View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Heriot Watt Pure



Received October 21, 2018, accepted November 14, 2018, date of publication November 20, 2018,
date of current version December 19, 2018.

*Digital Object Identifier 10.1109/ACCESS.2018.2882455*

# A General Framework for Accelerating Swarm Intelligence Algorithms on FPGAs, GPUs and Multi-Core CPUs

**DALIN LI[1,2], LAN HUANG[1], KANGPING WANG[1], WEI PANG[3], YOU ZHOU[1], AND RUI ZHANG[1]**

[1]College of Computer Science and Technology, Jilin University, Changchun 13002, China
[2]Zhuhai Laboratory of Key Laboratory of Symbol Computation and Knowledge Engineering of Ministry of Education, Department of Computer Science and Technology, Zhuhai College of Jilin University, Zhuhai 519041, China
[3]Department of Computing Science, University of Aberdeen, Aberdeen AB24 3UE, U.K.

Corresponding author: Kangping Wang (wangkp@jlu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61472159, Grant 61572227, and Grant 61772227, in part by the Development Project of Jilin Province of China under Grant 20160204022GX, Grant 20170101006JC, Grant 20170203002GX, Grant 2017C030-1, and Grant 2017C033, in part by the Premier-Discipline Enhancement Scheme through Zhuhai Government, in part by the Premier Key-Discipline Enhancement Scheme through the Guangdong Government Funds, and in part by the Jilin Provincial Key Laboratory of Big Data Intelligent Computing under Grant 20180622002JC.

**ABSTRACT** Swarm intelligence algorithms (SIAs) have demonstrated excellent performance when solving optimization problems including many real-world problems. However, because of their expensive computational cost for some complex problems, SIAs need to be accelerated effectively for better performance. This paper presents a high-performance general framework to accelerate SIAs (FASI). Different from the previous work which accelerates SIAs through enhancing the parallelization only, FASI considers both the memory architectures of hardware platforms and the dataflow of SIAs, and it reschedules the framework of SIAs as a converged dataflow to improve the memory access efficiency. FASI achieves higher acceleration ability by matching the algorithm framework to the hardware architectures. We also design deep optimized structures of the parallelization and convergence of FASI based on the characteristics of specific hardware platforms. We take the quantum behaved particle swarm optimization algorithm as a case to evaluate FASI. The results show that FASI improves the throughput of SIAs and provides better performance through optimizing the hardware implementations. In our experiments, FASI achieves a maximum of 290.7 Mb/s throughput which is higher than several existing systems, and FASI on FPGAs achieves a better speedup than that on GPUs and multi-core CPUs. FASI is up to 123 times and not less than 1.45 times faster in terms of optimization time on Xilinx Kintex Ultrascale xcku040 when compares to Intel Core i7-6700 CPU/ NVIDIA GTX1080 GPU. Finally, we compare the differences of deploying FASI on hardware platforms and provide some guidelines for promoting the acceleration performance according to the hardware architectures.

**INDEX TERMS** Field programmable gate arrays, multicore processing, parallel programming, particle swarm optimization, pipeline processing.

## I. INTRODUCTION

Swarm intelligence (SI) emerges from the collective behavior of decentralized and self-organized systems. A typical SI system consists of a population of individuals which follow very simple rules and interact with each other by acting on their local environments. The interactions between such individuals can lead to the emergence of very complex global behavior, far beyond the capability of single individual [1]–[3]. Examples in natural SI systems include bird flocking, ant foraging, and fish schooling, etc.

Inspired by the nature swarm intelligence, a collection of algorithms have been proposed for optimization problems, such as particle swarm optimization (PSO) [4], [5], ant colony optimization (ACO) [6], [7]. PSO is inspired by the social behavior of bird flocking or fish schooling and widely used for real-parameter optimization. ACO simulates the foraging behavior of ant colony, and it has been successfully applied to solve various combinatorial optimization problems. Typically, the SI algorithms (SIAs) consist of three stages: (1) updating the individuals respectively; (2) fitness



2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

evaluation of the individuals; (3) communication between all the individuals for global information sharing [8]. These stages will be iterated several times until the termination criteria are met.

Although SIAs achieve varying degree of success in solving many real-world problems where conventional arithmetic and numerical methods do not perform well, such as routing design in wireless sensor network [9]–[11] and path planning [12]–[14], they suffer from the drawback of intensive computation caused by the time-consuming fitness evaluation process, which greatly limits their applications. Much effort [15], [16] has been made to improve the performance of SIAs. Based on interactions within population, SIAs are naturally amenable to be parallelized. Such an intrinsic property of SIAs makes them very suitable to be deployed on hardware platforms with the ability of parallel computing, such as multi-core CPUs, graphics processing units (GPUs), and field programmable gate arrays (FPGAs).

However, there will not be sufficient performance improvement if we only focus on the characteristic of parallelization without paying enough attention on the properties of the hardware platforms. In addition to parallelization, the SIAs also have a convergence characteristic determined by the interactions among all individuals. The interactions should be implemented through the comparison operations between each of the individuals and the only individual with the best fitness value. If the comparisons are not well organized by considering the efficiency of memory access according to specific hardware platforms, the memory access bottleneck and the throughput limitation may occur when the size of a swarm is large, and thus the performance of algorithm will be significantly reduced.

To further optimize the parallelization and convergence performance of SIAs, we propose a general framework for accelerating SIAs (FASI) on FPGAs, GPUs and multi-core CPUs. FASI provides a unified framework implemented by the C++ language, and it can be deployed on different hardware platforms.

FASI reschedules the process of SIAs based on their dataflows: it assigns the update of individuals and the fitness evaluation into *map_x* functions, and the communications among individuals into *reduce_x* functions. The *map_x* functions are function units of FASI which are mapped to parallel cores on hardware platforms, and *reduce_x* functions are function units of FASI which are implemented for the convergence of SIAs. The new dataflow of SIAs is a converged one which begins from parallel update, followed by parallel fitness evaluation and completes at the interactions among individuals. The number of *map_x* functions is determined by the parallel processing ability of the specific hardware platform and set by the tuning knobs of FASI. If the number of individuals is larger than that of *map_x* functions, FASI divides the individuals into groups, and then the individuals are updated group by group. For optimizing the performance of memory access caused by the convergence of SIAs, FASI uses a hierarchical convergence implemented by *combine_x* and *reduce_x* functions. Before reducing, a *combine_x* function is used for pre-converging a group of individuals by generating a group best individual, then all the group best individuals are converged by the final *reduce_x* function through a low cost bandwidth. Before compiling, FASI should be optimized for higher throughput according to the specific parallel processing ability and memory accessing feature of the hardware platforms.

We take quantum behaved particle swarm optimization (QPSO) algorithm [17], [18], an improved PSO, as the case to implement and evaluate FASI, and deeply optimize the case for the hardware platforms separately. On FPGAs, a combination of SIMD and full pipelined FASI is implemented based on the flexible architecture of FPGAs. On GPUs, we design a low sequential ID thread reduce (LSITR) to increase the occupancy of GPUs, and we also use combination in block to reduce the communication cost between blocks. On multi-core CPUs, a *combine_x* function in each thread improves the cache hit ratio.

In our experiments, eight standard benchmark functions are adopted for the performance evaluation and comparison. Compared with the state-of-the-art work on accelerating PSO [8], [19], FASI achieves higher throughputs as shown in Section 6. FASI on FPGAs reaches higher speedup than on other hardware platforms; compared with multi-core CPUs, FPGAs reach a maximum of 123 times speedup on benchmark function Sphere; compared with GPUs, FPGAs reach a maximum of 9 times speedup on benchmark function SchwefelP2.22.

In summary, this research makes the following contributions:

1) We propose the first unified general framework for accelerating SIAs on different hardware platforms. The framework is designed by matching the algorithm framework to the architectures of the hardwares for better throughput and optimized parallel programming modules.
2) We propose memory access optimization methods for better accelerating performance of the framework according to the characteristics of the hardware platform to be deployed.
3) We compare the differences of deploying FASI on hardware platforms, and provide some guidelines for promoting the acceleration performance according to the hardware architectures.

The rest of this paper is organized as follows: in Section II, we present some related work; Section III describes the characteristics of the parallel hardware platforms; in Section IV, we introduce swarm intelligence algorithms and analyze the parallel and MapReduce features of the algorithms; Section V gives the design and implementation of FASI; Section VI evaluates the performance of FASI; Section VII concludes the paper and explores future work.

## II. RELATED WORK

Accelerated SIAs has been used on solving many problems [20], [21]. Although much effort has been made, research on accelerating SIAs is still in its infancy and rising phase; especially good and convincible performance criteria are yet to be proposed.

Due to the characteristic of intensive computation, a direct method of accelerating SIAs is investing more computation units, especially more hardware cores, to construct a parallelized computation environment. As a result, the hareware platforms with multi-cores are widely used.

FPGAs are considered as one of the candidate hardware platforms for the parallel implementation of SIAs, which could accelerate algorithms by their customizable hardware structures. FPGAs cannot only provide customized parallel structures, but also provide flexible pipelines. A very deep pipelined multiplier of FPGAs was proposed in [22] with some initial guidance on deep pipeline structure designing. However, the multiplier provided is dedicated for matrix multiplication, and it needs to be significantly modified while applied on SIAs. One can also use the throughput of the FPGAs implementations to evaluate the acceleration performance. The work in [23] and [24] provide good reference on how to improve the throughput for dedicated algorithms.

There are many explorations on accelerating SIAs by FPGAs. In [25], a micro-architecture specific for QPSO was proposed. The dimension update part of QPSO, which owns obvious parallelization features, is accelerated by parallelized computation units, whereas the fitness evaluation part is still serial. This work only achieves speedup compared with an ARM platform. Another work in [26] accelerates an improved ACO on FPGAs and applies it for path planning, which is also composed of parallelized dimension update and serial fitness evaluation. For the better hardware resource usage,in [27], the proposed PSO accelerator packages the fitness evaluation function in an SP module which only calculates the fitness value of the particles and is shared by the paralleled swarm unit modules. Since the fitness evaluation costs more resource of FPGAs due to the complexity of the benchmark functions, a shared fitness evaluation function module provides an economic method for improving the degree of parallelization.

When the hardware resource is limited on the FPGAs chips, the upper acceleration strategy performs well. Considering that fitness evaluations consume most of the operation resource and time, a hardware/software co-design accelerator was proposed in [28]. The accelerator is implemented on a heterogeneous architecture, and the algorithms are divided into two parts: the benchmark functions were designed through software in the CPU-core; the left parts are deployed in parallel in PFGAs. This acceleration strategy is flexible when the benchmark functions are changed. And also, it can save more hardware resource to improve the degree of parallelization. However, the communication between the benchmark functions and particle update becomes the new performance bottleneck.

In order to reach full parallelization of SIAs, in [19], [29]–[31], full parallelized accelerators of PSO are developed on single FPGAs chips. Each particle has its own position update unit and fitness evaluation unit. All the particles are operated concurrently except for global information sharing. However, the accelerators still do not reach good acceleration ratio mainly because they follow the original dataflow of PSO. The accelerating ratio reported in [19] is between hardware accelerator and Micorblaze soft-core implementation. In [30], the accelerating ratio is achieved by comparing with a Matlab implementation on an Intel Core Duo at 1.6 GHz. But neither of them provided ideal acceleration benchmark.

Since the size of the swarm scale may increase, in [32] and [33], a multi-swarm strategy is developed. The particles are divided into sub-swarms, and all sub-swarms complete the iteration process by themselves. There is a communication controller, and the best position will be synchronized to all the other sub-swarms through the communication structure when it receives a better global best position request. In addition, a paralleled Genetic Algorithm (GA) on Multiple FPGAs has also been implemented in [34]. Both of them just immigrate parallel programming modules to the FPGAs platform and focus on the communication between devices, considering insufficiently on the characteristic of FPGAs, especially the customizable pipeline.

GPUs are another kind of hardware platform for the parallel implementation of the SIAs. Compared with FPGAs, GPUs have more RAMs and can accelerate SIAs of larger data scale. A PSO implementation in CUDA architecture is proposed in [35], which is aiming to speed up the algorithm on problems which has large amounts of data. Each processing core of the GPUs is responsible for a portion of the overall processing operations, and the parallelization ability of GPUs could be fully utilized. In [36], a set of bio-inspired optimization methods (PSO, Genetic Algorithm (GA), Simulated Annealing (SA), and pattern search) on GPUs are implemented, which can be a good index of implementing SIAs on GPUs. However, the implementation details are not given. The work in [8] provides a brief overview of recent studies on GPUs-based SIAs. The implementations of SIAs on GPUs are summarized as four modules. According to the experiment results, a PSO implemented in multiphase parallel model reaches better performance.

In addition to deploying an algorithm on a single hardware platform as shown previously, implementing the same algorithm on different hardware platforms and comparing the performance of the implementations will provide good insights for better accelerating algorithms. In [37], Markov Chain Monte Carlo (MCMC) is implemented and deeply optimized for better performance on multi-core CPUs, GPUs, and FPGAs. The experiment results provide good reference on how to choose acceleration platforms. In [38], the three-point Viterbi decoding algorithm was implemented on multi-core CPUs, GPUs, and FPGAs respectively, and the performance, such as decoding throughput, programming costs, and price

costs, was discussed. Both of the work accelerate the algorithms respectively on each of the hardware platforms, which discuss less on constructing unified parallel modules of the algorithms on different hardware platforms.

Inspired by the above previous work, we propose a new acceleration framework of SIAs, which can be coded in C++ language in the same framework, compiled and deeply optimized for different hardware platforms.

## III. THE CHARACTERISTICS AND PROGRAMMING MODELS OF FPGAs, GPUs AND MULTI-CORE CPUs

This section introduces the characteristics and programming models of FPGAs, GPUs, and multi-core CPUs.

### A. FPGAs

FPGA, acronym for Field Programmable Gate Array, is a kind of integrated circuit (IC) that can be programmed in the field after manufacture. The gates are connected according to the programs. Because of this flexible hardware architecture, FPGA has almost no bus bandwidth limitation, and it offers gate-level parallelization and deep alterable pipeline.

A classic FPGA chip is composed of control logic blocks (CLBs), which are the basic function units. A CLB is composed of Flip-Flops and LUTs, where Flip-Flops are responsible for storage, and LUTs are mainly used for arithmetic. Considering the higher demands for storage and arithmetic by applications, the manufacturers integrate hard cores in FPGAs to meet these demands and for higher integration, such as BRAM for storage and DSP for arithmetic. The prevail FPGA architecture is shown in Fig. 1, which is a typical Xilinx FPGA.[1]

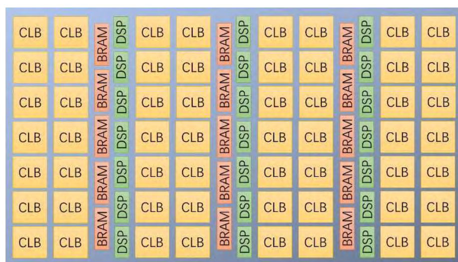



**FIGURE 1.** The architecture of FPGAs.

However, managing such a rich resource is not an trival task. The function modules are always described in hardware description language (HDL), e.g., VHDL and Verilog, which are designed for circuits description, but not logic description. A programmer has to transfer the arithmetic logic to the resistor transistor logic, which is challenging for programmers using high-level languages.

High-level synthesis (HLS) is an automated design process that interprets an algorithmic description of a desired behavior described in a high-level language such as SystemC and C/C++ and then creates HDL of the behavior. A designer typically describes the behaviors and the interconnect protocols; the high-level synthesis tools handle the micro-architecture and transform untimed or partially timed functional code into fully timed implementations. In recent years, HLS is successful in accelerating algorithms with low developing costs [39], [40].

### B. GPUs

GPUs have recently become popular as general computing devices due to their massively parallel architecture and improved accessibility provided by developing environments such as the Nvidia CUDA framework.

A GPU consists of multiple identical instances of computation units called Stream Multiprocessors (SMs). A SM is the computing unit where parallel execution of a group of threads, named thread blocks, happens. Each SM has one (or more) unit to fetch instructions, multiple ALUs (i.e., stream processors or CUDA cores) for parallel execution, a shared memory accessible by all threads in the SM, and a large register file which contains private register sets for each of the hardware threads. Each thread of a thread block is processed on an ALU. Since ALUs are grouped to share a single instruction unit, threads mapped on these ALUs execute the same instruction in each cycle, but with different data. Each logical group of threads sharing the same instructions is called a warp. Moreover, threads belonging to different warps can execute different instructions on the same ALUs, but in a different time slot. Actually, ALUs are time-shared between warps [41].

CUDA provides an accessible compiler and an API extension with familiar C-like constructs. In the programming model of CUDA, a task on GPUs is a grid, which contains blocks. A block is mapped to an SM on the GPUs, and consists of multiple threads. A thread is the minimum processing unit. A task accesses memory through a hierarchical structure as shown in Fig. 2. Each thread has private local memory. Each block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have to access the same global memory. Texture memory

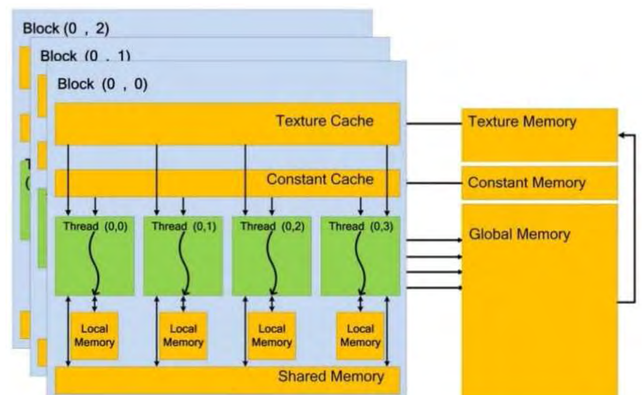



**FIGURE 2.** The architecture of GPUs.

[1]https://www.xilinx.com/products/silicon-devices/fpga.html

and constant memory are read-only, and they are cached for fast access. Different memories are quite different on bandwidth. Local memory and global memory are off-chip DRAM connecting to GPUs via GDDR5. Shared memory is essentially a block of programmable on-chip L1 cache with limited capacity. In order to improve the bandwidth of reading and writing, the share memory is divided into 32 blocks, named banks. A bank conflict will occur if multiple threads access different addresses of the same bank at the same time. The memory traffic must be carefully tuned to exploit the full bandwidth from all memory controllers.

### C. MULTI-CORE CPUs

A multi-core processor is a single computing component with two or more independent processing units called cores, which read and execute program instructions. The instructions are ordinary CPU instructions (such as add, data movement, and branch), but a single processor can run multiple instructions on separate cores at the same time (Multi-threading), thus increasing the overall speed for programs amenable to parallel computing [42].

OpenMP [43] is an implementation of multithreading, a method of parallelizing whereby a master thread forks a specified number of slave threads. The threads then run concurrently, with the runtime environment allocating threads to different processors. The communication between threads is completed by accessing share memory, which should be proceeded carefully to increase the throughput.

## IV. SWARM INTELLIGENCE ALGORITHMS (SIAs) AND DATAFLOW ANALYSIS

This section presents an overview and a dataflow analysis of SIAs through taking the quantum-behaved particle swarm optimization algorithm as a case.

### A. AN OVERVIEW OF SWARM INTELLIGENCE ALGORITHMS

The most respected and popular SIAs are particle swarm optimization (PSO), ant colony optimization (ACO), and etc. PSO is widely used for real value parameter optimization while ACO has been successfully applied to solve combinatorial optimization problems. As described in [8], although there are differences in details among SIAs, most of the SIAs follow the same computational framework and common features in the dataflow as shown in the left of Fig. 3, which makes it possible to provide a general acceleration framework.

### B. PARTICLE SWARM OPTIMIZATION AND QUANTUM-BEHAVED PARTICLE SWARM OPTIMIZATION

PSO, originally proposed by Eberhart and Kennedy in 1995 [4], is one of the classic SIAs. It is motivated by the behavior of organisms such as fish schooling and bird flocking. In PSO, each particle corresponding to an individual of the population is a candidate solution to the problem.

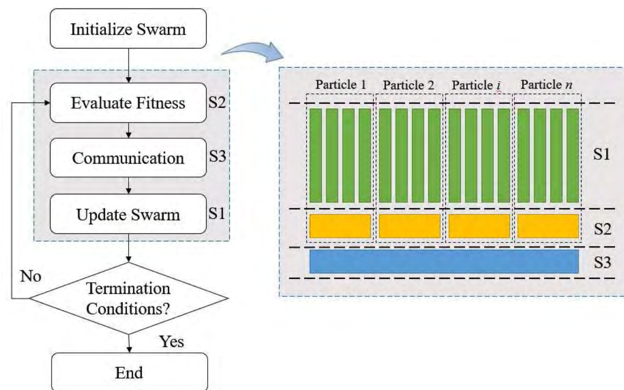



**FIGURE 3. The rescheduled dataflow of QPSO.**

The particles fly in a multi-dimensional search space to find an optimal or sub-optimal solution by competition as well as by cooperation among them. In terms of classical mechanism, a particle is depicted by its position vector $\vec{x}$ and velocity vector $\vec{v}$ , which determine the trajectory of the particle. The particle moves along a determined trajectory in Newtonian mechanics. The update equations in the classic PSO are shown as follow:

$$\begin{cases}\vec{v}_i \leftarrow \vec{v}_i + \vec{U}(0,\phi_1) \bigotimes (\vec{p}_i - \vec{x}_i) + \vec{U}(0,\phi_2) \bigotimes (\vec{p}_g - \vec{x}_i) \\ \vec{x}_i \leftarrow \vec{x}_i + \vec{v}_i,\end{cases} \tag{1}$$

where $\vec{U}(0, \phi_1)$ represents a vector of random numbers uniformly distributed in $[0, \phi_1]$ , and it is randomly generated at each iteration and for each particle. $\bigotimes$ is component-wise multiplication; $\vec{p}_i$ is the best ever position of particle $i$; and $\vec{p}_g$ is the global best ever position of all the particles.

PSO has undergone many modifications for better performance or solution of more specific problems since its introduction. In this research, we take the representative Quantum Behaved Particle Swarm Optimization (QPSO) algorithm [17], [18] as the instance of our algorithm acceleration framework. QPSO replaces the trajectory in Newtonian mechanics with the quantum mechanics. In the quantum world, the $\vec{x}$ and $\vec{v}$ of a particle cannot be determined simultaneously according to uncertainty principle, which lead to a better global convergence of the QPSO algorithm. The updating equations of QPSO are shown as follow:

$$\begin{cases}\vec{p}_i \leftarrow \vec{U}_1(0,\phi) \bigotimes \vec{p}_i + (1 - \vec{U}_1(0,\phi)) \bigotimes \vec{p}_g \\ \vec{x}_i \leftarrow \vec{p}_i \pm \vec{\beta} \bigotimes |\overrightarrow{mbest_i} - \vec{x}_i| \bigotimes ln(1/\vec{U}_2(0,\phi)),\end{cases} \tag{2}$$

where $\vec{U}_1(0, \phi)$ and $\vec{U}_2(0, \phi)$ represent the vectors of random numbers uniformly distributed in $[0, \phi]$ ,which is randomly generated at each iteration and for each particle. $\bigotimes$ is componentwise multiplication. $\vec{p}_i$ is the best ever position of particle $i$. $\vec{p}_g$ is the global best ever position of all the particles. $\overrightarrow{mbest_i}$ is defined as the mean of the $\vec{p}_i$ positions of all particles. $\beta$ is related to the times of iteration and calculated

by $\beta = (1.0 - 0.5) \cdot (T - t)/T + 0.5$, where $T$ is the total iteration times of the algorithm, and $t$ is the iterating count of the current iteration.

QPSO is one of the most efficient versions of PSO, and it keeps the same algorithm framework as the original PSO in [4]. The convergence of QPSO contains two parts: $\overrightarrow{mbest_i}$, which is implemented by accumulations, and $\vec{p}_g$, which is implemented by comparisons. The accumulations are the typical criterion for the memory access performance of a parallel program containing convergence [41]. Therefore, we take QPSO as an use case in our experiments.

### C. PARALLEL AND DATAFLOW ANALYSIS

In order to improve the performance of the intensive computation of SIAs, parallel computation can be used.

#### 1) PARALLELISING QPSO

A parallel analysis is necessary before designing a high performance parallel implementation of SIAs. The pseudocode of QPSO is shown in Algorithm 1. There are three stages in QPSO. Stage 1: line 7 $\sim$ 15, the positions of all particles are update separately; Stage 2: line 5, each particle evaluates its fitness in the new position by the fitness function, and compares the new fitness value with its best ever fitness value for the better fitted position of this iteration; Stage 3: line 6, all the particles compete for the global best position at this iteration, and this is called global information sharing. All the data and computations in Stage 1 are independent, which makes Stage 1 suitable for parallelization. In Stage 2, all the particles calculate their fitness values with the same fitness function, but there are no dependencies between the executions of the fitness function. Therefore, Stage 2 can also be parallelized. Stage 3 is the convergence of QPSO which could be optimized but not fully parallelisable.

**Algorithm 1** QPSO Algorithm

```
1:  for t = 1 to Maximum Iteration D do
2:    Compute the mean best position mbest;
3:    β = (1.0 − 0.5) · (T − t)/T + 0.5;
4:    for i = 1 to population size M do
5:      if if(x_i) < f(p_i) then p_i = x_i ; end if
6:      p_g = min(p_i);
7:      for j = 0 to dimension D do
8:        φ = rand(0,1); u = rand(0,1);
9:        p_ij = φ · p_ij + (1 − φ) · p_gj;
10:       if rand(0,1)> 0.5 then
11:         x_ij = p_ij + β · abs(mbest_j − x_ij) · log(1/2);
12:       else
13:         x_ij = p_ij − β · abs(mbest_j − x_ij) · log(1/2);
14:       end if
15:     end for
16:   end for
17: end for
```

#### 2) DATAFLOW ANALYSIS

In addition to parallelization, all data need to be read and written through the memories (registers, caches, RAMs) when QPSO is operated on processors. The efficiency of the memory access greatly affects the performance of the algorithms. We can improve the performance of the algorithms by a well-designed dataflow of the algorithms that takes account of the architectures of the memories.

We propose a rescheduled dataflow of QPSO as shown in Fig. 3. The left part of Fig. 3 is the dataflow of Algorithm 1, and the sections in the grey box will be iteratively excuted until the termination criterion is met. Stage 3 could be the bottleneck of the memory access because all the particles have to compare the fitness values generated from Stage 2 with that of the global best. The comparisons are serial, and the memory unit where the global best position stored will be accessed as many times as the number of particles. If the size of a swarm is large, the efficiency of Stage 3 will be very low. The right part of Fig. 3 is the rescheduled dataflow, which puts Stage 1 at the beginning, and is followed by Stage 2 and Stage 3 successively. S1 is processed in parallel, and the updated dimensions belonging to the same particle are transferred to the corresponding fitness evaluation unit in S2; the new fitness values from S2 will be received by the communication unit in S3. The rescheduled dataflow is convergent, and suitable for improving memory access through optimizations based on the memory architecture of specific processors.

## V. DESIGN AND IMPLEMENTATION OF FASI

This section presents the design and implementation of FASI. We first propose the baseline framework of FASI, then we implement FASI on FPGAs, GPUs and multi-core CPUs systems respectively. We also provide a deep optimization of FASI according to the architectures of the hardware platforms, which could provide some insights for the future work.

### A. BASELINE FRAMEWORK OF FASI

The previous studies on accelerating SIAs in the literature mainly focus on the parallelization of the algorithms that they implements the algorithms according to the data independence, and thus they do not pay enough attention to the architecture of specific platforms. These implementations have either a limited parallel scale and throughput [19] due to the low utilization of hardware resource or large data transmission delay caused by using the memory architecture inadequately [28], or low overall performance because of the frequent data exchange between RAM and cache [8].

In order to achieve better accelerating performance of SIAs and optimize the implementation of the algorithm by focusing on the architecture of specific platforms, we propose FASI. FASI is a general framework for accelerating SIAs on FPGAs, GPUs and multi-core CPUs, and it has the following features:

1) FASI provides a unified programming interface (*map_x* and *reduce_x*) and algorithm framework which could be compiled on different hardware platforms.

2) FASI adjusts the degree of concurrency through the tuning knobs to match the ability of the specific hardware platforms.
3) FASI is implemented in C++ language, and takes floating point as the data type for the portability across hardware platforms.
4) FASI reschedules the dataflow of SIAs for optimizing memory access.
5) FASI optimizes the parallelism of SIAs and the utilization of hardware based on the ability of the concurrent cores and the architecture of memories of the specific platforms.

**Algorithm 2** Baseline Framework of FASI

```
1:  for t = 1 to Maximum Iteration IT do
2:    for g = 1 to Group Number of Dimensions GD do
3:      for d = 1 to Dimension Number D do
4:        execute map_dimension();
5:      end for
6:    end for
7:    for g = 1 to Group Number of Individuals GI do
8:      for i = 1 to Individual Number I do
9:        execute map_particle();
10:     end for
11:     for c = 1 to Combination Number CN do
12:       execute combine_group();
13:     end for
14:   end for
15:   for r = 1 to Reduce Number R do
16:     execute reduce_group();
17:   end for
18:   execute reduce_global();
19: end for
```

The pseudocode of the baseline framework of FASI is shown in Algorithm 2. There are five user defined interfaces: *map_dimension*(), *map_particle*(), *combine_group*(), *reduce_group*() and *reduce_global*(). The implementation of the interfaces should be completed specifically by the designer based on the ability and features of particular hardware platforms. Each instance of *map_dimension*() is in charge of one paralleled update of dimension, i.e., Stage 1 described in Algorithm 1. *map_dimension*() has high concurrency and low computation intensity. Each instance of *map_particle*() is in charge of one paralleled fitness evaluation, i.e., Stage 2 described in Algorithm 1. The *map_particle*() has less concurrency than *map_dimension*() but it may be more computationally intensive because of the possible complexity of fitness functions. The *combine_group*() function combines a group of concurrent particles evaluated by *map_particle*(). A well designed *combine_group*() could decrease the access frequency of global memory through computing the intermediate global best position of the group of particles it belongs to. *reduce_group*() and *reduce_global*() construct a two-level convergence, and they are in charge of Stage 3 of Algorithm 1. *reduce_group*() is the level 1 convergence, which improves the utilization of cache through calculating the intermediate global best position generated by the *combine_group*() in groups if the size of the cache of the computation platform is small; *reduce_global*() is the level 2 convergence, which calculates the final global best position.

There are seven tuning knobs for the concurrency of the interfaces, i.e., $IT$, $GD$, $D$, $GI$, $I$, $CN$ and $R$. $IT$ is the times of iteration. $D$ is the number of paralleled dimension simultaneously, and the maximum value of $D$ is determined by the value of $C1$, the number of concurrent cores that could execute dimension update. Normally, $D = C1$. $GD$ is the group number that finishes all the dimension update. The relationship between $GD$ and $D$ is as follow:

$$GD = \frac{individuals \times dimensions\ per\ individual}{D} \tag{3}$$

$I$ is the number of the paralleled particles simultaneously, and the maximum value of $I$ is determined by $C2$ which is the number of concurrent cores that could execute fitness calculation. Normally, $I = C2$, and $C2 = C1$ while on CPUs and GPUs since dimension updating and fitness calculation are executed in the same core. However, $C2 < C1$ while on FPGAs, for the implementations of $C1$ and $C2$ cost resources on chip respectively. $GI = number\ of\ individuals/I$, which is the group number that finishes the computation of all the fitness value. $CN$ is the number of iteration of *combine_group*, while $R$ is the parallel number of *reduce_group*(). $CN$ will be kept at 1, and $R$ will be kept at 0 if the hardware platform on which FASI is deployed has enough cache.

We will introduce the principles of customizing *map_x* and *reduce_x* and the rules of choosing the values of the tuning knobs in the following sections while deploying FASI on different hardware platforms. All the implementations take QPSO as the example.

### B. FASI ON FPGAs

FPGAs offer the most flexible architecture for the hardware, including gate level parallel, variable memory bandwidth and all programmable memories without hierarchical structure. The advantage of FPGAs should be better explored through the combination of high concurrency and variable deep pipeline, which could achieve higher throughput when the pipeline is full. There will not be enough speedup if we only implement a paralleled structure of the algorithm.

#### 1) OPTIMIZED BASELINE FRAMEWORK OF FASI ON FPGAs

We optimize the baseline framework of FASI on FPGAs by setting the tuning knob $CN$ to 1 and $R$ to 0. The *reduce_global*() is moved to right after *combine_group*(), as shown in Algorithm 3. *#pragma* is the pre-compilation directive of Xilinx HLS. PIPELINE indicates to implement a for-loop with pipeline structures, and UNROLL indicates to implement a for-loop with parallel structures.

**Algorithm 3** Optimized Framework of FASI on FPGAs

1: **for** $t = 1$ to Maximum Iteration $IT$ **do**
2: #pragma HLS PIPELINE
3: **for** $g = 1$ to Group Number of Dimensions $GD$ **do**
4: #pragma HLS UNROLL
5: **for** $d = 1$ to Dimension Number $D$ **do**
6: execute *map_dimension()*;
7: **end for**
8: **end for**
9: **for** $g = 1$ to Group Number of Individuals $GI$ **do**
10: #pragma HLS UNROLL
11: **for** $i = 1$ to Individual Number $I$ **do**
12: execute *map_particle()*;
13: **end for**
14: execute *combine_group()*;
15: execute *reduce_global()*;
16: **end for**
17: **end for**
18: execute *reduce_tree()*;

### 2) FULL PIPELINED FASI ON FPGAs

The *reduce_global*() is for calculating the global best values (*mbest* and *Pg* of QPSO), and has to communicate with all of the *map_particle*() functions. If it is implemented on processors with hierarchical memories, such as GPUs and multi-core CPUs, the global best values should be stored in the global memory for the communications among threads, and barriers will be set before reductions or combinations for the synchronization of data of the threads. All of these will lead to a latency of the program. However, benefited from the flexible memory architecture of FPGAs, the communications and the synchronization on FPGAs could be better controlled without global memory access and the barriers because the dataflow of the program will be mapped to dedicated circuit connections, and the global best values could be stored in FFs. Therefore, we propose a full pipelined FASI on FPGAs, where the reductions and combinations are also pipelined together with the threads. As shown in Fig. 4, *map_dimension*(), *map_particle*(), *combiner*(), and *reducer_global*() are located in dimension map, particle map, combine, and reduce stages respectively, and the *reduce_global*() could be pipelined together with *map_dimension*() and *map_particle*() for higher throughput.

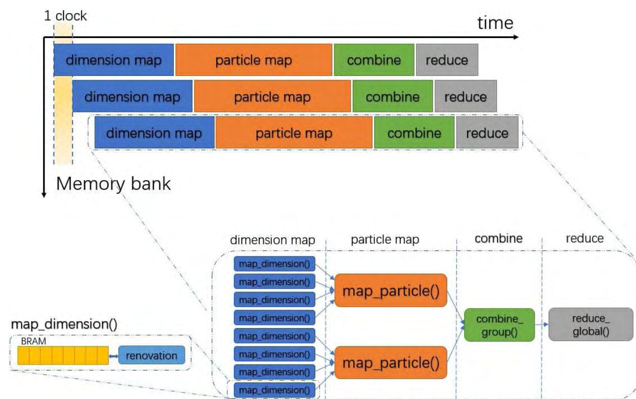



**FIGURE 4.** The full pipelined FASI on FPGAs.

### 3) SIMD

The *map_dimension*() functions are implemented in SIMD mode; the number of paralleled *map_dimension*() is the value of $C1$. Each *map_dimension*() stores the dimension data to be processed in a block of BRAM with FIFO mode. All the *map_dimension*() functions read dimension data in the same clock period. There are fixed connections between *map_dimension*() and *map_particle*() according to the relationship between dimensions and particles. The *map_particle*() functions work in SIMD, too.

Different from GPUs and CPUs, there are no fixed processing cores in FPGAs. Therefore, the numbers of paralleled *map_dimension*() and *map_particle*() are not limited by the number of processing cores, but limited by the bottleneck resource for the algorithm is accelerated, such as BRAMs, FFs, LUTs or DSPs. We use the following expression to determine the numbers of paralleled *map_dimension*(), i.e., the value of $C1$.

$$C1 = max\{C_{BRAM}, C_{FF}, C_{LUT}, C_{DPS}\}, \tag{4}$$

where $C_{BRAM}$ is calculated by the following expression:

$$C_{BRAM} = Q_{BRAM}/N_{BRAM}, \tag{5}$$

$Q_{BRAM}$ is the quantity of BRAMs of dedicated FPGA chip; NBRAM is the number of BRAMs occupied by a *map_dimension*().

Due to the different strategies that HLS translates C code to HDL code, we get the values of $C_{FF}$, $C_{LUT}$, and $C_{DPS}$, from the synthesis report of HLS tools. All the particle data are stored in BRAMs, and other intermediate values are stored in FFs, which we could find an explicit expression for the usage of BRAMs. However, the LUTs, DSPs, and FFs are used for arithmetic calculations, and the combination for a dedicated operation of them is accomplished by the HLS tools automatically. LUTs will also be used as routing resource for decreasing the fan-out of some central elements, and the ratio of LUTs usage should be obtained from the synthesis report of HLS tools.

*map_particle*() is mainly for fitness value calculation, which is a computationally intensive part of QPSO. As mentioned above, we cannot provide an explicit expression for the number of paralleled *map_particle*() functions, i.e., the value of $C2$.

However, there is a relationship between $C1$ and $C2$: a big value of $C1$ will lead to a big value of $C2$. A bigger $C2$ will lead to a sharp increase in the usage of the resource, which may extend the resource of a FPGA chip. So, the basic strategy is that a bigger $C1$ could be taken when the complexity of fitness function is lower, and a smaller $C1$ should be taken otherwise.

#### 4) FULL PIPELINE WITH 1 CLOCK INITIATION INTERVAL

Not only full pipelined, FASI is also a perfect 1 clock Initiation Interval pipeline, as shown in Fig. 4. Initiation Interval is defined as the number of clocks before the pipeline can accept new input data. In a pipelined structure, the value of Initiation Interval will affect the latency of the pipeline, i.e., the running time of the algorithm. The perfect 1 clock Initiation Interval requires the data being processed in 1 clock in each step of the pipeline. This could be achieved through increasing the depth of the pipeline and modifying the workload of each step, for instance, refining a float-point addition into multiple steps. However, only increasing the depth is not enough for QPSO because of the global information sharing in *reduce_global*(). As shown in lower part of Fig. 4, a group of dimension data in BRAMs will be read by *map_dimension*() every clock, and also the global best values of this paralleled group ($P_g^{GP}$ and $mbest^{GP}$ of QPSO) will be generated by *combine_group*() every clock, which is a typical pipeline process. Finally, *reduce_global*() calculates the $P_g^{GP}$ and $mbest^{GP}$ with the $P_g^{GP}$ and $mbest^{GP}$ of the previous iteration for the new global best values of this iteration ($P_g$ and $mbest$). The calculations in *reduce_global*() include floating-point comparation and addition which have to be finished in multiple clocks. Refining the floating-point comparation and addition into multiple steps to realize a 1 clock Initiation Interval pipeline is invalid in *reduce_global*(), because the new coming $P_g^{GP}$ and $mbest^{GP}$ cannot communicate with $P_g$ and $mbest$ until the previous floating-point operations are completed, and this is called carried dependency.

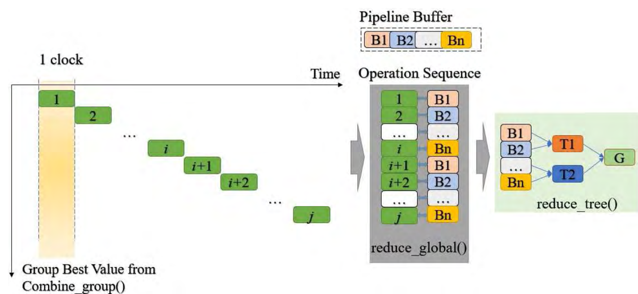



**FIGURE 5.** The pipeline buffer for 1 clock initiation interval.

In order to construct a full 1 clock Initiation Interval pipeline of FASI for accelerating QPSO, we design pipeline buffers for $P_g$ and $mbest$ respectively in *reduce_global*() as shown in Fig. 5. The size of the buffer $N_{PB}$ could be obtained according to $N_{PB} = N_c$, where $N_c$ is the clock number of a floating-point comparation or addition which is variable according to the frequency of FPGAs and should be obtained from HLS tools.

A pipeline buffer works as a circular queue by one temporary global best value ($P_g^T$ or $mbest^T$) in each of its elements. The new coming $P_g^{GP}$ or $mbest^{GP}$ of every clock chooses $P_g^T$ or $mbest^T$ from the tail of the queue for calculation, and the newly generated $P_g^T$ or $mbest^T$ will be put back where it comes from. The tail pointer of the queue increase, and the next new coming $P_g^{GP}$ or $mbest^{GP}$ will communicate with the next value in the queue. After Nc times of calculation, the tail pointer goes back to the head of the queue, and the calculation between $P_g^{GP}$ or $mbest^{GP}$ and the head element has just completed. In this way, a full 1 clock Initiation Interval pipeline is constructed. At the end of each iteration, a *reduce_tree*() calculates all the values in the queue for $P_g$ and $mbest$ by an adding tree or a comparing tree.

#### 5) BRAM BANDWIDTH

BRAMs provide a large capacity and discrete distribution but with limited bandwidth per block. If a higher bandwidth is required, the designer may take a lot of effort for assembling the BRAMs to fit the algorithms. In FASI, we designed a pattern of assembling BRAMs to meet the bandwidth requirement of SIMD as shown in Fig. 6. There are $D$ paralleled BRAM vectors with $GD$ depth per vector. Each vector belongs to a *map_dimension*(), and could be multiple serial BRAMs if the depth of one block is less than $GD$. According to this pattern, the bandwidth of BRAMs can be calculated by the following equation:

$$Bandwith = D \times wordwidth, \tag{6}$$

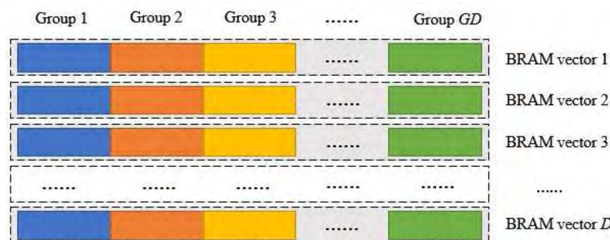



**FIGURE 6.** The BRAMs pattern.

#### 6) TRADE-OFF OF FASI ON FPGAs

*SIMD vs Micro − core*. A micro-core structure for function units, such as *map_dimension*() and *map_particle*(), has better integration and portability performance on FPGAs, which has been used by most of the previous work. However, the Initiation Interval of the pipeline will be increased because of the I/O communication overheads among micocores. And also, the micro-cores will be treated as atomic by HLS and cannot be refined into multiple steps if not well-designed. Finally, we choose SIMD structure for implementing *map_dimension*() and *map_particle*() functions.

*BRAMs vs FFs*. FFs provides flexible storage structure, larger bandwidth, but limited capacity, and BRAMs have larger capacity with limited bandwidth. Finally, we choose BRAMs for storing the particles and break the bandwidth limitation of BRAMs by an elaborated pattern of assembling BRAMs, which fits BRAMs for the SIMD so that FASI could handle larger scale of particles than before.

*SavingResource vs Convergence*. There are log functions in QPSO. The implementation of log functions on FPGAs

will costs many resources, and they become an obstacle for deploying scalable QPSO on FPGAs. We replace the log function with a lookup table of 256 log values, which will slower the convergence of QPSO but save a lot of resources.

### C. FASI ON GPUs

GPUs provide a many-core processor and hierarchical RAM architecture managed by CUDA. CUDA maps the parallel parts of the algorithms to kernel functions. Each kernel function is implemented as a grid on GPUs, while a grid contains multiple blocks, and a block contains multiple threads. Each thread is a processing core of the GPUs. The communication between threads of the same block is achieved through shared memory, and both the communications between grids and between blocks are achieved through global memory. An optimized program of GPUs should minimize the global memory access.

#### 1) OPTIMIZED BASELINE FRAMEWORK OF FASI ON GPUs

We optimize the baseline framework of FASI by setting the tuning knobs *GD*, *D*, *CN*, and *R* to be 0 and by moving the dimension update and combination into *map_particle*() as shown in Algorithm 4. Following the CUDA coding standard, we describe *map_particle*() and *reduce_global*() as kernel functions which will be explained later.

**Algorithm 4** Optimized Framework of FASI on GPUs

1: **for** $t = 1$ to Maximum Iteration *IT* **do**
2: execute *map_particle*< < < *GI*, *I* > > >();
3: execute *reduce_global*< < < 1, *GI* > > >();
4: **end for**

#### 2) PARALLEL STRUCTURE OF FASI ON GPUs

We propose a paralleled structure of FASI on GPUs as shown in Fig. 7. *map_particle*() and *reduce_global*() are mapped to kernel functions which are executed in parallel on GPUs. *map_particle*() is implemented by *GI* blocks, and each block contains *I* threads. Each thread is in charge of the position update and fitness evaluation of one particle. *reduce_global*() is implemented in 1 block and *GI* threads, and it is responsible for calculating the global best value.

#### 3) BLOCK AND THREAD CONFIGURATION

There are three constrains of limitations for choosing the values of *GI* and *I*: The Compute Capability (CC) of GPUs ($C_1$); the requirements of share memory, registers, and warps of a block ($C_2$); the feature of the algorithm ($C_3$).

The solutions of $C_1$ and $C_2$ are provided by the manual of GPUs and CUDA tool, and $C_3$ should be resolved according to the scale of the algorithms. For QPSO, the last constraint is described by the following two expressions:

$$N_B \times N_T \leq N_p, \quad (7)$$

$$N_T = 2^x, \quad (8)$$

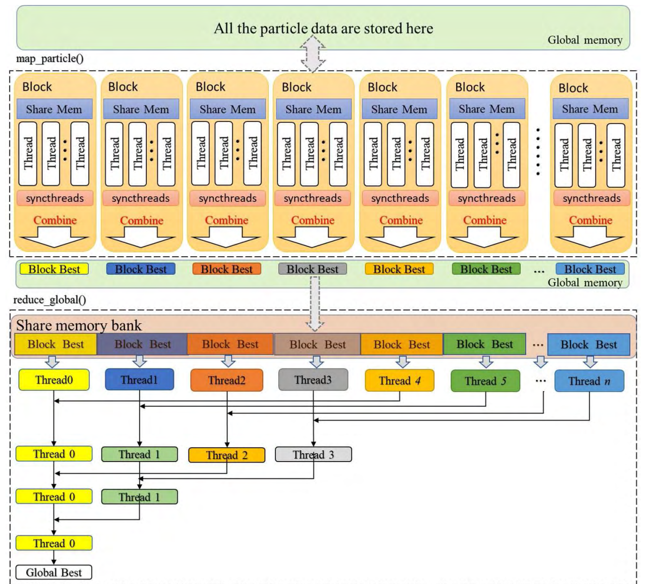



**FIGURE 7.** The parallel structure of FASI on GPUs.

where $N_B$ is the number of blocks, $N_T$ is the number of threads, and $N_p$ is the number of particles. The first expression ensures no idle threads, and the second one provides a high efficiency in the *reduce_global*() kernel function. And the final values of blocks and threads are determined by:

$$V_{B,T} = min(L_1, L_2, L_3), \quad (9)$$

#### 4) DATA STORAGE AND COMMUNICATIONS

In order to decrease the global memory access and improve the communication efficiency among threads, we deeply optimize the implementation of FASI on GPUs. All the particle data are stored in the global memory and divided into *mb* groups, just the same value as the number of blocks of *map_particle*(). Each thread of *map_particle*() reads the particle data from the global memory, updates the data and calculates the fitness value. After that, the threads rewrite the new particle data into the global memory and store the fitness values into the share memory. Before the reduction in the block, we use a CUDA system call function *__syncthreads*() to synchronize the threads. Then, all the threads belonging to the same block share the fitness values to calculate the block best values through a reduction process. At the end, the block best values of all the blocks are written into the global memory for calculating the global best values in *reduce_global*(). *reduce_global*() communicates with *map_particle*() through the global memory. It reads all the block best values into the share memory and calculates the global best values through a reducing process. In this way, only the limited block best values are shared between blocks, which can save the global memory bandwidth and shorten the runtime of the whole program.

### 5) LOW SEQUENTIAL ID THREAD REDUCTION

To deeply optimize the reduction process according to the architecture of GPUs, we propose Low Sequential ID Thread Reduction (LSITR) in *reduce_global*() as shown in Fig. 7. All the block best values generated from *map_particle*() are first moved into the share memory of the block, which can speed up the data reading and writing in the later operations. Each thread of *reduce_global*() can obtain the block best values from the corresponding share memory bank. The reduction processes are performed iteratively. Every reduction process is executed in the first half of all the current threads (the threads with low sequential IDs). This brings two benefits: first, all the operating threads locate in the same warp, which improves the occupancy of warps to decrease the thread latency; secondly, the threads access the share memory sequentially, which prevents the bank conflict so that the runtime of *reduce_global*() can be shortened and the performance of the whole program will be improved. Finally, the global best values are obtained in Thread 0.

## D. FASI ON MULTI-CORE CPUs

Multi-core CPUs are well managed by OpenMP. Each processor of the multi-core CPUs is equipped with the local cache. It is much faster a processor accesses its own local cache rather than accesses the main memory or the cache of other processors. An optimized design of multi-core CPUs should utilize the local cache efficiently.

### 1) OPTIMIZED BASELINE FRAMEWORK OF FASI

We optimized the baseline framework of FASI by setting the tuning knob *GD*, *D*, and *R* as 0, setting tuning knob *CN* as 1, and moving the dimension updating into *map_particle*(), which is shown as Algorithm 5. *#pragma* is the pre-compilation directive in OpenMP. $N_t$ indicates the number of parallel threads. Moving the dimension update into *map_particle*() enables the threads to load more dimension data to the cache, which could increase the cache hit rate.

**Algorithm 5** Optimized Framework of FASI on Multi-Core CPUs

```
1:  for t = 1 to Maximum Iteration IT  do
2:    #pragma omp parallel num_threads(N_t)
3:    for g = 1 to Group Number of Individuals GI  do
4:      for i = 1 to Individual Number I  do
5:        execute map_particle();
6:      end for
7:      #pragma omp critical
8:      execute combine_group();
9:    end for
10:   execute reduce_global();
11: end for
```

### 2) PARALLEL STRUCTURE OF FASI ON MULTI-CORE CPUs

We propose a paralleled structure of FASI on multi-core CPUs as shown in Fig.8. Each thread is in charge of $1/n$ part of the particles, where $n$ is the number of processors. The group best values of the particles of a thread are calculated by the *combine_group*(), and then stored in the share memory. In this way, there are only $n$ values that need to be read from the share memory for the *reduce_global*(). The Barrier keeps the synchronization between threads before *reduce_global*(). For the validity of the global best value, the *reduce_global*() has to be executed in a critical region in the main thread. The critical region guarantees the data to be read and calculated in sequence.

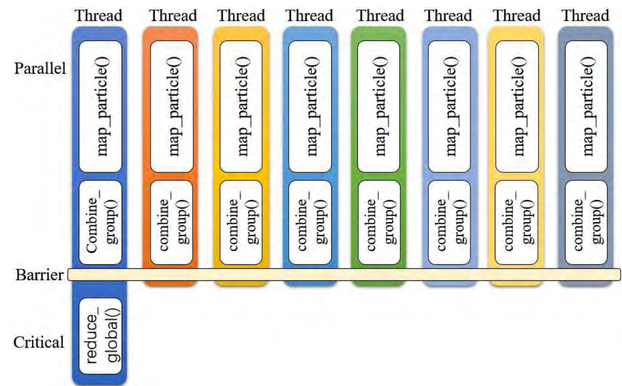



**FIGURE 8.** The parallel structure of FASI on multi-core CPUs.

In conclusion, the heavy workload of each thread could improve the cache hit rate, and the group best values could decrease the share memory access, which helps us provide an optimized FASI on multi-core CPUs.

## E. A COMPARISON BETWEEN THE IMPLEMENTATIONS WITH AND WITHOUT DATAFLOW RESCHEDULING

We compare the difference between the implementations with and without dataflow rescheduling on each of the hardware platforms separately.

### 1) THE COMPARISON OF IMPLEMENTATIONS OF FASI ON FPGAs

The most efficient part of FASI on FPGAs is the full pipeline with 1 clock initiation interval as discussed in Section V-B.4. To construct a pipeline structure for an algorithm, it is better to set the dataflow of the algorithm as a set of continuous forward operations without breaks from the beginning to the end. Before the rescheduling as shown in the left part of Fig. 3, the communication locates in the middle of the dataflow. The global best values will be generated only when all the particle data arrive. There is a break in the dataflow if implementing the algorithm with a pipeline. Without rescheduling, the algorithm has to be implemented with two pipelines before and after the communication. The efficiency of the whole implementation will be therefore decreased.

### 2) THE COMPARISON OF IMPLEMENTATIONS OF FASI ON GPUs

The communication efficiency between blocks and threads is very important for the performance of GPU programs,

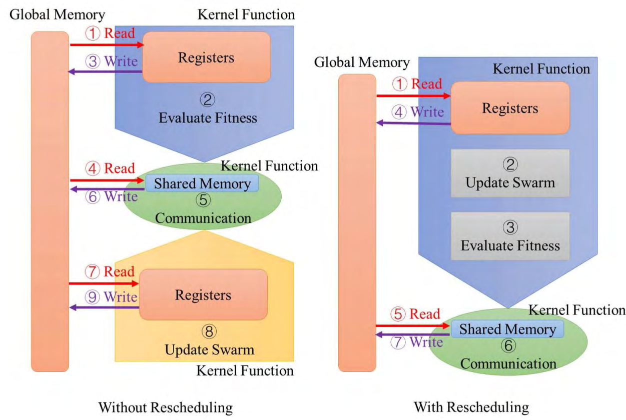



**FIGURE 9.** The memory access schedule of SIAs on GPUs.

which mainly depends on memory access. The memory access schedule of SIAs on GPUs is shown in Fig. 9. Before rescheduling, the dataflow is divided into three parts which have to be implemented by three kernel functions because of the communication among them. Therefore, there are three times of global memory access in each iteration of the algorithms. Benefited from the rescheduling, the "Update Swarm" and the "Evaluate Fitness" is implemented in the same kernel function, and one time global memory access is saved. The performance of the whole program will be improved.

#### 3) THE COMPARISON OF IMPLEMENTATION OF FASI ON MULTI-CORE CPUs

The memory access efficiency is very important for the performance of GPUs programs. Each core of the multi-core CPUs has its own cache, the communication among the cores is implemented through the shared memory. As shown in Fig. 10, before the rescheduling, there are three times of shared memory access in each iteration of the algorithms. However, there will be only twice shared memory access after rescheduling. Therefore, the runtime of the program will be shorter.

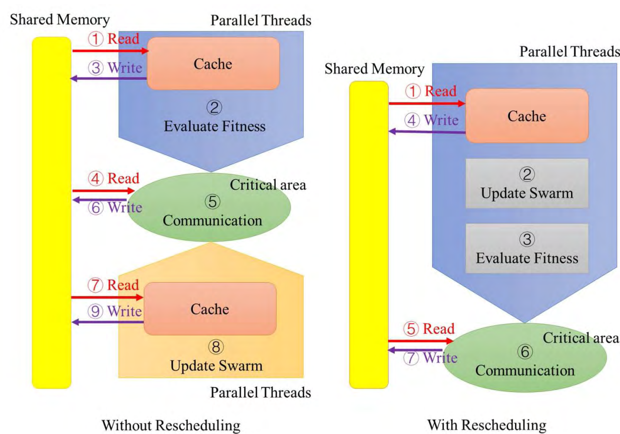



**FIGURE 10.** The memory access schedule of SIAs on CPUs.

## VI. EXPERIMENTS AND RESULTS

### A. EVALUATION PLATFORMS

We evaluate the performance of FASI using the following devices: For the FPGA platform, we present results for KCU105 (xcku040-ffva1156-2-e) evaluation board (@400MHz) hosted by Intel Xeon E5-2660 with 8 GBs of RAM and Xilinx Vivado 2017.2 tools. The FPGAs communicated with the host PC through a serial port. All the results on FPGAs are obtained from physical KCU105 board. For the GPUs platform, we use NVIDIA GeForce GTX 1080, visual studio 2015 and CUDA 8.0. For the multi-core CPUs, we use Intel Core i7-6700HQ (@2.6 GHz) with 8 threads. We use 32 GBs of RAM and OpenMP on Ubuntu Linux16.04. The -O3 optimization is used by the compiler on both multi-core CPUs and GPUs.

We also implemented the baseline framework of FASI on the Intel Core i7-6700HQ (@2.6 GHz) with one CPU core and -O3 optimization activated.

A resource comparison is listed in Table 1.

**TABLE 1.** Resource comparison of plateforms.

| | max Freq. | technology | Main Resource |
|---|---|---|---|
| KCU040 | 594MHz | 20nm | Flip-Flops:484800; LUTs:242400; BRAM(18k): 1200; DSP: 1920 |
| i7-6700 | 2.6GHz | 14nm | kernel:4; thread 8; L1/L2:256KB/1MB L3:6MB |
| GTX1080 | 1.6GHz | 16nm | shader:2560; L1/L2: 48KB/256KB; Shared Mem: 96KB |

We use eight benchmark functions for the evaluation of fitness of QPSO as shown in Table 2.

In order to provide a universal comparison criterion for the hardware platforms, and take the conclusion of [8] as a reference, we used the particle scenarios as shown in Table 3 in our experiments. The Particles per Group are divided into two sections ((f1, f2) and (f3-f8)) based on the complexity of the bench mark functions. f3-f8 involves sin/cos/exp which costs more resource on FPGAs to permit less parallel particles. For exploring the differences among the memory architectures, we set the biggest group number as 4000, which could make all the particle data just stored in the BRAMs on FPGA chips without data transmission between the BRAMs and the RAMs out of the FPGA chip.

### B. PERFORMANCE OF FASI ON FPGAs

#### 1) RESOURCE UTILIZATION

From the resource utilization, we could find out whether the resources of FPGAs are used effectively and where is

**TABLE 2.** Benchmark function.

| Function | Expression | Searching space | $X$ | $f_{min}$ |
|---|---|---|---|---|
| Sphere | $f_1(x) = \sum_{i=1}^{D} \lvert x_i^2 \rvert$ | [-100,100] | $\{0\}^D$ | 0.0 |
| SchwefelP2.22 | $f_2(x) = \sum_{i=1}^{D} \lvert x_i \rvert + \prod_{i=1}^{D} \lvert x_i \rvert$ | [-10,10] | $\{0\}^D$ | 0.0 |
| Ackley | $f_3(x) = exp(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}) - exp(\frac{1}{D}\sum_{i=1}^{D} cos(2\pi x_i)) + 20 + e$ | [-32,32] | $\{0\}^D$ | 0.0 |
| Griewank | $f_4(x) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} cos(\frac{x_i}{\sqrt{i}}) + 1$ | [-600,600] | $\{0\}^D$ | 0.0 |
| Rastrigin | $f_5(x) = \sum_{i=1}^{D}(x_i^2 - 10cos(2\pi x_i) + 10)$ | [-5.12,5.12] | $\{0\}^D$ | 0.0 |
| SchwefelP1.2 | $f_6(x) = 418.98288727216249D - \sum_{i=1}^{D} x_i sin(\sqrt{\lvert x_i \rvert})$ | [-500,500] | $\{402.96\}^D$ | 0.0 |
| Alpine | $f_7(x) = \sum_{i=1}^{D} \lvert x_i \sin(x_i) + 0.1x_i \rvert$ | [-10,10] | $\{0\}^D$ | 0.0 |
| Salomon | $f_8(x) = -\cos(2\pi\sqrt{\sum_{i=1}^{D} x_i^2}) + 0.1\sqrt{\sum_{i=1}^{D} x_i^2} + 1$ | [-100,100] | $\{0\}^D$ | 0.0 |

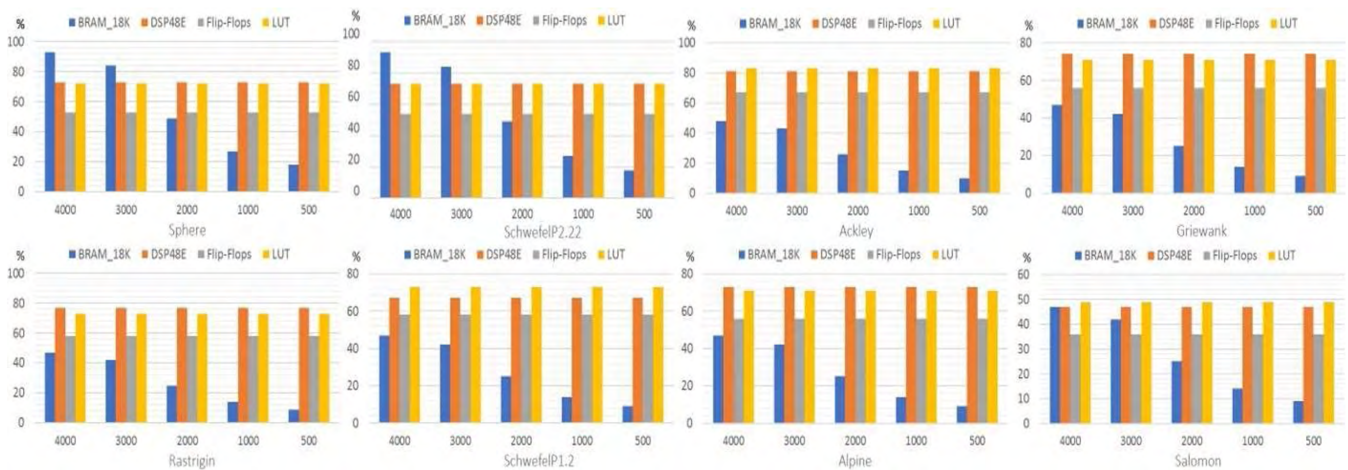



**FIGURE 11.** The resource occupation ratio of QPSO on FPGAs.

**TABLE 3.** Particle scenarios of experiments.

| Iteration | Dimension | Group Number (GI) | Particles per Group(I) (f1, f2) | Particles per Group(I) (f3-f8) |
|---|---|---|---|---|
| 2000 | 4 | 500 | 12 | 6 |
| 2000 | 4 | 1000 | 12 | 6 |
| 2000 | 4 | 2000 | 12 | 6 |
| 2000 | 4 | 3000 | 12 | 6 |
| 2000 | 4 | 4000 | 12 | 6 |

the resource bottleneck for the implementation of an algorithm. A comparison of the percentage of required hardware resources for the eight benchmark functions is illustrated in Fig. 11. The utilization of BRAMs in each benchmark function grows linearly with the increase of the number of the groups. There is a maximum of 93% utilization in $f1$ with 4,000 groups. Different from BRAMs, the other kinds of resources used for arithmetic have a fixed percentage of utilization in different groups, but change according to the complexity of different benchmark functions. There is a maximum of 81% utilization of DSP48E in $f3$. We also find that the BRAMs will be the bottleneck of the implementation when the complexity of the algorithms, mainly the complexity of the benchmark functions in QPSO, is low; on the other hand, the arithmetic resource (DSPs, LUTs) will be the bottleneck when the complexity of the algorithms is high. The implementation of the algorithm on FPGA will fail if the resource is not enough for the complexity. This is one of the most important rules which should be considered before implementing an algorithm on FPGAs.

### 2) PERFORMANCE

The runtime of each benchmark function is shown in Fig. 12, which also has a positive linear relationship with the number of groups. As described in Section V-B.2, a pipeline structure was designed in FASI on FPGAs, and the particles are divided into groups to flow into the pipeline group by group, which leads to the growth of the run time. The algorithm implemented by FPGAs will be synthesized to certain circuit connections on the chip without operating system, resource competition and so on. Therefore, there is a definite linear relationship between the runtime and the scale of the algorithm.

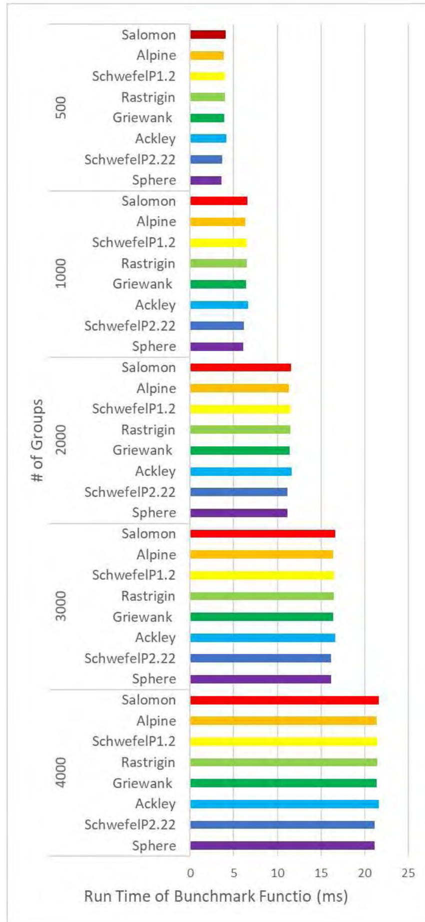



**FIGURE 12.** The run time of benchmark functions on FPGA.

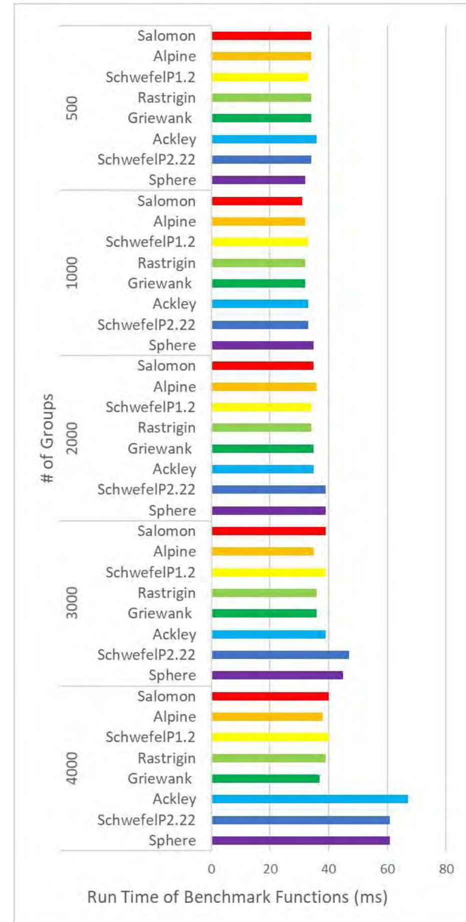



**FIGURE 13.** The run time of benchmark functions on GPUs.

**TABLE 4.** Pipeline depth of different benchmark function.

| Function | Pipeline Length |
|---|---|
| Sphere | 160 |
| SchwefelP2.22 | 164 |
| Ackley | 260 |
| Griewank | 211 |
| Rastrigin | 227 |
| SchwefelP1.2 | 235 |
| Alpine | 207 |
| Salomon | 255 |

The depth of the pipeline of FASI on FPGAs is alterable according to the complexity of the benchmark function as shown in Table 4, where the maximum 260-level depth is achieved on the Ackley benchmark function. The alterable pipeline is another advantage of FPGAs compared with GPUs and multi-core CPUs. However, a high performance pipeline, such as 1 clock II in FASI, is very challenging for a designer.

The degree of parallelism is shown bellow:

$$P = N \times D, \tag{10}$$

where $P$ denotes degree the of parallelism, $N$ denotes the number of particles per group, and $D$ denotes the number of dimensions per particle. Therefore, we got a $48 = 12 \times 4$ degree of parallel on $f1$ and $f2$, and a $24 = 6 \times 4$ degree of parallel on $f3$ $f8$, which is higher than multi-core CPUs.

There are performance improvements of FASI compared with the work in [19]. FASI proposes a structure that combines the variable deep pipeline and the parallelism, while only parallel structure was implemented in [19]. Another improvement of FASI is that the BRAMs are used for the storage of the particles, but the particles are only stored in FFs in [19]. Finally, FASI achieves a maximum of 290.7Mbit/s throughput on benchmark function Sphere, where in [19] the maximum value of throughput is only 22.2Kbit/s.

### C. PERFORMANCE OF FASI ON GPUs

We use the Nsight[2] tool for analyzing the performance of FASI on GPUs, and Nsight is a commonly used CUDA performance optimizing tool. For the comparison between platforms, the runtime of FASI on GPUs only includes the execution of the two kernel functions, but without data preparation time.

The occupancy of the kernel functions is shown in Table 5, which means the activation ratio of the Warps of a streaming multiprocessor (SM). According to the occupancy, we could find that FASI is well optimized on GPUs.

[2]http://www.nvidia.com/object/nsight.html

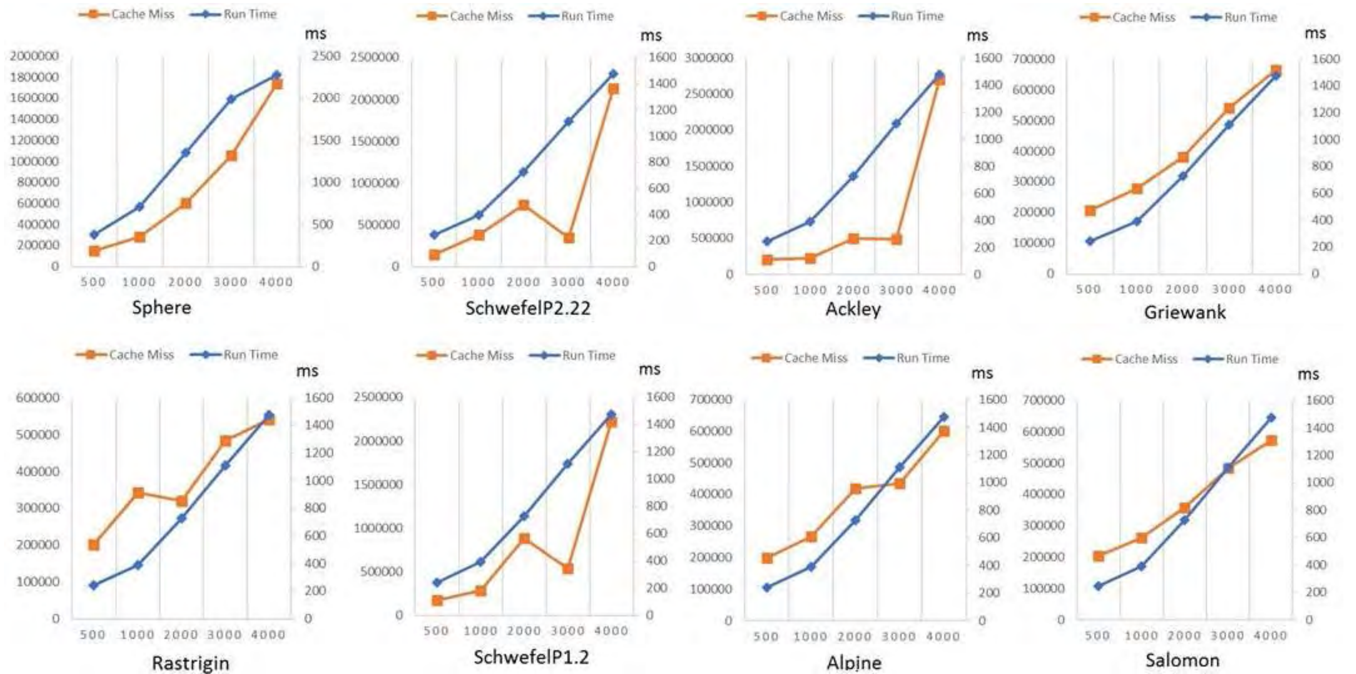



**FIGURE 14.** The run time and the cache hit ratio of benchmark functions on multi-core CPUs.

**TABLE 5.** Resource occupancy of kernel functions.

| Kernel function | occupancy |
|---|---|
| *map_particle()* | 62.5% |
| *reduce_global()* | 100% |

Considering the randomness of SIAs, we take the average value of 10 trails of program execution as the runtime for each benchmark function. The runtime of FASI on GPUs is shown in Fig. 13. There is an overall trend that the runtime increases with the increase of the size of the group. However, the trend between two neighbor groups is not as distinct as that of FPGAs. The increase in run time on GPUs is mainly due to the warp switching among threads and the data transmission among the shared memory and the local memory. The times of warp switching are almost the same in different group numbers because of the fixed numbers of paralleled particles, as shown in Table 3. The transmission latency between memories is small because of the sufficient bandwidth of GPUs comparing to the bandwidth required by the paralleled particles. Finally, the times of warp switching dominates the latency of FASI on GPUs, which leads to the increase of the run time as shown in Fig. 13.

The work in [8] only implements a multiphase parallel model on GPUs, which deploys communication (global information sharing) on multi-core CPUs and the other two stages of PSO on GPUs with twice the amount of data transmissions between GPUs and multi-core CPUs in the multiphase parallel model. Different from [8], FASI reschedules the dataflow which optimizes the memory access between different types of memories of GPUs, and deploy all sections of QPSO on GPUs. The data of particles are moved from multi-core CPUs to GPUs at the beginning, calculated on GPUs during the whole process, and transferred back to multi-core CPUs at the end. FASI achieves a maximum of 102.4Mbit/s throughput on benchmark function Sphere. We cannot compare the throughputs between FASI and the system in [8] due to the fact that only the speedup is published in [8].

### D. PERFORMANCE OF FASI ON MULTI-CORE CPUS

We use the *Perf* [3] tool for analyzing the performance of FASI on multi-core CPUs, which is a commonly used performance analysis tool on Linux. The run time is recorded by the system call *clock_gettime*(), which can calculate the running time of a thread. We take the average value of 10 times of program execution as the runtime for each benchmark function. The cache missing is recorded by *Perf*. The run time and cache missing of FASI on multi-core CPUs are shown in Fig. 14. Both the run time and cache missing increase in line with the growth of the group, which demonstrates that our dataflow rescheduling of QPSO is valid on the memory access. The rescheduling decreases the cross access of memory; only the increase in the size of the group dominates the increase in cache missing and the RAM access. FASI on multi-core CPUs achieves a maximum of 5.095Mbit/s throughput on benchmark function Salomon.

[3] http://www.eclipse.org/linuxtools/projectPages/perf/

### E. SPEEDUP COMPARISON BETWEEN FPGAs, GPUs AND MULTI-CORE CPUs

Based on the previous analysis, the maximum speedup of FASI occurs on FPGAs, which is mainly attributed to the following characteristics: 1) the efficiency of the dataflow of SIAs could be improved by customized pipeline structures; 2) constructing pipeline structures is the superiority of FPGAs; 3) the memory architecture of FPGAs is more flexible than that of GPUs and multi-core CPUs.

We also compare the run time among FPGAs, GPUs and multi-core CPUs. The speedup of FPGAs to multi-core CPUs is shown in Fig. 15. The variation of the speedup is not obvious between groups since FASI on FPGAs and FASI on multi-core CPUs have the same run time increase trends as the analysis in Section VI-B and Section VI-D.

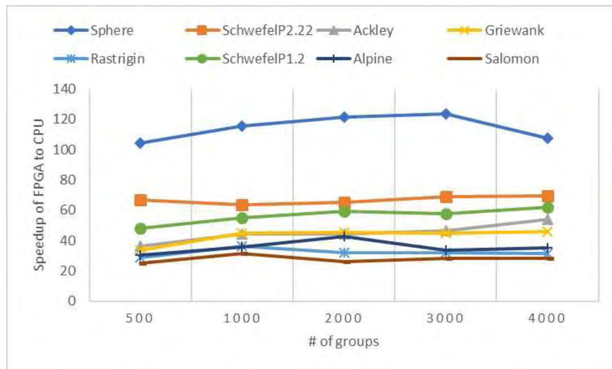



**FIGURE 15.** The speedup of FPGAs to multi-core CPUs.

The speedup of FPGAs to GPUs is shown in Fig. 16. The value of speedup decreases as the number of groups increases since the run time of FASI on GPUs did not increase as rapidly as the run time on FPGAs as discussed in VI-C. We can conclude that FASI will achieve better speedup on GPUs as the number of particle increases, and FASI on FPGAs can reach better speedup with a small number of particles in the group.

The speedup of GPUs to multi-core CPUs is shown in Fig. 17. As the number of particle increases, the GPUs achieves better speedup.

## VII. DISCUSSIONS AND CONCLUSIONS

FASI is a general acceleration framework for SIAs based on the uniform dataflow of the algorithms. Most of the SIAs has the same update and fitness evaluation progresses but different communication models. FASI separates these three stages into dedicated modules for further optimizations. The communication will be the throughput bottleneck of parallelizing SIAs due to the heavy data transmission and the hierarchal memory architectures of the hardware platforms. FASI reschedules the dataflow of SIAs to decrease the data transmission between different memory hierarchies, which improves the overall throughput. Because of the distributed memory architecture and the customized deep variable pipeline of FPAGs, FASI on FPGAs achieves better throughput.

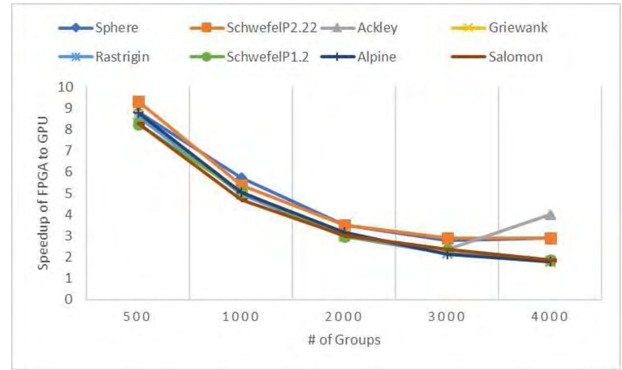



**FIGURE 16.** The speedup of FPGAs to GPUs.

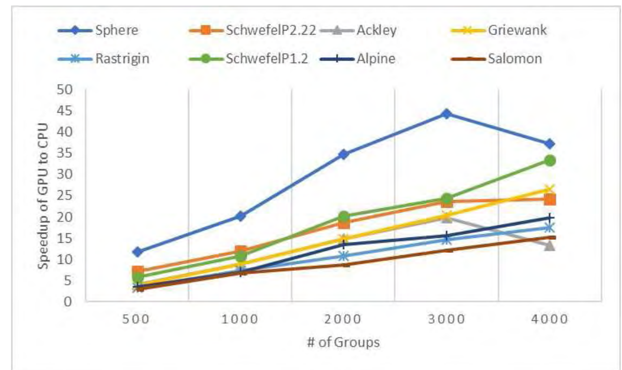



**FIGURE 17.** The speedup of GPUs to multi-core CPUs.

Another improvement of FASI compared to the previous studies is that FASI does not parallelize all the particles at the same time, and instead, it divides the paralleled particles into groups by considering both the number of parallel cores of the processors and the efficiency of the memory access. There could be a tradeoff between the parallelization and the bandwidth of the memory; a better acceleration for the algorithms should achieve a larger overall throughput.

Benefited from HLS, FASI can be implemented in C++ language and portable across hardware platforms. FASI offers open interfaces, e.g. *map_x*, *combine_x*, and *reduce_x*, and tuning knobs. A designer could adapt a particular SIA into the baseline framework of FASI first, then customize the structure of FASI by setting the values of tuning knobs and making appropriate modifications of the code based on the characteristics of the hardware platforms.

The different characteristics of hardware platforms incur varying cost for customizing the structure of FASI for better performance. In this research, we take QPSO as the case, which contains typical operations of the parallel computation, and QPSO achieves the highest speedup on FPGAs, followed by the speedup on GPUs, and the lowest speedup on muli-core CPUs. However, QPSO on FPGAs also costs our longest time for structure design and optimization, especially the *reduce_x* function, which is the biggest obstacle of 1 clock II pipeline. On GPUs, the design of LNTR takes up most of the time when implementing FASI, which needs a designer to be very experienced on the architecture of GPUs. On multi-core CPUs, the customization costs least, but it achieves the

lowest speedup. The efficiency of the *reduce_x* functions becomes the bottleneck of the whole workflow. According to the experimental results, FPGAs achieves the higher speedup while the size of the swarm is small, and the speedup of FPGAs to GPUs will decrease with the increase of the size of a swarm. We suggest that the designers deploy FASI on FPGAs while there are high performance requirement and sufficient time for designing the algorithm.

As described in [8], although there are differences in details among SIAs, most of them follow the same computational framework and common features in the dataflow. FASI constructs a general framework for SIAs based on the dataflow, which makes FASI work well for most of the SIAs. Furthermore, FASI optimizes the dataflow for the hardware platforms, which could achieve better speedup compared to the systems in [8] and [19].

The computation intensity of SIAs locates in the benchmark functions, and the benchmark funcions adopted in SIAs are almost the same ones among SIAs. Therefore, the main work that implements FASI for different SIAs is the implementation and optimization of the *reduce_x* functions due to the different communication strategies of SIAs, which still has to be completed specifically by the designer for a particular SIA in FASI at the moment.

In the future, we will further investigate and summarize different communication strategies of SIAs and provide common frameworks in the *reduce_x* functions for the SIAs with the same communication strategy so that we can further refine and improve FASI.

## REFERENCES


[1] R. C. Eberhart, Y. Shi, and J. Kennedy, *Swarm Intelligence*. San Francisco, CA, USA: Morgan Kaufmann, 2001.

[2] Y. Tan, *Fireworks Algorithm: A Novel Swarm Intelligence Method*. Berlin, Germany: Springer, 2015.

[3] A. P. Engelbrecht, *Fundamentals of Computational Swarm Intelligence*. Hoboken, NJ, USA: Wiley, 2005.

[4] R. Eberchart and J. Kennedy, "Particle swarm optimization," in *Proc. IEEE Int. Conf. Neural Netw.*, Perth, WA, Australia, Nov./Dec. 1995, pp. 1942–1948.

[5] D. Bratton and J. Kennedy, "Defining a standard for particle swarm optimization," in *Proc. IEEE Swarm Intell. Symp. (SIS)*, Apr. 2007, pp. 120–127.

[6] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 53–66, Apr. 1997.

[7] M. Dorigo and C. Blum, "Ant colony optimization theory: A survey," *Theor. Comput. Sci.*, vol. 344, nos. 2–3, pp. 243–278, 2005.

[8] Y. Tan and K. Ding, "A survey on GPU-based implementation of swarm intelligence algorithms," *IEEE Trans. Cybern.*, vol. 46, no. 9, pp. 2028–2041, Sep. 2016.

[9] P. S. Mann and S. Singh, "Energy-efficient hierarchical routing for wireless sensor networks: A swarm intelligence approach," *Wireless Pers. Commun.*, vol. 92, no. 2, pp. 785–805, 2017.

[10] N. Srivastava and P. Raghav, "A review on swarm intelligence based routing algorithms in mobile adhoc network," in *Proc. 8th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2017, pp. 1–7.

[11] W. Wang, S. Wu, and K. Lu, "Duck pack algorithm—A new swarm intelligence algorithm for route planning based on imprinting behavior," in *Proc. 29th Chin. Control Decis. Conf. (CCDC)*, May 2017, pp. 2392–2396.

[12] G. Li, Q. Liu, Y. Yang, F. Zhao, Y. Zhou, and C. Guo, "An improved differential evolution based artificial fish swarm algorithm and its application to AGV path planning problems," in *Proc. 36th Chin. Control Conf. (CCC)*, Jul. 2017, pp. 2556–2561.

[13] A. Hidalgo-Paniagua, M. A. Vega-Rodríguez, J. Ferruz, and N. Pavón, "Solving the multi-objective path planning problem in mobile robotics with a firefly-based approach," *Soft Comput.*, vol. 21, no. 4, pp. 949–964, 2017.

[14] Y. K. Ever, "Using simplified swarm optimization on path planning for intelligent mobile robot," *Procedia Comput. Sci.*, vol. 120, pp. 83–90, 2017. [Online]. Available: https://www.journals.elsevier.com/procedia-computer-science/

[15] W. Deng, R. Chen, B. He, Y. Liu, L. Yin, and J. Guo, "A novel two-stage hybrid swarm intelligence optimization algorithm and application," *Soft Comput.*, vol. 16, no. 10, pp. 1707–1722, 2012.

[16] S. M. Duan, J. L. Mao, J. L. Li, and L. X. Fu, "Design implementation and application of swarm intelligence algorithm optimization function simulation platform," in *Proc. Int. Conf. Softw. Eng. Inf. Technol. (SEIT)*, 2016, pp. 196–203.

[17] J. Sun, B. Feng, and W. Xu, "Particle swarm optimization with particles having quantum behavior," in *Proc. Congr. Evol. Comput.*, vol. 1, Jun. 2004, pp. 325–331.

[18] J. Sun, W. Xu, and B. Feng, "A global search strategy of quantum-behaved particle swarm optimization," in *Proc. IEEE Conf. Cybern. Intell. Syst.*, vol. 1, Dec. 2004, pp. 111–116.

[19] R. M. Calazan, N. Nedjah, and L. M. Mourelle, "A hardware accelerator for particle swarm optimization," *Appl. Soft Comput.*, vol. 14, pp. 347–356, Jan. 2014.

[20] Ö. Polat and T. Yildirim, "FPGA implementation of a general regression neural network: An embedded pattern classification system," *Digit. Signal Process.*, vol. 20, no. 3, pp. 881–886, 2010.

[21] S. E. Papadakis and A. G. Bakrtzis, "A GPU accelerated PSO with application to economic dispatch problem," in *Proc. Int. Conf. Intell. Syst. Appl. Power Syst.*, Sep. 2011, pp. 1–6.

[22] A. Panato, S. Silva, F. Wagner, M. Johann, R. Reis, and S. Bampi, "Design of very deep pipelined multipliers for FPGAs," in *Proc. Design, Automat. Test Eur. Conf. Exhib.*, vol. 3, Feb. 2004, pp. 52–57.

[23] M. F. Brejza, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A high-throughput FPGA architecture for joint source and channel decoding," *IEEE Access*, vol. 5, pp. 2921–2944, 2017.

[24] X.-T. Nguyen, T.-T. Hoang, H.-T. Nguyen, K. Inoue, and C.-K. Pham, "An FPGA-based hardware accelerator for energy-efficient bitmap index creation," *IEEE Access*, vol. 6, pp. 16046–16059, 2018.

[25] Z. Chai, J. Sun, R. Cai, and W. Xu, "Implementing quantum-behaved particle swarm optimization algorithm in FPGA for embedded real-time applications," in *Proc. 4th Int. Conf. Comput. Sci. Converg. Inf. Technol. (ICCIT)*, Nov. 2009, pp. 886–890.

[26] C.-C. Hsu, W.-Y. Wang, Y.-H. Chien, R.-Y. Hou, and C.-W. Tao, "FPGA implementation of improved ant colony optimization algorithm for path planning," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2016, pp. 4516–4521.

[27] Y. Maeda and N. Matsushita, "Simultaneous perturbation particle swarm optimization using FPGA," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Aug. 2007, pp. 2695–2700.

[28] S.-A. Li, C.-C. Hsu, C.-C. Wong, and C.-J. Yu, "Hardware/software co-design for particle swarm optimization algorithm," *Inf. Sci.*, vol. 181, no. 20, pp. 4582–4596, 2011.

[29] A. Rathod and R. A. Thakker, "FPGA realization of particle swarm optimization algorithm using floating point arithmetic," in *Proc. Int. Conf. High Perform. Comput. Appl. (ICHPCA)*, Dec. 2014, pp. 1–6.

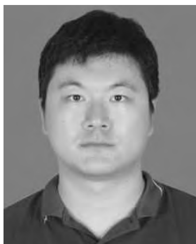
[30] D. M. Muñoz, C. H. Llanos, L. dos S. Coelho, and M. Ayala-Rincón, "Comparison between two FPGA implementations of the particle swarm optimization algorithm for high-performance embedded applications," in *Proc. IEEE 5th Int. Conf. Bio-Inspired Comput., Theories Appl. (BIC-TA)*, Sep. 2010, pp. 1637–1645.

[31] D. M. M. Arboleda, C. H. Llanos, and M. Ayala-Rincón, "Hardware architecture for particle swarm optimization using floating-point arithmetic," in *Proc. 9th Int. Conf. Intell. Syst. Design Appl. (ISDA)*, Nov./Dec. 2009, pp. 243–248.

[32] G. S. Tewolde, D. M. Hanna, and R. E. Haskell, "Multi-swarm parallel PSO: Hardware implementation," in *Proc. IEEE Swarm Intell. Symp. (SIS)*, Mar./Apr. 2009, pp. 60–66.

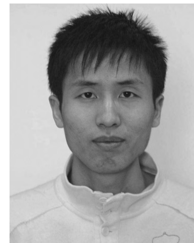
[33] G. S. Tewolde, D. M. Hanna, and R. E. Haskell, "A modular and efficient hardware architecture for particle swarm optimization algorithm," *Microprocess. Microsyst.*, vol. 36, no. 4, pp. 289–302, 2012.

[34] L. Guo, A. I. Funie, D. B. Thomas, H. Fu, and W. Luk, "Parallel genetic algorithms on multiple FPGAs," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 86–93, 2016.
[35] D. L. Souza, G. D. Monteiro, T. C. Martins, and V. A. Dmitriev, "PSO-GPU: Accelerating particle swarm optimization in CUDA-based graphics processing units," in *Proc. Annu. Conf. Companion Genetic Evol. Comput.*, 2011, pp. 837–838.
[36] F. Valdez, P. Melin, and O. Castillo, "Bio-inspired optimization methods on graphic processing unit for minimization of complex mathematical functions," in *Recent Advances on Hybrid Intelligent Systems*. Berlin, Germany: Springer, 2013, pp. 313–322.
[37] G. Mingas and C.-S. Bouganis, "Population-based MCMC on multi-core CPUs, GPUs and FPGAs," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1283–1296, Apr. 2016.
[38] R. Li, Y. Dou, and D. Zou, "Efficient parallel implementation of three-point Viterbi decoding algorithm on CPU, GPU, and FPGA," *Concurrency Comput., Pract. Exper.*, vol. 26, no. 3, pp. 821–840, 2014.
[39] L. Ma, L. Lavagno, M. T. Lazarescu, and A. Arif, "Acceleration by inline cache for memory-intensive algorithms on FPGA via high-level synthesis," *IEEE Access*, vol. 5, pp. 18953–18974, 2017.
[40] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.
[41] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. 16th ACM Symp. Princ. Pract. Parallel Program. (PPoPP)*, 2011, pp. 267–276.
[42] R. Buchty, V. Heuveline, W. Karl, and J.-P. Weiss, "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators," *Concurrency Comput., Pract. Exper.*, vol. 24, no. 7, pp. 663–675, 2012.
[43] OpenMP Community. (2015). *OpenMP 4.5 Complete Specifications*. Accessed: Nov. 2015. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf



**DALIN LI** received the bachelor's and master's degrees in mechatronic engineering from Liaoning Technical University, Fuxin, China, in 2005 and 2008, respectively. He is currently pursuing the Ph.D. degree in computer science with the College of Computer Science and Technology, Jilin University, Changchun, China. His current research interests include high performance computing, swarm intelligence, and machine learning.

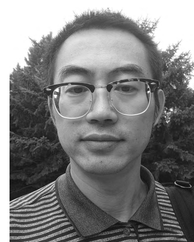
**LAN HUANG** received the Ph.D. degree. He is currently a Professor and a supervisor for Ph.D. candidates. She is mainly engaged in business intelligence theory and application research. She was one of the outstanding youth project funding winners of Jilin Province in 2005, and the person in charge of Young and Middle-aged Leader and Innovation Team of Jilin Province in 2012. She was invited to Italy Trento University as a Senior Visitor in 2010. As PI and Co-PI, he has been undertaking or accomplished more than 10 teaching and scientific research projects, granted by the National 863 Hi-tech Research and Development Program, the National Science Foundation China, provincial/ministerial foundations, and other sources. The works that she participated as main investigator, were awarded the first prize for the National Commercial Science and Technology Award bestowed by the China General Chamber of Commerce (the first prizewinner in 2010), the second prize for the Jilin Province Scientific and Technological Progress Award (first prizewinner in 2011), the second prize for the National Commercial Science and Technology Award (forth prizewinner in 2007, sixth prizewinner in 2004), the second prize for the Jilin Province Scientific and Technological Progress Award (forth prizewinner in 2004), and the second prize of Jilin Province Teaching Achievements (second prizewinner in 2005). In recent years, her research interests focus on business intelligence application and social network mining algorithm. She has published 64 academic papers, and obtained seven software copyrights. The software results researched and developed by her team have brought good economic benefit for the cooperative enterprises and application enterprises.



**KANGPING WANG** received the bachelor's, master's, and Ph.D. degrees from Jilin University in 2000, 2003, and 2008, respectively. He is currently a Faculty Member with the College of Computer Science and Technology, Jilin University. In past several years, his research interests include heterogeneous computing and deep learning.

**WEI PANG** received the Ph.D. degree from the University of Aberdeen in 2009. He is currently a Senior Lecturer with the School of Natural and Computing Sciences, University of Aberdeen. In past several years, his research interests include machine learning, qualitative reasoning, and evolutionary computing.

**YOU ZHOU** received the bachelor's and Ph.D. degrees from Jilin University in 2002 and 2008, respectively. He is currently an Associate Professor with the College of Computer Science and Technology, Jilin University. In past several years, his research interests include heterogeneous computing and machine learning.



**RUI ZHANG** received the Ph.D. degree from the University of Trento in 2009. He is currently an Associate Professor with the College of Computer Science and Technology, Jilin University. In past several years, his research interests include knowledge representation and machine learning.