A_{min}.

The bat's position and velocity update equations are defined as follows:

$$v_i^{g+1} = v_i^g + f_i \cdot (x_i^g - x_{gbest}^g) \tag{3}$$

$$x_i^{g+1} = x_i^g + v_i^{g+1} \tag{4}$$

Among them, g is the current iteration number. v_i and x_i represent the velocity and position of the ith bat, respectively, $i = (1, 2, \ldots, NP)$. NP is the population size. X_{gbest} is the current local optimum for the bat population. f_i is the sound frequency of the ith bat, and the calculation equation is:

$$f_i = f_{min} + (f_{max} - f_{min}) \cdot \beta \tag{5}$$

Where f_{min} and f_{max} represent the minimum and maximum sonic frequencies, respectively. They take the values 0 and 1 respectively. β is a random vector in the interval [0,1].

During the random flight of the bat, a new local optimal solution is generated by Eq. (6).

$$x_{new} = x_{old} + \varepsilon \cdot A^g \tag{6}$$

Where x_{old} is a random flying bat. ε is a random number in the interval $[-1, 1]$. A is the average loudness of all bats in the current iteration. The update equations for loudness A and frequency r are Eqs. (7) and (8), respectively:

$$A_i^{g+1} = \alpha \cdot A_i^g \tag{7}$$

$$r_i^{g+1} = r_i^0[1 - exp(-\gamma t)] \tag{8}$$

Where α is the acoustic loudness attenuation coefficient, $\alpha \in (0, 1)$. γ is the pulse frequency enhancement coefficient, $\gamma > 0$. r_i^0 represents the initial pulse frequency of the ith bat.

3.3. Whale optimization algorithm

The WOA algorithm is an intelligent optimization algorithm proposed by Mirjalili in 2016 based on the hunting behavior of the whale. The algorithm mainly includes three stages: encircling prey, bubble-net attacking prey, and searching for the prey. In the WOA algorithm, the position of each whale represents a potential solution. The algorithm finds the global optimal solution by iteratively updating the whale's position.

3.3.1. Encircling prey

First, the whale population finds its prey and surrounds it. The contraction behavior of the whale is defined by Eqs. (9) and (10).

$$D_i = |C \cdot X_{gbest}^g - X_i^g| \tag{9}$$

$$X_i^{g+1} = X_{gbest}^g - A \cdot D_i \tag{10}$$

Among them, g is the current iteration number. X_i represents the position of the ith whale, $i = (1, 2, \ldots, NP)$. NP is the population size. X_{gbest} is the current global optimal solution. A and C are coefficient vectors, and they are calculated as follows:

$$A = 2a \cdot r1 - a \tag{11}$$

$$C = 2 \cdot r2 \tag{12}$$

Where a decreases linearly from 2 to 0 as the number of iterations increases. $r1$ and $r2$ are random vectors in the interval [0, 1].

3.3.2. Bubble-net attacking prey

In the process, the algorithm creates a spiral equation based on the distance between the whale and its prey. The whale's position is updated according to the spiral equation.

$$D_i' = |X_{gbest}^g - X_i^g| \tag{13}$$

$$X_i^{g+1} = X_{gbest}^g + D_i' \cdot e^{bl} \cdot \cos(2\pi l) \tag{14}$$

Where D' represents the distance between the individual and the optimal solution. b is a constant that determines the shape of the spiral and takes the value 1. l is a random number in the interval $[-1, 1]$.

There are two main ways that the whales get close to their prey: encircling prey and bubble-net attacking prey. The WOA algorithm chooses which way to update the individual position according to the probability p.

$$X_i^{g+1} = \begin{cases} X_{gbest}^g - A \cdot D_i, & p < 0.5 \\ X_{gbest}^g + D_i' \cdot e^{bl} \cdot \cos(2\pi l), & p \geq 0.5 \end{cases} \tag{15}$$

3.3.3. Searching for the prey

In this process, the parameter A is used to decide whether the algorithm performs the exploration phase or the exploitation phase. When $|A| < 1$, the algorithm executes the exploitation phase. Among them, the exploitation phase corresponds to two stages of encircling prey and bubble-net attacking prey. When $|A| \geq 1$, the algorithm performs the exploration phase according to Eqs. (16) and (17).

$$D_i'' = |C \cdot X_{rand}^g - X_i^g| \tag{16}$$

$$X_i^{g+1} = X_{rand}^g - A \cdot D_i'' \tag{17}$$

Where X_{rand} is a random individual in the whale population. D'' is the distance between the searched individual and the random individual.

3.4. Grey wolf optimizer

The GWO algorithm is a swarm intelligence optimization algorithm proposed by Mirjalili et al. in 2014. It is inspired by the hunting behavior of the grey wolf population. There is a social hierarchy in the grey wolf population. The leader of the pack is called the alpha (α) wolf. Beta (β) wolf and delta (δ) wolf belong to the second and third tiers, respectively. They follow the orders of the α wolf. The rest of the wolves are collectively known as omega (ω) wolves. They are at the bottom of the hierarchy. Collective hunting is a social behavior of the grey wolf population. This process is completed under the leadership of the α wolf. The main steps of the GWO algorithm simulate the hunting behavior of the grey wolf population.

3.4.1. Encircling prey

The behavior of the grey wolves to round up their prey is defined by Eqs. (18) and (19):

$$D = |C \cdot X_p^g - X_i^g| \tag{18}$$

$$X_i^{g+1} = X_p^g - A \cdot D \tag{19}$$

Eq. (18) represents the distance between the individual grey wolf and its prey. Eq. (19) is the position update equation of the individual grey wolf. g is the current iteration number. X_i represents the position of the ith grey wolf, $i = (1, 2, \ldots, NP)$. NP is the population size. X_p is the position of the prey. A and C are coefficient vectors. Their calculation equations are as follows:

$$A = 2a \cdot r_1 - a \tag{20}$$

$$C = 2 \cdot r_2 \tag{21}$$

Where r_1 and r_2 are random numbers in the interval [0, 1]. a is the convergence factor. Its value decreases linearly from 2 to 0 as the number of iterations increases.

3.4.2. Hunting

The wolves locate the location of their prey. Under the command of α, β, δ wolves, the grey wolf population surrounds the prey. The distance between each ω wolf and the α, β, δ wolves is calculated by Eq. (22).

$$\begin{cases} D_\alpha = |C_1 \cdot X_\alpha^g - X_i^g| \\ D_\beta = |C_2 \cdot X_\beta^g - X_i^g| \\ D_\delta = |C_3 \cdot X_\delta^g - X_i^g| \end{cases} \tag{22}$$

X_α, X_β, and X_δ are the positions of the α, β and δ wolves, respectively. X_i is the position of the ith ω wolf.

According to D_α, D_β, D_δ, the three position components X_1, X_2, X_3 required for ω wolf update can be obtained.

$$\begin{cases} X_1^g = X_\alpha^g - A_1 \cdot D_\alpha \\ X_2^g = X_\beta^g - A_2 \cdot D_\beta \\ X_3^g = X_\delta^g - A_3 \cdot D_\delta \end{cases} \tag{23}$$

The final position of the ω wolf is shown in Eq. (24).

$$X_i^{g+1} = \frac{X_1^g + X_2^g + X_3^g}{3} \tag{24}$$

3.4.3. Attacking prey

This process corresponds to the exploitation phase of the algorithm. The algorithm uses the linear decrease of a to simulate the state of wolves gradually approaching the prey. The fluctuation range of the A value also gradually decreases with the change of the a value. When $|A| \leq 1$, wolves attack their prey. At the same time, this behavior may cause the algorithm to get stuck in a local optimum.

3.4.4. Searching for prey

This stage corresponds to the exploration phase of the algorithm. The grey wolf population spread out in search of more suitable prey. When $|A| > 1$, the individual grey wolf is far away from the current prey (the local optimum). The value of parameter C is a random vector between 0 and 2. Its randomness increases the probability that the algorithm will find the global optimum.

3.5. Covariance matrix adaptive evolution strategy

The CMAES algorithm is an unconstrained optimization algorithm proposed by Nikolaus Hansen et al. The algorithm mainly includes three parts: population sampling, individual evaluation and selection, covariance matrix and step size updating.

3.5.1. Population sampling

This part is used to generate a new population, and the update equation is as follows:

$$x_i^{g+1} = \langle x \rangle_\mu^g + \sigma^g \cdot \underbrace{B^g \cdot D^g \cdot z_i^{g+1}}_{\sim N(0, C^g)} \tag{25}$$

Where g is the current iteration number. x_i represents the position of the ith individual, $i = (1, 2, \ldots, NP)$. NP is the population size. $\langle x \rangle_\mu$ is the population center point (also called the expectation). μ is a constant. σ is the distribution step size of the population. C represents the covariance matrix of the population distribution. Initially it is an identity matrix of size $Dim \times Dim$. Dim means dimension. The column vectors of the orthonormal matrix B consist of the orthonormal basis of the eigenvectors of the matrix C. The elements on the main diagonal of the diagonal matrix D consist of the square roots of the eigenvalues of the matrix C. The random vector z_i of the ith individual is generated by the normal distribution $N(0, 1)$.

3.5.2. Individual evaluation and selection

The algorithm evaluates and ranks the fitness of the newly generated individuals x. Then, the algorithm selects the first μ individuals to calculate the new population center point $\langle x \rangle_\mu$ and the center point $\langle z \rangle_\mu$ of the random variable z.

$$\langle x \rangle_\mu^{g+1} = \frac{\sum_{j=1}^{\mu} x_j^{g+1} \cdot \omega_j}{sum(\omega)} \tag{26}$$

$$\langle z \rangle_\mu^{g+1} = \frac{\sum_{j=1}^{\mu} z_j^{g+1} \cdot \omega_j}{sum(\omega)} \tag{27}$$

Where x_j is the j-ranked individual. z_j represents the random variable value of the x_j. ω_j is the weight of the x_j, $(\omega_1 > \omega_2 > \cdots > \omega_\mu)$.

3.5.3. Covariance matrix and step size updating

The update equation of the covariance matrix C is defined as follows:

$$C^{g+1} = (1 - c_{cov}) \cdot C^g + c_{cov} \cdot p_c^{g+1} \cdot (p_c^{g+1})^T \tag{28}$$

Where p_c is the evolution path of the covariance matrix C, and the calculation equation is:

$$p_c^{g+1} = (1 - c_c) \cdot p_c^g + \sqrt{c_c \cdot (2 - c_c)} \cdot \sqrt{\mu} \cdot B^g \cdot D^g \cdot \langle z \rangle_\mu^{g+1} \tag{29}$$

The update equation of the distribution step size σ of the population is:

$$\sigma^{g+1} = \sigma^g \cdot exp\left(\frac{1}{d_\sigma} \cdot \frac{\|p_\sigma^{g+1}\| - \hat{\chi}}{\hat{\chi}} \right) \tag{30}$$

Where p_σ is the evolution path of the distribution step *sigma*, and the calculation equation is:

$$p_\sigma^{g+1} = (1 - c_\sigma) \cdot p_\sigma^g + \sqrt{c_\sigma \cdot (2 - c_\sigma)} \cdot \sqrt{\mu} \cdot B^g \cdot \langle z \rangle_\mu^{g+1} \tag{31}$$

The values of the default parameters c_c, c_{cov}, c_σ, d_σ, $\hat{\chi}$ mentioned in Eqs. (28) to (31) are calculated according to the dimension Dim, as shown in Eqs. (32) and (33).

$$c_c = \frac{4}{4 + Dim}, \quad c_{cov} = \frac{2}{(\sqrt{2} + Dim)^2}, \quad c_\sigma = \frac{4}{4 + Dim}, \tag{32}$$

$$d_\sigma = c_\sigma^{-1} + 1, \quad \hat{\chi} \approx \sqrt{Dim} \cdot \left(1 - \frac{1}{4 \cdot Dim} + \frac{1}{21 \cdot Dim^2} \right) \tag{33}$$

3.6. Equilibrium optimizer

The EO algorithm is a novel intelligent optimization algorithm proposed by Faramarzia et al. in 2019. It is inspired by the model that control volume and mass balance. The particle concentration is updated according to the equilibrium candidate solution, and finally the model reaches an equilibrium state. The EO algorithm is mainly divided into three stages: initializing the population, establishing an equilibrium pool, and updating the particle concentration.

3.6.1. Initializing the population

The algorithm is initialized randomly within the boundary. The initial population consists of particles randomly distributed in the search space.

$$C_i^0 = LB + R_i \cdot (UB - LB) \tag{34}$$

Among them, C_i represents the concentration of the ith particle, $i = (1, 2, \ldots, NP)$. NP is the population size. R is a random vector in the interval [0, 1]. UB and LB denote the upper and lower bounds of the search space, respectively.

3.6.2. Establishing the equilibrium pool

The EO algorithm establishes an equilibrium pool to improve the ability of exploration and exploitation. The equilibrium pool consists of five equilibrium candidate particles. They are the four particles with the best fitness values ($C_{eq\_1} \sim C_{eq\_4}$), and their average value (C_{eq_ave}). After that, a candidate solution (C_{eq}) is randomly selected from the equilibrium pool. It is used for particle concentration update.

$$C_p = \left\{C_{eq\_1} \sim C_{eq\_4}, C_{eq\_ave}\right\} \tag{35}$$

$$C_{eq} = Rand(C_p) \tag{36}$$

3.6.3. Updating concentration

The particle concentration update equation is defined as follows:

$$C_i^{g+1} = C_{eq} + (C_i^g - C_{eq}) \cdot F + \frac{G}{\lambda V} \cdot (1 - F) \tag{37}$$

Where g is the current iteration number. λ is a random vector in the interval [0, 1]. V represents the unit volume and takes the value of 1.0. F is an exponential term used to balance exploration and exploitation. G is the generation rate, which is used to improve the exploitation ability of the algorithm. F is defined as:

$$F = a_1 \cdot sign(r - 0.5) \cdot [e^{-\lambda t} - 1] \tag{38}$$

$$t = (1 - \frac{g}{Max\_iter})^{a_2 \cdot \frac{g}{Max\_iter}} \tag{39}$$

Where a_1 is a constant with a value of 2.0. $sign(r - 0.5)$ is used to control the direction of exploration and exploitation. Max_iter denotes the maximum number of iterations. a_2 is a constant that controls exploitation capability, and takes the value of 1.0. G is defined as:

$$G = G_0 \cdot F \tag{40}$$

Where:

$$G_0 = GCP(C_{eq} - \lambda \cdot C) \tag{41}$$

$$GCP = \begin{cases} 0.5r_1, & r_2 \geq GP \\ 0, & r_2 < GP \end{cases} \tag{42}$$

Where r_1 and r_2 are random numbers in the interval [0, 1]. GCP is an intermediate variable. GP determines the update state of the particle. Its value is 0.5.

4. Implementation of the software–hardware co-design scheme

This section introduces the specific implementation of the software–hardware co-design of intelligent optimization algorithms. Fig. 1 is the overall architecture diagram of the scheme. As shown in the figure, the scheme is mainly divided into two parts: ARM side (PS) and FPGA side (PL). The ARM side is mainly responsible for: (1) device configuration and initialization; (2) initial population generation; (3) iterative loop control; (4) fitness module calculation. The program on the ARM side is completed in the development tool Xilinx SDK. The FPGA side is mainly responsible for: (1) parameter update; (2) optimal value update; (3) position update; (4) boundary judgment. The program on the FPGA side is completed in the development tool Vivado HLS. ARM and FPGA transmit data through the AXI bus. The layout and routing diagram of the design scheme is completed in the development tool Vivado. In addition, it should be noted that the programming language in both Xilinx SDK and Vivado HLS is C++. The difference is that the random numbers in the Xilinx SDK can be generated by the *rand*() function in the C++ library. The program in the Vivado HLS does not support the *rand*() function. Therefore, this paper uses a linear feedback shift register (LFSR) in the Vivado HLS to generate random numbers. In the following subsections, Section 4.1 presents the basic principles of the linear feedback shift registers. Section 4.2 describes the design and optimization of the FPGA side programs in the Vivado HLS. Section 4.3 shows the layout design of the scheme in the Vivado. Section 4.4 describes the program design in the Xilinx SDK.

4.1. Linear feedback shift register

The LFSR has many applications in both software and hardware. In this paper, we use the LFSR to generate pseudo-random numbers on hardware. The essence of the LFSR is to use the linear function of the output of the previous state as the input of the current state. The feedback circuit of the LFSR consists of XOR gates. The XOR operation is the most common single-bit linear function operation. First, it performs XOR operations on some bits in the register. Then, the resulting value is input into the register to perform an overall shift of the individual bits. The feedback function of the LFSR is usually a characteristic polynomial. This paper uses a 32-bit LFSR. Its feedback function is $f(x) = x^{32} + x^{22} + x^2 + x + 1$. Through the characteristic polynomial, the LFSR can generate a pseudo-random sequence with a very long cycle period. In addition, the LFSR needs a seed number before it starts working. It is the initial value of the register. Its value is any number between 1 and $2^n - 1$. n is the number of bits in the register. Fig. 2 draws the circuit diagram of a 32-bit LFSR.

4.2. Vivado High Level Synthesis

The Vivado HLS is a development tool provided by Xilinx. Fig. 3 shows the design flow of the Vivado HLS. Its most basic function is to synthesize C/C++ code into VHDL/Verilog code. And it can automatically convert the program into hardware-recognized RTL language through C simulation. The use of the Vivado HLS will greatly reduce the development time of the FPGA. Fig. 4 shows the comparison of the HLS-based FPGA with other hardware platforms.

Unlike general C/C++ code, programs in the Vivado HLS cannot all be compatible with C/C++ language data types and functions. In addition, the input and output of data in the Vivado HLS need to specify which interface is used. Whether the calculation process of the program requires a pipeline design. This paper takes the main structure of the intelligent optimization algorithm

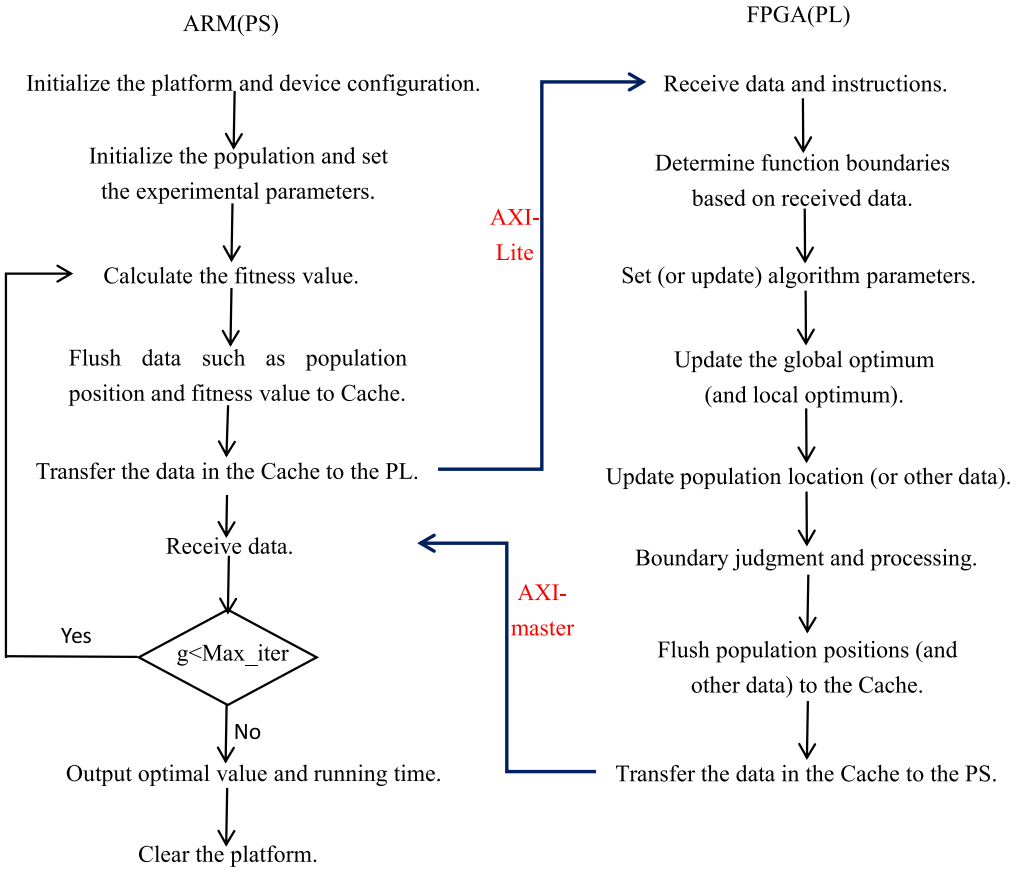

Fig. 1. The overall architecture diagram of the design scheme.

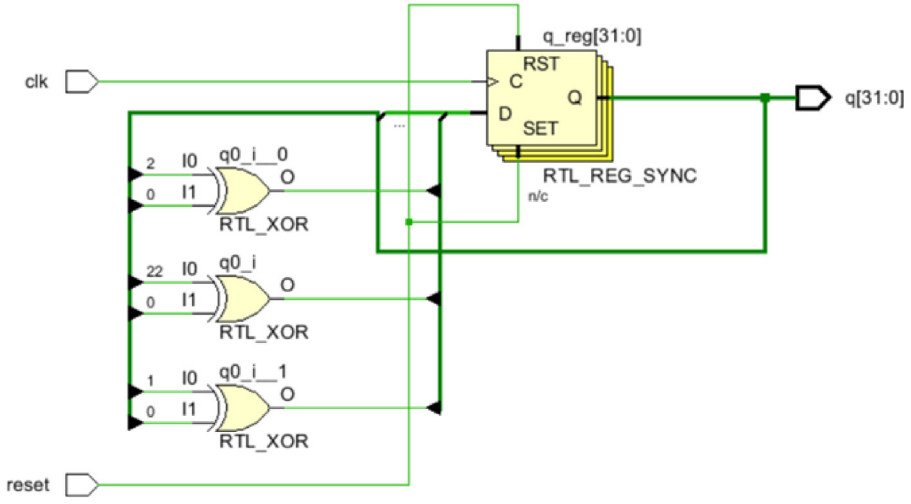

Fig. 2. The circuit diagram of 32-bit LFSR.

as the top-level function. The data type mainly uses floating-point data. The data of the input and output ports are transmitted through the AXI bus address. According to the characteristics of the algorithm, we add "pipeline" optimization instructions to the program. The "pipeline" directive allows programs to execute concurrently to shorten the startup interval of the loop. It reduces latency by pipelining the program, which in turn reduces the execution time of the algorithm. Its usage is to put pragmas inside the body of the function or loop. Algorithms 1–6 show the pseudocodes of the PSO, BA, WOA, GWO, CMAES, and EO algorithms after adding the "pipeline" instruction, respectively.

In the Algorithms 1–6, the code from the sixth to the penultimate line is implemented in the Vivado HLS, that is, the program of the algorithm on the FPGA side. The "# pragma HLS pipeline" in the pseudocodes means adding the "pipeline" instruction to the loop. In addition, because the C++ code of the CMAES algorithm is cumbersome, some double-layer and multi-layer loop structures are not described in detail in the pseudo-code. Here are our rules for adding the "pipeline" directive to the loop structure:

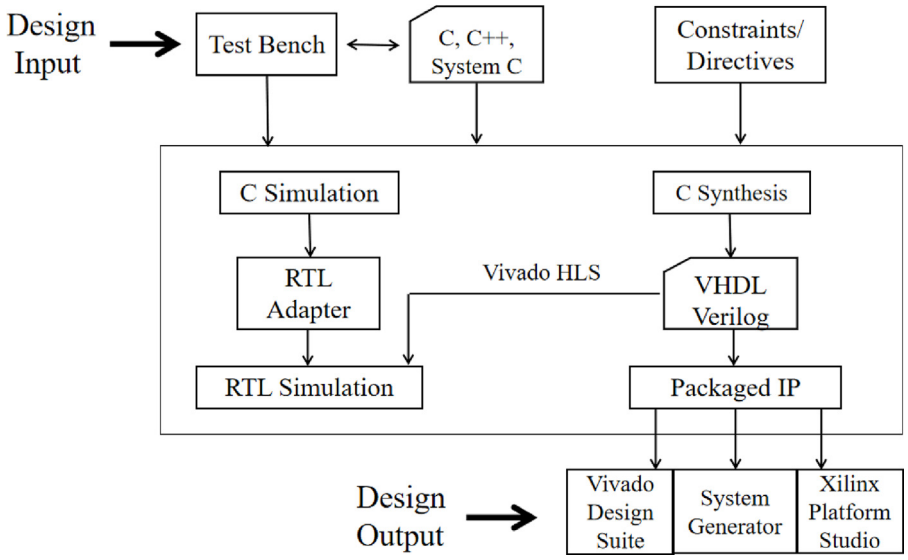

Fig. 3. The design flow of the Vivado HLS.

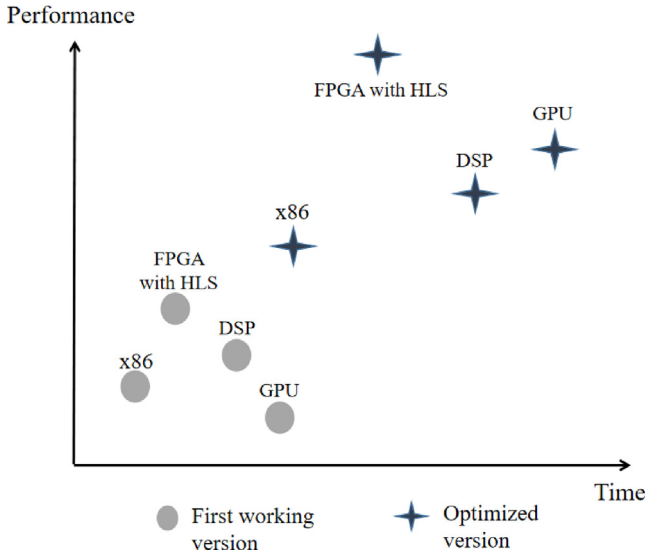

Fig. 4. Comparison of FPGA with HLS and other design platforms.

1. For the single-level loop, the "pipeline" directive is added directly.

2. For the double-layer loop, if there is no loop statement between the inner and outer layers, the "pipeline" directive is added to the inner loop. If there are parallelizable loop statements between the inner and outer layers, the "pipeline" directive is added to the outer loop.

3. For the three-level loop, if the loop statement only exists in the innermost loop, the "pipeline" directive is added to the innermost loop. If there are parallelizable loop statements between the innermost layer and the middle layer, the "pipeline" directive is added to the middle layer loop. If there are parallelizable loop statements between the middle layer and the outermost layer, the "pipeline" directive is added to the outermost loop.

It is worth noting that the above rules are not absolute. If the program achieves the best optimization effect, it needs to be analyzed according to the specific algorithm. The above rule is a general rule when we deal with the algorithm optimization in this paper. For the special loop structure, we need to add the optimization instruction based on the dependencies between the codes. It can be seen from Fig. 3 that after adding the optimization instructions, the Vivado HLS tool estimates the delay and resource occupation of the top-level function through C synthesis. After that, it is verified whether the output state of the program is correct through the co-simulation of C and RTL. Finally, the Vivado HLS packages the main structure of the algorithm into an IP core.

4.3. The design in the Vivado

The Vivado design can establish the connection between Vivado HLS program and Xilinx SDK program. That is, it is the key step to realize the data transmission between ARM and FPGA. First, the IP core exported by the Vivado HLS is added to the Block Design of the Vivado for design. The Figs. 5–10 show the layout design of the PSO, BA, WOA, GWO, CMAES, and EO algorithms in the Block Design, respectively. The core module in the figure is the ZYNQ core and the IP core of the algorithm. The ZYNQ core is the processing system of the development board. It handles the program on the ARM side in Fig. 1. As mentioned above, the program on the FPGA side in Fig. 1 is encapsulated in the IP core. In the Figs. 5–10, the Processor System Reset core, the AXI SmartConnect core, and the AXI Interconnect core establish the connection and communication between the ZYNQ core and the IP core.

In the scheme design, the data is transmitted between ARM and FPGA through the AXI bus. This means that the data is transmitted between the ZYNQ core and the IP core through the AXI bus in Figs. 5–10. Among them, the port data of the IP core is mainly pointer type data. When it transmits data to the ZYNQ core, it needs to select the *AXI_master* bus as the data transmission line. When the ZYNQ core transmits data to the IP core, the lightweight *AXI_Lite* bus is used as the data transmission line. The data transmission process between the ZYNQ core and the IP core is as follows. When the ZYNQ core transmits data to the IP core, the ZYNQ core is the master and the IP core is

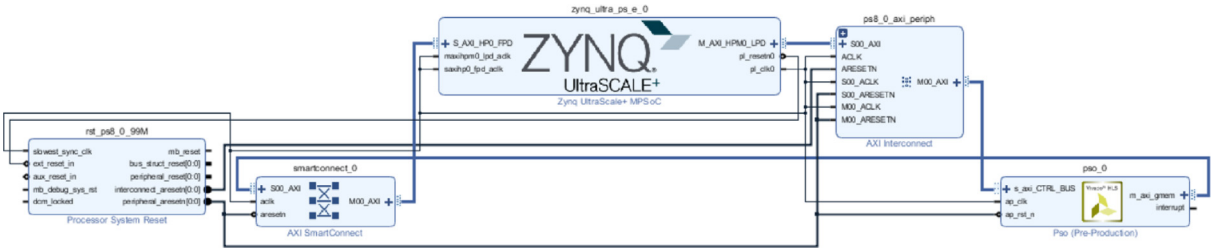

Fig. 5. The layout diagram of the PSO algorithm in Block Design.

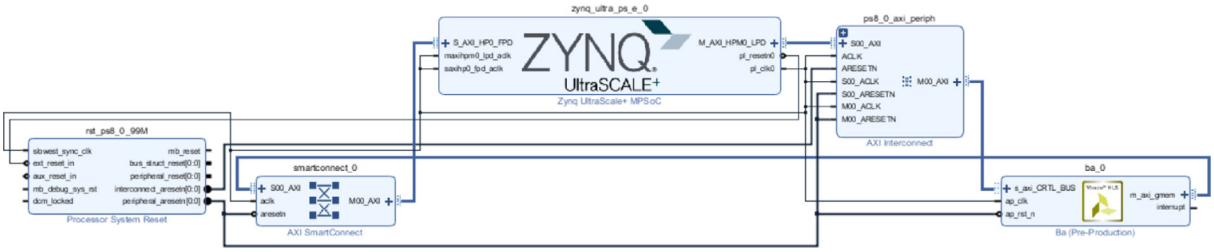

Fig. 6. The layout diagram of the BA algorithm in Block Design.

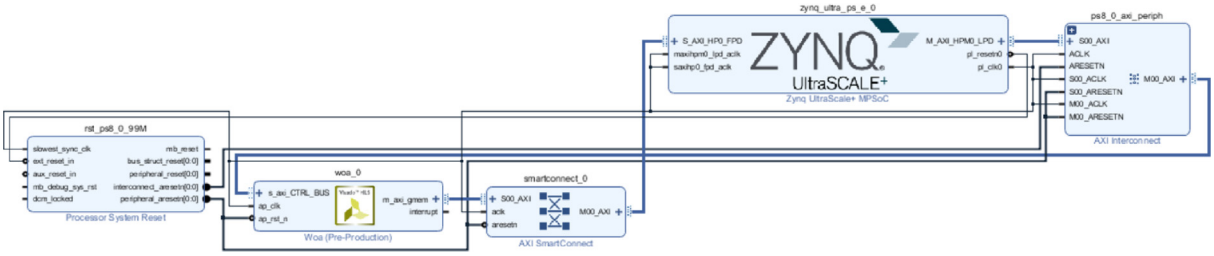

Fig. 7. The layout diagram of the WOA algorithm in Block Design.

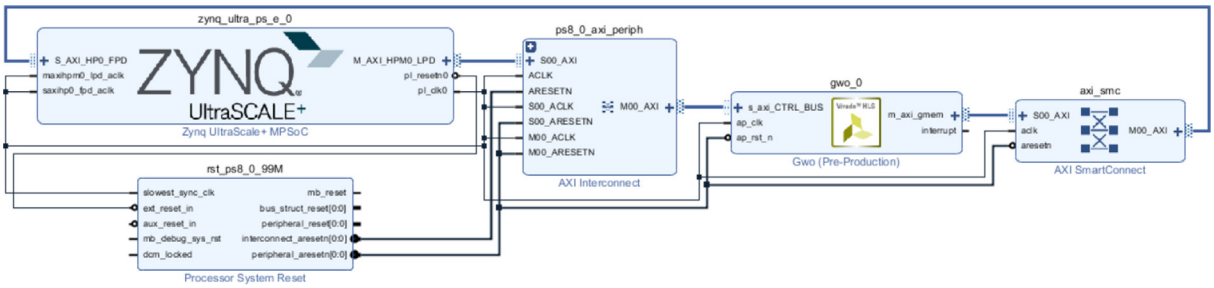

Fig. 8. The layout diagram of the GWO algorithm in Block Design.

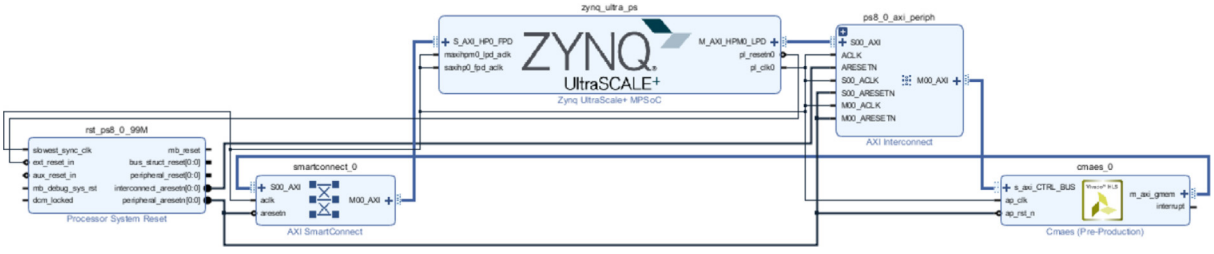

Fig. 9. The layout diagram of the CMAES algorithm in Block Design.

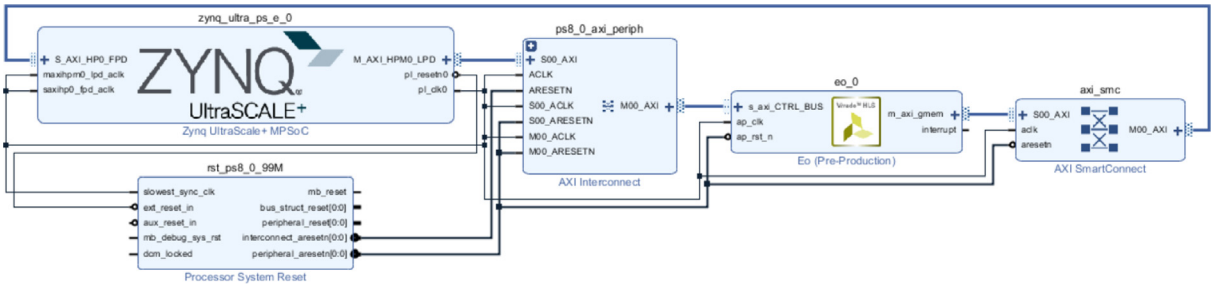

Fig. 10. The layout diagram of the EO algorithm in Block Design.

Algorithm 1 Pseudocode for the PSO algorithm with HLS optimized instructions.

Set the population size (*NP*), dimension (*Dim*), and maximum iteration number (*Max_iter*).
Initialize population position and velocity.
Initialize the individual historical optimal (*pbest*) and the population global optimal (*gbest*).
for $g = 1$ to *Max_iter* **do**
    Calculate the fitness value (*fit*) of the particle according to the benchmark function.
    Select its corresponding boundary range according to the benchmark function.
    Set the values of parameters $c1$, $c2$ and ω.
    for $i = 1$ to *NP* **do**
        if $fit(i) < pbest(i)$ **then**
            for $j = 1$ to *Dim* **do**
                # *pragma HLS pipeline*
                Update the individual historical optimal (*pbest*) of the *i-th* particle and its position.
            end for
        end if
        if $fit(i) < gbest$ **then**
            for $j = 1$ to *Dim* **do**
                # *pragma HLS pipeline*
                Update the population global optimum (*gbest*) and its position.
            end for
        end if
        for $k = 1$ to *Dim* **do**
            # *pragma HLS pipeline*
            Update the *k-th* dimension velocity of the *i-th* particle according to Eq. (1).
            Update the *k-th* dimension position of the *i-th* particle according to Eq. (2).
            Determine whether the *k-th* dimension position exceeds the boundary range.
        end for
    end for
end for

the slave. We set the "offset" of the IP core port to the "slave" state. In this case, the ZYNQ core can control the IP core through commands. First, the ZYNQ core transfers data out through the *M_AXI_HPM0_LPD* interface. The data reaches the input port of the IP core through the *AXI_Lite* bus. Then, the ZYNQ core sends the command to start the computation through the port. After receiving the command, the IP core starts to work. The calculation

Algorithm 2 Pseudocode for the BA algorithm with HLS optimized instructions.

Set the population size (*NP*), dimension (*Dim*), and maximum iteration number (*Max_iter*).
Initialize population position and velocity.
Initialize the individual historical optimal (*pbest*) and the population global optimal (*gbest*).
for $g = 1$ to *Max_iter* **do**
    Calculate the fitness value (*fit*) of the bat according to the benchmark function.
    Select its corresponding boundary range according to the benchmark function.
    Set the values of parameters A, r, ω, f_{min} and f_{max}.
    for $i = 1$ to *NP* **do**
        if $fit(i) < pbest(i)\&\&rand() < A$ **then**
            for $j = 1$ to *Dim* **do**
                # *pragma HLS pipeline*
                Update the individual historical optimal (*pbest*) of the *i-th* bat and its position.
            end for
        end if
        if $fit(i) < gbest$ **then**
            for $j = 1$ to *Dim* **do**
                # *pragma HLS pipeline*
                Update the global optimum (*gbest*) of the bat population and its position.
            end for
        end if
        Calculate the frequency f_i of the *i-th* bat according to Eq. (5).
        for $k = 1$ to *Dim* **do**
            # *pragma HLS pipeline*
            Update the *k-th* dimension velocity of the *i-th* bat according to Eq. (3).
            Update the *k-th* dimension position of the *i-th* bat according to Eq. (4).
        end for
        if $rand() > r$ **then**
            for $k = 1$ to *Dim* **do**
                # *pragma HLS pipeline*
                Update the *k-th* dimension position of the *i-th* bat according to Eq. (6).
            end for
        end if
        Determine whether the *k-th* dimension position exceeds the boundary range.
    end for
end for

Algorithm 3 Pseudocode for the WOA algorithm with HLS optimized instructions.

Set the population size (*NP*), dimension (*Dim*), and maximum iteration number (*Max_iter*).
Initialize population position.
Initialize the population global optimal (*gbest*).
for $g = 1$ to *Max_iter* **do**
    Calculate the fitness value (*fit*) of the whale according to the benchmark function.
    Select its corresponding boundary range according to the benchmark function.
    for $i = 1$ to *NP* **do**
        # pragma HLS pipeline
        if $fit(i) < gbest$ **then**
            Update the global optimum (*gbest*) of the whale population and its position.
        end if
    end for
    for $i = 1$ to *NP* **do**
        Parameters A and C are updated according to Eqs. (11) and (12), respectively.
        Generate a value of p between 0 and 1.
        for $k = 1$ to *Dim* **do**
            # pragma HLS pipeline
            if $p < 0.5$ **then**
                if $|A| \geq 1$ **then**
                    Update the *k-th* dimension position of the *i-th* whale according to Eqs. (16) and (17).
                else if $|A| < 1$ **then**
                    Update the *k-th* dimension position of the *i-th* whale according to Eqs. (9) and (10).
                end if
            else if $p \geq 0.5$ **then**
                Update the *k-th* dimension position of the *i-th* whale according to Eqs. (13) and (14).
            end if
            Determine whether the *k-th* dimension position exceeds the boundary range.
        end for
    end for
end for

ends when the state of the IP core becomes "done". After the IP core completes the computing task, it starts to transmit data to the ZYNQ core. When the IP core transmits data to the ZYNQ core, the IP core is the master and the ZYNQ core is the slave. The IP core transmits data through the *m_axi_gmem* interface. The data reaches the *S_AXI_HP0_FPD* interface of the ZYNQ core through the *AXI_master* bus. So far, a data transmission process between the ZYNQ core and the IP core is completed. After that, the ZYNQ core processes the received data and then transmits it to the IP core, and so on.

After the layout design in the Block Design is verified successfully, the design source file is generated into a top-level file and an external executable product. After that, it is converted into the bitstream file.

4.4. The design in the Xilinx SDK

The program in the development tool Xilinx SDK writes the control flow of the algorithm on the ARM side. To ensure the deployment of the platform and the execution of the algorithm, we need to load the bitstream file and the driver file generated by the Vivado into the Xilinx SDK. Algorithm 7 is the pseudo

Algorithm 4 Pseudocode for the GWO algorithm with HLS optimized instructions.

Set the population size (*NP*), dimension (*Dim*), and maximum iteration number (*Max_iter*).
Initialize population position.
Initialize the fitness of α wolf, β wolf, δ wolf and their positions.
for $g = 1$ to *Max_iter* **do**
    Calculate the fitness value (*fit*) of the ω wolves according to the benchmark function.
    Select its corresponding boundary range according to the benchmark function.
    for $j = 1$ to *Dim* **do**
        # pragma HLS pipeline
        Update α wolf, β wolf, δ wolf and their positions.
    end for
    for $i = 1$ to *NP* **do**
        for $k = 1$ to *Dim* **do**
            # pragma HLS pipeline
            Calculate the values of A and C according to Eqs. (20) and (21), respectively.
            Calculate the values of D_α, D_β, and D_δ according to Eq. (22).
            Calculate the values of X_1, X_2, and X_3 according to Eq. (23).
            Update the *k-th* dimension position of the *i-th* ω wolf according to Eq. (24).
            Determine whether the *k-th* dimension position exceeds the boundary range.
        end for
    end for
end for

code of the program control flow. As shown in the Algorithm 7, the initialization module and the fitness module of the algorithm are executed in the Xilinx SDK program. In addition, we add Cache buffers in the program. When using the AXI bus to transfer data, the program originally needs to access the data in the DDR memory. The use of the Cache buffer can reduce program access to DDR memory. According to the knowledge in the operating system, the time for the program to access the Cache buffer is less than the time for the program to access the DDR memory. Therefore, the addition of the Cache buffer can reduce the time for the program to access data, thereby reducing the overall running time of the program.

5. Experiments and analysis

In this section, we verify the feasibility and effectiveness of the proposed scheme through the experiments. First, in Section 5.1, we introduce thirteen benchmark functions for experimental testing and analyze their resource occupancy. Secondly, in Section 5.2, we analyze the impact of HLS instruction optimization on the resource occupancy and execution time of the algorithm. After that, in Section 5.3, we compare and analyze the experimental data with the software execution results. In addition, it can be known from Section 2 that the literature [47] adopts a different scheme from this paper to realize the metaheuristic algorithm. To compare the effect both it and the design scheme of this paper, we analyze the experimental results and the data of the literature [47] in Section 5.4. It should be noted that the test platform we used in the experiment is the Vivado Design Suite 2017 version. The type of development board is Xiililnx UltraScale MPsoc AXU3EG. It contains 432 Block Random

Algorithm 5 Pseudocode for the CMAES algorithm with HLS optimized instructions.

Set the population size (*NP*), dimension (*Dim*), and maximum iteration number (*Max_iter*).
Initialize the population position and the global optimum (*gbest*).
Set the initial values of parameters σ, μ, C, B, D, z, p_c and p_σ.
for $g = 1$ to *Max_iter* **do**
    Calculate the fitness value (*fit*) of the individual according to the benchmark function.
    Select its corresponding boundary range according to the benchmark function.
    Calculate c_c, c_{cov}, c_σ, d_σ and $\hat{\chi}$ according to Eqs. (32) and (33).
    # pragma HLS pipeline
    Sort the population individuals according to their fitness values.
    if $g > 1$ **then**
        for $k = 1$ to μ **do**
            # pragma HLS pipeline
            Calculate the center point $\langle z \rangle_\mu$ of the first μ random vector z according to Eq. (27).
        end for
        for $j = 1$ to *Dim* **do**
            # pragma HLS pipeline
            Update p_c and p_σ according to Eqs. (29) and (31), respectively.
        end for
        Update σ according to Eq. (30).
        for $j = 1$ to *Dim* **do**
            for $j_1 = 1$ to *Dim* **do**
                # pragma HLS pipeline
                Update the covariance matrix C according to Eq. (28).
            end for
        end for
        # pragma HLS pipeline
        Calculate the orthogonal matrix B and diagonal matrix D according to the matrix C.
    end if
    for $i = 1$ to *NP* **do**
        for $j = 1$ to *Dim* **do**
            # pragma HLS pipeline
            Generate the random vector z that follows the $N(0, 1)$ distribution.
        end for
    end for
    for $k = 1$ to μ **do**
        # pragma HLS pipeline
        Calculate the center point $\langle x \rangle_\mu$ of the first μ individual x according to Eq. (26).
    end for
    # pragma HLS pipeline
    Calculate the product of matrix B, matrix D, and the random vector z.
    for $i = 1$ to *NP* **do**
        for $j = 1$ to *Dim* **do**
            # pragma HLS pipeline
            Update the *j-th* dimension position of the *i-th* individual according to Eq. (25).
            Determine whether the *j-th* dimension position exceeds the boundary range.
        end for
    end for
end for

Algorithm 6 Pseudocode for the EO algorithm with HLS optimized instructions.

Set the population size (*NP*), dimension (*Dim*), and maximum iteration number (*Max_iter*).
Initialize population concentration (position).
Initialize equilibrium candidate particles $C_{eq\_1} \sim C_{eq\_4}$.
for $g = 1$ to *Max_iter* **do**
    Calculate the fitness value (*fit*) of the particles according to the benchmark function.
    Select its corresponding boundary range according to the benchmark function.
    Set the values of parameters a_1, a_2, GP and V.
    for $i = 1$ to *NP* **do**
        if $fit(i) < C_{eq\_1}$ **then**
            # pragma HLS pipeline
            Update C_{eq_1} and its concentration.
        else if $fit(i) \geq C_{eq\_1}$ && $fit(i) < C_{eq\_2}$ **then**
            # pragma HLS pipeline
            Update C_{eq_2} and its concentration.
        else if $fit(i) \geq C_{eq\_1}$ && $fit(i) \geq C_{eq\_2}$&&$fit(i) < C_{eq\_3}$ **then**
            # pragma HLS pipeline
            Update C_{eq_3} and its concentration.
        else if $fit(i) \geq C_{eq\_1}$ && $fit(i) \geq C_{eq\_2}$&&$fit(i) \geq C_{eq\_3}$&&$fit(i) < C_{eq\_4}$ **then**
            # pragma HLS pipeline
            Update C_{eq_4} and its concentration.
        end if
    end for
    for $j = 1$ to *Dim* **do**
        # pragma HLS pipeline
        Calculate C_{eq_ave} according to the equilibrium candidate particles $C_{eq\_1} \sim C_{eq\_4}$.
        Build the equilibrium pool according to Eq. (35).
    end for
    Calculate the value of parameter t according to Eq. (39).
    for $i = 1$ to *NP* **do**
        for $k = 1$ to *Dim* **do**
            # pragma HLS pipeline
            Parameters λ and r are randomly generated on the interval [0,1].
            Calculate the value of vector F according to Eq. (38).
            Calculate the value of vector GCP according to Eq. (42).
            Calculate the value of vector G_0 according to Eq. (41).
            Calculate the value of vector G according to Eq. (40).
            Update the *k-th* dimension concentration of the *i-th* particle according to Eq. (37).
            Determine whether the *k-th* dimension concentration exceeds the boundary range.
        end for
    end for
end for

Access Memory (BRAM). The number of Digital Signal Processing (DSP48E) is 360. The number of Flip-Flop (FF) is 141120. It also has 70560 Look Up Table (LUT).

5.1. Benchmark functions

We choose thirteen benchmark functions to calculate the fitness value of the individual. Table 1 records the mathematical

Table 1
The details of the benchmark functions.

Function name	Mathematical definition	Dimension	Input domain	Global minima
Drop-Wave	$f(x) = -\frac{1+\cos(12\sqrt{x_1^2+x_2^2})}{0.5(x_1^2+x_2^2)+2}$	2	$x_1 \in [-5.12, 5.12]$, $x_2 \in [-5.12, 5.12]$	$f(x^*) = -1$, at $x^* = (0, 0)$
Shubert	$f(x) = (\sum_{i=1}^{5} i\cos((i+1)x_1 + i)) \times (\sum_{i=1}^{5} i\cos((i+1)x_2 + i))$	2	$x_1 \in [-5.12, 5.12]$, $x_2 \in [-5.12, 5.12]$	$f(x^*) = -186.7309$
Three-hump camel	$f(x) = 2x_1^2 - 1.05x_1^4 + \frac{x_1^6}{6} + x_1x_2 + x_2^2$	2	$x_1 \in [-5, 5]$, $x_2 \in [-5, 5]$	$f(x^*) = 0$, at $x^* = (0, 0)$
Egg holder	$f(x) = -(x_2+47)\sin(\sqrt{\lvert x_2 + \frac{x_1}{2} + 47\rvert}) - x_1\sin(\sqrt{\lvert x_1 - (x_2+47)\rvert})$	2	$x_1 \in [-512, 512]$, $x_2 \in [-512, 512]$	$f(x^*) = -959.6407$, at $x^* = (512, 404.2319)$
Six-hump camel	$f(x) = (4 - 2.1x_1^2 + \frac{x_1^4}{3})x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2$	2	$x_1 \in [-3, 3]$, $x_2 \in [-2, 2]$	$f(x^*) = -1.0316$, at $x^* = (0.0898, -0.7126)$ and $(-0.0898, 0.7126)$
Bukin n.6	$f(x) = 100\sqrt{\lvert x_2 - 0.01x_1^2\rvert} + 0.01\lvert x_1 + 10\rvert$	2	$x_1 \in [-15, -5]$, $x_2 \in [-3, 3]$	$f(x^*) = 0$, at $x^* = (-10, 1)$
Levy n.13	$f(x) = \sin^2(3\pi x_1) + (x_1-1)^2(1+\sin^2(3\pi x_2)) + (x_2-1)^2(1+\sin^2(2\pi x_2))$	2	$x_1 \in [-10, 10]$, $x_2 \in [-10, 10]$	$f(x^*) = 0$, at $x^* = (1, 1)$
Goldstein-price	$f(x) = [1 + (x_1+x_2+1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)] \times [30 + (2x_1 - 3x_2)^2 (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$	2	$x_1 \in [-2, 2]$, $x_2 \in [-2, 2]$	$f(x^*) = 0$, at $x^* = (0, -1)$
Sphere	$f(x) = \sum_{i=1}^{10} x_i^2$	10	$x_i \in [-5.12, 5.12]$	$f(x^*) = 0$, at $x^* = (0, 0)$
Rastrigin	$f(x) = 100 + \sum_{i=1}^{10}[x_i^2 - 10\cos(2\pi x_i^2)]$	10	$x_i \in [-5.12, 5.12]$	$f(x^*) = 0$, at $x^* = (0, 0)$
Sum squares	$f(x) = \sum_{i=1}^{10} ix_i^2$	10	$x_i \in [-5.12, 5.12]$	$f(x^*) = 0$, at $x^* = (0, 0)$
Styblinski-tang	$f(x) = \frac{1}{2}\sum_{i=1}^{10}(x_i^4 - 16x_i^2 + 5x_i)$	10	$x_i \in [-5, 5]$	$f(x^*) = -391.6599$, at $x^* = (-2.903534, \ldots, -2.903534)$
Schwefel	$f(x) = 4189.829 - \sum_{i=1}^{10} x_i\sin(\sqrt{\lvert x_i\rvert})$	10	$x_i \in [-500, 500]$	$f(x^*) = 0$, at $x^* = (420.9687, \ldots, 420.9687)$

Algorithm 7 Pseudocode for the program in the Vivado SDK.

```
Initialize the platform.
Initialize the IP core of the intelligent optimization algorithm.
Sets the start time of program execution.
Set the population size (NP), dimension (Dim), and maximum iteration number (Max_iter).
Initialize the population and parameters that need to be transferred.
for g = 1 to Max_iter do
    Calculate the fitness value of individuals in the population according to the benchmark function.
    Flush the data that needs to be transferred to the Cache.
    Send the data in the Cache according to the address.
    Send the instruction to start execution to the IP core.
    Determine the execution status of the IP core.
    Access data such as population position, global optimal value, etc. from the Cache.
end for
Output the global optimal value of the intelligent optimization algorithm.
Sets the end time for program execution.
Calculate and output the execution time of the program.
Clear the platform.
```

definitions, dimensions and target values of the thirteen benchmark functions. It is worth noting that this dimension value is the dimension value used in the following experiments. These benchmark functions have different characteristics, as shown in Table 2. Their function graphs are shown in Fig. 11. They can evaluate the convergence of the algorithm, the ability to jump out of the local optima, etc. For example, the unimodal continuous functions, such as the sphere function and the sum squares function, can detect the convergence of the algorithm. The multimodal complex functions, such as levy n.13 function, bukin n.6 function and goldstein-price function, can test the optimization ability of the algorithm. The multiple local optimum functions, such as the egg holder function and the rastrigin function, can evaluate the ability of the algorithm to jump out of the local optima. In addition, in the design scheme of this paper, we put the fitness module on the ARM side to implement. There are two main reasons.

On the one hand, the implementation of the fitness module on ARM can reduce the occupation of hardware resources. Table 3 lists the resource occupancy and latency of the thirteen functions under their corresponding dimensions for 200 iterations. It can be seen from the results that the more complex the function expression is, the more resources are occupied. In particular, the number of DSP and FF increases exponentially when the function expressions contain trigonometric and high-power functions. Since the hardware resources on the FPGA are limited, we implement the fitness module on the ARM.

On the other hand, the implementation of the fitness module on ARM can increase the flexibility of the program. If the fitness module is encapsulated into the IP core of the FPGA, it needs to be re-planned to generate a new IP core when it is modified. When the fitness module is implemented on the ARM, only the main structure of the optimization algorithm is encapsulated in the IP core on the FPGA. When testing the algorithm with different

Table 2
The characteristics of the benchmark functions.

Function name	Description and features
Drop-Wave	Continuous; non-convex; unimodal; differentiable; non-separable; non-random; a local optimum.
Shubert	Continuous; differentiable; non-separable; multiple local optima; multiple global optima.
Three-hump camel	Continuous; non-convex; multimodal; differentiable; non-separable; three local optima
Egg holder	Continuous; non-convex; multimodal; differentiable; non-separable; non-random; non-parametric.
Six-hump camel	Continuous; non-scaler; multimodal; differentiable; non-separable.
Bukin n.6	Multiple local optima; all of which lie in a ridge.
Levy n.13	Continuous; non-separable; multimodal; differentiable.
Goldstein-price	With several local optima.
Sphere	Unimodal; convex; continuous; scalable; a local optimum.
Rastrigin	Multimodal; separable; multiple local optima.
Sum squares	Continuous; convex; unimodal; only the global optimum.
Styblinski-tang	Continuous; non-convex; multimodal; differentiable; inseparable; non-scalable.
Schwefel	Complex; with many local minima.

Table 3
The resource occupancy and latency of the benchmark functions.

	BRAM	DSP48E	FF	LUT	Latency
Drop-Wave	10	95	8662	13771	103
Shubert	10	84	4357	7909	1651
Three-hump camel	2	83	6773	7131	73
Egg holder	24	182	11952	20684	90
Six-hump camel	2	82	8591	9537	90
Bukin n.6	2	52	5550	5971	81
Levy n.13	32	314	13521	24130	72
Goldstein-price	0	96	7844	6552	66
Sphere	0	10	2313	1635	191
Rastrigin	10	87	4335	7944	1230
Sum squares	0	16	2793	1999	191
Styblinski-tang	0	51	8615	5599	281
Schwefel	10	82	5389	9643	1248

benchmark functions in this case, we only need to change the software part of the program.

5.2. Impact of the HLS instruction optimizations on results

To understand the impact of the HLS instruction optimization on results, we conduct experiments on the six algorithms in terms of resource occupancy, convergence value, and running time. We uniformly set the evaluation times of all algorithms to $200 \times NP$. NP is the population size with the value of 30. Each algorithm is run independently and consecutively 10 times to obtain its average. The dimensions and boundary ranges of each benchmark function are shown in Table 1. In addition, it can be seen from Table 1 that the dimensions of the benchmark function are mainly divided into 2-Dim and 10-Dim. Therefore, we analyze the resource occupancy of the six algorithms with or without instruction optimization in 2-Dim and 10-Dim. The experimental data are recorded in Tables 4 and 5, respectively. Table 6 records the execution time of the six algorithms with and without instruction optimization on thirteen benchmark functions. Table 7 records the convergence values of the six algorithms on thirteen benchmark functions.

The "N" in Tables 4 and 5 indicates that no HLS optimization instruction is added to the program. The "Y" indicates that HLS optimization instruction is added to the program. The "Latency" means the delay from an input to an output. As can be seen from the data in the tables, the instruction optimization can reduce the latency of the algorithm. Comparing the "Latency" in the two tables, as the dimension increases, the improvement effect of the instruction optimization on the latency is more obvious. However, the addition of optimization instructions also increases the resource occupancy such as BRAM and LUT. This is inevitable. The reason is that the optimization instructions make the program execute in parallel. This often has the adverse effect of increasing resource occupancy. As described in Section 4.2, we try to use the most reasonable optimizations to reduce this adverse effect. In addition, we encapsulate the judgment and processing of the boundary range of the benchmark function in the IP core of the algorithm. As can be seen from Table 1, we use more 2-Dim functions than 10-Dim functions. And like the six-hump camel function and the bukin n.6 function, the boundaries of each dimension are different. Therefore, the program uses slightly more logic function generators in 2-Dim than in 10-Dim. And the logic function generator belongs to the LUT resource. Therefore, the 2-Dim program uses slightly more LUT resources than the 10-Dim program.

In order to visually observe the resource occupancy of the six algorithms, we draw the data in Tables 4 and 5 into the histograms, as shown in Figs. 12–13. Fig. 12(a) shows the resource occupancy of the six algorithms in 2-Dim without instruction optimization. Fig. 12(b) shows the resource occupancy of the six algorithms in 2-Dim with instruction optimization. Fig. 13(a) shows the resource occupancy of the six algorithms in 10-Dim without instruction optimization. Fig. 13(b) shows the resource occupancy of the six algorithms in 10-Dim with instruction optimization. In addition, we process the data in Tables 4 and 5 to facilitate observation and comparison. The DSP and BRAM occupancy in Figs. 12–13 is magnified by a factor of 100. The "Latency" of the CMAES algorithm in Fig. 13(a) is reduced by half. As can be seen from the figure, the complex CMAES algorithm has the largest delay and resource occupancy. The simple PSO algorithm has the least delay and resource occupancy. Among all the resources occupied, the LUT resources are the most and the BRAM resources are the least. Comparing Fig. 13(a) and Fig. 13(b), the instruction optimization greatly reduces the delay of the GWO algorithm. This shows that the GWO algorithm is suitable for parallel processing.

The meanings of "N" and "Y" in Table 6 are the same as those in Table 4. As can be seen from the data in the table, the instruction optimization can reduce the running time of the algorithm. In particular, the acceleration effect of the algorithm is more obvious on high-dimensional functions, such as the sphere function and the sum squares function. Compared with the simple PSO algorithm, the execution speed of the complex CMAES algorithm is increased by 2–3 times after instruction optimization. Among the six algorithms, the instruction optimization improves the execution time of the CMAES algorithm the most, and improves the execution time of the WOA algorithm the least. This is because the CMAES algorithm is complex. The code of the CMAES algorithm implemented by C++ language contains many loop statements. The instruction optimization can make some programs execute in parallel, thereby increasing the execution speed of the algorithm. The structure of the WOA algorithm is simple, and it requires logical decisions at each stage. Which limits the parallel execution of the program. Therefore, the instruction optimization does not have a significant impact on the

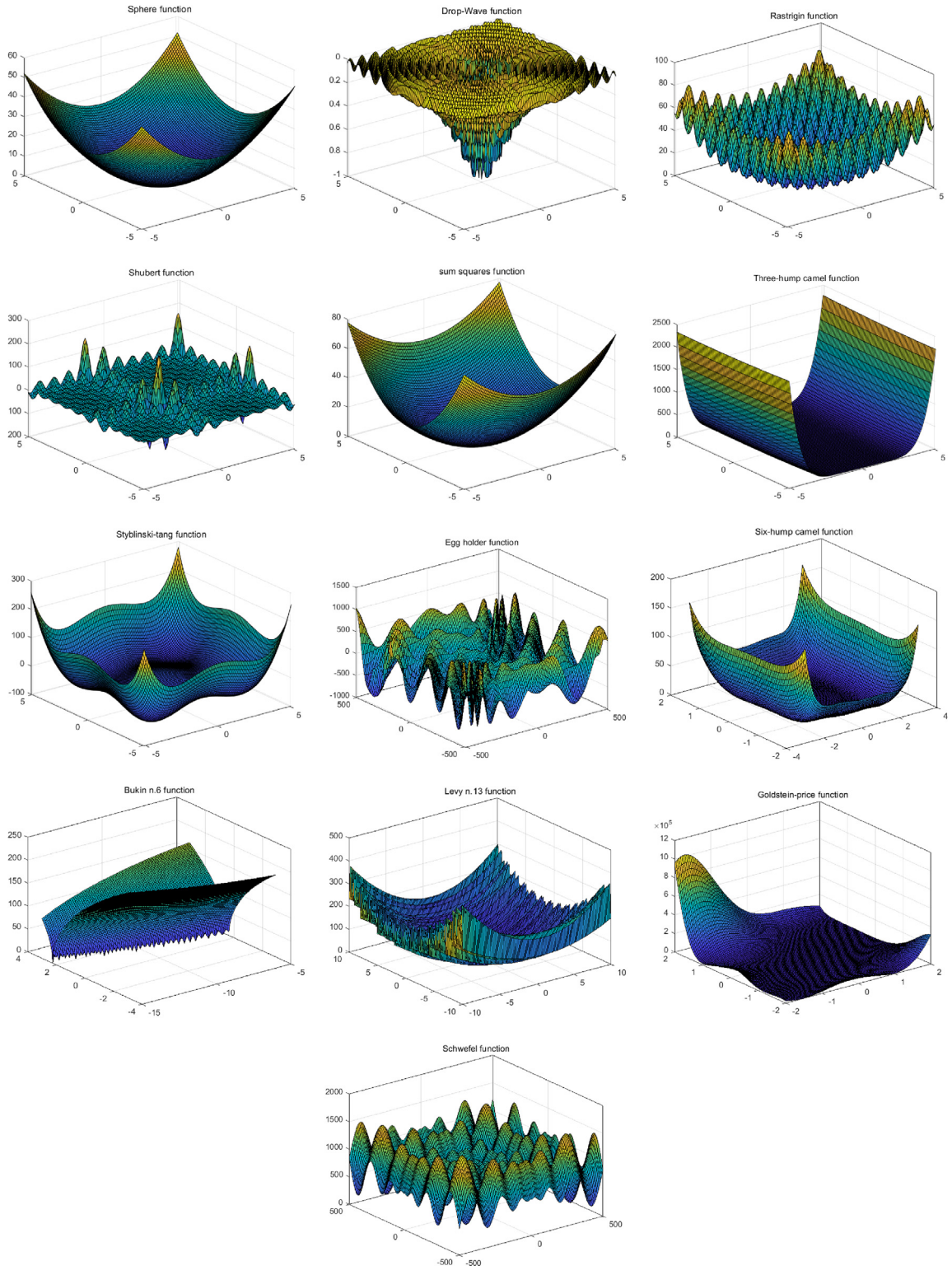

Fig. 11. The 3D image of the benchmark functions.

execution time of the WOA algorithm. In addition, the instruction optimization does not change any parameters of the algorithm. It just makes programs execute in parallel. Therefore, its convergence value does not change whether or not the optimization

Table 4
The resource occupancy and latency of the six algorithms on 2-Dim functions.

	PSO		WOA		BA		GWO		CMAES		EO	
	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y
Latency	3198	2090	3478	2399	4680	3212	2997	1736	10892	4757	3645	2371
BRAM	10	11	13	17	10	11	4	4	20	23	16	16
DSP48E	13	31	115	117	22	26	24	76	178	189	70	112
FF	3131	4933	8649	10012	4150	5183	5630	10512	20097	22250	8433	11981
LUT	6674	8043	18588	18547	7929	8956	11159	14592	35946	37561	17832	20495

Table 5
The resource occupancy and latency of the six algorithms on 10-Dim functions.

	PSO		WOA		BA		GWO		CMAES		EO	
	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y
Latency	14013	4895	14113	3824	18498	6844	13941	1015	122837	48002	14947	3184
BRAM	9	12	13	17	9	12	4	4	24	29	14	15
DSP48E	13	31	115	115	22	26	24	76	178	193	70	112
FF	3237	5917	8577	10408	4141	5973	5494	10567	16604	23581	8426	12038
LUT	6371	9344	17740	17834	7929	9894	10160	13728	29496	34642	16762	19547

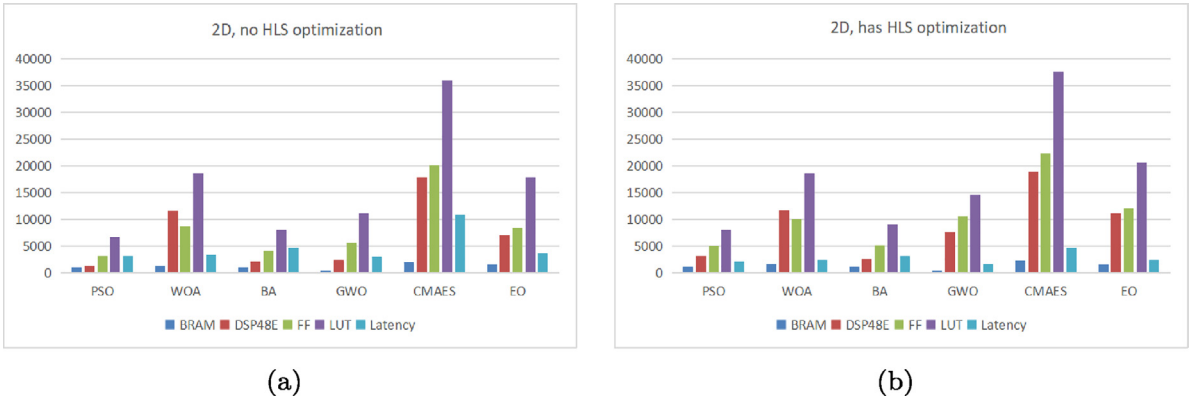

Fig. 12. The histograms of resource occupancy of the six algorithms on 2-Dim functions.

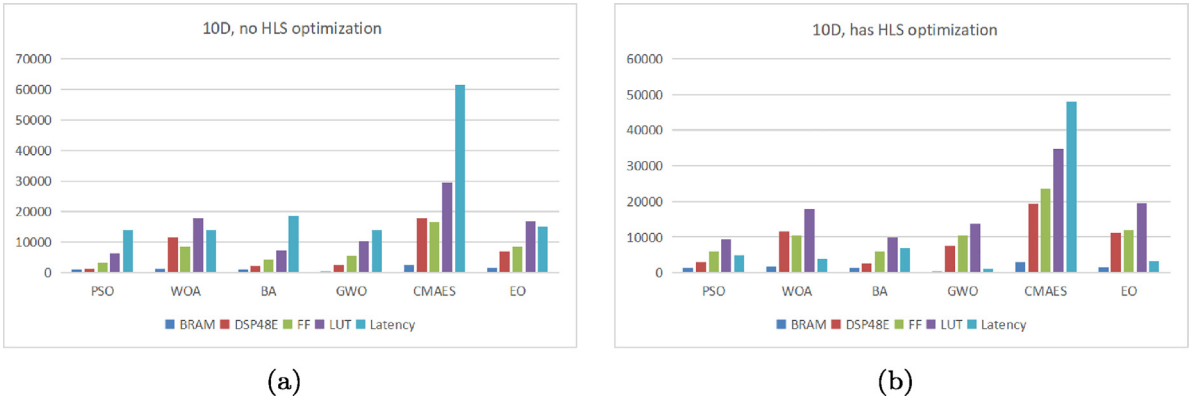

Fig. 13. The histograms of resource occupancy of the six algorithms on 10-Dim functions.

instructions are added to the program. The convergence values of the six algorithms on thirteen benchmark functions are shown in Table 7.

5.3. Comparison with software results

In order to understand the difference between the design in this paper and the software implementation, we compare the experimental results with the software implementation results. The software environment is the ARM with the 666MHz running memory. The settings of the experimental parameters such as the number of evaluations and the population size are the same as those in Section 5.2. Table 8 records the execution time of the software implementation and the implementation of this paper for the six algorithms on the thirteen benchmark functions. Table 9 records the software-implemented convergence values of the six algorithms on thirteen benchmark functions.

In Table 8, the "ARM" represents the execution time of the algorithm on the software. The "FPGA" means the execution time of the algorithm on the scheme of this paper. As can be seen from the data in the table, the six algorithms can achieve

Table 6
The running time (ms) of the six algorithms on thirteen benchmark functions.

	PSO		WOA		BA		GWO		CMAES		EO	
	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y
Drop-Wave	8	6	7	6.4	9	7	7.8	5	16	5.9	9.2	7
Shubert	16	14.2	15.4	15	17	15.2	16.9	14.2	23	13	17.2	15.1
Three-hump camel	13	11	12	11	14	12	7	5	20.9	10	14	12
Egg holder	9	7	8.3	8	10	8	9	6.5	17	6.9	11	8.1
Six-hump camel	10	8	9	9	11	9	10	7	18	8	12	9
Bukin n.6	7	5	7	6	8	6	7	4	15	5	9	6.3
Levy n.13	10	8	9	8	11	9	8.7	6.3	18	7	11.1	9
Goldstein-price	8	6	7	6	9	7	7	5	16	5	9	7
Sphere	32	14	25	12	33	17	32	6	187	65	37	14
Rastrigin	41.4	24.3	33.4	19.8	43	27.4	43.6	17	197.1	74.9	46.1	23.1
Sum squares	31	14	25	11	32.4	17	33.1	6	187	65	36	13
Styblinski-tang	60.5	43	54	41	62	46	33	7	216	94	66.1	43.1
Schwefel	43	25	36	22.4	43.8	28	43	16.3	198	76	48.4	25.4

Table 7
The convergence values of the six algorithms on thirteen benchmark functions.

	PSO	WOA	BA	GWO	CMAES	EO
Drop-Wave	−0.9936	−0.9745	−0.9467	−0.9949	−0.9426	−0.9937
Shubert	−186.73	−186.73	−176.00	−181.15	−186.73	−186.73
Three-hump camel	6.70E−19	8.96E−02	2.57E−32	0	8.45E−40	5.14E−39
Egg holder	−904.90	−925.98	−842.19	−829.29	−617.81	−779.49
Six-hump camel	−1.0316	−1.0316	−1.0293	−1.0247	−1.0316	−1.0316
Bukin n.6	7.88E−02	9.18E−02	0.63159	0.22062	73.3588	8.68E−02
Levy n.13	2.54E−07	2.01E−04	2.54E−07	5.50E−02	2.54E−07	2.54E−07
Goldstein-price	11.0999	3.00169	11.0999	3.21617	3	3
Sphere	6.14E−09	1.92E−33	1.09E−01	3.02E−24	1.39E−29	9.69E−27
Rastrigin	8.95463	2.65325	24.6021	1.36573	14.8249	2.19664
Sum squares	2.30E−08	1.42E−31	1.14768	1.42E−24	3.86E−26	9.88E−26
Styblinski-tang	−339.36	−362.89	−336.20	−300.66	−391.66	−391.65
Schwefel	1786.59	1181.83	1890.89	2524.00	1739.21	1408.32

Table 8
Comparison with the software implementation method on the running time (ms).

	PSO		WOA		BA		GWO		CMAES		EO	
	ARM	FPGA	ARM	FPGA	ARM	FPGA	ARM	FPGA	ARM	FPGA	ARM	FPGA
Drop-Wave	14.6	6	17.4	6.4	15	7	18	5	40.8	5.9	22.3	7
Shubert	22	14.2	25	15	21	15.2	27	14.2	49.1	13	31.1	15.1
Three-hump camel	7	11	9	11	7	12	11	5	34.4	10	15	12
Egg holder	22	7	25	8	22	8	27.2	6.5	50.6	6.9	32.4	8.1
Six-hump camel	7	8	9	9	6	9	11	7	33	8	15	9
Bukin n.6	12.8	5	16	6	13.2	6	18.1	4	36.7	5	22.4	6.3
Levy n.13	11	8	13	8	11	9	15	6.3	39	7	20	9
Goldstein-price	7	6	10	6	7	7	12	5	34.3	5	16	7
Sphere	30	14	36	12	26.7	17	51	6	279	65	64	14
Rastrigin	46.2	24.3	49	19.8	42.7	27.4	63.5	17	284.8	74.9	79.4	23.1
Sum squares	31	14	36.3	11	27	17	52	6	280	65	65	13
Styblinski-tang	32.1	43	38.3	41	29	46	53	7	314.7	94	68	43.1
Schwefel	111.9	25	122	22.4	112	28	134.3	16.3	353	76	146.9	25.4

hardware acceleration on most functions. Only for the three-bump camel function, the execution time of the PSO, WOA and BA algorithms on the "FPGA" is slightly shorter than that on the "ARM". For the schwefel function, all six algorithms can achieve hardware acceleration on the "FPGA". Their execution speed on the "FPGA" is more than four times that on the "ARM". On the 2-Dim functions, the CMAES algorithm has the best acceleration effect. Its execution speed on the "FPGA" is about 3–8 times faster than on the "ARM". On the 10-Dim functions, the EO algorithm has the best acceleration effect. Its execution speed on the "FPGA" is about 3–9 times faster than on the "ARM". On the whole, the acceleration effect of the GWO, CMAES, and EO algorithms implemented by the scheme in this paper is better than that of the PSO, WOA, and BA algorithms. This is because we encapsulate the main structure of the algorithm in the IP core. The inherent parallelism and powerful computing power of the FPGA can accelerate the execution speed of the algorithms. In general, the more complex the algorithm, the more obvious the acceleration effect.

Tables 7 and 9 give the convergence values of the algorithm implemented in this paper and implemented by software, respectively. To facilitate comparison of the data in the two tables, we summarize the results in Table 10. In Table 10, the "0" indicates that the software-implemented convergence value is better. The "1" means that the convergence value implemented in this paper is better. The "=" indicates that the convergence value implemented by the proposed scheme is equal to the convergence value implemented by the software. The "0-num", "1-num" and "=-num" represent the number of "0", "1" and "=" respectively in a column. In terms of convergence values, the CMAES and EO algorithms implemented in this paper are better than those implemented by software in most functions. The convergence values of the BA algorithm implemented in this paper is similar to that of the software implementation. The PSO, WOA and GWO algorithms implemented in this paper are worse than those implemented by software in most functions. We guess that random generators have a greater impact on the algorithms with

Table 9
The convergence values for the six algorithms implemented in software.

	PSO	WOA	BA	GWO	CMAES	EO
Drop-Wave	−1	−0.9617	−0.9212	−1	−0.9362	−1
Shubert	−186.73	−186.73	−186.73	−186.61	−176.00	−186.73
Three-hump camel	1.61E−23	5.97E−02	6.01E−32	0	0	0
Egg holder	−860.47	−943.36	−851.96	−933.23	−564.34	−928.84
Six-hump camel	−1.0316	−1.0316	−1.0316	−1.0316	−1.0316	−1.0316
Bukin n.6	8.16E−02	7.33E−02	5.18E−02	3.61E−01	3.82E−02	1.04E−01
Levy n.13	2.54E−07	9.49E−05	2.19E−02	2.42E−04	2.54E−07	2.54E−07
Goldstein-price	3	3	3	3	3	3
Sphere	4.29E−10	3.40E−32	5.20E−02	1.32E−29	5.77E−29	2.42E−24
Rastrigin	10.2994	2.59127	26.1783	1.47877	12.9345	2.20181
Sum squares	9.60E−10	9.90E−32	6.85E−01	7.11E−29	2.14E−25	4.45E−23
Styblinski-tang	−357.73	−374.86	−329.16	−359.83	−349.25	−360.55
Schwefel	684.365	1008.89	1988.43	1892.44	2339.45	1126.92

Table 10
Comparison with the software implementation method on the convergence values.

	PSO	WOA	BA	GWO	CMAES	EO
Drop-Wave	0	1	1	0	1	0
Shubert	=	=	0	0	1	=
Three-hump camel	0	0	1	=	0	0
Egg holder	1	0	0	0	1	0
Six-hump camel	=	=	0	0	=	=
Bukin n.6	1	0	0	1	0	1
Levy n.13	=	0	1	0	=	=
Goldstein-price	0	0	0	0	=	=
Sphere	0	1	0	0	1	1
Rastrigin	1	0	1	1	0	1
Sum squares	0	0	0	0	1	1
Styblinski-tang	0	0	1	0	1	1
Schwefel	0	0	1	0	1	0
0-num	**7**	**9**	**7**	**10**	**3**	**4**
1-num	**3**	**2**	**6**	**2**	**7**	**5**
=-num	**3**	**2**	**0**	**1**	**3**	**4**

simple structures and strong randomness. This results in their poor convergence accuracy. Comparing Tables 7 and 9, although there are certain differences between the proposed scheme and the software scheme, the difference value between them is not large in most functions.

5.4. Comparison with literature [47]

Literature [47] introduces the graphical implementation of several meta-heuristic algorithms on the LabView FPGA platform. Both we and the literature [47] implement the PSO, BA and GWO algorithms. At the same time, the benchmark functions include seven functions such as the dropwave function and the sphere function. To compare the effects of different design schemes, we conduct experiments on the design schemes in this paper according to the parameters in the literature [47]. The number of evaluations is set to $100 \times NP$. NP is the population size with the value of 15. The dimension is 2. It is known that there are experimental results based on the above parameters in the literature [47]. First, using the sphere function as the test function, we compare the resource occupancy with the literature [47], as shown in Table 11. Secondly, we compare the convergence values of the three algorithms on the seven functions with the literature [47], as shown in Table 12.

The "-" in Table 11 indicates that the platform does not provide the occupancy of this resource. The "Time" is the running time of the algorithm in milliseconds. The "Ref. [47]" represents the experimental data in the literature [47]. The "ours" represents the experimental results of the design scheme in this paper. As can be seen from the table, we occupy more BRAM and DSP48E resources than the literature [47]. However, our occupancy on the LUT resources is much less than in the literature [47]. Overall, our resource occupancy is less than the literature [47]. Furthermore, our results have a significant advantage over the literature [47] in terms of execution time. Especially for the GWO algorithm, our program can be completed within 1 ms, while the literature [47] needs 19.9 ms. This is because our design scheme can maximize the parallel execution of the program, which greatly reduces the running time of the algorithm.

Table 11
Comparison with literature [47] on the resource occupancy and running time.

	PSO		BA		GWO	
	Ref. [37]	Ours	Ref. [37]	Ours	Ref. [37]	Ours
BRAM	3	10	3	10	3	4
DSP48E	17	31	10	26	26	76
FF	–	4950	–	5194	–	10586
Slice registers	40504	–	38993	–	27568	–
LUT	31827	7878	29967	8751	28369	14436
Time (ms)	3.715	1	5.799	1	19.9	1

The bold font in Table 12 indicates the better values between "Ref.[47]" and "ours". The "win" in the last line indicates the number of the winners of "Ref. [47]" or "ours" on the seven benchmark functions. It can be seen from the table that our experimental results on the POS algorithm are slightly worse than the literature [47]. For the BA algorithm, our convergence values are better than the literature [47] on all seven benchmark functions. For the GWO algorithm, our convergence values are better than the literature [47] on the five benchmark functions such as dropwave function and sphere function. Overall, compared with the literature [47], our design scheme can also show some advantages in terms of the convergence value.

6. Conclusion

This paper proposes a software–hardware co-design scheme of the intelligent optimization algorithm. In the design scheme, the update module of the algorithm is executed in parallel on the FPGA. The initialization module and the fitness module are executed sequentially on the ARM. The data between ARM and FPGA is transmitted through AXI bus. This paper implements the PSO, BA, WOA, GWO, CMAES and EO algorithms with the above design scheme. The six algorithms are tested on thirteen benchmark functions. The feasibility and effectiveness of this design scheme are proved by the experiments. It not only facilitates the change of the test function, but also increases the flexibility of the program. Moreover, it also uses the parallelism and powerful computing power of the FPGA platform to speed up the execution of the algorithm. Compared with software and other implementations, the design scheme in this paper shows certain advantages in terms of convergence value, resource occupation and execution

Table 12
Comparison with literature [47] on the convergence values.

	PSO		BA		GWO	
	Ref. [37]	Ours	Ref. [37]	Ours	Ref. [37]	Ours
Drop-Wave	−0.986	**−0.993**	−0.754	**−0.943**	−0.966	**−0.982**
Egg holder	−825.36	**−858.75**	−737.25	**−785.19**	**−759.64**	−758.15
Bukin n.6	0.233	**0.198**	3.287	**1.23**	2.334	**0.452**
Levy n.13	**2.94E−10**	3.09E−07	2.081	**0.022**	**2.21E−03**	0.144
Sphere	**1.17E−11**	1.02E−06	0.919	**1.89E−14**	2.40E−21	**4.80E−23**
Rastrigin	**0.020**	0.100	4.373	**1.691**	0.447	**0.045**
Styblinski-tang	**−77.060**	−76.919	−71.404	**−72.678**	−76.693	**−76.909**
win	4	3	0	7	2	5

time. However, in the experiment, we find that the convergence effect of the simple algorithm on some benchmark functions is poor. Therefore, how to improve the convergence ability of the algorithm in the design scheme will be our next work direction.

Declarations

Not applicable.

Funding

None. No fundings to declare.

CRediT authorship contribution statement

Zonglin Fu: Methodology, Software, Writing – original draft. **Shu-Chuan Chu:** Validation, Writing – editing. **Junzo Watada:** Data curation. **Chia-Cheng Hu:** Writing – editing. **Jeng-Shyang Pan:** Conceptualization, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] F. Tao, Y. Laili, L. Zhang, Brief history and overview of intelligent optimization algorithms, in: Configurable Intelligent Optimization Algorithm, Springer, 2015, pp. 3–33.
[2] W. Li, G.-G. Wang, A.H. Gandomi, A survey of learning-based intelligent optimization algorithms, Arch. Comput. Methods Eng. 28 (5) (2021) 3781–3799.
[3] Z. Beheshti, S.M.H. Shamsuddin, A review of population-based meta-heuristic algorithms, Int. J. Adv. Soft Comput. Appl 5 (1) (2013) 1–35.
[4] A. Chakraborty, A.K. Kar, Swarm intelligence: A review of algorithms, Nat.-Inspir. Comput. Optim. (2017) 475–494.
[5] J.H. Holland, Genetic algorithms, Sci. Am. 267 (1) (1992) 66–73.
[6] C. Blum, Ant colony optimization: Introduction and recent trends, Phys. Life Rev. 2 (4) (2005) 353–373.
[7] P.J. Van Laarhoven, E.H. Aarts, Simulated annealing, in: Simulated Annealing: Theory and Applications, Springer, 1987, pp. 7–15.
[8] S. Mirjalili, SCA: a sine cosine algorithm for solving optimization problems, Knowl.-Based Syst. 96 (2016) 120–133.
[9] D. Dasgupta, Artificial Immune Systems and their Applications, Springer Science & Business Media, 2012.
[10] A.S. Joshi, O. Kulkarni, G.M. Kakandikar, V.M. Nandedkar, Cuckoo search optimization-a review, Mater. Today: Proc. 4 (8) (2017) 7262–7269.
[11] J.-S. Pan, N. Liu, S.-C. Chu, T. Lai, An efficient surrogate-assisted hybrid optimization algorithm for expensive optimization problems, Inform. Sci. 561 (2021) 304–325.
[12] L. Lv, X.-D. Zhou, P. Kang, X.-F. Fu, X.-M. Tian, Multi-objective firefly algorithm with hierarchical learning, J. Netw. Intell. 6 (3) (2021) 411–427.
[13] J.-S. Pan, T.-K. Dao, T.-S. Pan, T.-T. Nguyen, S.-C. Chu, J.F. Roddick, An improvement of flower pollination algorithm for node localization optimization in wsn, J. Inf. Hiding Multim. Signal Process. 8 (2) (2017) 486–499.
[14] C.-W. Tsai, P.-W. Tsai, J.-S. Pan, H.-C. Chao, Metaheuristics for the deployment problem of WSN: A review, Microprocess. Microsyst. 39 (8) (2015) 1305–1317.
[15] P. Raja, S. Pugazhenthi, Optimal path planning of mobile robots: A review, Int. J. Phys. Sci. 7 (9) (2012) 1314–1320.
[16] J. Tian, W. Yu, S. Xie, An ant colony optimization algorithm for image edge detection, in: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), IEEE, 2008, pp. 751–756.
[17] M.N.A. Wahab, S. Nefti-Meziani, A. Atyabi, A comprehensive review of swarm optimization algorithms, PLoS One 10 (5) (2015) 1–36.
[18] J.R. Woodward, J.R. Neil, No free lunch, program induction and combinatorial problems, in: European Conference on Genetic Programming, Springer, 2003, pp. 475–484.
[19] Z. Cao, Y. Shi, X. Rong, B. Liu, Z. Du, B. Yang, Random grouping brain storm optimization algorithm with a new dynamically changing step size, in: International Conference in Swarm Intelligence, Springer, 2015, pp. 357–364.
[20] A.A. Ewees, M. Abd Elaziz, E.H. Houssein, Improved grasshopper optimization algorithm using opposition-based learning, Expert Syst. Appl. 112 (2018) 156–172.
[21] G.R. Harik, F.G. Lobo, D.E. Goldberg, The compact genetic algorithm, IEEE Trans. Evol. Comput. 3 (4) (1999) 287–297.
[22] J.-S. Pan, P. Hu, S.-C. Chu, Binary fish migration optimization for solving unit commitment, Energy 226 (2021) 120329.
[23] P. Hu, J.-S. Pan, S.-C. Chu, Improved binary grey wolf optimizer and its application for feature selection, Knowl.-Based Syst. 195 (2020) 105746.
[24] M.H. Sulaiman, Z. Mustaffa, M.R. Mohamed, O. Aliman, Using the gray wolf optimizer for solving optimal reactive power dispatch problem, Appl. Soft Comput. 32 (2015) 286–292.
[25] N.H. Awad, M.Z. Ali, P.N. Suganthan, R.G. Reynolds, An ensemble sinusoidal parameter adaptation incorporated with L-SHADE for solving CEC2014 benchmark problems, in: 2016 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2016, pp. 2958–2965.
[26] A.A. Heidari, P. Pahlavani, An efficient modified grey wolf optimizer with Lévy flight for optimization tasks, Appl. Soft Comput. 60 (2017) 115–134.
[27] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet Things J. 3 (5) (2016) 637–646.
[28] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, D.S. Nikolopoulos, Challenges and opportunities in edge computing, in: 2016 IEEE International Conference on Smart Cloud, SmartCloud, IEEE, 2016, pp. 20–26.
[29] J. Zhu, B. Du, Image encryption algorithm based on chaos and its implementation on fpga., J. Inf. Hiding Multim. Signal Process. 10 (2) (2019) 278–288.
[30] S.D. Scott, A. Samal, S. Seth, HGA: A hardware-based genetic algorithm, in: Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays, 1995, pp. 53–59.
[31] C.-F. Juang, C.-M. Lu, C. Lo, C.-Y. Wang, Ant colony optimization algorithm for fuzzy controller design and its FPGA implementation, IEEE Trans. Ind. Electron. 55 (3) (2008) 1453–1462.
[32] S.M. Trimberger, Field-Programmable Gate Array Technology, Springer Science & Business Media, 2012.
[33] S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic, Field-programmable gate arrays, Vol. 180, Springer Science & Business Media, 1992.
[34] T.J. Thompson, An overview of microprocessor central processing units (CPUs), Edu. Technol. 19 (10) (1979) 41–44.
[35] A. Cano, A survey on graphic processing unit computing for large-scale data mining, Wiley Interdisc. Rev.: Data Min. Knowl. Discov. 8 (1) (2018) e1232.
[36] M.J.S. Smith, Application-Specific Integrated Circuits, Vol. 7, Addison-Wesley Reading, MA, 1997.
[37] T. Feist, Vivado design suite, White Pap. 5 (2012) 30.

[38] Y. Shi, et al., Particle swarm optimization: developments, applications and resources, in: Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546), Vol. 1, IEEE, 2001, pp. 81–86.
[39] J. Wu, M. Xu, F.-F. Liu, M. Huang, L. Ma, Z.-M. Lu, Solar wireless sensor network routing algorithm based on multi-objective particle swarm optimization, J. Inf. Hiding Multim. Signal Process. 12 (1) (2021) 1–11.
[40] X.-S. Yang, X. He, Bat algorithm: literature review and applications, Int. J. Bio-Insp. Comput. 5 (3) (2013) 141–149.
[41] F.S. Gharehchopogh, H. Gholizadeh, A comprehensive survey: Whale optimization algorithm and its applications, Swarm Evol. Comput. 48 (2019) 1–24.
[42] S. Mirjalili, S.M. Mirjalili, A. Lewis, Grey wolf optimizer, Adv. Eng. Softw. 69 (2014) 46–61.
[43] N. Hansen, S.D. Müller, P. Koumoutsakos, Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES), Evol. Comput. 11 (1) (2003) 1–18.
[44] A. Faramarzi, M. Heidarinejad, B. Stephens, S. Mirjalili, Equilibrium optimizer: A novel optimization algorithm, Knowl.-Based Syst. 191 (2020) 105190.
[45] M. Molga, C. Smutnicki, Test functions for optimization needs, Test Funct. Optim. Needs 101 (2005) 48.
[46] M.M. Ali, C. Khompatraporn, Z.B. Zabinsky, A numerical evaluation of several stochastic algorithms on selected continuous global optimization test problems, J. Global Optim. 31 (4) (2005) 635–672.
[47] A. Ortiz, E. Mendez, D. Balderas, P. Ponce, I. Macias, A. Molina, Hardware implementation of metaheuristics through LabVIEW FPGA, Appl. Soft Comput. 113 (2021) 107908.
[48] B. Scheuermann, K. So, M. Guntsch, M. Middendorf, O. Diessel, H. El-Gindy, H. Schmeck, FPGA implementation of population-based ant colony optimization, Appl. Soft Comput. 4 (3) (2004) 303–322.
[49] A. Hassanein, M. El-Abd, I. Damaj, H.U. Rehman, Parallel hardware implementation of the brain storm optimization algorithm using FPGAs, Microprocess. Microsyst. 74 (2020) 103005.
[50] H. Sadeeq, A.M. Abdulazeez, Hardware implementation of firefly optimization algorithm using FPGAs, in: 2018 International Conference on Advanced Science and Engineering, ICOASE, IEEE, 2018, pp. 30–35.
[51] L. Urbina, C.A. Duchanoy, M.A. Moreno-Armendáriz, D. Lara, H. Calvo, Implementación sobre FPGA de la estrategia evolutiva CMA-ES para optimización numérica, Res. Comput. Sci. 105 (2015) 97–106.
[52] A.L. Da Costa, C.A. Silva, M.F. Torquato, M.A. Fernandes, Parallel implementation of particle swarm optimization on fpga, IEEE Trans. Circ. Syst. II: Express Briefs 66 (11) (2019) 1875–1879.
[53] M.A. Cavuslu, C. Karakuzu, F. Karakaya, Neural identification of dynamic systems on FPGA with improved PSO learning, Appl. Soft Comput. 12 (9) (2012) 2707–2718.
[54] M.S. Ben Ameur, A. Sakly, FPGA implementation of parallel particle swarm optimization algorithm and compared with genetic algorithm, Int. J. Adv. Comput. Sci. Appl. 7 (8) (2016) 57–64.
[55] M.S.B. Ameur, A. Sakly, FPGA based hardware implementation of bat algorithm, Appl. Soft Comput. 58 (2017) 378–387.
[56] M.S.B. Ameur, A. Sakly, A. Mtibaa, A hardware optimization of BAT algorithms implemented on FPGA, in: 2015 16th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering, STA, IEEE, 2015, pp. 146–151.
[57] Q. Jiang, Y. Guo, Z. Yang, X. Zhou, A parallel whale optimization algorithm and its implementation on FPGA, in: 2020 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2020, pp. 1–8.
[58] Q. Jiang, Y. Guo, Z. Yang, Z. Wang, D. Yang, X. Zhou, Improving the performance of whale optimization algorithm through OpenCL-based FPGA accelerator, Complexity 2020 (2020).