Available online at www.ijpe-online.com

International Journal of Performability Engineering

Vol. 13, No. 6, October 2017, pp. 854-863
DOI: 10.23940/ijpe.17.06.p7.854863

HACO-F: An Accelerating HLS-Based Floating-Point Ant Colony Optimization Algorithm on FPGA

Shuo Zhang[a,b], Zhangqin Huang[a,b,*], Weidong Wang[a,b,*], Rui Tian[a,b], Jian He[a,b]

[a]*Beijing Advanced Innovation Center for Future Internet Technology, Beijing University of Technology, Beijing 100124,China*
[b]*Beijing Engineering Research Center for IOT Software and Systems, Beijing University of Technology, Beijing 100124,China*

Abstract

In this paper, a novel accelerating Ant Colony Optimization (ACO) algorithm based on High-Level Synthesis (HLS) on FPGA (Field Programmable Gate Array) is proposed. The proposed algorithm (HACO-F) is implemented by C/C++ programming language and calculated by floating-point. For the sake of accelerating, the algorithm mainly employs the data optimization strategy to redefine the variables precision in HACO-F to reduce resource utilization and energy consumption. Then, we explore a loop optimization strategy including pipeline and unroll to parallelize loops in HACO-F to decrease computation time. The experimental results show that the HACO-F algorithm can achieve more than 6 times accelerating performance than that of the AS (Ant System) at the same search ability. The resource utilization in HACO-F is 1% FF, 4% LUT, and 9% BRAM decrease. The total on-chip energy consumption of HACO-F is reduced by 23.9%.

Keywords: high-level synthesis; algorithm acceleration; ant colony optimization; FPGA; embedded system

(Submitted on July 25, 2017; Revised on August 30, 2017; Accepted on September 15, 2017)

(This paper was presented at the Third International Symposium on System and Software Reliability.)

1. Introduction

With developments in integration, reconfiguration, and performance optimization in FPGA research, it has attracted more attention from academia and industry, especially in the algorithm optimization on FPGA (e.g. ACO algorithm optimization). Hence, much research has investigated how to accelerate the ACO algorithm on FPGA [5,6,12,13,14,15].

In the early stage, research for ACO algorithm acceleration on FPGA was based on Hardware Description Language (HDL). Guntsch et al. [5] first implemented the ACO on FPGA devices using VHDL language, *i.e.* P-ACO [5,14]. In their research, it is hard to map resource on FPGA. For instance, the calculations of pheromone values are executed when the FIFO-queue determines that a better solution can replace an older population. Moreover, it might need a lot of the hardware knowledge to implement the P-ACO on FPGA, which raised the bar for accelerating the ACO algorithm. After that, Scheuermann et al. [13] proposed a counter-based ant colony optimization (C-ACO) for the resource mapping on FPGA. The C-ACO is a more suitable method than P-ACO to implement on FPGA. Juang et al. [9] proposed the ACO-FC based on a fuzzy controller system design. In their design, the key values of heuristic information are ignored in calculating transition probabilities. This may result in a low-quality convergence.

Currently, research has turned to accelerating the ACO algorithm using both hardware and software co-design methods [7,8,16]. Hsu et al. [7] designed an improved ACO algorithm and implemented on the FPGA. To avoid local minima, they continuously tuned a setting parameter and employed novel mechanisms for updating partial pheromone and opposite

* Corresponding author.
E-mail address: zhuang@bjut.edu.cn, wangweidong@bjut.edu.cn

pheromone. Huang et al. [8] presented an ant colony optimization-particle swarm optimization (ACO-PSO) algorithm by combining the system-on-a-programmable chip (SoPC) and HW/SW co-design method. In their ACO-PSO algorithm, the communication among different models could be a performance barrier on FPGA.

Above all, these accelerating methods are more or less based on the HDL, which needs a lot of manual work and time for simulation, synthetization and implementation [8]. Moreover, to avoid the complexity of implementation on FPGA, the ACO could be simplified or cut down on some functions. This could lead to the downgrade of algorithm performance. Complementary to the above methods, this paper proposes a novel accelerating algorithm, which employs a High-Level Synthesis (HLS) [2] method to accelerate the floating-point ACO algorithm (HACO-F) on FPGA by "C/C++" language.

The contributions of the proposed HACO-F are listed as follows.

1) The proposed HACO-F can decrease resource utilization under an acceptable accuracy by a data optimization strategy. We employ data optimization to redefine the variable precision of HACO-F to reduce resource utilization and energy consumption. Moreover, the definition of the overflow type also improves the search ability of HACO-F.
2) The execution time of HACO-F is greatly reduced by adopting loop optimization strategies in HLS. We use loop optimization strategies including PIPELINE and UNROLL to parallelize loops in HACO-F to decrease execution time.

In addition, we implement a prototype system for HACO-F including communication interfaces. The rest of this paper is organized as follows. Section 2 introduces the optimization strategies of the HLS. Section 3 describes the design and implementation of HACO-F algorithm in detail. Section 4 presents the experimental results. Section 5 concludes this paper and outlines the future work.

2. Preliminary

The HLS provides many optimization strategies for accelerating algorithms that mainly include two aspects: data optimizations and loop optimizations. The HACO-F design is based on these acceleration strategies.

2.1. Data optimization

In the FPGA, resources such as hardware memory are very limited and precious. For example, the byte-based data storage implemented by the standard C/C++ could result in the waste of these memory resources. This is because that when a bit is required to represent a bool variable, the use of an 8-bit byte variable for data storage may result in other 7-bit waste. Especially when the valid bits are up to 34 bits, a float type to conduct a multiplication may use a 64-bit memory storage to guarantee the accuracy. The extra utilization of bits-based memory storage not only consumes a lot of resources, but also takes a long computation time. Hence, it is reasonable to believe that a reasonable definition of the variables' bits in memory in an algorithm can improve the performance of algorithm.

Besides the variables' valid bits, the behaviors of overflow and underflow can also be defined. When overflowing, the variables can be saturated to the maximum value, zero, or just wrapped around to throw the overflow bits. The wrap-around way will lose the most significant bit, leaving just the follow bits. However, a suitable definition of the variables' overflow type may lead to a brand new efficiency of the algorithm, especially when used to a probabilistic variable. The least significant bits of the variables can be saturated to the nearest value when underflow happens.

In addition, different protocols to transfer data may take different effects on speed. If the handshaking signals of the interface protocol are too frequent, the transfer of large size data may cost more delay cycles. Therefore, to achieve fast variable data transfers, the efficient protocols chosen may also influence the efficiency of the algorithms.

2.2. Loop optimization

Loop optimization uses configurable computation resources to complete loops or functions in parallel. In FPGA, there are some available logic and memory resources, and they can be used to accelerate a loop by specifying a loop optimization method as follows.

Loop pipelining (PIPELINE) can execute following instructions before total completion of concurrent operations. For example, it is reasonable to execute a read operator and an add operator in a loop at the same clock cycle, as there is no resource overlap between the two operators, thereby reducing the waiting time. This is a much better strategy in increasing the loop throughput and speed when there are limited logic and memory resources.

Loop unrolling (UNROLL) utilizes redundant resources to make all loop instructions completed in one clock cycle. As opposed to with PIPELINE, UNROLL uses the same resources to enter the loop partly or totally in parallel. This optimization strategy can greatly cut down execution delays of loops, namely trading space for time.

3. HACO-F design based on HLS

The HACO-F is designed on HLS by integrating the Ant System（AS) algorithm [4], as the computation scale is larger than those of other ACOs. The HACO-F design is composed of two aspects. One is data optimization design, which can be employed to redefine the precision of variables for further reduction of resource usage. The other is loop optimization design, which can be used to make the AS algorithm paralleled and integrated in FPGA. The notations of HACO-F are explained in Table 1.

Table 1 Notations of HACO-F

Notations	Specification
d	city location data
n	city quantity
m	ant quantity
NC	trip quantity
IN	pheromone initial value
seed	random function seed
ρ	volatilization coefficient
Q	constant coefficient for the calculating of $\Delta\tau$
η	heuristic information , the visibility of the road
tabu	visited cities list of ants
allowed	unvisited cites list of ants
L	tour length of all the ants
τ	road pheromone
$\Delta\tau$	update pheromone
p	transition probability
besttour	best tour map
tourlength	best tour length
α	parameter of τ
β	parameter of η
i	index variable of n
j	index variable of n
s	index variable of n
k	index variable of m

3.1. HACO-F data optimization design

Data optimization is to redefine the precision of the variables used in the algorithm to further reduce the resource usage. In the process of data optimization, it mainly focuses on the boundaries of input variables, which may differ in different applications. According to the boundaries of input variables, it is easy to redefine the valid bits of these input variables. Especially for float type variables, we redefine the precision by the following transition principle. For the precision insensitive variables, 6bits in decimal (20bits in binary) is defined, and the total bits should be defined smaller than 32bits (the same with float type). Otherwise there would be little optimization or even worse. For some precision sensitive variables, the total bits may be up to 64bits (the same with *double* type). In particular, we may approximate the percentage values with 6bits in binary. Other variables' valid bits are defined based on the relations such as the operations relations and valid bits dependency relations.

To enhance the flexibility, we adopt variables as input. The HACO-F input variables include city location (*d*), city quantity (*n*), ant quantity (*m*), trip quantity (*NC*), pheromone initial value (*IN*), random seed (*seed*), volatilization coefficient (ρ), and constant coefficient (Q). Intermediary variables include the visibility of the road (η), visited cities list (*tabu*), unvisited cites list (*allowed*), the tour length (*L*), pheromone (τ), pheromone update ($\Delta\tau$), and the transition probability (*p*). Output variables contain best tour map (*besttour*) and tour length (*tourlength*). In addition, there are some predefined parameters such as, α, β and index variables such as i,j,k.

The scope of ρ is [0,1], we redefine it as percentage values, ufixed(7,1), integer 1bits, and decimal 6 bits. Q is a constant in AS, We redefine Q as ufixed(32,16), integer 16 bits, decimal 16 bits, and as a input variable for the flexibility of HACO-

F. *seed* and *NC* are defined with *int* type for better randomness and greater searching ability. Input variable *IN* is dependent on τ. To limit the total resource of HACO-F, we redefine *d* as uint(16) and *m* as uint(7). As opposed to the input variables above, the valid bits of input variables *n* are different in different applications.

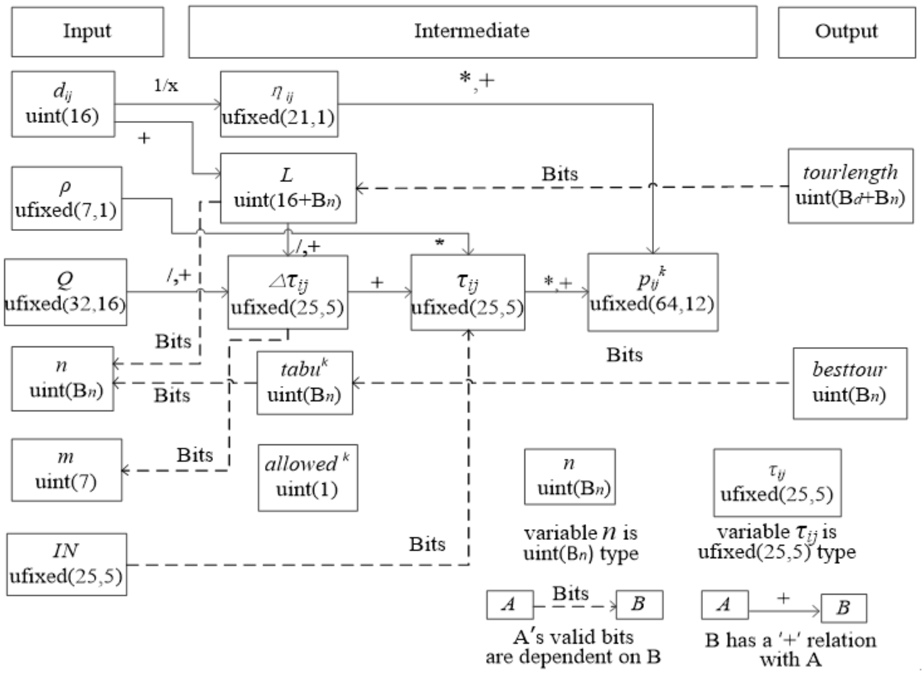


Figure.1 the relations and precision of notations in HACO-F

We do not specify the actual valid bits here (with an unknown B_n below), as we leave it to section 5 experiments.

With the defined valid bits of the input variables above, we present the other variables' valid bits defined based on the relations and the design principle described in the first paragraph of this section, Figure 1. For example, p_{ij}^k is the sum of the product of τ_{ij}'s square and η's square. Actually, the precision will be fixed-point ufixed(92,12) to store all the bits. We just redefine p_{ij}^k to fixed-point ufixed(64,12), integer 12bits, and decimal 52bits to reduce the memory and logic resources. *L's* precision depends on the input variables *n* and *d*, and is the sum of them. The precision of output variable *besttour* is redefined based on the intermediate variable *tuba,* which depends on *n*. The other output variable *tourlength's* precision is based on *L*.

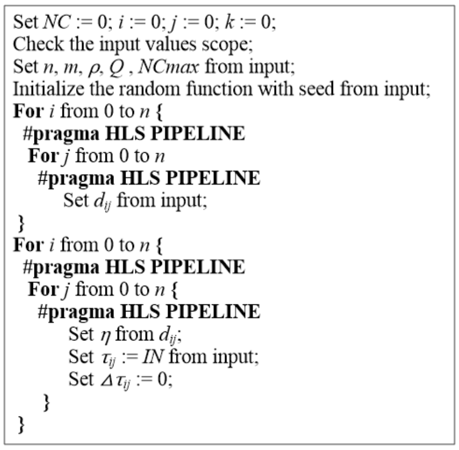
In particular, we redefine the overflow types of τ_{ij} and $\Delta\tau$ as 'AP_WRAP'. The 'AP_WRAP' type is to throw the overflow bits, just leaving the following valid bits. This is an appropriate idea to avoid the excessive heuristic information with no extra logic resource increases. The overflow types of other float type variables are redefined as 'AP_SAT'. The 'AP_SAT' type is to saturate the value to the nearest maximum absolute value when overflowing. The underflow types of all the variables in HACO-F are defined as 'AP_RND', which is to round the value to the nearest maximum absolute value. In this way, the minimum values of them will not be zero, avoiding total loss of the heuristic information.

Moreover, the heuristic parameters are set as α=2, β=2. The index variables' precision is not redefined for reducing variable conversions.

3.2. HACO-F loop optimization design

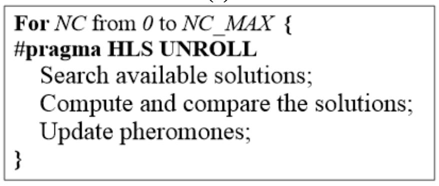
HACO-F contains two main parts, initialization and search the best solutions. There are some inner loops in these two parts respectively. We use UNROLL to parallel the search the best solutions loop and use PIPELINE to optimize all the other loops, shown as follows.

Step 1: Initialize parameters. This step is to initialize all the variables of HACO-F, such as the input parameters *n*, *m*, NC_{MAX} and the index variables *i*, *j*, *k*. Before assignments, there need to be some scope checks for the input values to avoid overflow of *n* and *d*. As variable η is the reciprocal of *d*, we separate the initialization of η and *d* to two separate loops to avoid data dependence for better pipelining. Then, there will be two loops: one is the initialization of *d*, and the other loop is the initializations of η, τ *and* $\Delta\tau$. Both of these loops are dual loops from 0 to n. We use PIPELINE to optimize the two dual loops, shown as Figure 2 (a).



```
Set $NC$ := 0; $i$ := 0; $j$ := 0; $k$ := 0;
Check the input values scope;
Set $n$, $m$, $\rho$, $Q$ , $NCmax$ from input;
Initialize the random function with seed from input;
For $i$ from 0 to $n$ {
  #pragma HLS PIPELINE
   For $j$ from 0 to $n$
    #pragma HLS PIPELINE
       Set $d_{ij}$ from input;
}
For $i$ from 0 to $n$ {
  #pragma HLS PIPELINE
   For $j$ from 0 to $n$ {
    #pragma HLS PIPELINE
       Set $\eta$ from $d_{ij}$;
       Set $\tau_{ij}$ := $IN$ from input;
       Set $\Delta\tau_{ij}$ := 0;
    }
}
```

(a)



```
For $NC$ from $0$ to $NC\_MAX$ {
#pragma HLS UNROLL
    Search available solutions;
    Compute and compare the solutions;
    Update pheromones;
}
```

(b)

Figure 2. (a) initialization of HACO-F (b) the best solution search

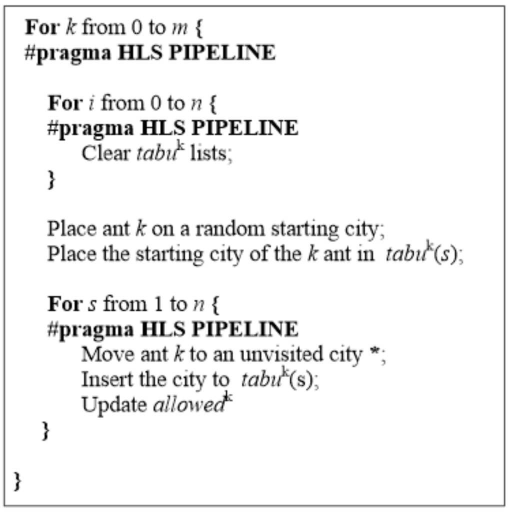
The input variable *seed* is used to initialize the random generated function. The random numbers used in HACO-F are pseudo-random numbers generated by mixed congruence method. We use the input variable *seed* as the initial value of the random function, and generate the pseudo-random numbers by the multiplication-addition operator. Firstly, multiply *seed* with a predefined value, and we get the result as *seed1*. Secondly, add another predefined value to *seed1*, and then we get the first random number *rand1*. After that, set the random value *rand1* as the new *seed* value, and go on as before. As *seed* can be set with different value from input, the random numbers generated will be different in different times. In this way, we can get random numbers continually. Generally, '%' operator is used to narrow the scope of random numbers in C/C++. Here, we exchange the operator '%' to operator '&' and '>>' operators as the '%' operator is time-consuming. Such as R_0=R_1%48, R_1 is the random number generated, R_0 is the result which is from 0 to 47. We may transfer to three separate instructions: first R_0=R_1&0x1F, second R_0+=(R_1>>5)&0xF, third R_0+=(R_1>>9)&0x1. It is to say that R_0 is composed of the dissected bits of R_1 from bit0 to bit9. The maximum of R_0 is equal to 0x1F+0xF+0x1=47. Similarly, the minimum of R_0 is 0. So the scope of R_0 is [0,48). As R_1 is a random number and R_0 is got from the dissected bits of R_1, the result of R_0 is also a random number.

Step 2: Search the best solutions. In this step, there are three parts inside. We unroll the outer loop to make the inner three parts execute in parallel, shown as Figure 2 (b), and describe the inner parts as three separate steps as follows.

Step 3: Search available solutions. In this step, the outer loop is to finish the solution searching of all ants. And its upper bound is the ant number *m*. The first inner loop is to clear the *tabu* list of ant *k*. The following inner loop is to finish ant *k*'s solution searching of all the cities. We pipeline these three loops, as there are temporal order relations among them, shown as Figure 3 (a).

In this step, Move ant k to an unvisited city (with '*' in Figure 3 (a)), we use the pseudo-random-proportional rule [3] to determine the next city that ants would move to (as it is like roulette, we called roulette follows), shown as Figure 3 (b). Firstly, compute the sum of the transition probabilities of unvisited cities, as p_{sum}. Secondly, generate a random percentage p_k. Thirdly, do roulette, namely, compute the sum of the transition probabilities of unvisited cities in order, as p_{pro}, until the transition percentage (p_{pro}/p_{sum}) is bigger or equal than p_k. Then the city roulette success is that the ant chooses.

For optimization, we make some tricks for the implementation of the roulette rule. We merge the code of compute the transition probability (with '#' in Figure 3 (b)) and the follow if conditional to one if conditional statement. We generate the random probability p_R as a random integer, which is more suitable for the process computation than percentage values. Then we multiply 127 in both sides of the inequality in the if conditional. So the inequality is transferred to p_{pro}*127 >= p_{sum}*p_R', where p_R' is got from p_R&0x7F. In this way, the computation of the transition probability with '/' operator is removed and it does not change the rule.



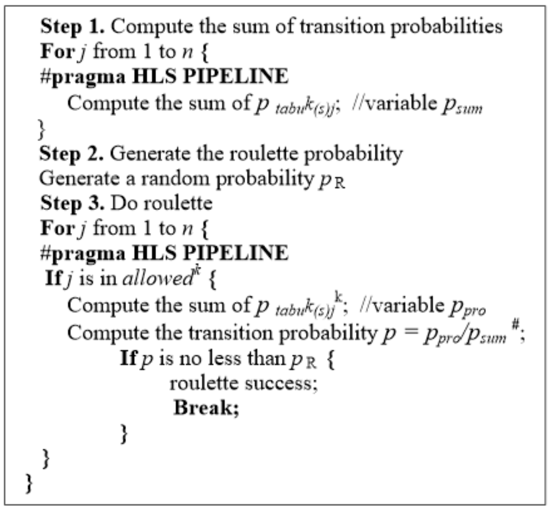
```
For k from 0 to m {
#pragma HLS PIPELINE

    For i from 0 to n {
    #pragma HLS PIPELINE
        Clear $tabu^k$ lists;
    }

    Place ant k on a random starting city;
    Place the starting city of the k ant in $tabu^k(s)$;

    For s from 1 to n {
    #pragma HLS PIPELINE
        Move ant k to an unvisited city *;
        Insert the city to $tabu^k(s)$;
        Update $allowed^k$
    }

}
```



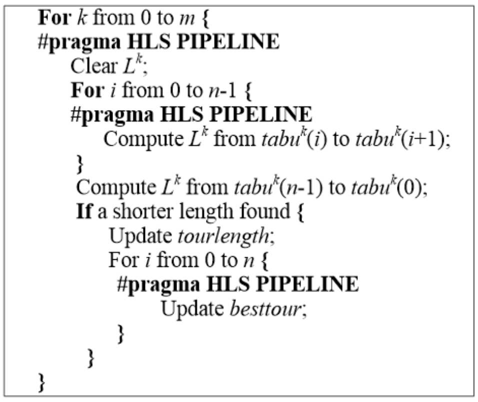
```
Step 1. Compute the sum of transition probabilities
For j from 1 to n {
#pragma HLS PIPELINE
    Compute the sum of $p_{tabu^k(s)j}$; //variable $p_{sum}$
}
Step 2. Generate the roulette probability
Generate a random probability $p_R$
Step 3. Do roulette
For j from 1 to n {
#pragma HLS PIPELINE
 If j is in $allowed^k$ {
    Compute the sum of $p_{tabu^k(s)j}{}^k$; //variable $p_{pro}$
    Compute the transition probability $p = p_{pro}/p_{sum}$ #;
        If p is no less than $p_R$ {
            roulette success;
            Break;
        }
 }
}
```

(a) (b)

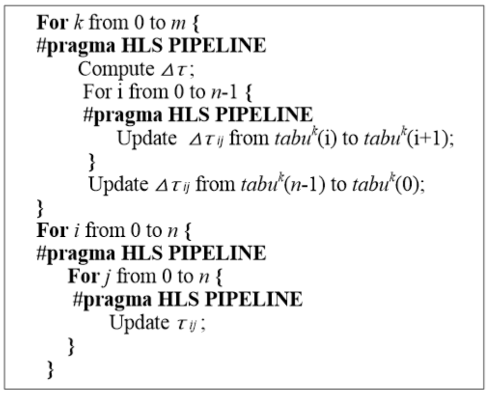
Figure 3. (a) available solutions search (b) transition rule

In addition, for better pipelining, we use three additional variables *a, b, c,* when computing the sum of transition probabilities. It splits the *pow* function to three individual multiplies to decrease the computation intervals. The precision of *a* and *b* is dependent on the assignment variables η_{ij} and τ_{ij}, and the variable *c*'s precision is defined as ufixed(32,6), integer 6 bits, decimal 26 bits, based on the principle described in data optimization.

Step 4: Compute and compare the solutions. In this part, there is one outer loop with two inner loops. The outer loop is to finish road length computations in current cycle of all ants. The loop's scope is from 0 to *m*. The first inner loop is to compute ant *k*'s road length. The follow inner loop is to update the shortest road and shortest length, when a shorter length is found. We pipeline these three loops, shown in Figure 4 (a).



```
For k from 0 to m {
#pragma HLS PIPELINE
    Clear $L^k$;
    For i from 0 to n-1 {
    #pragma HLS PIPELINE
        Compute $L^k$ from $tabu^k(i)$ to $tabu^k(i+1)$;
    }
    Compute $L^k$ from $tabu^k(n-1)$ to $tabu^k(0)$;
    If a shorter length found {
        Update tourlength;
        For i from 0 to n {
         #pragma HLS PIPELINE
            Update besttour;
        }
    }
}
```



```
For k from 0 to m {
#pragma HLS PIPELINE
    Compute $\Delta\tau$;
    For i from 0 to n-1 {
    #pragma HLS PIPELINE
        Update $\Delta\tau_{ij}$ from $tabu^k(i)$ to $tabu^k(i+1)$;
    }
    Update $\Delta\tau_{ij}$ from $tabu^k(n-1)$ to $tabu^k(0)$;
}
For i from 0 to n {
#pragma HLS PIPELINE
    For j from 0 to n {
    #pragma HLS PIPELINE
        Update $\tau_{ij}$;
    }
}
```

(a) (b)

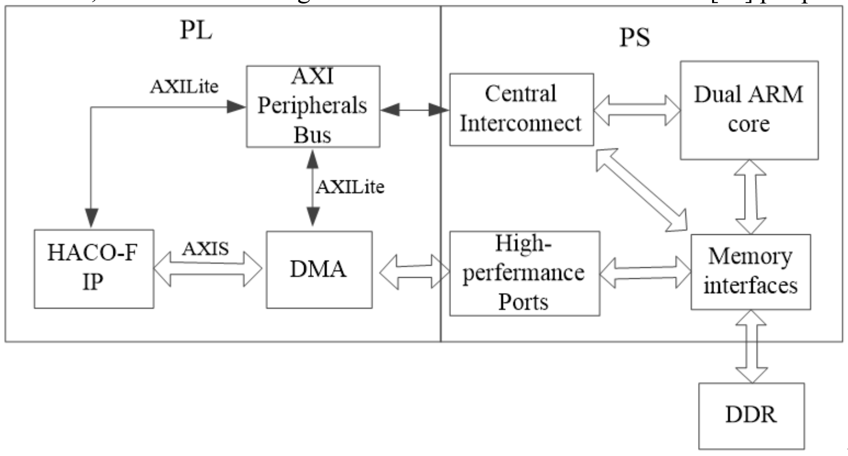
Figure 4. (a) the comparisons of solutions (b) road pheromone update

Step 5: Update road pheromones based on new solutions. In this step, there are two loops. The first loop is to update each road's $\Delta\tau$ based on the new tour map of ant *k*. The update rule of $\Delta\tau$ is based on paper [4] with no simplified. The following inner loop is to update the road pheromone τ based on $\Delta\tau$. As the precision of τ has been redefined, the values of τ can be confined automatically without the if-else conditional statement, which is a good method for code optimization. We pipeline these three loops, shown as Figure 4 (b).

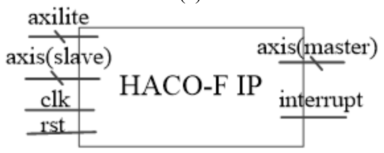
The complexity of HACO-F is the same with AS, and the complexity of pseudo-random-proportional is O(n).

3.3. Prototype

The prototype implementation of the HACO-F IP core is based on the Zynq [1] FPGA platform, shown as Figure 5 (a). The PS part of Zynq is dual-core ARM CPU, which is able to run a Linux OS for system control and the data transfer control. The PL part of Zynq is the FPGA, where we can integrate the HACO-F IP core. The axilite [11] peripherals bus is applied to



(a)



(b)

Figure 5. (a) the framework of HACO-F prototype system (b) IP core of HACO-F

integrate the HACO-F IP core and DMA as devices in the system, meanwhile to transfer variables' data. The DMA is used to connect the HACO-F IP core and the PS high performance bus ports through axis [11] protocol to directly access DDR memory. In this way, the HACO-F IP core is independent with the PS, which is able to leave the PS resource to other processes.

The IP core of HACO-F is shown in Figure 5 (b). Input variable *d* is a B_n*B_n array, we implement it as an axis slave port. The slave port is designed to connect to the axis master port of the DMA. Output variable *besttour* is also a B_n size array, we implement it as an axis master port, and connect to the axis slave port of the DMA. All the other parameters described in data optimization, we use the axilite bus to finish the data transfer, as the data size is small. The bus width of both axis and axilite are 32 bits. In addition, the clock and reset ports are needed for HACO-F IP core. After that, the interrupt port is used to indicate the end-of-run of HACO-F.

4. Experiments

The experiments are conducted on the Zynq FPGA platform. The clock of PS is 667MHz, and the clock of PL, e.g. FPGA part, is set 150MHz.

In order to show the performance of HACO-F, we typically choose a real-world traveling salesman problem (TSP) and att48 in TSPLIB [10] can be mapped as cities' locations. In att48, the valid bits of *n* are 6 bits because the value of *n* is 48. Subsequently the valid bits of *L* is 22(6+16) bits. For the purpose of pre-reserving bits for the error code, the final valid bits of *L* are set as 25 bits. Other parameters such as *seed*, *IN*, *ρ*, *Q*, *m*, and *NC* can be dynamically changed in testing.

4.1. Execution time

Table 2 shows the latency time of HACO-F and AS in synthesis. In Table 2, we can see that the maximum accelerate rate of HACO-F is 10.2, and the minimum accelerate rate is 3.9. The average accelerate rate is approximately 7.

Table 2 Latency of HACO-F and AS

latency	HACO-F	AS	rate
min	46446037	472007119	10.2
max	241986037	945317119	3.9

To obtain the performance of acceleration, we test the execution time of HACO-F and AS on real embedded system on Zynq. The PS of Zynq executes a C/C++ AS algorithm as the baseline of experimental results. According to the complexity of HACO-F, its execution time is mainly impacted by these variables *i.e. n, m* and *NC*. Hence, we dynamically change the value of *m*, and observe *NC*, then in turn. The results are shown in Table 3.

Table 3 Execution time of HACO-F and AS

parameters		HACO-F/s	AS/s	rate
m=50	NC=10	0.053408	0.312318	5.85
	NC=20	0.103381	0.622123	6.02
	NC=50	0.252929	1.563667	6.18
	NC=80	0.402664	2.502337	6.21
	NC=100	0.502404	3.098722	6.17
NC=80	m=10	0.084125	0.51183	6.08
	m=20	0.163298	1.006051	6.16
	m=40	0.322276	2.004416	6.22
	m=80	0.640687	4.000108	6.24
	m=100	0.798378	4.993827	6.25

In Table 3, we can observe that the execution time of HACO-F IP core is less than 6 times that of AS algorithm running in the ARM hard core. Moreover, when the values of *m* and *NC* are small, the time for DMA transferring data may take up more percentage of the execution time. Therefore, the accelerate rate is less than the following tests. These results also show that loop optimization and data optimization in HACO-F are very effective and efficient.

4.2. Accuracy

As the variables τ and $\Delta\tau$ are redefined, the scopes of them are much smaller than AS. According to the relations shown in the data optimization section, the input variables *IN*, ρ and *Q* have a great impact on accuracy of τ and $\Delta\tau$. If the values of *IN*, ρ and *Q* are too big, the variables τ and $\Delta\tau$ will be quickly saturated. So search results perform well when the values of *IN* and ρ are small. Since both HACO-F and AS have randomness, the results are different in different conditions. We randomly select 10 groups with *m*=50 and *NC*=100. Table 4 shows the search results of HACO-F and AS.

The results show that the accuracy of HACO-F and AS are almost in the same level. Moreover, as we define τ and $\Delta\tau$ '*Wrap-around*' type, which can make the ants more extra chances to search different roads near the convergence of HACO-F after τ and $\Delta\tau$ are saturation. This is the reason for the improvement of search ability of HACO-F.

Table 4 Accuracy of HACO-F and AS

parameters				road length		
ρ	IN	Q	seed	HACO-F	AS	distance
0.25	0.25	5	3	11353	11463	-110
0.375	1	10	5	11395	11643	-248
0.5	0.25	20	85	11074	11314	-240
0.875	0.375	50	66	11187	11209	-22
0.25	0.25	200	888	11867	11844	23
0.5	0.5	10	56	11154	11430	-276
0.25	0.25	20	81	11465	11388	77
0.75	0.875	50	90	11458	11251	207
0.875	0.25	1000	55	11648	11570	78
0.25	0.375	2000	29	11790	11417	373

4.3. Resource utilization and energy consumption

Table 5 shows the resource utilization before and after HLS accelerating.

Table 5 Resource of HACO-F IP core

Resource	AS	HACO-F
BRAM	51(18%)	27(9%)
DSP	9(4%)	35(15%)
FF	8511(7%)	6570(6%)
LUT	12111(22%)	9904 (18%)

We can observe the resource utilization is just DSP 15%, 6% FF, 18% LUT, 9% BRAM. The resource utilization of HACO-F are 1% FF, 4% LUT, 9% BRAM decrease than AS, and 11% DSP increase. The increase of DSP is because that we redefine the variables' precision in HAOC-F, and the redefined variables' multiplications are implemented by DSP resource. Moreover, in HACO-F we split the multiplication-addition operator of p in substep, which adds more multiplication operators. The additional multiplications executing in pipeline finally result in the increase of DSP resource. The division operator in assignment of η may consume other DSP resource.

Obviously, the conversion of '%' operator to '&' operators consumes less LUTs and FFs' resource. As the variables' precision is cut down, the resource of BRAM dynamically decreases. Note that the variable *tabu* reduces the utilization of BRAM from 16 to 3, as the precision is redefined from 32(*int*) to 6.

Figure 6 shows the energy consumption before and after HLS accelerating in the integrated system. Due to the decrease of logic resource, the on-chip power of system drops 0.111W. The power of DSP resource grows while the power of FF, BRAM, LUT, Clock and Signals decrease.

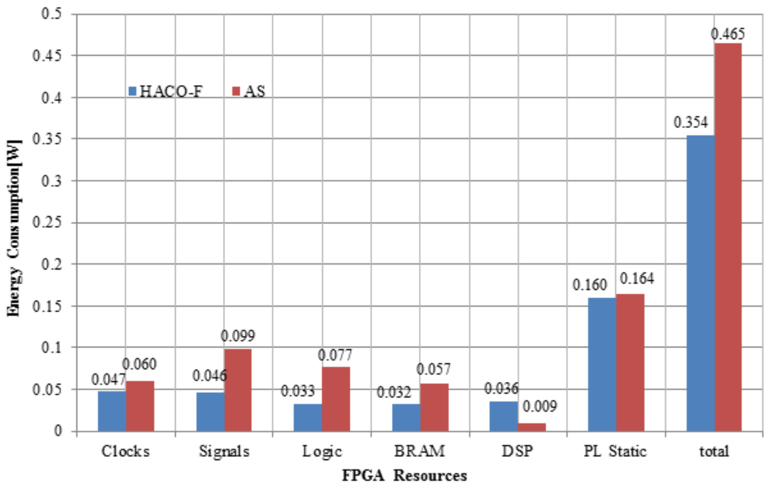


Figure 6. Energy consumption of HACO-F IP core

5. Conclusions

In this paper, we have proposed HACO-F, which is the accelerating algorithm based on HLS designed by C/C++ programing language. The data optimization by redefining the variables in HACO-F decreases resource utilization by 1% FF, 4% LUT, 9% BRAM. The redefined overflow type of floating-point variables τ and $\Delta\tau$ are able to improve search ability near the convergence of HACO-F with no extra resource consumption. Then, we have explored the loop optimization strategies including pipeline and unroll in HLS to parallelize the loops in HACO-F. The experimental results show that the HACO-F algorithm can accelerate more than 6 times speed than that of the AS. Moreover, the system on-chip energy consumption decreases by 23.9%.

In the future work, we will consider to extend our prototype system to accelerate most of the intelligent algorithms (*e.g.* Convolutional Neural Network, CNN). In addition, we will also consider to investigate the energy saving strategy (e.g. dormancy strategy) for further reducing the energy consumption.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grant No.61502018) and the Science and technology plan project of Beijing science and Technology Commission (D171100004017003).

References

1. S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "A 16-nm Multiprocessing System-on-Chip Field-Programmable Gate Array Platform," *IEEE Micro*, vol. 36, no. 2, pp. 48-62, April 2016

2. J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for Fpgas: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473-491, March 2011
3. M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53-66, April 1997
4. M. Dorigo, V. Maniezzo, and A. Colorni, "Ant System: Optimization by a Colony of Cooperating Agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29-41, February 1996
5. M. Guntsch, M. Middendorf, B. Scheuermann, O. Diessel, H. ElGindy, H. Schmeck, and K. So, "Population based ant colony optimization on FPGA," *IEEE International Conference on Field-Programmable Technology*, pp.125-132, Hong Kong, China, December 2002
6. R. M. Hamou, H. A. Bouarara, and A. Amine, "Bio-inspired techniques in the clustering of texts: Synthesis and comparative study," *International Journal of Applied Metaheuristic Computing*, vol. 6, no. 4, pp. 39-68, October 2015
7. 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," *IEEE Congress on Evolutionary Computation (CEC)*, pp.4516-4521, Vancouver, Canada, July 2016
8. H. C. Huang, "A Taguchi-Based Heterogeneous Parallel Metaheuristic ACO-PSO and Its FPGA Realization to Optimal Polar-Space Locomotion Control of Four-Wheeled Redundant Mobile Robots," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 915-922, June 2015
9. C. F. Juang, C. M. Lu, C. Lo, and C. Y. Wang, "Ant Colony Optimization Algorithm for Fuzzy Controller Design and Its FPGA Implementation," *IEEE Transactions on Industrial Electronics*, vol. 55, no. 3, pp. 1453-1462, March 2008
10. M. Mavrovouniotis, F. M. Muller, and S. Yang, "Ant Colony Optimization with Local Search for Dynamic Traveling Salesman Problems," *IEEE Transactions on Cybernetics*, vol. 47, no. 7, pp. 1743-1756, April 2016
11. M. Ramirez, M. Daneshtalab, J. Plosila, and P. Liljeberg, "NoC-AXI interface for FPGA-based MPSoC platforms," *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 479-480, Oslo, Norway, August 2012
12. A. Sakthivel, P. Vijayakumar, A. Senthilkumar, L. Lakshminarasimman, and S. Paramasivam, "Experimental investigations on Ant Colony Optimized PI control algorithm for Shunt Active Power Filter to improve Power Quality," *Control Engineering Practice*, vol. 42, pp. 153-169, June 2015
13. B. Scheuermann and M. Middendorf, "Counter-Based ant colony optimization as a hardware-oriented meta-heuristic," *European Conference on Applications of Evolutionary Computing*, pp. 235-244, Berlin, Germany, March 2005
14. B. Scheuermann, K. So, M. Guntsch, M. Middendorf, O. Diessel, H. Elgindy, and H. Schmeck, "FPGA implementation of population-based ant colony optimization," *Applied Soft Computing*, vol. 4, no. 3, pp. 303-322, March 2004
15. N. Venugopal, V. Shobana, and R. Manimegalai, "Analysis of optimization techniques in FPGA placement," *International Conference on Computer Communication and Informatics(*ICCCI), pp. 1-5, Coimbatore, India, January 2014
16. Z. H. Xiong, J. H. Chen, and S. K. Li, "Hardware/software partitioning for platform-based design method," *Asia and South Pacific Design Automation Conference(ASP-DAC)*, vol. 2, pp. 691-696, Shanghai, China, January 2005