Knowledge-Based Systems 276 (2023) 110774

Contents lists available at ScienceDirect

Knowledge-Based Systems

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

FPGA implementation of QUasi-Affine TRansformation evolutionary algorithm

Jeng-Shyang Pan [a,b], Qingyong Yang [a], Jyh-Horng Chou [c,d], Chia-Cheng Hu [e], Shu-Chuan Chu [a,*]

[a] College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China
[b] Department of Information Management, Chaoyang University of Technology, Taichung, Taiwan
[c] Department of Healthcare Administration and Medical Informatics, Kaohsiung Medical University, Kaohsiung 807, Taiwan
[d] Department of Mechanical and Computer-Aided Engineering, Feng-Chia University, Taichung 407, Taiwan
[e] College of Artificial Intelligence, Yango University, Fuzhou, 330015, China

ARTICLE INFO

Article history:
Received 19 April 2022
Received in revised form 27 June 2023
Accepted 30 June 2023
Available online 6 July 2023

Keywords:
QUasi-Affine TRansformation evolutionary algorithm
Field-programmable gate array
Vivado high-level synthesis
Hardware implementation

ABSTRACT

The QUasi-Affine TRansformation Evolutionary (QUATRE) algorithm, a new intelligence optimization algorithm, has been widely used in many optimization fields. In this paper, a hardware-based QUATRE algorithm is designed and implemented on a field-programmable gate array (FPGA). To facilitate the implementation of the QUATRE algorithm on hardware, this paper simplifies the co-evolutionary matrix generation process. Compared with the original QUATRE algorithm, the simplified QUATRE algorithm may reduce latency and resource occupation. The Vivado High-Level Synthesis (HLS) design tool is used to complete the IP core design of the QUATRE algorithm. Through the benchmark function test under different population sizes, compared with the QUATRE algorithm implemented by software, both the running speed and optimization performance of the QUATRE algorithm implemented by hardware are significantly better than the former. Compared with the GA, DE, and PSO algorithms implemented by hardware, the QUATRE algorithm also shows strong competitiveness.

1. Introduction

"Optimization problems" exist in various fields. The so-called "optimization problem" refers to finding the most suitable solution (a set of suitable variables) in the optimization space that meets the constraints so that the goal of the solution reaches the optimal value. According to the search space, the "optimization problem" is divided into a function optimization problem and a combinatorial optimization problem [1]. Between them, the solution space corresponding to the function optimization problem is continuous [2]. The engineering optimization problem [3] is a typical function optimization problem. The solution space corresponding to the combinatorial optimization problem is discrete. Common combinatorial optimization problems include the traveling salesman problem (TSP), job-shop scheduling problem (JSSP), bin-packing problem, knapsack problem, graph coloring problem, transportation problem, etc.

The solution of optimization problems has always been a hot topic in academia and industry. With the deepening of research, many optimization algorithms have been proposed. These optimization algorithms can be roughly divided into two categories: exact algorithms [4] and approximate algorithms (also called heuristic algorithms) [5]. In the initial research, it was mainly based on exact algorithms, such as linear programming, integer programming, and dynamic programming. However, the computational complexity of this type of algorithm is usually very large, and it is only suitable for solving smaller-scale problems. With the increasing scale and complexity of optimization problems, these exact algorithms will be very difficult and time-consuming to process. Although the heuristic algorithm cannot guarantee that the ideal optimal value can be obtained every time, the complexity of the algorithm design is relatively low, and it can give an optimal solution in an acceptable time. Therefore, how to construct a high-quality heuristic solution algorithm has become the focus of subsequent research. At present, many heuristic algorithms have been proposed, which can be simply divided into the following three categories: simple heuristic algorithms, meta-heuristic algorithms [6] and hyper-heuristic algorithms [7]. Among the algorithms, meta-heuristic algorithms (sometimes called intelligent optimization algorithms) are important. After years of research and development, many meta-heuristic algorithms have been proposed. The inspirations for these algorithm designs come

* Corresponding author.
E-mail addresses: jspan@cc.kuas.edu.tw (J.-S. Pan), yang_qy@sdust.edu.cn (Q. Yang), choujh@nkust.edu.tw (J.-H. Chou), cchu.chiachenghu@gmail.com (C.-C. Hu), scchu0803@sdust.edu.cn (S.-C. Chu).

https://doi.org/10.1016/j.knosys.2023.110774
0950-7051/

from various biological, chemical, physical and other phenomena in nature and humanistic society [8]. The simulated annealing (SA) algorithm [9], genetic algorithm (GA) [10], tabu search (TS) [11], neural network (NN) [12], particle swarm optimization (PSO) [13], ant colony optimization (ACO) [14], agent heroes and cowards algorithm (AHC) [15], gradient-based optimizer (GBO) [16], archimedes optimization algorithm (AOA) [17], groundwater flow algorithm (GWFA) [18], etc., belong to the category of meta-heuristic algorithms. At present, meta-heuristic algorithms have been successfully applied to solve optimization problems in various fields [19–21].

According to the number of agents contained in the meta-heuristic algorithm, it can be divided into individual-based meta-heuristic algorithms and population-based meta-heuristic algorithms. Individual-based meta-heuristic algorithms include the SA algorithm [9] and the TS [11]. Population-based intelligent optimization algorithms can continue to be divided into evolutionary algorithms and swarm intelligence optimization algorithms. Common evolutionary algorithms include the GA [10], differential evolution (DE) [22], etc. There are many types of swarm intelligence optimization algorithms [23], such as the PSO [13], ACO [14], the artificial bee colony (ABC) algorithm [24], the bat algorithm (BA) [25], and the grey wolf optimizer (GWO) [26,27]. The QUasi-Affine TRansformation Evolutionary (QUATRE) algorithm [28] is a new evolutionary algorithm proposed by Meng et al. in 2016. The algorithm has the characteristics of few parameter setting and fast convergence speed, and shows strong optimization ability. To further improve the optimization ability of the QUATRE algorithm, some literatures have been improved by introducing different algorithm strategies [29–32]. At present, the QUATRE algorithm and its improved version have been successfully applied to solve practical application problems in different fields, such as function optimization [29,33,34], wireless sensor networks [31], image processing [30], transportation [35] and other fields [32].

However, the above algorithms, including the QUATRE algorithm, are usually implemented on computer software to solve different optimization problems. Compared with the traditional optimization algorithm, although the meta-heuristic algorithm can still find an approximate solution in the face of high-dimensional complex optimization problems, its solution efficiency is usually low on the software platform. This is due to the sequential execution structure of the software platform [36,37]. When faced with some complex problems in real life (such as massive data processing in smart cities, automatic driving, industrial control), this sequential execution structure will seriously restrict the execution speed and computational efficiency of the algorithm. Deploying the algorithm to the hardware platform can improve its computation speed and efficiency with the help of the parallel processing capability of the hardware platform, thus significantly reducing the time and cost required for problem processing [37]. Common hardware platforms include graphic processing units (GPUs) [38] and field-programmable gate arrays (FPGAs). Compared with the GPU platform, the FPGA platform has the advantages of low power consumption, low cost and flexible configuration, which is more suitable for the execution of edge computing tasks.

Through reviewing relevant literature, the QUATRE algorithm and its variants are all implemented by MATLAB software. Therefore, this paper will also be the first hardware implementation of the QUATRE algorithm. To facilitate the hardware implementation of the algorithm, this paper simplifies part of the process of the QUATRE algorithm. In this paper, the Vivado HLS design tool is used to complete the loop optimization of the simplified QUATRE algorithm and FPGA IP core design. The algorithm is finally deployed on the Xilinx ZYNQ 7020 FPGA platform. Through testing, the running speed and optimization performance of the hardware-implemented QUATRE algorithm to solve the benchmark functions are better than those of the software-implemented version. Compared with the GA,DE and PSO algorithms implemented by hardware, the QUATRE algorithm also has strong competitiveness. The contributions made in this paper are as follows:

1. The hardware implementation of the QUATRE algorithm is carried out for the first time and successfully deployed on FPGA.
2. To facilitate the hardware implementation of the QUATRE algorithm, two simplified versions of co-evolutionary matrix generation methods are proposed.
3. Through the control signal and data flow, the connection and interaction between the modules of the hardware-implemented QUATRE algorithm are realized. A reasonable loop optimization instruction adding rule is formulated.
4. Six benchmark functions are used to evaluate the operating efficiency and solution performance of the hardware-implemented QUATRE algorithm.

The remaining sections of this paper are arranged as follows: Section 2 introduces some intelligent optimization algorithms that have been deployed on FPGAs. The principles of the QUATRE algorithm and the introduction of the HLS platform are presented in Section 3. Section 4 describes the simplified process of the QUATRE algorithm and the hardware implementation process. Section 5 shows and discusses the resource occupancy and the experimental results of the algorithm. Section 6 is the summary and prospects of this paper.

2. Related works

At present, some intelligent optimization algorithms have been deployed and implemented on FPGA platforms. By reviewing the relevant literature, the intelligent optimization algorithms that have been implemented include the genetic algorithm (GA) [39], differential evolution (DE) [40], particle swarm optimization (PSO) [41–44], ant colony optimization (ACO) [45], brain storm optimization (BSO) [46], bat algorithm (BA) [42,47], firefly algorithm (FA) [48], cuckoo search algorithm (CSA) [49], whale optimization algorithm (WOA) [50], grey wolf optimizer (GWO) [42] and artificial bee colony (ABC) algorithm [51].

The hardware implementation architecture of the GA algorithm on FPGA is given in Ref. [39]. The updating process of the algorithm and the individual fitness evaluation part are designed according to modularity, so that they can be called asynchronously. To improve the generality of the proposed architecture, the parameters contained in the algorithm are all customizable. According to the needs of the problem, the required parameters can be adjusted and the appropriate modules can be selected. The whole process of the algorithm is designed and developed using the VHDL design language. Through the test of five benchmark functions and the TSP problem, the generality of the proposed architecture is proven.

In Ref. [40], the author designs the hardware implementation architecture of the DE algorithm using the VHDL language, and successfully deploys it on FPGA. To improve the applicability, the paper uses a double-precision floating-point format to represent variable data. Through the test of five benchmark functions, the optimization ability of the DE algorithm implemented by hardware is verified.

The hardware implementation process of the PSO algorithm on FPGA is given in Ref. [41–44]. In Ref. [41], the authors use a finite state machine (FSM) to reasonably schedule each execution process of the algorithm and parallelize the particle velocity and position update process in the PSO algorithm. Ref. [42] uses LabVIEW software to implement the PSO algorithm in hardware

and successfully deploys it on FPGA. Compared with the software-implemented PSO algorithm, the hardware-implemented PSO algorithm improves the running speed by approximately 67.83 times. It is worth noting that Refs. [41,42] fully deploy the entire algorithm on FPGA. Refs. [43,44] adopt the idea of software–hardware co-design. The software part performs the overall scheduling of the algorithm (such as the control of the iterative process). The hardware part parallelizes the repeated processes to improve the running speed of the algorithm. Through experiments, the PSO algorithm architecture designed by software–hardware co-design is also faster than the PSO algorithm implemented by pure software.

In addition to the implementation of the PSO algorithm, Ref. [42] also provides hardware design solutions for the BA algorithm, GWO algorithm, earthquake algorithm (EA), and Nelder–Mead (NM) algorithm. Ref. [47] also gives the same hardware implementation of the BA algorithm, but uses the VHDL design language. Refs. [48–50] give the hardware implementation process of the FA algorithm, CSA algorithm and WOA algorithm, respectively. Ref. [50] uses the OpenCL language and designs an FPGA acceleration framework for the heterogeneous parallel WOA algorithm. Both Ref. [48,49] use the VHDL design language for algorithm design.

In the above mentioned literature, the hardware implementations of the original version of the algorithm are carried out. Refs. [45,46,51] first simplified or improved the algorithm before hardware implementation and deployment. Ref. [45] considers that it is very difficult to deploy the original ACO algorithm directly on FPGA. Therefore, a population-based ACO algorithm is proposed for FPGA hardware implementation. The random group BSO (RGBSO) algorithm is selected for hardware implementation on FPGA in Ref. [46]. Compared with the original BSO algorithm, the RGBSO algorithm has lower complexity and is easier to implement in hardware. Ref. [51] first introduces the opposite learning strategy to improve the ABC algorithm. Then the improved ABC algorithm is implemented in hardware.

To facilitate hardware implementation, this paper first simplifies the process of the QUATRE algorithm. For the hardware design of the algorithm, the Vivado HLS tool is used in this paper to complete the loop optimization of the algorithm and the export of the IP core. In the next section, this paper will introduce the principle of the original QUATRE algorithm and the HLS platform.

3. Basic algorithm and platform introduction

3.1. QUasi-affine Transform Evolutionary algorithm

The QUasi-Affine TRansformation Evolutionary (QUATRE) algorithm [28] is a new intelligent optimization algorithm proposed by Meng et al. in 2016. It combines the advantages of the PSO algorithm and the DE algorithm, and has the characteristics of fast convergence speed and few parameters. The evolutionary formula of the QUATRE algorithm is shown in Eq. (1).

$$X = \bar{M} \bigotimes X_{pbest} + M \bigotimes B \tag{1}$$

where X and X_{pbest} are both $ps*D$ matrices that are used to represent the updated population and the individual historical optimal solution of the population, respectively. ps is the size of the population and D is the dimension of the problem to be solved. The symbol $\bigotimes$ represents the multiplication operation of the corresponding elements of the matrix. M is the co-evolutionary matrix, and $\bar{M}$ (obtained by inverting the elements in the matrix M) is the correlation matrix of M. B is the evolutionary guidance matrix. The matrices M, $\bar{M}$ and B have the same size of $ps * D$. Table 1 lists eight ways to generate the evolutionary guidance matrix B in the QUATRE algorithm.

Table 1
The parameter settings of related algorithms.

No.	QUATRE/x/y	Equation
1	QUATRE/best/1	$B = X_{gbest} + F \cdot (X_{r_1} - X_{r_2})$
2	QUATRE/rand/1	$B = X_{r_0} + F \cdot (X_{r_1} - X_{r_2})$
3	QUATRE/target/1	$B = X + F \cdot (X_{r_1} - X_{r_2})$
4	QUATRE/target-to-rand/1	$B = X + F \cdot (X_{r_0} - X) + F \cdot (X_{r_1} - X_{r_2})$
5	QUATRE/target-to-best/1	$B = X + F \cdot (X_{gbest} - X) + F \cdot (X_{r_1} - X_{r_2})$
6	QUATRE/target/2	$B = X + F \cdot (X_{r_1} - X_{r_2}) + F \cdot (X_{r_3} - X_{r_4})$
7	QUATRE/rand/2	$B = X_{r_0} + F \cdot (X_{r_1} - X_{r_2}) + F \cdot (X_{r_3} - X_{r_4})$
8	QUATRE/best/2	$B = X_{gbest} + F \cdot (X_{r_1} - X_{r_2}) + F \cdot (X_{r_3} - X_{r_4})$

In Table 1, r_0, r_1, r_2, r_3 and r_4 are individual indices generated by random total permutation. X_{gbest} represents the optimal individual of the current population. F is the scaling factor, and the default value is 0.7. By default, the QUATRE algorithm uses "QUATRE/best/1" to generate matrix B. This paper will also adopt this generation method to implement the QUATRE algorithm on hardware.

The co-evolutionary matrix M is used to implement the crossover operation between parents and offsprings. It is transformed from a lower triangular matrix M_{init} with element 1 by two random full permutations. The first time is to randomly and fully arrange the row elements of all rows, and the second time is to randomly and fully arrange all rows. Eq. (2) gives an example of transformation. When the ps scale satisfies $ps = s \times D + k$ and $ps\%D = k$, the first $s \times D$ rows of the M_{init} matrix are stacked by the lower triangular matrix with the size $D \times D$. The last k rows are the first k rows of the lower triangular matrix with a size of $D \times D$.

$$M_{init} = \begin{bmatrix} 1 & & & \\ 1 & 1 & & \\ & & \cdots & \\ 1 & 1 & \cdots & 1 \\ 1 & & & \\ 1 & 1 & & \\ & & \cdots & \\ 1 & 1 & \cdots & 1 \\ & & \vdots & \\ 1 & & & \\ 1 & 1 & & \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & & \\ & & \cdots & \\ 1 & 1 & \cdots & 1 \\ & & 1 & \\ & \cdots & & \\ 1 & & & \\ 1 & & \cdots & 1 \\ & 1 & \cdots & 1 \\ & & \vdots & \\ & & & 1 \\ 1 & \cdots & 1 & \end{bmatrix} = M \tag{2}$$

Algorithm 1 describes the pseudocode of the QUATRE algorithm.

Algorithm 1 The QUATRE algorithm

Input: $f(x)$: objective function; ps: population size; D: problem dimension; Max_gen: maximum number of iterations.
Output: Optimal X_{gbest} and optimal value $gbest val$.

```
Initialize population matrix X and calculate the fitness value val using f(X)
Set X_pbest = X, pbest val = val
Set gbest val = min(val) and X_gbest=X_index(min(val))
while iter < Max_gen do
    Generate the co-evolutionary matrix M
    Generate evolutionary guidance matrix B using "QUATRE/best/1"
    Update population matrix X using Eq. (1)
    Boundary detection
    Update X_pbest and pbest val
    Update X_gbest and gbest val
    iter = iter + 1
end while
```

Notably, the above QUATRE algorithm flow is very easy to implement in the MATLAB software environment. For example,

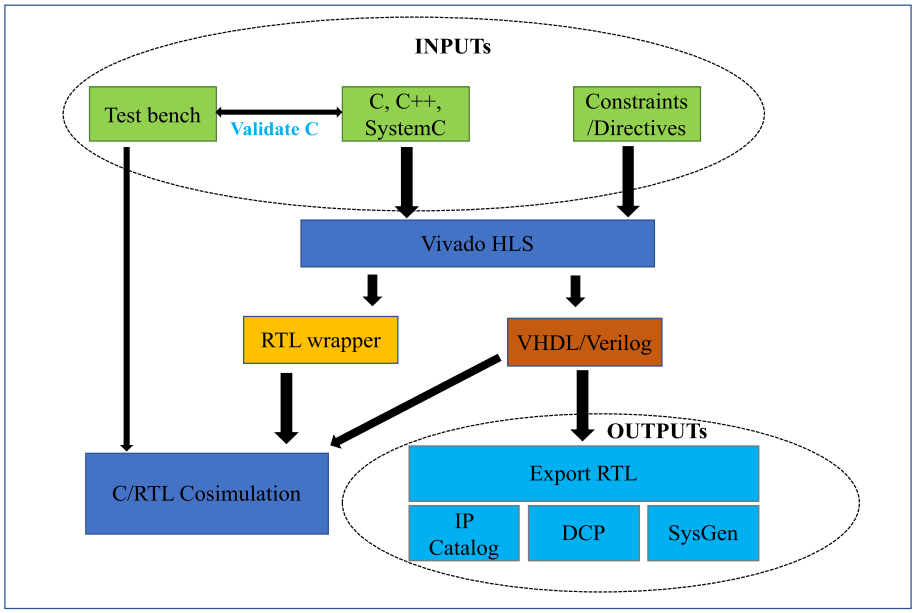


Fig. 1. Vivado HLS design flow.

the generation of co-evolutionary matrix M depends on operations such as "tril", "randperm" and copying of arrays. The "tril" operation is used for the generation of the lower triangular matrix M_{init}, and the "randperm" operation is used for the two random full permutations of the matrix M_{init} to generate the co-evolutionary matrix M. However, in the hardware environment, these libraries do not exist. These operations need to rely on a large number of loops to complete the generation of matrix M, and occupy more hardware resources. Therefore, it is necessary to properly simplify the QUATRE algorithm for easy implementation and deployment in hardware environments. In Section 4, we will simplify the QUATRE algorithm and use the Vivado HLS tool to design its IP core.

3.2. Vivado High-Level Synthesis

Vivado High-Level Synthesis (HLS) [52] is a high-level comprehensive tool introduced by the Xilinx Company. It can directly use C, C++ or SystemC to program Xilinx series FPGA and synthesize C code into RTL-level HDL or Verilog description. The time required for FPGA development using traditional RTL descriptions (such as Verilog or VHDL) is reduced, and the development difficulty of FPGA is also greatly reduced. The Fig. 1 shows the general design process when using Vivado HLS tools for FPGA development.

As shown in Fig. 1, the Vivado HLS tool will eventually export an IP core with user function definition. The IP core can continue to be added to other Xilinx design tools (such as Vivado and System Generator). Vivado HLS also provides developers with some instructions to optimize the design of applications. Common optimization instructions are as follows:

- *Pipeline*: Pipeline the code under the loop.
- *Unroll*: Unroll the loop body.
- *Merge*: Realize the merger between loops.
- *Array Partition*: Realize matrix segmentation.

The Vivado HLS tool also includes different interface types, such as RAM interfaces, FIFO interfaces, and BUS-type interfaces. In the design process, users can specify the appropriate interface type according to their functional requirements. At the same time, the tool supports user-defined data types. In the hardware implementation of the QUATRE algorithm, this platform will be used for the design of the algorithm IP core in this paper. The details of the implementation will be described in the next section.

4. Simplification and hardware implementation of the QUATRE algorithm

4.1. Basic design idea

To facilitate the successful implementation of the QUATRE algorithm on hardware and reduce the difficulty of implementation. In this section, first, some processes in the QUATRE algorithm are simplified, specifically the generation method of the co-evolutionary matrix M. Second, according to the algorithm execution process, the hardware implementation architecture diagram of the QUATRE algorithm is given. Based on the HLS platform, each module included in the QUATRE algorithm is designed and implemented. Finally, the cycle part contained in the algorithm is optimized by designing the optimization instruction adding rules.

4.2. Simplification of the QUATRE algorithm

In this subsection, some of the processes in the QUATRE algorithm are simplified. We mainly simplify the generation of co-evolutionary matrix M in the QUATRE algorithm. In the original QUATRE algorithm, the purpose of two random full permutations is to generate the intersection between each offspring and the parent. In other words, the position where the element value is 1 in the matrix M needs to be updated, and the position with the value of 0 is inherited from the parent. After two transformations, each row of elements in the matrix M is random. Therefore, to facilitate implementation, we remove the generation process of the lower triangular matrix M_{init}. In turn, the matrix M is generated by randomly generating the number and position of each row with a value of 1.

We use *cnt_one* to represent the number of elements with a value of 1 in each row. In this paper, two methods can be used for *cnt_one* generation.

- $cnt\_one = randperm(round(D/2), 1)$
- $cnt\_one = floor(one\_sum/ps)$

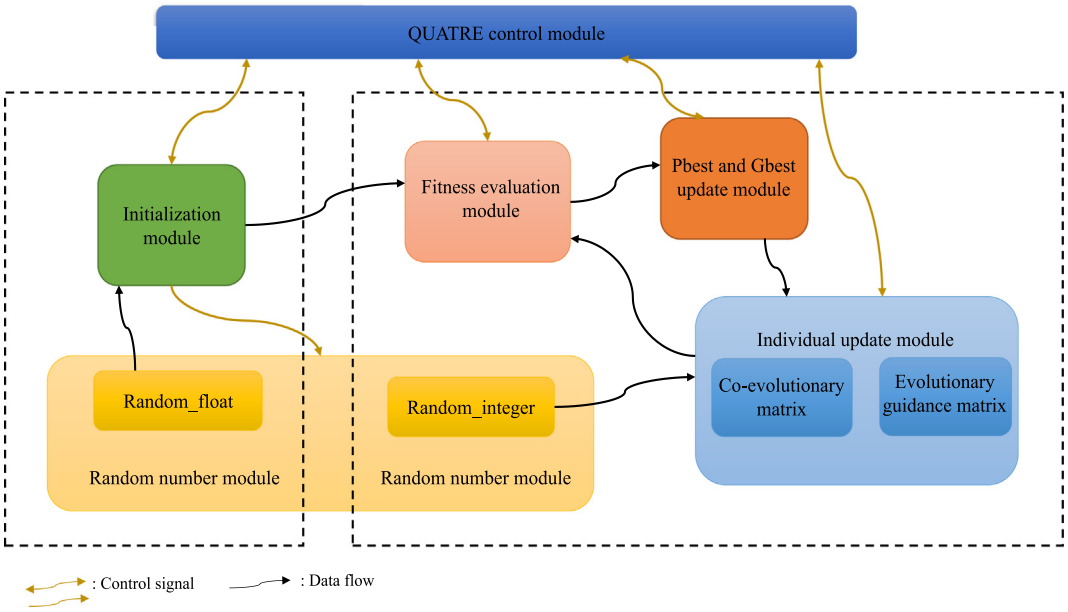


Fig. 2. The hardware implementation architecture of the QUATRE algorithm.

- $one\_sum = K * (D * (D + 1)/2) + mod\_cnt * (mod\_cnt + 1)/2$
- $K = floor(ps/D)$
- $mod\_cnt = mod(ps, D)$

The value cnt_one in the first method is random, and the number of elements in each row is a random number belonging to the interval $[1, round(D/2)]$. In the second method, the value cnt_one in each row is determined and can be calculated by the above formulas. one_sum corresponds to the total number of elements of matrix M with a value of 1 in the original QUATRE algorithm. Here, we assume that $K * D = ps$. The matrix M generation process of the original QUATRE algorithm takes approximately $(13 * ps * D/2 + 5 * ps/2 + D)$ cycles. However, in the method proposed in this paper, the total number of cycles is only $ps * (D + cnt\_one)$. Compared to the original generation method, the time complexity of the proposed method in this paper is lower. This will be conducive to the smooth implementation of the QUATRE algorithm in the C/C++ environment while reducing the consumption of development platform resources.

4.3. Hardware implementation of the QUATRE algorithm

In Section 3.1, this paper describes the working principle and flow of the QUATRE algorithm. Referring to the pseudocode, the QUATRE algorithm mainly includes the following processes: initialization, the generation of co-evolutionary matrix M, the generation of evolutionary guidance matrix B, the update of individuals, the calculation of fitness, the update of individual history optimization and global optimal solution. In the process of implementation, while we simplify the co-evolutionary matrix M, we also merge and adjust part of the process of the algorithm. Finally, the hardware implementation architecture of the algorithm is shown in Fig. 2.

The brown and black lines in the figure represent control signals and data transmission between modules, respectively. In this paper, the *float* data type is used to store and calculate individual location and fitness values. In the following, we introduce the implementation process of these modules.

4.3.1. QUATRE control module

The *QUATRE control* module is the overall control unit of the architecture proposed in this paper, which controls the operating state of each module through signals. Referring to Fig. 2, the signal transmission and data transmission processes controlled by this module are as follows:

(1) Start by sending a start signal to the *Initialization* module. First, after receiving the start signal, the *Initialization* module immediately sends a signal to the *Random number* module to enable random number generation. The floating-point data generated by the *Random number* module are transmitted to the *Initialization* module in the form of data flow to initialize the individual population. The generated integer data are also transmitted to the *Individual update* module in the form of data flow for updating population individuals. After that, the initial population is passed to the *Fitness evaluation* module in the form of data flow. Finally, the *Initialization* module sends an end signal to the *QUATRE control* module after completing the initialization task.

(2) After receiving the end signal from the *Initialization* module, the *QUATRE control* module sends the signal to the *Fitness evaluation* module to start the calculation. The *Fitness evaluation* module calculates the fitness value of individuals in the population according to the evaluation function number. After that, the population individuals and fitness values are passed to the *Pbest and Gbest update* module in the form of data flow. The *Fitness evaluation* module sends the end signal to the control module after the calculation task is completed.

(3) After receiving the end signal from the *Fitness evaluation* module, the *QUATRE control* module sends the signal to start the update to the *Pbest and Gbest update* module. After receiving the signal, the *Pbest and Gbest update* module starts to update the individual historical optimal solution and the global optimal solution. Then, this module passes the population individuals, fitness values and updated data to the *Individual update* module. The *Pbest and Gbest update* module sends the end signal after the update task is completed.

(4) The *QUATRE control* module sends the update start signal to the *Individual update* module after receiving the end signal from the *Pbest and Gbest update* module. After receiving the signal, the *Individual update* module updates all individuals in the population. In this process, the *Individual update* module needs

to use the integer random numbers from the *Random number* module. After the update task is completed, the *Individual update* module transfers the updated population to the *Fitness evaluation* module and sends the end signal.

(5) The program enters the iterative loop state after the *QUATRE control* module receives the end signal from the *Pbest and Gbest update* module for the first time. Processes (2)-(4) are repeatedly executed until the end condition is reached. After the iteration loop ends, the *QUATRE control* module outputs the global optimal solution and the number of iterations when the algorithm reaches the optimal solution.

4.3.2. Random number module

The implementation principles of the *Random_float* module and *Random_integer* module are the same, so we unify them into the *Random number* module. At present, most intelligent optimization algorithms are random optimization algorithms. As a type of random optimization algorithm, the QUATRE algorithm requires the participation of random numbers in the process of individual initialization and co-evolutionary matrix generation. In C/C++, there is a *rand*() library, which can be used to generate random numbers. However, the library cannot be used for algorithm synthesis in the HLS environment. Therefore, the user needs to customize the generation of random numbers.

To generate random numbers, the linear feedback shift register (LFSR) is selected in this paper. The LFSR is a shift register in which an XOR operation of certain bits is performed before the shift, and the XOR result is used as an input. Among them, the bit involved in the XOR is called a tap. The tap sequence can be used to describe the feedback polynomial of the LFSR. For a 32 bit linear feedback shift register, the feedback polynomial is $f(x) = x^{32} + x^{22} + x^2 + x + 1$.

The initial value assigned to the register is called the "seed". In this paper, the "seed" is a random integer in the interval $[20000, 70000]$, which is customized by the user. In the *Random_float* module and the *Random_integer* module, the "seed" is loaded by the *load* signal from the *Initialization* module, and the value after each shift is recorded by the "lfsr" static variable. In the *Random_float* module, only the last sixteen bits of the "lfsr" variable are intercepted to generate random *float* decimals between 0 and 1. These data are transmitted to the *Initialization* module for the generation of the initial population. The *Random_integer* module transmits the value of "lfsr" with the data type *unsigned int* to the *Individual update* module for the generation of the co-evolutionary matrix M and the evolutionary guidance matrix B.

4.3.3. Initialization module

The load signal initialization of the *Random number* module, the population individuals and fitness values, and the global individual X_{gbest} will be initialized in this module. The "load" signals in the *Random_float* and *Random_integer* modules are set to 0 by default. In the *Initialization* module, the "load" signals in the two modules are first assigned a value of 1 to load the "seed". Then, the population is initialized. The individual initialization formula is as follows:

$$X_{i,j} = lb + (ub - lb) * pseudo\_random() \tag{3}$$

The *pseudo_random*() in the formula is the *Random_float* module. lb and ub denote the upper and lower bounds of the individual dimension, respectively. The fitness value of each individual is calculated by the *Fitness evaluation* module, and the global optimal individual X_{gbest} is initialized to the individual with the best fitness value in the population.

4.3.4. Fitness evaluation module

In this paper, six benchmark functions are selected for the performance test of the algorithm, and detailed descriptions are given in Section 5.1. By calculating the corresponding evaluation function, the fitness value of each individual in the population is obtained and passed into the *Pbest and Gbest update* module.

4.3.5. Pbest and Gbest update module

This module is responsible for updating the individual historical optimal solution X_{pbest} and the global optimal solution X_{gbest}. When the module receives the data from the *Fitness evaluation* module for the first time, the program has not yet entered the main loop. X_{pbest} is initialized to the population generated in the *Initialization* module.

4.3.6. Individual update module

The key to the individual update of the QUATRE algorithm is the participation of the co-evolutionary matrix M and evolutionary guidance matrix B. The matrix M is generated in the way that we have simplified in Section 4.2. In the hardware implementation, this is defined as the function *Rand_Label*. The function contains two parameters: one is the array *label*, and the other is variable *cnt_one*. The value of the variable *cnt_one* can be obtained in two ways (described in Section 4.2) to represent the maximum number of values to be assigned to 1 in the array *label* (the default initial value is 0). *cnt_one* different positions are generated by the *Random_integer* module. These positions in the array *label* are assigned 1, and the remaining positions remain 0.

On the implementation of the QUATRE algorithm in this paper, the generation formula of the evolution guidance matrix B is "QUATRE/best/1". Referring to Table 1, the focus of this matrix generation is on the latter two random individual arrays. In the QUATRE algorithm, the individual index is generated in a full permutation, that is, a non-repeating sequence from 1 to *ps* is generated. Here, this paper uses the Knuth–Durstenfeld Shuffle algorithm [53] for implementation. It is an algorithm for disrupting sequences in situ, with a time complexity of $O(N)$. The principle can be simply understood as: a random number is taken from the unprocessed data each time, and then this number is placed at the end of the array. The pseudocode of the algorithm is as follows.

Algorithm 2 The Knuth–Durstenfeld Shuffle algorithm

Input: *ps*: the size of array; *Rand_Seq*: array to be processed (stores values from 1 to *ps*);
Output: *Rand_Seq*: processed array (random full permutation from 1 to ps).

```
1: for i = ps − 1; i >= 0; i − − do
2:     swap(Rand_Seq[i], Rand_Seq[Random_integer()%(i + 1)])
3: end for
```

For dimensions that exceed the boundary, they are uniformly assigned the boundary value.

4.4. Cycle optimization

From the above process and introduction, it can be seen that the QUATRE algorithm contains many loops. The sequential execution of the loop will cause a large time delay. Vivado HLS provides loop optimization processing instructions such as Pipeline and Unroll for loop optimization. In the Vivado HLS tool, some optimization processing instructions are provided, such as *Pipeline* and *Unroll*, for loop optimization. To improve the running speed of the algorithm and reduce the time delay, this paper optimizes the loop part of the QUATRE algorithm. The following rules for adding optimization instructions are formulated:

1. For a single-layer loop:

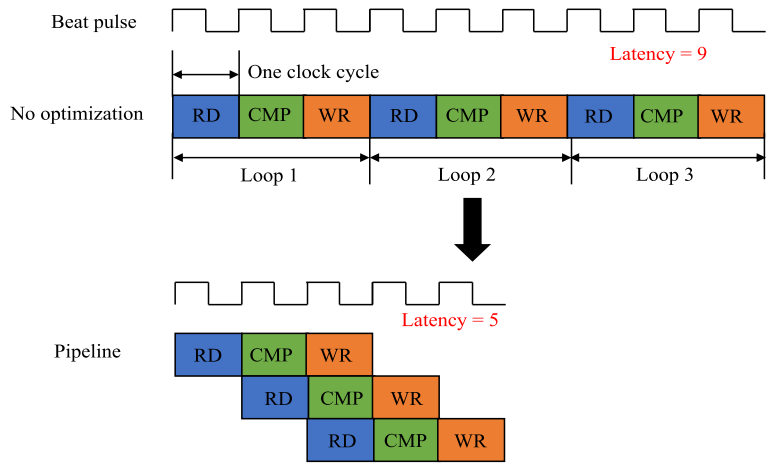


Fig. 3. The schematic diagrams for optimization instruction *Pipeline*.

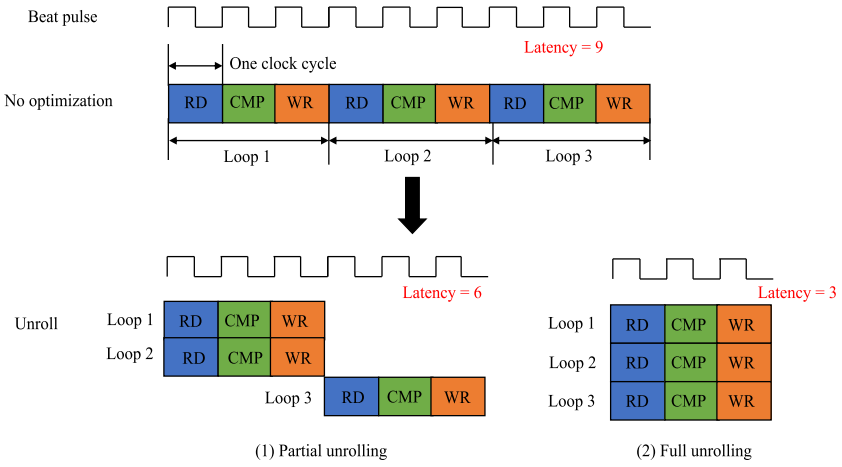


Fig. 4. The schematic diagrams for optimization instruction *Unroll*.

(1) If the statement is relatively simple, such as containing only one array assignment statement, the *Unroll* instruction is used for optimization.

(2) If there are multiple complex statements and there is an association between the statements, such as a comparison statement, the *Pipeline* instruction is used for optimization.

2. For a double-layer loop, such as an individual update process. The *Pipeline* instruction is used in the outer loop, and the *Unroll* instruction is used in the inner loop.

The schematic diagrams for these two instructions are shown in Figs. 3 and 4. *RD* represents the read operation, *CMP* is the calculation operation, and *WR* is the write operation.

5. Experimental results

5.1. Experimental settings

To verify the feasibility and effectiveness of the proposed hardware implementation scheme for the QUATRE algorithm, relevant experiments are carried out in this section. In this paper, the development board Xilinx ZYNQ 7020 is selected for the hardware deployment of the QUATRE algorithm. The on-board resources are shown in Table 2.

Xilinx's 7000 series FPGA development boards consist of two parts. One part is the programmable logic (PL) side, that is, the FPGA side. The other part is the processing system (PS) side, namely, the ARM side. The two sides communicate through the Advanced eXtensible Interface (AXI) bus, which can form a heterogeneous processing system. In the system construction of this paper, the IP core of the QUATRE algorithm derived from Vivado HLS will be deployed on the PL side. The number of iterations to reach the optimal value and the optimal values collected during the whole iteration process will be transmitted to the PS side. The PS side receives the data transmitted by the PL side and connects the serial port to print the data. We integrate the fitness module into the IP core. Therefore, the PS side needs to transmit the corresponding function number to the PL side. Finally, the block design (BD) project built in Vivado is shown in Fig. 5.

Table 2
The on-board resources.

Resource	Available
Block Random Access Memory (BRAM)	280
Digital Signal Processing (DSP48E)	220
Flip-Flop (FF)	106,400
Look Up Table (LUT)	53,200

The *Fitness evaluation* module contains six benchmark functions. Table 3 records the details of these functions. During the experiment, the number of iterations of each algorithm is set to 500. The algorithm is run 10 times consecutively. The optimal value, the mean, the standard deviation, and the number of iterations (*gen*) when obtaining the optimal value are recorded. The population size *ps* is set to 20, 50 and 100. The scaling factor *F* in the hardware-implemented QUATRE algorithm is set to 0.4.

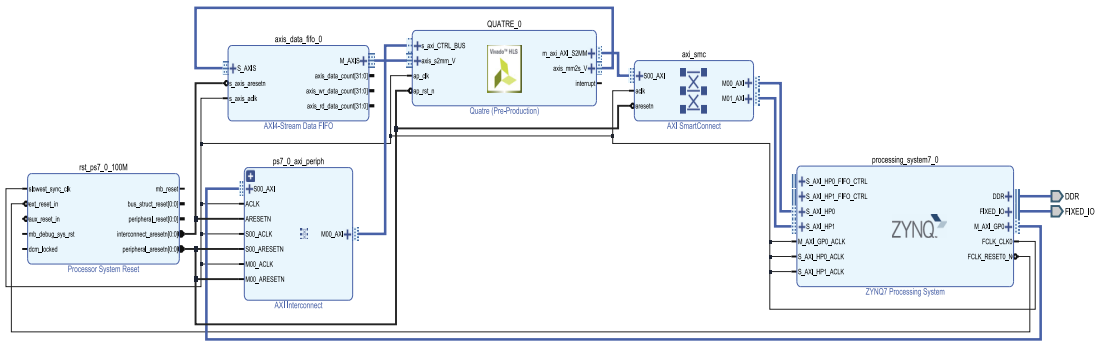


Fig. 5. The BD project of the QUATRE IP core.

Table 3
The details of six benchmark functions.

Function	Function name	Function details	D	Bound	f^*
F1	Sphere Function	$f = \sum_{i=1}^{D} x_i^2$	2	$[-5.12, 5.12]$	0
F2	Schwefel's Function 1.2	$f = \sum_{i=1}^{D} (\sum_{j=1}^{i} x_j)^2$	2	$[-100, 100]$	0
F3	Schwefel's Function 2.21	$f = \sum_{i=1}^{D} \lvert x_i \rvert + \prod_{i=1}^{D} \lvert x_i \rvert$	2	$[-10, 10]$	0
F4	Schwefel's Function 2.22	$f = \max\{\lvert x_i \rvert, 1 \le i \le D\}$	2	$[-100, 100]$	0
F5	Rosenbrock Function	$f = \sum_{i=1}^{D-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	2	$[-2.048, 2.048]$	0
F6	Dixon-price Function	$f = (x_1 - 1)^2 + \sum_{i=2}^{D} i(2x_i^2 - x_{i-1})^2$	2	$[-10, 10]$	0

Table 4
The comparison of the QUATRE algorithm and the simplified QUATRE (S_QUATRE) algorithms.

	OQUATRE		S_QUATRE1		S_QUATRE2	
	Total	Utilization (%)	Total	Utilization (%)	Total	Utilization (%)
$ps = 20$						
BRAM	11	3	10	3	10	3
DSP48E	41	18	41	18	41	18
FF	9218	8	8558	8	8558	8
LUT	18105	34	16790	31	16774	31
Latency	1932062∼2205002		1295062∼1558002		1295062∼1558002	
$ps = 50$						
BRAM	13	4	12	4	12	4
DSP48E	41	18	41	18	41	18
FF	9223	8	8539	8	8539	8
LUT	18104	34	16871	31	16765	31
Latency	5003671∼5682271		3361671∼4015271		3361671∼4015271	
$ps = 100$						
BRAM	13	4	12	4	12	4
DSP48E	41	18	41	18	41	18
FF	9290	8	8582	8	8582	8
LUT	18131	34	16797	31	16871	31
Latency	10106321∼11461021		6789321∼8094021		6789321∼8094021	

5.2. Comparison of the QUATRE algorithm before and after simplification

In the Vivado HLS environment, this paper synthesizes the original QUATRE (OQUATRE) algorithm and the simplified QUATRE (S_QUATRE) algorithm to verify the effectiveness of the improvement method. The synthesized results are shown in Table 4. It can be seen that the two S_QUATRE algorithms are lower than the OQUATRE algorithm in terms of resource occupancy and latency. As the number of solutions increases, the latency gap between the OQUATRE algorithm and the S_QUATRE algorithms gradually increases. The two S_QUATRE algorithms remain essentially the same in terms of resource occupancy and latency. Since the first method needs to call the *Random_integer* module one more time, it occupies more LUT resources than the second method.

5.3. Comparison with and without optimization instructions

In Section 4.4, this paper optimizes the loops of the S_QUATRE algorithm by adding optimization instructions. In this subsection, this paper analyzes the influence of optimization instructions on the algorithm from the aspects of resource occupancy and running time. Under different population sizes, Table 5 records the resource occupancy and latency comparison of the two S_QUATRE algorithms with or without optimization instructions.

As seen from Table 5, after adding the optimization instructions, the occupancy of DSP48E, FF and LUT is increased by approximately 1.6 times, but it is still within an acceptable range. Only when $ps = 100$ does the occupied LUT exceed 50% of the total onboard resources. The resources occupied by other items are still low. In terms of latency, the algorithm is significantly reduced by about 6∼7 times compared to before adding the optimization instructions. Overall, this optimization strategy is

Table 5
The comparison of the S_QUATRE algorithm before and after optimization.

	S_QUATRE1		S_QUATRE1_op		S_QUATRE2		S_QUATRE2_op	
	Total	Utilization (%)	Total	Utilization (%)	Total	Utilization (%)	Total	Utilization (%)
ps = 20								
BRAM	10	3	9	3	10	3	9	3
DSP48E	41	18	85	38	41	18	85	38
FF	8558	8	14445	13	8558	8	14445	13
LUT	16790	31	26082	49	16774	31	26066	48
Latency	1295062∼1558002		262646∼262646		1295062∼1558002		262646∼262646	
ps = 50								
BRAM	12	4	16	5	12	4	16	5
DSP48E	41	18	85	38	41	18	85	38
FF	8539	8	14437	13	8539	8	14437	13
LUT	16871	31	26971	50	16765	31	26955	50
Latency	3361671∼4015271		578563∼578563		3361671∼4015271		578563∼578563	
ps = 100								
BRAM	12	4	16	5	12	4	16	5
DSP48E	41	18	85	38	41	18	85	38
FF	8582	8	14851	13	8582	8	14851	13
LUT	16797	31	28719	53	16871	31	28703	53
Latency	6789321∼8094021		1105087∼1105087		6789321∼8094021		1105087∼1105087	

Table 6
The running time and solution results of the algorithms deployed on the FPGA platform.

Function	S_QUATRE1		S_QUATRE1_op		S_QUATRE2		S_QUATRE2_op	
	mean	Time (ms)	mean	Time (ms)	mean	Time (ms)	mean	Time (ms)
ps = 20								
F1	0.00E+00	14	0.00E+00	2	0.00E+00	14	0.00E+00	2
F2	0.00E+00	14	0.00E+00	2	0.00E+00	14	0.00E+00	2
F3	0.00E+00	14	0.00E+00	2	0.00E+00	14	0.00E+00	2
F4	0.00E+00	13	0.00E+00	2	0.00E+00	13	0.00E+00	2
F5	1.60E−03	14	1.60E−03	2	1.79E−02	14	1.79E−02	2
F6	3.05E−06	14	3.05E−06	2	1.42E−06	14	1.42E−06	2
ps = 50								
F1	0.00E+00	37	0.00E+00	6	0.00E+00	37	0.00E+00	6
F2	0.00E+00	37	0.00E+00	6	0.00E+00	37	0.00E+00	6
F3	0.00E+00	37	0.00E+00	6	0.00E+00	37	0.00E+00	6
F4	0.00E+00	34	0.00E+00	5	0.00E+00	34	0.00E+00	5
F5	2.49E−04	37	2.49E−04	6	1.46E−04	37	1.46E−04	6
F6	4.86E−08	37	4.86E−08	6	1.89E−07	37	1.89E−07	6
ps = 100								
F1	0.00E+00	74	0.00E+00	11	0.00E+00	74	0.00E+00	11
F2	0.00E+00	76	0.00E+00	12	0.00E+00	76	0.00E+00	12
F3	0.00E+00	74	0.00E+00	11	0.00E+00	74	0.00E+00	11
F4	0.00E+00	68	0.00E+00	10	0.00E+00	68	0.00E+00	10
F5	2.36E−05	75	2.36E−05	11	1.19E−04	75	1.19E−04	11
F6	1.94E−08	75	1.94E−08	11	1.97E−08	75	1.97E−08	11

worthwhile in exchange for a significant speed increase by increasing the occupancy of a small number of resources. Table 6 records the actual running time and the average results of the algorithms after deployment to the development board. After adding the optimization instructions, the running result of the algorithm is not affected. In terms of running time, the two optimized S_QUATRE algorithms are approximately 6∼7 times faster than before, which is consistent with the latency comparison in Table 5.

5.4. Comparison of software-implemented and hardware-implemented QUATRE algorithms

Table 7 records the comparison of the test results of the software-implemented QUATRE algorithm (S_QT) and hardware-implemented QUATRE algorithm (H_QT) at different population sizes. In this paper, the S_QT algorithm runs on the PS side of the development board, that is the ARM side. Bolded data in the table represent the best value among all the objects to be compared. N/A indicates that the theoretical optimal value is not found.

On function F1, the four test algorithms obtain the theoretical optimal value 10 consecutive times, but the number of iterations spent by H_QT1 and H_QT2 to reach the theoretical optimal value is significantly less than that of S_QT1 and S_QT2. On functions F2 and F3, the optimal values obtained by the four algorithms in 10 solutions are all theoretical optimal values. However, H_QT1 and H_QT2 also take fewer iterations than S_QT1 and S_QT2, and perform better in terms of the mean and standard deviation (stability). On function F4, neither S_QT1 nor S_QT2 reach the expected ideal value. However, both H_QT1 and H_QT2 achieve convergence of the results, and obtain the theoretical optimal value 10 consecutive times. Although on functions F5 and F6, the software implementation and hardware implementation of the QUATRE algorithm do not obtain the theoretical optimal value in 10 solutions, the convergence accuracy and stability of H_QT1 and H_QT2 are overall better than those of S_QT1 and S_QT2. Only on function F5, when $ps = 20$, does S_QT2 perform better than H_QT1 and H_QT2. It can also be seen from the table that the optimization performance of the two methods of generating the

Table 7
The comparison of the results of software-implemented and hardware-implemented QUATRE algorithms.

Function		ps = 20				ps = 50				ps = 100			
		S_QT1	S_QT2	H_QT1	H_QT2	S_QT1	S_QT2	H_QT1	H_QT2	S_QT1	S_QT2	H_QT1	H_QT2
F1	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
	gen	169	151	92	**89**	156	154	**83**	**83**	154	156	86	**81**
F2	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
	mean	2.05E−19	8.17E−17	**0.00E+00**	**0.00E+00**	6.74E−37	6.47E−29	**0.00E+00**	**0.00E+00**	8.34E−27	6.77E−32	**0.00E+00**	**0.00E+00**
	std	5.95E−19	2.14E−16	**0.00E+00**	**0.00E+00**	1.40E−36	1.94E−28	**0.00E+00**	**0.00E+00**	2.50E−26	2.00E−31	**0.00E+00**	**0.00E+00**
	gen	227	286	88	**80**	372	274	**86**	**86**	212	234	92	84
F3	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
	mean	2.10E−45	**0.00E+00**	**0.00E+00**	**0.00E+00**	2.80E−46	2.21E−42	**0.00E+00**	**0.00E+00**	2.80E−46	4.20E−46	**0.00E+00**	**0.00E+00**
	std	6.31E−45	**0.00E+00**	**0.00E+00**	**0.00E+00**	8.41E−46	6.64E−42	**0.00E+00**	**0.00E+00**	8.41E−46	1.26E−45	**0.00E+00**	**0.00E+00**
	gen	316	308	183	**171**	347	366	177	**171**	353	346	186	**185**
F4	best	4.66E−22	3.00E−19	**0.00E+00**	**0.00E+00**	1.17E−24	4.87E−21	**0.00E+00**	**0.00E+00**	1.05E−27	2.59E−21	**0.00E+00**	**0.00E+00**
	mean	1.34E−10	2.70E−05	**0.00E+00**	**0.00E+00**	5.54E−10	4.04E−11	**0.00E+00**	**0.00E+00**	7.10E−13	3.51E−10	**0.00E+00**	**0.00E+00**
	std	4.02E−10	8.10E−05	**0.00E+00**	**0.00E+00**	1.66E−09	8.69E−11	**0.00E+00**	**0.00E+00**	2.05E−12	1.05E−09	**0.00E+00**	**0.00E+00**
	gen	N/A	N/A	194	191	N/A	N/A	208	**207**	N/A	N/A	237	**225**
F5	best	1.88E−05	**3.00E−19**	1.04E−04	1.78E−05	1.03E−04	1.24E−05	**6.46E−06**	7.56E−06	9.68E−05	8.24E−06	**7.79E−07**	9.48E−06
	mean	6.03E−02	**2.70E−05**	1.60E−03	1.79E−02	9.93E−03	4.83E−03	2.49E−04	**1.46E−04**	1.61E−03	1.04E−03	**2.36E−05**	1.19E−04
	std	8.15E−02	**8.10E−05**	1.39E−03	3.60E−02	1.38E−02	8.86E−03	3.32E−04	**2.30E−04**	1.75E−03	1.65E−03	**3.40E−05**	1.19E−04
	gen	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
F6	best	6.12E−08	5.62E−07	7.07E−09	**4.33E−09**	9.23E−09	7.07E−09	1.03E−09	**7.72E−10**	1.11E−08	7.14E−09	5.76E−10	**9.22E−11**
	mean	2.10E−04	5.70E−05	3.05E−06	**1.42E−06**	7.11E−05	2.10E−06	**4.86E−08**	1.89E−07	7.86E−07	2.21E−06	**1.94E−08**	1.97E−08
	std	3.36E−04	1.00E−04	8.54E−06	**2.79E−06**	1.94E−04	3.96E−06	**5.93E−08**	4.60E−07	1.42E−06	4.59E−06	**2.12E−08**	2.45E−08
	gen	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 8
The running time and speed-up ratio of software-implemented and hardware-implemented QUATRE algorithms.

Function		ps = 20				ps = 50				ps = 100			
		S_QT1	S_QT2	H_QT1	H_QT2	S_QT1	S_QT2	H_QT1	H_QT2	S_QT1	S_QT2	H_QT1	H_QT2
F1	Time (ms)	13	12	2	2	35	32	6	6	71	66	11	11
	speed-up	1	1	6.5	6	1	1	5.83	5.33	1	1	6.46	6
F2	Time (ms)	13	12	2	2	35	33	6	6	71	68	12	12
	speed-up	1	1	6.5	6	1	1	5.83	5.5	1	1	6.46	5.67
F3	Time (ms)	14	13	2	2	38	35	6	6	76	73	11	11
	speed-up	1	1	7	6.5	1	1	6.33	5.83	1	1	6.91	6.64
F4	Time (ms)	14	13	2	2	37	35	5	5	74	70	10	10
	speed-up	1	1	7	6.5	1	1	7.4	7	1	1	7.4	7
F5	Time (ms)	13	12	2	2	35	33	6	6	72	68	11	11
	speed-up	1	1	6.5	6	1	1	5.83	5.5	1	1	6.55	6.18
F6	Time (ms)	13	12	2	2	35	32	6	6	71	67	11	11
	speed-up	1	1	6.5	6	1	1	5.83	5.33	1	1	6.46	6.09

co-evolutionary matrix M is relatively close. The second method is slightly better than the first method.

Table 8 records the time for the software/hardware implementation of the QUATRE algorithm to solve each function and the speed-up ratio. The speed-up ratio is calculated as follows:

$$S = \frac{T_{S\_QT}}{T_{H\_QT}} \tag{4}$$

where T_{S_QT} represents the running time of the software -implemented QUATRE algorithm, and T_{H_QT} represents the execution time of the hardware-implemented QUATRE algorithm. Thanks to the parallelism of FPGA and the blessing of pipeline architecture, the execution speed of the hardware-implemented QUATRE algorithm is better than the software-implemented QUATRE algorithm. Under different population sizes, the solution speed is increased by approximately 5.5∼7 times. To intuitively reflect the difference in the time it takes for the QUATRE algorithm to solve the problem on software and hardware, this paper expresses the solution time in Table 8 with a histogram, and the results are shown in Fig. 6.

We also test the convergence performance of the algorithms implemented by software and hardware. Fig. 7 gives the convergence curves of the algorithms solving different test functions under $ps = 50$. As seen from the convergence curves, the convergence performance of the hardware-implemented QUATRE algorithm is also better than that of the software-implemented QUATRE algorithm. H_QT1 is slightly worse than H_QT2 in terms of convergence performance.

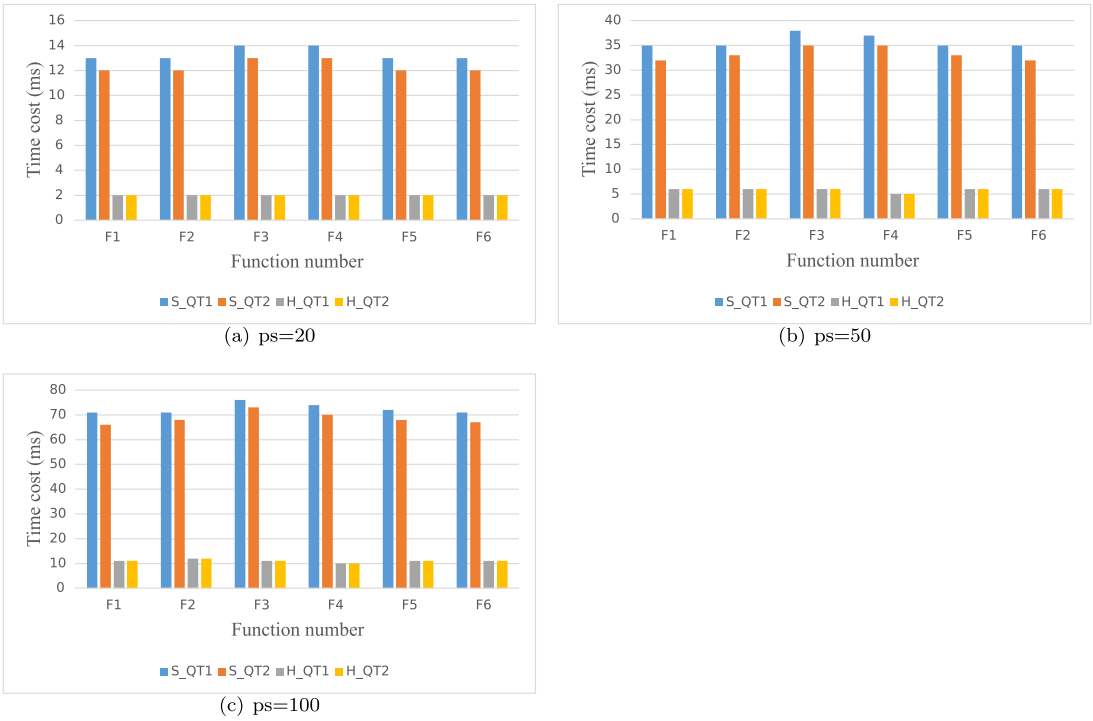
5.5. Comparison with the experimental results of the ga, DE and PSO algorithms

To further verify the ability of the QUATRE algorithm to solve optimization problems on hardware, this paper selects the classical GA, DE and PSO algorithms, which have been implemented on hardware, for comparison. The parameters of these three algorithms are set as follows.

- GA: Roulette wheel selection, mutation probability $mu = 0.01$, crossover probability $gamma = 0.4$
- DE: $F = 0.7$, $pCR = 0.2$
- PSO: $\omega = 0.8$, $c1 = c2 = 2.0$

The hardware implementation flow of the three algorithms is consistent with the QUATRE algorithm, and the loops in the algorithm also add optimization instructions. Table 9 records the resource occupancy of each algorithm when the population size ps is 50. As seen from the data in the table, the QUATRE algorithm occupies the most resources in the first three items compared to the other three algorithms. In the resource occupancy of LUT, the QUATRE algorithm is only better than the GA algorithm.

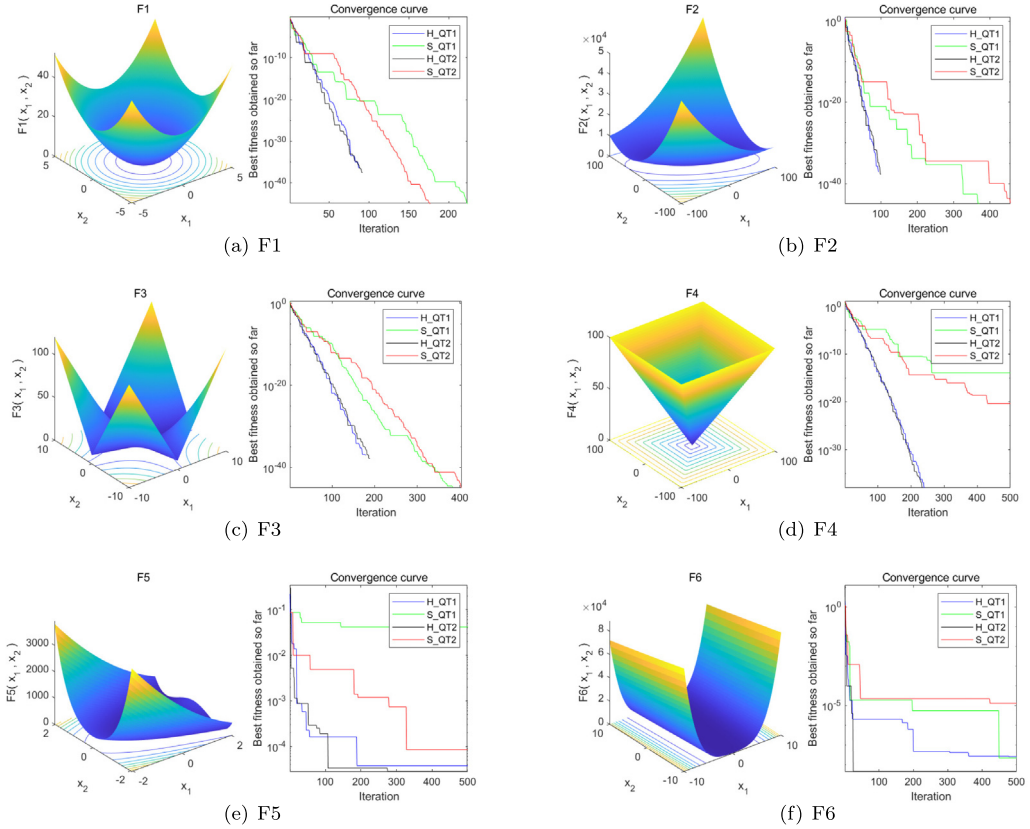
Table 10 gives the solution results of each algorithm, and the best values are also bolded. Compared with the other three algorithms, the hardware implementation of the QUATRE algorithm has strong competitiveness. On functions F1–F4, the DE



(a) ps=20 (b) ps=50

(c) ps=100

Fig. 6. The running time histograms for software-implemented and hardware-implemented QUATRE algorithms under different *ps*.



(a) F1 (b) F2

(c) F3 (d) F4

(e) F5 (f) F6

Fig. 7. The convergence curves of the software-implemented and hardware-implemented QUATRE algorithms solving six benchmark functions.

Table 9
The resource occupancy of five algorithms ($ps = 50$).

	GA		DE		PSO		H_QT1		H_QT2	
	Total	Utilization (%)	Total	Utilization (%)	Total	Utilization (%)	Total	Utilization (%)	Total	Utilization (%)
BRAM	14	5	7	2	12	4	16	5	16	5
DSP48E	62	28	35	15	59	26	85	38	85	38
FF	13097	12	11435	10	8737	8	14437	13	14437	13
LUT	28581	53	20500	38	16214	30	26971	50	26955	50

Table 10
The comparison of the QUATRE algorithms and the other three algorithms ($ps = 50$).

Function		GA	DE	PSO	H_QT1	H_QT2
F1	best	7.95E−06	**0.00E+00**	9.54E−26	**0.00E+00**	**0.00E+00**
	mean	2.36E−02	**0.00E+00**	1.11E−09	**0.00E+00**	**0.00E+00**
	std	3.12E−02	**0.00E+00**	3.15E−09	**0.00E+00**	**0.00E+00**
	gen	N/A	121	N/A	**83**	**83**
F2	best	9.31E−06	**0.00E+00**	4.83E−35	**0.00E+00**	**0.00E+00**
	mean	4.80E−03	**0.00E+00**	4.37E−18	**0.00E+00**	**0.00E+00**
	std	1.12E−02	**0.00E+00**	1.22E−17	**0.00E+00**	**0.00E+00**
	gen	N/A	99	N/A	**86**	**86**
F3	best	3.11E−05	**0.00E+00**	1.32E−11	**0.00E+00**	**0.00E+00**
	mean	1.21E−01	**0.00E+00**	2.19E−04	**0.00E+00**	**0.00E+00**
	std	2.63E−01	**0.00E+00**	6.52E−04	**0.00E+00**	**0.00E+00**
	gen	N/A	242	N/A	177	**171**
F4	best	5.66E−02	**0.00E+00**	2.23E−11	**0.00E+00**	**0.00E+00**
	mean	2.04E+00	**0.00E+00**	9.47E−05	**0.00E+00**	**0.00E+00**
	std	3.10E+00	**0.00E+00**	2.55E−04	**0.00E+00**	**0.00E+00**
	gen	N/A	343	N/A	208	**207**
F5	best	3.98E−03	**0.00E+00**	2.27E−11	6.46E−06	7.56E−06
	mean	8.63E−02	1.22E−04	**9.22E−07**	2.49E−04	1.46E−04
	std	8.00E−02	1.50E−04	**2.22E−06**	3.32E−04	2.30E−04
	gen	N/A	152	N/A	N/A	N/A
F6	best	1.35E−03	**1.26E−29**	4.00E−14	1.03E−09	7.72E−10
	mean	1.29E+00	2.55E−07	7.44E−08	**4.86E−08**	1.89E−07
	std	2.56E+00	4.66E−07	1.62E−07	**5.93E−08**	4.60E−07
	gen	N/A	N/A	N/A	N/A	N/A

algorithm and the QUATRE algorithm are able to obtain the theoretical optimal value 10 consecutive times, but the DE algorithm takes more iterations than the QUATRE algorithm. On function F5, the QUATRE algorithm performs poorly, only better than the GA algorithm. On function F6, the QUATRE algorithm is worse than the PSO algorithm and the DE algorithm in the accuracy of the optimal value, but the stability is better than these two algorithms. We also count the average solution time for the GA, DE and PSO algorithms, and Fig. 8 shows the histogram of the solution time for each algorithm.

As seen from the figure, the running time of the hardware-implemented QUATRE algorithm is significantly faster than that of the other three optimization algorithms. Through calculation, the solution time of the GA algorithm is approximately 9.5∼12 times that of the QUATRE algorithm, the DE algorithm is 9.16∼10.6 times, and the PSO algorithm is 3.33∼4.2 times. Therefore, considering the optimization ability and the time spent solving (real-time), the performance of the QUATRE algorithm on the hardware is better than the other three algorithms and is more competitive.

5.6. Analysis of parameter F

This subsection explores the impact of the parameter F on the performance of the hardware-implemented QUATRE algorithm. F is set to 0.1, 0.4–1.0, and 2.0. The population size ps is set to 50, and the algorithm runs continuously 10 times. Tables 11 and 12 record the experimental results of the two hardware-implemented QUATRE algorithms under different F values. The best values are also bolded. On functions F1–F4, the first four values of F are able to find the theoretical optimal value 10 times. On functions F5 and F6, the mean and standard deviation of the two hardware-implemented QUATRE algorithms perform best when F is 0.4. The optimal value is slightly worse in accuracy than the other values. On the whole, when F is set to 0.4, the QUATRE algorithm performs best on hardware.

6. Conclusion

In this paper, a hardware-implemented QUATRE algorithm is proposed and deployed on FPGA. By analyzing the flow of the QUATRE algorithm, the generation process of co-evolutionary matrix is complicated and inconvenient to implement in the C/C++ environment. To facilitate hardware implementation, this paper simplifies the process and proposes two different methods to generate co-evolutionary matrix. The simplified QUATRE algorithm may reduce latency and resource occupation. By using the Vivado HLS design tool, this paper completes the export and loop optimization of the QUATRE algorithm IP core. Through testing on the Xilinx ZYNQ 7020 development board, the running speed and

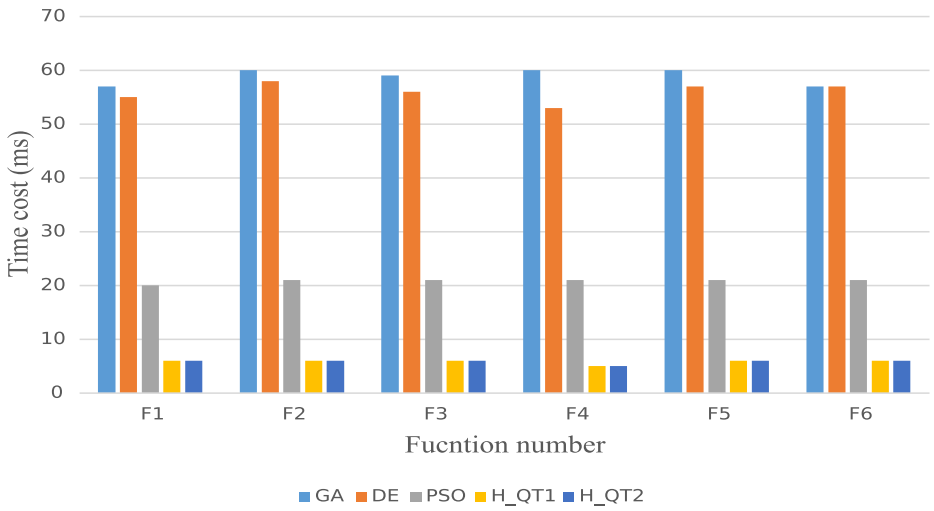


Fig. 8. The histogram of solution time for each algorithm.

Table 11
The experimental results of the hardware-implemented QUATRE algorithm under different F values (the first method).

Function		$F = 0.1$	$F = 0.4$	$F = 0.5$	$F = 0.6$	$F = 0.7$	$F = 0.8$	$F = 0.9$	$F = 1$	$F = 2$
F1	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	2.05E−08
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	2.16E−13	2.78E−05
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	6.11E−13	4.50E−05
	gen	43	83	108	122	151	184	212	74	N/A
F2	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	9.31E−06
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	3.73E−06	2.79E−01
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	4.56E−06	8.29E−01
	gen	39	86	101	127	147	197	198	32	N/A
F3	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	1.22E−03
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	3.57E−38	3.05E−05	1.25E−02
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	8.71E−38	9.16E−05	1.13E−02
	gen	81	177	222	268	330	380	450	33	N/A
F4	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	3.32E−38	6.44E−31	3.92E−25	0.00E+00	2.91E−01
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	4.14E−38	1.09E−30	4.62E−25	0.00E+00	4.80E−01
	gen	111	208	269	311	368	431	498	62	55
F5	best	4.52E−05	6.46E−06	1.51E−05	1.73E−05	5.01E−05	9.83E−05	1.50E−04	**6.30E−06**	7.57E−05
	mean	3.60E−02	**2.49E−04**	7.66E−04	4.74E−04	5.33E−04	9.48E−04	1.35E−03	1.61E−03	1.43E−02
	std	4.11E−02	**3.32E−04**	1.49E−03	7.93E−04	5.63E−04	1.31E−03	1.33E−03	1.61E−03	1.17E−02
	gen	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
F6	best	1.27E−09	**1.03E−09**	7.83E−09	1.40E−09	1.06E−08	1.34E−08	8.44E−09	7.60E−08	3.71E−06
	mean	3.53E−05	**4.86E−08**	8.63E−08	8.15E−07	1.05E−06	6.38E−07	2.81E−06	4.14E−06	1.45E−03
	std	5.25E−05	**5.93E−08**	6.81E−08	1.13E−06	1.42E−06	8.55E−07	4.46E−06	3.93E−06	3.96E−03
	gen	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

optimization performance of the hardware-implemented QUATRE algorithm are better than those of the software-implemented QUATRE algorithm. Compared to the GA, DE and PSO algorithms implemented by hardware, the hardware-implemented QUATRE algorithm also shows strong competitiveness and runs faster than these three algorithms. Through experimental comparison, it can be found that the resource occupation of the QUATRE algorithm is relatively high. Therefore, in the future, we will focus on how to reduce the resource occupation and further improve the execution speed of the QUATRE algorithm.

Funding

None. No fundings to declare.

CRediT authorship contribution statement

Jeng-Shyang Pan: Formal analysis, Conceptualization. **Qingyong Yang:** Writing – original draft, Software, Methodology. **Jyh-Horng Chou:** Data curation. **Chia-Cheng Hu:** Writing – review & editing. **Shu-Chuan Chu:** Writing – review & editing, Validation.

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.

Table 12
The experimental results of the hardware-implemented QUATRE algorithm under different F values (the second method).

Function		$F = 0.1$	$F = 0.4$	$F = 0.5$	$F = 0.6$	$F = 0.7$	$F = 0.8$	$F = 0.9$	$F = 1$	$F = 2$
F1	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	2.88E−08
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	2.64E−09	1.01E−05
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	7.30E−09	1.81E−05
	gen	44	83	97	126	146	183	212	78	N/A
F2	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	9.31E−06
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	9.31E−09	2.79E−06	2.83E−02
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	2.79E−08	4.27E−06	7.84E−02
	gen	40	86	98	129	143	172	213	225	N/A
F3	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	9.16E−04
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	1.20E−26	3.05E−05	3.36E−02
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	3.60E−26	9.16E−05	7.50E−02
	gen	89	171	222	268	309	377	443	456	N/A
F4	best	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	3.05E−03
	mean	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	5.67E−38	1.00E−30	1.45E−25	3.05E−04	1.20E−01
	std	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	8.45E−38	1.16E−30	1.59E−25	9.16E−04	1.13E−01
	gen	115	207	248	308	375	416	476	65	N/A
F5	best	5.09E−05	7.56E−06	1.16E−05	1.39E−05	5.62E−06	3.79E−06	1.54E−04	**9.47E−07**	3.06E−05
	mean	2.96E−02	**1.46E−04**	3.49E−04	2.59E−04	1.60E−03	8.25E−04	1.69E−03	8.67E−04	3.63E−03
	std	2.61E−02	**2.30E−04**	6.61E−04	3.51E−04	2.93E−03	1.06E−03	9.40E−04	8.99E−04	4.26E−03
	gen	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
F6	best	2.77E−09	7.72E−10	**5.55E−10**	1.08E−07	1.04E−09	5.56E−07	2.93E−07	4.90E−09	1.50E−06
	mean	4.33E−05	**1.89E−07**	5.82E−07	6.45E−07	9.84E−06	2.80E−06	2.24E−05	4.77E−06	1.61E−03
	std	4.80E−05	**4.60E−07**	1.55E−06	8.14E−07	2.30E−05	2.61E−06	6.37E−05	5.04E−06	2.75E−03
	gen	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

References

[1] L. Hulianytskyi, I. Riasna, Formalization and classification of combinatorial optimization problems, in: Optimization Methods and Applications, Springer, 2017, pp. 239–250.
[2] M.A. Muñoz, Y. Sun, M. Kirley, S.K. Halgamuge, Algorithm selection for black-box continuous optimization problems: A survey on methods and challenges, Inform. Sci. 317 (2015) 224–245.
[3] A. López-Jaimes, C.A.C. Coello, Including preferences into a multiobjective evolutionary algorithm to deal with many-objective engineering optimization problems, Inform. Sci. 277 (2014) 1–20.
[4] G.J. Woeginger, Exact algorithms for NP-hard problems: A survey, in: Combinatorial Optimization—Eureka, You Shrink!, Springer, 2003, pp. 185–207.
[5] N. Kokash, An introduction to heuristic algorithms, Department of Informatics and Telecommunications, Citeseer, 2005, pp. 1–8.
[6] M. Abdel-Basset, L. Abdel-Fatah, A.K. Sangaiah, Metaheuristic algorithms: A comprehensive review, in: Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications, Elsevier, 2018, pp. 185–231.
[7] E.K. Burke, M.R. Hyde, G. Kendall, G. Ochoa, E. Özcan, J.R. Woodward, A classification of hyper-heuristic approaches: revisited, in: Handbook of Metaheuristics, Springer, 2019, pp. 453–477.
[8] A.K. Kar, Bio inspired computing–a review of algorithms and scope of applications, Expert Syst. Appl. 59 (2016) 20–32.
[9] S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[10] S. Katoch, S.S. Chauhan, V. Kumar, A review on genetic algorithm: past, present, and future, Multimedia Tools Appl. 80 (5) (2021) 8091–8126.
[11] F. Glover, M. Laguna, Tabu search, in: Handbook of Combinatorial Optimization, Springer, 1998, pp. 2093–2229.
[12] O.I. Abiodun, A. Jantan, A.E. Omolara, K.V. Dada, N.A. Mohamed, H. Arshad, State-of-the-art in artificial neural network applications: A survey, Heliyon 4 (11) (2018) e00938.
[13] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95-International Conference on Neural Networks, Vol. 4, IEEE, 1995, pp. 1942–1948.
[14] M. Dorigo, M. Birattari, T. Stutzle, Ant colony optimization, IEEE Comput. Intell. Mag. 1 (4) (2006) 28–39.
[15] E. Cuevas, J. Gálvez, K. Avila, M. Toski, V. Rafe, A new metaheuristic approach based on agent systems principles, J. Comput. Sci. 47 (2020) 101244.
[16] I. Ahmadianfar, O. Bozorg-Haddad, X. Chu, Gradient-based optimizer: A new metaheuristic optimization algorithm, Inform. Sci. 540 (2020) 131–159.
[17] F.A. Hashim, K. Hussain, E.H. Houssein, M.S. Mabrouk, W. Al-Atabany, Archimedes optimization algorithm: a new metaheuristic algorithm for solving optimization problems, Appl. Intell. 51 (2021) 1531–1551.
[18] R. Guha, S. Ghosh, K.K. Ghosh, E. Cuevas, M. Perez-Cisneros, R. Sarkar, Groundwater flow algorithm: a novel hydro-geology based optimization algorithm, IEEE Access 10 (2022) 132193–132211.
[19] G.-G. Wang, Y. Tan, Improving metaheuristic algorithms with information feedback models, IEEE Trans. Cybern. 49 (2) (2017) 542–555.
[20] E. Osaba, E. Villar-Rodriguez, J. Del Ser, A.J. Nebro, D. Molina, A. LaTorre, P.N. Suganthan, C.A.C. Coello, F. Herrera, A tutorial on the design, experimentation and application of metaheuristic algorithms to real-world optimization problems, Swarm Evol. Comput. 64 (2021) 100888.
[21] H. Zhou, M. Song, W. Pedrycz, A comparative study of improved GA and PSO in solving multiple traveling salesmen problem, Appl. Soft Comput. 64 (2018) 564–580.
[22] S. Das, P.N. Suganthan, Differential evolution: A survey of the state-of-the-art, IEEE Trans. Evol. Comput. 15 (1) (2010) 4–31.
[23] X.-S. Yang, M. Karamanoglu, Nature-inspired computation and swarm intelligence: a state-of-the-art overview, in: Nature-Inspired Computation and Swarm Intelligence, Elsevier, 2020, pp. 3–18.
[24] D. Karaboga, B. Akay, A comparative study of artificial bee colony algorithm, Appl. Math. Comput. 214 (1) (2009) 108–132.
[25] X.-S. Yang, A.H. Gandomi, Bat algorithm: a novel approach for global engineering optimization, Eng. Comput. 29 (5) (2012) 464–483.
[26] S. Mirjalili, S.M. Mirjalili, A. Lewis, Grey wolf optimizer, Adv. Eng. Softw. 69 (2014) 46–61.
[27] A. Rodríguez, O. Camarena, E. Cuevas, I. Aranguren, A. Valdivia-G, B. Morales-Castañeda, D. Zaldívar, M. Pérez-Cisneros, Group-based synchronous-asynchronous grey wolf optimizer, Appl. Math. Model. 93 (2021) 226–243.
[28] Z. Meng, J.-S. Pan, H. Xu, Quasi-affine transformation evolutionary (QUATRE) algorithm: A cooperative swarm based algorithm for global optimization, Knowl.-Based Syst. 109 (2016) 104–121.
[29] Z. Meng, J.-S. Pan, QUasi-Affine TRansformation Evolution with External ARchive (QUATRE-EAR): an enhanced structure for differential evolution, Knowl.-Based Syst. 155 (2018) 35–53.
[30] S.-C. Chu, Z. Zhuang, J. Li, J.-S. Pan, A novel binary QUasi-affine transformation evolutionary (QUATRE) algorithm, Appl. Sci. 11 (5) (2021) 2251.
[31] Z.-G. Du, J.-S. Pan, S.-C. Chu, H.-J. Luo, P. Hu, Quasi-affine transformation evolutionary algorithm with communication schemes for application of RSSI in wireless sensor networks, Ieee Access 8 (2020) 8583–8594.
[32] X. Wang, S.-C. Chu, V. Snášel, L. Kong, J.-S. Pan, H.A. Shehadeh, A two-phase quasi-affine transformation evolution with feedback for parameter identification of photovoltaic models, Appl. Soft Comput. 113 (2021) 107978.
[33] N. Liu, J.-S. Pan, C. Sun, S.-C. Chu, An efficient surrogate-assisted quasi-affine transformation evolutionary algorithm for expensive optimization problems, Knowl.-Based Syst. 209 (2020) 106418.

[34] Z. Meng, Y. Chen, X. Li, C. Yang, Y. Zhong, Enhancing QUasi-Affine TRansformation Evolution (QUATRE) with adaptation scheme on numerical optimization, Knowl.-Based Syst. 197 (2020) 105908.
[35] N. Liu, J.-S. Pan, S.-C. Chu, A competitive learning quasi affine transformation evolutionary for global optimization and its application in CVRP, J. Int. Technol. 21 (7) (2020) 1863–1883.
[36] H. Yu, Y. Tan, J. Zeng, C. Sun, Y. Jin, Surrogate-assisted hierarchical particle swarm optimization, Inform. Sci. 454 (2018) 59–72.
[37] D. Li, L. Huang, K. Wang, W. Pang, Y. Zhou, R. Zhang, A general framework for accelerating swarm intelligence algorithms on FPGAs, GPUS and multi-core CPUS, IEEE Access 6 (2018) 72327–72344.
[38] Y. Tan, K. Ding, A survey on GPU-based implementation of swarm intelligence algorithms, IEEE Trans. Cybern. 46 (9) (2015) 2028–2041.
[39] M. Peker, A fully customizable hardware implementation for general purpose genetic algorithms, Appl. Soft Comput. 62 (2018) 1066–1076.
[40] P. Cortés-Antonio, J. Rangel-González, L.A. Villa-Vargas, M.A. Ramírez-Salinas, H. Molina-Lozano, I. Batyrshin, Design and implementation of differential evolution algorithm on fpga for double-precision floating-point representation, Acta Polytech. Hungarica 11 (4) (2014) 139–153.
[41] S. Anis, et al., Fpga implementation of parallel particle swarm optimization algorithm and compared with genetic algorithm, Int. J. Adv. Comput. Sci. Appl. 1 (7) (2016) 57–64.
[42] 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.
[43] R.M. Calazan, N. Nedjah, L.M. Mourelle, A hardware accelerator for particle swarm optimization, Appl. Soft Comput. 14 (2014) 347–356.
[44] M. Ettouil, H. Smei, A. Jemai, Particle swarm optimization on fpga, in: 2018 30th International Conference on Microelectronics, ICM, IEEE, 2018, pp. 32–35.
[45] 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.
[46] 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.
[47] M.S.B. Ameur, A. Sakly, FPGA based hardware implementation of bat algorithm, Appl. Soft Comput. 58 (2017) 378–387.
[48] 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.
[49] H.H. Issa, S.M.E. Ahmed, FPGA implementation of floating point based cuckoo search algorithm, IEEE Access 7 (2019) 134434–134447.
[50] 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) 1–15.
[51] D.M. Muñoz, C.H. Llanos, L.d.S. Coelho, M. Ayala-Rincón, Accelerating the artificial bee colony algorithm by hardware parallel implementations, in: 2012 IEEE 3rd Latin American Symposium on Circuits and Systems, LASCAS, IEEE, 2012, pp. 1–4.
[52] V. Xilinx, Vivado design suite user guide-high-level synthesis, 2014.
[53] E.K. Donald, et al., Seminumerical algorithms, in: The Art of Computer Programming, Vol. 2, Addison-Wesley, 1969, pp. 351–354.