A parallel whale optimization algorithm and its implementation on FPGA

Qiangqiang Jiang, Yuanjun Guo, Zhile Yang* and Xianyu Zhou

***Abstract*—Whale Optimization Algorithm (WOA), as a novel nature-inspired swarm optimization algorithm, has demonstrated superior performance in solving optimization problems. However, the performance deteriorates when applied to large-scale complex problems due to rapidly increasing running time required for huge computational tasks. Based on interactions within population, WOA is naturally amenable to parallelism, prompting an effective approach to mitigate the drawbacks of sequential WOA. Field Programmable Gate Array (FPGA) is an acceleration device of high parallelism and programmability. Meanwhile, Open Computing Language (OpenCL) provides a general architecture for heterogeneous development. In this paper, an efficient implementation of parallel WOA on FPGA is proposed named FPWOA. Experiment studies are conducted by performing WOA on CPU and FPWOA on FPGA respectively to solve ten well known benchmark functions. Numerical results show that our approach achieves a favourable speedup while maintaining optimization performance.**

***Index Terms*—Whale Optimization Algorithm (WOA), parallelism, Field Programmable Gate Array (FPGA), heterogeneous computing, Open Computing Language (OpenCL).**

I. Introduction

Swarm intelligence algorithms have become a significant methodology to solve optimization problems and utilized in a wide range of scientific researches and practical applications. Whale optimization algorithm (WOA), a novel swarm intelligence based meta-heuristic algorithm, was proposed by Mirjalili and Lewis in 2016 [1]. Inspired by the special hunting behavior of humpback whales, the WOA algorithm shows better performance compared with several existing popular methods and has drawn great research interests. Mohamed et al.[2] integrated the WOA with a locals search strategy for tackling the permutation flow shop scheduling problem. Mafarja and Mirjalili [3] proposed a hybrid WOA with simulated annealing for feature selection. Aljarah et al.[4] introduced WOA-based trainer to train multilayer perceptron (MLP) neural networks. There are also some works that tried to solve multi-objective problems using the WOA algorithm [5], [6], [7], [8].

Nevertheless, when solving problems with high-dimension or complex mathematical model, swarm intelligence algorithms including WOA may encounter difficulties that the optimization performance decreases due to extensive computational cost. In the light of this, a growing number of scholars have started to study and design parallel swarm algorithms and implement them under various accelerating platforms. In the recent years, distributed and parallel Particle Swarm Optimization (PSO) has been implemented. Some studies [9], [10], [11], [12], [13] applied GPU to parallel PSO implementation for specific problems, putting forward diverse parallel strategies. Hajewski et al. [14] developed fast cache-aware parallel PSO relying on OpenMP. Ant Colony Optimization (ACO) [15] and Artificial Bee Colony (ABC) [16] were also parallelized by GPU. With respect to Brain Storm Optimization (BSO), Chen et al.[17] presented GPU-based manner while Ma et al.[18] proposed parallelized BSO algorithm based on Spark framework for association rule mining. Similar works [19] and [20] used GPU and FPGA to accelerate Genetic Algorithm (GA). What deserve attention is that Garcia et al. [21] achieved parallel implementation and comparison of Teaching-Learning Based Optimization (TLBO) and Jaya on many-core GPU. As for WOA, Khalil et al. [22] proposed a simple and robust distributed WOA using Hadoop MapReduce, reaching a promising speedup.

After investigating above-mentioned studies for parallel swarm algorithms, we can discover that there are several typical kinds of parallel techniques including OpenMP, MapReduce, Spark, and heterogeneous architecture based on dedicated accelerators, such as GPU and FPGA. GPU has become popular for general purpose computing and and great success has been achieved in developing parallel swarm intelligence algorithms, gaining remarkable performance[23]. With high-parallelism and flexible programmability, FPGA is gradually widely applied to heterogeneous computing with OpenCL and algorithms accelerating[24], [25], [26], [27], [28].

The experiments conducted by [29] showed that swarm algorithm on FPGAs achieved a better speedup than that on GPUs and multi-core CPUs. However, designing a near-optimal accelerator is not an easy task. Implementing CPU-oriented codes on FPGA rarely increases the performance and even reduces the performance compared to CPU. Therefore, it requires not only the hardware architecture design experience, but also the knowledge of how to write appropriate OpenCL codes to implement the desired architecture on an FPGA [30]. To the best of our knowledge, very few research works have been investigated on FPGA implementation of swarm intelligence algorithms, especially WOA. In this paper, inspired by the previous studies, a parallel WOA on FPGA is proposed

This research is financially supported by China NSFC under grants 51607177, Natural Science Foundation of Guangdong Province under grants 2018A030310671, Outstanding Young Researcher Innovation Fund of Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences (201822)

Qiangqiang Jiang, Yuanjun Guo and Zhile Yang are with Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, Guangdong, 518055, China (e-mail: joejq@outlook.com, yj.guo,zl.yang@siat.ac.cn), Xianyu Zhou is with Northeastern University, Shenyang, China (e-mail: 18813003797@163.com)

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

aiming to obtaining computation performance for large-scale complex problems.

The rest of the paper is organized as follows: the OpenCL-based FPGA heterogeneous computing is introduced in Section II; Section III presents the theory of WOA. In Section IV, we propose the FPGA implementation of parallel WOA, and it is followed by the experimental results and statistical analysis in Section V. Finally, some conclusions are made in Section VI.

II. OpenCL-based FPGA heterogeneous computing

A. OpenCL and FPGA

Open Computing Language (OpenCL), maintained by Khronos Group, is an open standard for general purpose parallel computing [31]. Various hardware devices, such as CPU, FPGA, GPU, DSP and other accelerators, can be supported in OpenCL, making it possible to implement high efficacy algorithms across powerful heterogeneous computing platform. Additionally, OpenCL specifies a $C99$ based programming API for developers. A typical OpenCL program consists of a host source code an a kernel source code.

Field Programmable Gate Array (FPGA) is a configurable integrated circuit that can be reconfigured repeatedly to perform an infinite number of functions. It generally includes programmable core logics, hierarchical reconfigurable interconnects, I/O elements, memory blocks, DSPs and etc. With these substantial logical resources, FPGA achieves a high level of programmability and flexibility. However, traditional development on FPGA is mainly describing hardware at register transfer level (RTL) or even at the gate level using hardware description languages (HDL) such as Verilog and VHDL, which is a high-cost and time-consuming process. To address this problem, FPGA vendors like Intel and Xilinx released OpenCL framework with FPGA support, endowing software developers the possibility to design FPGA applications effectively.

B. Intel FPGA SDK for OpenCL

The Intel FPGA SDK for OpenCL [32] allows users to create high-level FPGA implementation with OpenCL. The SDK generates a heterogeneous computing environment where OpenCL kernels are compiled by an off-line compiler (AOC) for programming FPGA at runtime. In this paradigm, Intel achieves design optimization while hiding low-level hardware details of FPGA. OpenCL-based FPGA logic framework is illustrated in Fig.1 where several modules are specifically explained as follows:

- **Kernel Pipeline**: The core module of the entire framework, which is an implementation of specific functions. The kernel code is compiled by *AOC* off-line compiler, and will be synthesized into highly parallel optimized logic circuit referring to the internal architecture of FPGA.
- **Processor**: A host processor, typically CPU, used to control programs running on FPGA device.

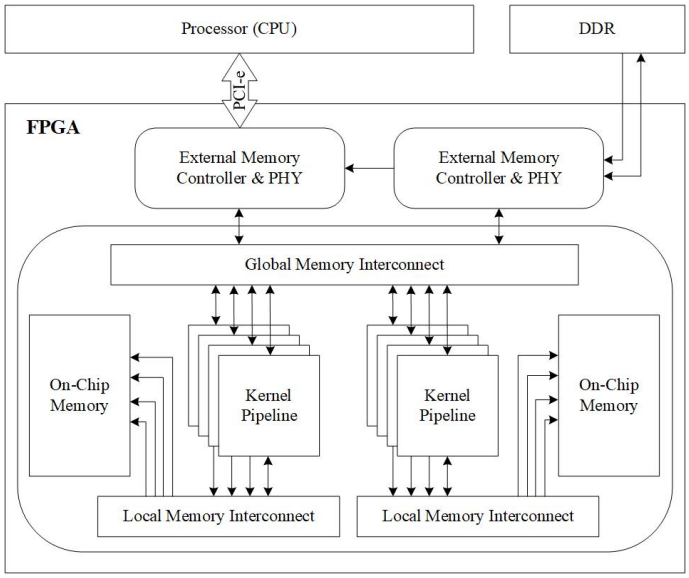

Fig. 1. OpenCL-based FPGA logic framework figure

- **DDR**: Off-chip memory, including global and constant memory in OpenCL memory model. Intel Cyclone V FPGA device used in this context, has a DDR3 with the capacity of 1GB. By default, the constant cache size is 16KB and can be modified in accordance with practical requirements.
- **PCI-e**: High-speed data exchanging interface, responsible for transporting data and instruction between host and device.
- **On-chip Memory**: Internal memory of the FPGA device, equivalent to local and private memory in OpenCL memory model. With small capacity but high speed, it is mainly used for storing input and output temporary data, reducing the number of accesses to global memory. Thus, we may take advantage of on-chip memory to improve the efficiency of OpenCL program.
- **Local Memory Interconnect**: A bridge between executing unit and memory.
- **External Memory Controller & PHY**: A controller which is in charge of controlling data sending and receiving via DDR.

III. Whale Optimization Algorithm

Inspired by the hunting behavior of humpback whales, the WOA algorithm implements two main phases, exploitation and exploration, through emulating shrinking encircling, bubble-net attacking and searching for prey. The following subsections explain in details the mathematical models of each phase.

A. Exploitation phase (encircling and bubble-net attacking)

To hunt preys, humpback whales first recognize the location of preys and encircle them. The mathematical model of

shrinking encircling behavior is represented by the following equations:

$$\boldsymbol{D} = \left| C \cdot \boldsymbol{X}^*_{(t)} - \boldsymbol{X}_{(t)} \right| \tag{1}$$

$$\boldsymbol{X}_{(t+1)} = \boldsymbol{X}^*_{(t)} - A \cdot \boldsymbol{D} \tag{2}$$

where $\boldsymbol{X}$ is the position vector, $\boldsymbol{X}^*$ represents the position of the best solution obtained so far, t indicates the current number of iteration, $|\ |$ denotes the absolute operation and $\cdot$ means an element-by-element multiplication.

A and C are two parameters, which are calculated as follows:

$$A = 2a \cdot r - a \tag{3}$$

$$C = 2 \cdot r \tag{4}$$

where a is linearly decreasing from 2 to 0 over the course of iterations (in both exploitation and exploration phases) and r is a random number in $[0, 1]$. The value of a is calculated by $a = 2 - t\frac{2}{MaxIter}$ and $MaxIter$ is the maximum number of iterations.

Another method used in the exploitation phase is spiral updating position, which in coordination with aforementioned shrinking encircling constitutes the bubble-net attacking strategy of humpback whales. The mathematical equations are as follows:

$$\boldsymbol{D'} = \left| \boldsymbol{X}^*_{(t)} - \boldsymbol{X}_{(t)} \right| \tag{5}$$

$$\boldsymbol{X}_{(t+1)} = \boldsymbol{D'} \cdot e^{bl} \cdot \cos(2\pi l) + \boldsymbol{X}^*_{(t)} \tag{6}$$

where b is a constant for determining the shape of the logarithmic spiral, l is a random number in $[-1, 1]$. Shrinking encircling and spiral updating position are used simultaneously during exploitation phase. The mathematical model is as follows:

$$\boldsymbol{X}_{(t+1)} = \begin{cases} \boldsymbol{X}^*_{(t)} - A \cdot \boldsymbol{D}, & p < 0.5 \\ \boldsymbol{D'} \cdot e^{bl} \cdot \cos(2\pi l) + \boldsymbol{X}^*_{(t)}, & p \geq 0.5 \end{cases} \tag{7}$$

where p is a random value in $[-1, 1]$ which stands for that there is a probability of 50% to choose either the shrinking encircling method or the spiral-shaped mechanism to update the position of whales during optimization process.

B. Exploration phase (searching for preys)

In addition to exploitation phase, a stochastic searching technique is also adopted to enhance the exploration in WOA. Unlike exploitation, a random whale $\boldsymbol{X}_{rand}$ is selected from swarm to navigate the search space, so as to find a better optimal solution (prey) than the existing one. This phase can efficiently prevents the algorithm from falling into local optima stagnation. Subsequently, based on the parameter A, a decision is made on which mechanism to be used for updating the position of whales. Exploration is done if $|A| \geq 1$, meanwhile if $|A| < 1$. This methodology is mathematically modelled as follows,

$$\boldsymbol{D} = \left| C \cdot \boldsymbol{X}_{rand} - \boldsymbol{X}_{(t)} \right| \tag{8}$$

$$\boldsymbol{X}_{(t+1)} = \boldsymbol{X}_{rand} - A \cdot \boldsymbol{D} \tag{9}$$

where $\boldsymbol{X}_{rand}$ is a random position of the whale chosen from the current population, and C is calculated by Eq.(4). Algorithm 1 presents the pseudo code of the WOA algorithm.

Algorithm 1 Whale Optimization Algorithm

```
Generate initial population X_i(i = 1, 2, ··· , n)
Evaluate the fitness of each search agent
X*=the best search agent
while (t < MaxIter) do
    for each search agent do
        Update a, A, C, l and p
        if (p < 0.5) then
            if (|A| < 1) then
                Update the position of the current search agent by Eq.(2)
            else if (|A| ≥ 1) then
                Select a random search agent (X_rand)
                Update the position of the current search agent by Eq.(9)
            end if
        else if (p ≥ 0.5) then
            Update the position of the current search agent by Eq.(6)
        end if
    end for
    Amend search agents which go beyond the search space
    Calculate the fitness of each search agent
    Replace X* with a better solution (if found)
    t = t + 1
end while
return X*
```

At the beginning of the algorithm, an initial random population is generated, and each individual gets evaluated by fitness function and X^* is the current best solution. Then, the algorithm repeatedly executes until the end criterion is satisfied. At each iteration, search agents update their position according to either a random chosen individual when $|A| \geq 1$, or the optimum solution obtained so far when $|A| < 1$. Depending on p, the WOA algorithm makes decision between using circular or spiral movement.

IV. FPGA IMPLEMENTATION OF PARALLEL WOA

WOA naturally has the speciality of being parallelized. Such an intrinsic property of WOA makes it very suitable to be deployed on heterogeneous platforms with the ability of parallel computing.

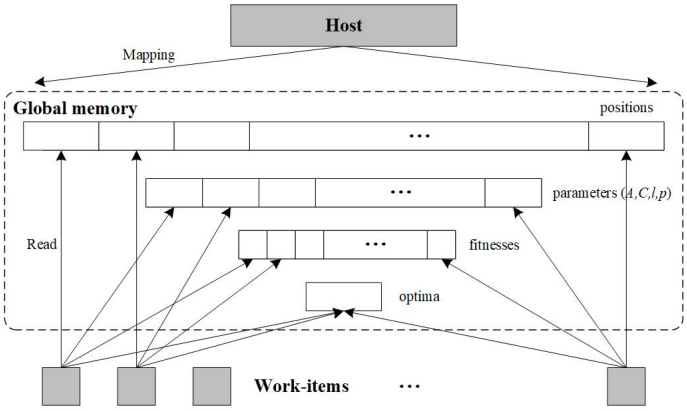

Fig. 2. Dataflow of parallel WOA between host and kernel

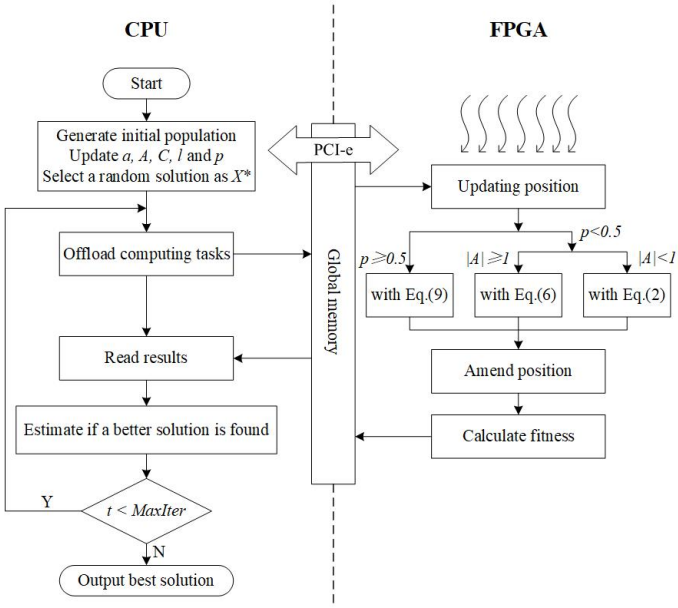

Fig. 3. Flow of parallel WOA

A. Parallel analysis on WOA

For implementing a high performance parallel algorithm, a parallel analysis on original algorithm is essential. Similar to other swarm optimization algorithms, WOA unavoidably suffers from the drawback of intensive computation caused by the time-consuming fitness evaluation, which greatly limits its execution speed[29]. As demonstrated in line $2, 18$ and 19 from Algorithm 1, the process of evaluation as well as amendment can be operated simultaneously in parallelization to the computation efficiency of WOA. On top of that, the codes in line $5 \sim 17$ can be executed concurrently as well, which means that the positions of all search agents are updated separately by corresponding moving mechanism, more specifically reflecting the real hunting of humpback whales.

B. Dataflow analysis on parallel WOA between host and kernel

In the proposed implementation, the dataflow between host and FPGA kernel is mainly depended on global memory by means of PCI-e interface. As illustrated in Fig.2, the set of data contains the position and fitness of all search agents, the global optima X^*, and four parameters comprising A, C, l and p. On the host side, memory buffers are created and the data used is mapped to these buffers, which will be further sent to kernel by global memory. On the kernel side, each work-item seen as a basic processing element, reads the data from global memory and complete the kernel function.

In that global memory access has a great influence on performance, realizing optimization for global memory access can be beneficial. We utilize aggregate data type ($float4$) to wrap above-mentioned parameters up and static memory coalescing methodology for the position and fitness information during global memory operation. Additionally, local memory is considerably smaller than global memory, but it has significantly higher throughput and much lower latency. Using local memory to pre-load data will reduce the number of times the kernel accesses global memory, effectually enhancing the performance of data transportation[33]. Hence, the development in this work reads the current optima (X^*) and parameters (A, C, l and p) from global memory and load it to local memory blocks, preparing for next steps.

C. Design and implementation of parallel WOA

In consideration of OpenCL-based FPGA computing framework and WOA theory, a parallel scheme for WOA is designed, shown in Fig.3.

1) Host logic flow: In terms of host (CPU), it undertakes non-computation-intensive tasks or runs these codes which need to be sequentially performed or cannot be parallelized, and the remaining tasks of algorithm are offloaded and performed by kernel program. Meanwhile, host program generates and organizes the data set required in the computation offloaded. Afterwards, it apply for memory buffers for the set of data, finally transported to kernel through global memory.

Since OpenCL does not implement random number generator and plenty of random numbers are used in WOA, C/C++ library about random number will be applied at host side to generate initial population and parameters (A, C, l and p). The random numbers appear in each iteration of WOA, namely data transportation between host and kernel in each iteration is required. It will become a bottleneck for the running speed due to mountains of data to be transported[9]. To alleviate this drawback, we generate all random numbers on CPU, and then send them to FPGA once by global memory. Furthermore, atomic function is considered to be expensive to implement in FPGA, which might degrade kernel performance and lead to hardware resources waste[33]. Hence, the process of estimating if a better solution is found, is also scheduled in host program, highly taking advantage of the computational horsepower of CPU.

2) Kernel logic flow: FPGA device in our algorithm acts as an accelerator to execute kernel program. Host offloads

computationally intensive tasks onto FPGA for paralleling running. With OpenCL programming model, the parallel parts of algorithm are mapped to kernel function to be executed by each work-item independently [23], [29]. In the proposed implementation, each work-item takes charge of a search agent, which calculates their respective fitness, update and amend respective position in parallelization. Fig.3 illustrates that, according to the parameters (A and p), each work-item (search agent) performs different mechanisms simultaneously: shrinking encircling, spiral updating and stochastic searching. Once the kernel has finished executing, the results related will be sent back to host.

To optimize the processing efficiency of kernel on the FPGA, some strategies can be adopted [34]. First of all, $restrict$ keyword is inserted into pointer arguments, avoiding pointer aliasing. In addition, in order to mitigate latency and overhead on the FPGA, we use loop unrolling pragma on the loop part where there are no loop-carried dependencies, and private memory implemented by FPGA registers or block RAMs to store intermediate variables during updating positions. At last, when the kernel is defined, we make it specified with required work-group size to optimize hardware usage without involving excess logic resources.

V. Experimental results and analysis

A. Experimental setup

Our experimental platform contains two main hardware devices: CPU and FPGA. For the CPU platform, we use Intel Core i5-7200 CPU with 8GB RAM. For the FPGA platform, we use Intel FPGA Cyclone V GT with 1GB DDR3 and 64MB SDRAM. The entire development environment is based on Microsoft Windows 10 and Intel FPGA SDK for OpenCL 17.1 version.

In this paper, ten benchmark functions [35], listed in Table 1, are used to make performance comparisons between serial WOA (CPU implementation) and parallel WOA (FPGA implementation). Among these benchmark functions, $f1 \sim f5$ are unimodal functions and $f6 \sim f10$ are multimodal functions. The dimensions of all functions tested are set to be 256.

Concerning other parameters in the WOA algorithm, the spiral coefficient b in spiral updating model and the maximum number of iterations $MaxIter$ are held constant in the whole evaluations and set to be 1 and 500, respectively. The population sizes are set to 256, 512, 1024 and 2048 for each test case. Additionally, for each implementation with a specific parameter setting, 30 independently runs are executed and the average performance is considered.

B. Running times and computation results

In this section, speedup is calculated from the running time of different population sizes and defined as follows:

$$Speedup = \frac{T_{WOA}}{T_{FPWOA}} \tag{10}$$

where T_{WOA} and T_{FPWOA} denote the running times of serial WOA and FPGA implementation of parallel WOA simplified as FPWOA, respectively. As per the results obtained in TableII-V, several analysis and conclusions can be made

TABLE I
Benchmark functions with the dimension set to 256

Func	Expression	Range	f_{min}
$f1$	$f(x) = \sum_{i=1}^{D} x_i^2$	$[-100, 100]^D$	0
$f2$	$f(x) = \sum_{i=1}^{D} \lvert x_i \rvert + \prod_{i=1}^{D} \lvert x_i \rvert$	$[-10, 10]^D$	0
$f3$	$f(x) = \sum_{i=1}^{D} i x_i^2$	$[-10, 10]^D$	0
$f4$	$f(x) = \max_{1 \le i \le D} \lvert x_i \rvert$	$[-100, 100]^D$	0
$f5$	$f(x) = \sum_{i=1}^{D} (\lfloor x_i + 0.5 \rfloor)^2$	$[-100, 100]^D$	0
$f6$	$f(x) = \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right]$	$[-5.12, 5.12]^D$	0
$f7$	$f(x) = -20e^{-0.02\sqrt{D^{-1}\sum_{i=1}^{D}}} - e^{D^{-1}\sum_{i=1}^{D}\cos(2\pi x_i)} + 20 + e$	$[-32, 32]^D$	0
$f8$	$f(x) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos(\frac{x_i}{\sqrt{i}}) + 1$	$[-600, 600]^D$	0
$f9$	$f(x) = \sum_{i=1}^{D} \lvert x_i \sin(x_i) + 0.1 x_i \rvert$	$[-10, 10]^D$	0
$f10$	$f(x) = \sum_{i=1}^{D} x_i^6 (2 + \sin\frac{1}{x_i})$	$[-1, 1]^D$	0

TABLE II
Comparison for the running times and results of WOA and FPWOA (population size is 256)

function	WOA		FPWOA		Speedup
	Mean	Time(s)	Mean	Time(s)	
$f1$	**3.19E-112**	2.2382	2.11E-110	0.4195	5.3354
$f2$	**1.54E-62**	2.3048	1.63E-62	0.4190	5.5008
$f3$	**2.02E-112**	2.2862	1.56E-111	0.4737	4.8262
$f4$	2.10E-16	2.2599	**1.50E-16**	0.4129	5.4733
$f5$	0	2.7232	0	0.4317	6.3082
$f6$	0	4.4341	0	0.5741	7.7236
$f7$	**2.11E-16**	4.3771	2.69E-15	0.5260	8.3216
$f8$	0	5.3035	0	0.5385	9.8486
$f9$	**7.03E-64**	3.9881	9.58E-63	0.5500	7.2511
$f10$	**4.04E-281**	5.7846	4.54E-281	0.6007	9.6297

TABLE III
Comparison for the running times and results of WOA and FPWOA (population size is 512)

function	WOA		FPWOA		Speedup
	Mean	Time(s)	Mean	Time(s)	
$f1$	1.47E-117	4.3858	**1.02E-119**	0.6648	6.5973
$f2$	3.78E-65	4.5478	**2.49E-66**	0.7048	6.4526
$f3$	6.73E-119	4.5307	**8.76E-120**	0.6534	6.9341
$f4$	6.45E-21	4.4472	**5.01E-24**	0.6843	6.4986
$f5$	0	5.3459	0	0.6556	8.1542
$f6$	0	8.9344	0	1.0893	8.2020
$f7$	**3.30E-16**	8.7900	1.60E-15	0.9821	8.9502
$f8$	0	10.6067	0	1.0220	10.3784
$f9$	2.33E-66	7.9469	**1.79E-67**	1.0608	7.4914
$f10$	**4.63E-288**	9.6482	2.75E-287	1.1735	8.2218

TABLE IV
COMPARISON FOR THE RUNNING TIMES AND RESULTS OF WOA AND FPWOA (POPULATION SIZE IS 1024)

function	WOA		FPWOA		Speedup
	Mean	Time(s)	Mean	Time(s)	
$f1$	3.20E-124	8.8261	**1.22E-124**	1.3132	6.7211
$f2$	9.62E-68	9.0653	**1.52E-68**	1.3267	6.8330
$f3$	1.09E-123	8.9422	**9.18E-124**	1.3361	6.6928
$f4$	5.72E-23	8.7697	**9.18E-29**	1.3219	6.6342
$f5$	0	10.5293	0	1.3135	8.0162
$f6$	0	17.6137	0	1.6786	10.4931
$f7$	**2.56E-17**	17.5276	2.19E-16	1.5690	11.1712
$f8$	0	20.1839	0	1.5915	12.6823
$f9$	3.28E-69	15.9423	**1.22E-69**	1.5964	9.9864
$f10$	**1.40E-296**	18.5671	5.77E-296	1.7976	10.3287

TABLE V
COMPARISON FOR THE RUNNING TIMES AND RESULTS OF WOA AND FPWOA (POPULATION SIZE IS 2048)

function	WOA		FPWOA		Speedup
	Mean	Time(s)	Mean	Time(s)	
$f1$	1.31E-128	18.3651	**1.26E-129**	3.2094	5.7223
$f2$	1.66E-70	18.1968	**5.14E-71**	3.1821	5.7185
$f3$	3.91E-128	17.7379	**2.00E-129**	3.0552	5.8058
$f4$	2.02E-26	17.6620	**1.98E-30**	3.1473	5.6118
$f5$	0	21.1211	0	3.1566	6.6911
$f6$	0	35.2745	0	3.2029	11.0133
$f7$	9.18E-16	34.9231	**1.05E-16**	3.4867	10.0161
$f8$	0	39.5615	0	3.2190	12.2900
$f9$	1.05E-70	31.9132	**4.06E-71**	3.2230	9.9017
$f10$	0	36.8537	0	3.4097	10.8085

below.

1) Statistical results and comparison: The $Mean$ and $Time$ in each table refer to the average values of results and running times obtained after the algorithm runs 30 times for each benchmark function. With the increasing of the population size, both WOA and FPWOA can find better solutions, which indicates that the population size affects the optimization performance to some extent. When the population size is 256, on the tests that almost all functions except for $f4$, WOA provides more effective optimization ability than FPWOA. On the contrary, FPWOA shows preferable solving efficacy as the population size grows (from 512 to 2048). For details, FPWOA transcends and outperforms WOA under the benchmark functions with $population = 512$ and $population = 1024$, except for $f7$ and $f10$, and has an absolute advantage in all test cases when the population size is 2048. From a whole, comparisons of FPWOA and WOA utilized to solve ten benchmark functions, reveal the efficient optimization of FPWOA (even better than original WOA, especially with large population size).

2) Running time and speedup: It can be seen from tables that the running times of WOA and FPWOA for each test case raise by nearly 2 times with the two folds of the population size. With the same population size, it takes more time for both WOA and FPWOA to optimize multimodal functions ($f6 \sim f10$) than unimodal functions ($f1 \sim f5$), which is more obvious for WOA, but gets gradually negligible for FPWOA as the population size becomes larger, especially with the population size being 2048. That is because multimodal functions are frequently linked with higher arithmetical complexity than unimodal functions [9], [10]. Thus, facing larger-scale complex problems, FPGA might display better performance by means of outstanding parallel power.

After analyzing the running times given in TableII-V, comparing with WOA on different test cases, a speedup of FPWOA can be obtained. For unimodal functions, the speedups can reach around 5 to 7 and be more than 8 in terms of multimodal functions. In the best condition, FPWOA can achieve a maximum speedup of 12.68 when the population size is 1024, settling multimodal function $f8$. It is obvious to notice that a competitive speedup can be obtained under the cases of multimodal funtions and larger population size ($population = 1024$ and 2048). Generally speaking, with powerful parallel computing feature, FPWOA would run faster than WOA for all test cases.

C. Convergence analysis for FPWOA

In this part, we investigate the best solution found by FPWOA at each iteration under different population size and ten benchmark functions. From Fig.4(a)-4(j), where N means the population size. It can be noticed that the population size is a crucial factor for the rate of convergence. As the number of whales increases, the proposed algorithm converges faster. In other words, a better optimal output can be gained at the same number of iterations. The curve starts to converge after around 20 iterations when solving the $f1$ with $N = 256$, $f3$ with$N = 256$ and $f4$ with $N = 512$, respectively shown in Fig.4(a), 4(c) and 4(d), which might be the slowest convergence rate in all test cases. The fastest rate of convergence, in contrast, appears after 5 iterations under $f10$ function and $N = 2048$. It is noteworthy that the algorithm behaves unstable for $f4$ function, which in turn affects its convergence rate to some extent. Besides, $f8$ and $f10$ are special cases where convergence rate of the algorithm is less affected by population size. In short, the conducted experiment indicates that the proposed algorithm for all benchmark functions in the context can achieve a promising performance in terms of the rate of convergence.

VI. CONCLUSION

In this work, a parallel implementation of WOA on FPGA is presented for large swarm and high dimensional problems. After analysing sequential model of WOA and FPGA architecture, we take some programming strategies for optimizing data processing efficiency, referring to OpenCL based development paradigm for FPGA. Moreover, several changes considering WOA have been made to adjust to parallel framework. Numerical experiments are conducted to draw comparisons about the running time and optimization performance between WOA and the proposed FPWOA, utilizing ten benchmarks. Experimental

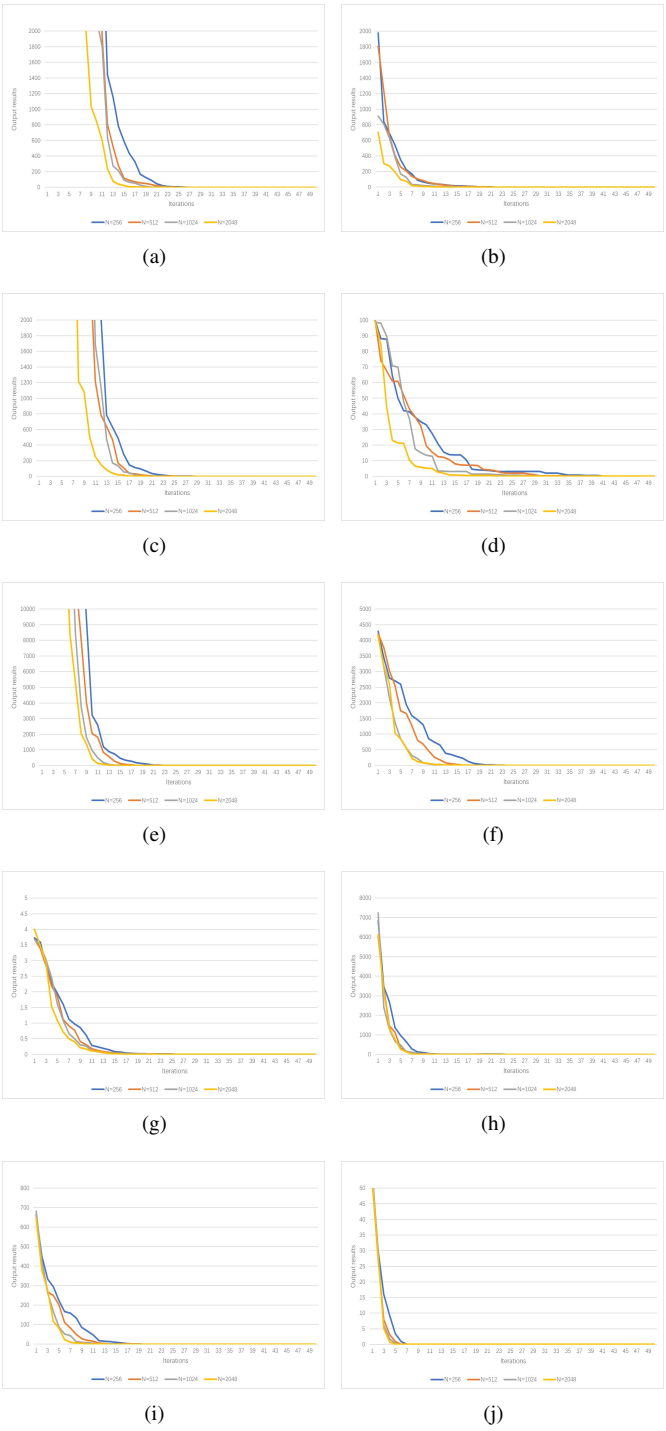

Fig. 4. Convergence curves for ten benchmark functions ($f1 \sim 10$)

results obtained show that significant improvement in executing efficiency can be achieved for FPWOA which also holds great solving quality and a fast convergence rate. However, it should be pointed that only one kind of heterogeneous platform based on FPGA is used to implement parallel WOA. Future work will be addressing the design and implement parallel WOA on other heterogeneous platforms such as GPU and multiple devices.

REFERENCES

[1] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in engineering software*, vol. 95, pp. 51–67, 2016.

[2] M. Abdel-Basset, G. Manogaran, D. El-Shahat, and S. Mirjalili, "A hybrid whale optimization algorithm based on local search strategy for the permutation flow shop scheduling problem," *Future Generation Computer Systems*, vol. 85, pp. 129–145, 2018.

[3] M. M. Mafarja and S. Mirjalili, "Hybrid whale optimization algorithm with simulated annealing for feature selection," *Neurocomputing*, vol. 260, pp. 302–312, 2017.

[4] I. Aljarah, H. Faris, and S. Mirjalili, "Optimizing connection weights in neural networks using the whale optimization algorithm," *Soft Computing*, vol. 22, no. 1, pp. 1–15, 2018.

[5] J. Wang, P. Du, T. Niu, and W. Yang, "A novel hybrid system based on a new proposed algorithm—multi-objective whale optimization algorithm for wind speed forecasting," *Applied energy*, vol. 208, pp. 344–360, 2017.

[6] M. A. El Aziz, A. A. Ewees, and A. E. Hassanien, "Multi-objective whale optimization algorithm for content-based image retrieval," *Multimedia tools and applications*, vol. 77, no. 19, pp. 26 135–26 172, 2018.

[7] I. R. Kumawat, S. J. Nanda, and R. K. Maddila, "Multi-objective whale optimization," pp. 2747–2752, 2017.

[8] A. Got, A. Moussaoui, and D. Zouache, "A guided population archive whale optimization algorithm for solving multiobjective optimization problems," *Expert Systems with Applications*, vol. 141, p. 112972, 2020.

[9] Y. Zhou and Y. Tan, "Gpu-based parallel particle swarm optimization," in *2009 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2009, pp. 1493–1500.

[10] M. Jin and H. Lu, "Parallel particle swarm optimization with genetic communication strategy and its implementation on gpu," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 1. IEEE, 2012, pp. 99–104.

[11] D. Narjess and B. Sadok, "A new hybrid gpu-pso approach for solving max-csps," pp. 119–120, 2016.

[12] M. P. Wachowiak, M. C. Timson, and D. J. DuVal, "Adaptive particle swarm optimization with heterogeneous multicore parallelism and gpu acceleration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2784–2793, 2017.

[13] J. Kumar, L. Singh, and S. Paul, "Gpu based parallel cooperative particle swarm optimization using c-cuda: a case study," in *2013 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE, 2013, pp. 1–8.

[14] J. Hajewski and S. Oliveira, "Two simple tricks for fast cache-aware parallel particle swarm optimization," in *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2019, pp. 1374–1381.

[15] B. A. Menezes, H. Kuchen, H. A. A. Neto, and F. B. de Lima Neto, "Parallelization strategies for gpu-based ant colony optimization solving the traveling salesman problem," in *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2019, pp. 3094–3101.

[16] Y. Djenouri, D. Djenouri, A. Belhadi, P. Fournier-Viger, J. C.-W. Lin, and A. Bendjoudi, "Exploiting gpu parallelism in improving bees swarm optimization for mining big transactional databases," *Information Sciences*, vol. 496, pp. 326–342, 2019.

[17] C. Jin and A. K. Qin, "A gpu-based implementation of brain storm optimization," in *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2017, pp. 2698–2705.

[18] L. Ma, T. Zhang, R. Wang, G. Yang, and Y. Zhang, "Pbar: Parallelized brain storm optimization for association rule mining," in *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2019, pp. 1148–1156.

[19] B. Chen, B. Chen, H. Liu, and X. Zhang, "A fast parallel genetic algorithm for graph coloring problem based on cuda," in *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. IEEE, 2015, pp. 145–148.

[20] Y. Ma and L. S. Indrusiak, "Hardware-accelerated parallel genetic algorithm for fitness functions with variable execution times," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 829–836.

[21] H. Rico-Garcia, J.-L. Sanchez-Romero, A. Jimeno-Morenilla, H. Migallon-Gomis, H. Mora-Mora, and R. Rao, "Comparison of high performance parallel implementations of tlbo and jaya optimization methods on manycore gpu," *IEEE Access*, vol. 7, pp. 133 822–133 831, 2019.

[22] Y. Khalil, M. Alshayeji, and I. Ahmad, "Distributed whale optimization algorithm based on mapreduce," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, p. e4872, 2019.
[23] Y. Tan and K. Ding, "A survey on gpu-based implementation of swarm intelligence algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 46, no. 9, pp. 2028–2041, 2016.
[24] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of fpgas for heterogeneous platforms in hpc," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, 2015.
[25] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "Opencl-based fpga-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2016.
[26] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. Tahoori, "Energy efficient scientific computing on fpgas using opencl," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 247–256.
[27] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 25–34.
[28] K. Shata, M. K. Elteir, and A. A. EL-Zoghabi, "Optimized implementation of opencl kernels on fpgas," *Journal of Systems Architecture*, vol. 97, pp. 491–505, 2019.
[29] D. Li, L. Huang, K. Wang, W. Pang, Y. Zhou, and R. Zhang, "A general framework for accelerating swarm intelligence algorithms on fpgas, gpus and multi-core cpus," *IEEE Access*, vol. 6, pp. 72 327–72 344, 2018.
[30] H. M. Waidyasooriya, M. Hariyama, M. J. Miyama, and M. Ohzeki, "Opencl-based design of an fpga accelerator for quantum annealing simulation," *The Journal of Supercomputing*, vol. 75, no. 8, pp. 5019–5039, 2019.
[31] "Opencl overview," https://www.khronos.org/opencl/, 2019.
[32] "Intel fpga sdk for opencl," https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, 2019.
[33] "Intel fpga sdk for opencl pro edition: Best practices guide," https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf, 2019.
[34] "Intel fpga sdk for opencl pro edition: Programming guide," https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf, 2019.
[35] M. Jamil and X. S. Yang, "A literature survey of benchmark functions for global optimisation problems," *International Journal of Mathematical Modelling and Numerical Optimisation*, vol. 4, no. 2, pp. 150–194, 2013.