*electronics*

MDPI

*Article*

# A Highly Configurable High-Level Synthesis Functional Pattern Library

**Lan Huang [1,2,‡], Teng Gao [1,‡], Dalin Li [1,†], Zihao Wang [1] and Kangping Wang [1,2,*]**

[1] College of Computer Science and Technology, Jilin University, Changchun 130012, China; huanglan@jlu.edu.cn (L.H.); gaoteng18@mails.jlu.edu.cn (T.G.); dlli16@mails.jlu.edu.cn (D.L.); zhwang19@mails.jlu.edu.cn (Z.W.)

[2] Key Laboratory of Symbolic Computation and Knowledge Engineering, Jilin University, Changchun 130012, China

* Correspondence: wangkp@jlu.edu.cn

† Current address: 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.

‡ These authors contributed equally to this work.

**Abstract:** FPGA has recently played an increasingly important role in heterogeneous computing, but Register Transfer Level design flows are not only inefficient in design, but also require designers to be familiar with the circuit architecture. High-level synthesis (HLS) allows developers to design FPGA circuits more efficiently with a more familiar programming language, a higher level of abstraction, and automatic adaptation of timing constraints. When using HLS tools, such as Xilinx Vivado HLS, specific design patterns and techniques are required in order to create high-performance circuits. Moreover, designing efficient concurrency and data flow structures requires a deep understanding of the hardware, imposing more learning costs on programmers. In this paper, we propose a set of functional patterns libraries based on the MapReduce model, implemented by C++ templates, which can quickly implement high-performance parallel pipelined computing models on FPGA with specified simple parameters. The usage of this pattern library allows flexible adaptation of parallel and flow structures in algorithms, which greatly improves the coding efficiency. The contributions of this paper are as follows. (1) Four standard functional operators suitable for hardware parallel computing are defined. (2) Functional concurrent programming patterns are described based on C++ templates and Xilinx HLS. (3) The efficiency of this programming paradigm is verified with two algorithms with different complexity.

**Keywords:** field programmable gate arrays; parallel programming; C++ template; functional programming; pipeline processing



**Citation:** Huang, L.; Gao,T.; Li, D.; Wang, Z.; Wang, K. A Highly Configurable High-Level Synthesis Functional Pattern Library. *Electronics* **2021**, *10*, 532. https://doi.org/10.3390/electronics10050532

Academic Editor: Luis Gomes

Received: 10 January 2021
Accepted: 20 February 2021
Published: 25 February 2021



## 1. Introduction

FPGA is widely used and has rapidly developed in high-performance computing, due to its parallel execution, high computational performance, low power consumption, and short development cycle compared to Application Specific Integrated Circuit (ASIC). However, with the increasing Scale of system-on-a-chip, the user designing process becomes more and more complex, and the drawbacks of traditional Register Transfer Level (RTL) approaches become prominent. Moreover, FPGA mainly uses hardware description language (HDL) development, which is difficult to use, has few practitioners, a long development cycle, and is not conducive to the rapid update of products.

Many designers still manually rewrite their sequential algorithms in HDL. In order to increase productivity and promote FPGA to a wider user community, new design methodologies in high-level design abstraction present in recent years, including FPGA HLS [1].

HLS has advantages in the following aspects: (1) A higher level expression abstraction than HDL. (2) C++ language is more familiar to software developers. (3) Designers can incrementally optimize the code by gradually replacing the loop structure with the functional pattern without compromising the portability of the code. HLS also has obvious drawbacks: designers remain exposed to various aspects of hardware design, development cycles are still time-consuming, and the quality of results of HLS tools is far behind that of RTL flows. For instance, unrolling is a standard HLS optimization. However, loop code segments cannot be fully expanded and executed concurrently due to access conflicts and data dependency. It is not easy to implement a full pipeline with minimal initiation interval (II).

Other HL-related Domain-Specific Languages (DSL) include Chisel [2] and SpinalHDL [3]; they are a specific language designed for a certain domain. Because of the domain restrictions, the problem to be solved is delimited, so the language does not need to be complex to have precise expressiveness. Furthermore, these languages are usually easy to learn and use; however, because they are domain languages, the generality of this kind of language is far less than C++, and this kind of language is only used to generate Verilog code; its optimization of timing constraints is far less than vivado HLS.

In this paper, we propose high-performance computing pattern using template-based hardware generation strategy through extending C++-based Xilinx Vivado HLS tools. We design computational patterns based on the MapReduce model that can be rapidly adapted to high-performance parallel flow algorithms on FPGA. The patterns are implemented in C++ templates and expressed by functional programming. HLS language-specific structure optimization (specific pragma) is added to the template, and the details of the internal structure are parametrized so that the user can adjust the parameters to achieve different parallelization and pipeline levels, and finally achieve an efficient pipeline structure (II = 1) regardless of the computational operators.

Templating computations have the following benefits: (1) Parametric C++ templates provide enough flexibility to exploit the structural properties. (2) Independent of the analysis capabilities of HLS tools, template-based constructs allow us to manually extend the implementation to the resource and data bandwidth constraints of the target device, thus improving coding efficiency. (3) Predefined templates integrated with specific computational instances can be quickly adapted to the application.

We evaluate our work in two algorithms: the vector distances algorithm and the Quantum-behaved Particle Swarm Optimization (QPSO) algorithm. For the vector distances algorithm, multiplying and adding are frequent operations in deep learning computations. We design experiments with different concurrency and flow rates for the relevant computations, analyze the differences in resource utilization, and explore how different parameter types affect the flow rate hierarchy. The goal of the QPSO is to find the optimal solution for all particles in a multidimensional hypervolume. All particles in the space are first assigned an initial random position and an initial random velocity. The position of each particle is then advanced according to its velocity, the known optimal global position in the problem space, and the known optimal position of the particle in turn. As the computation progresses, the particles cluster or aggregate around one or more optimal positions by exploring and exploiting the known favorable positions in the search space. The mystery design of the algorithm is that it preserves information about both the optimal global position and the particle's known optimal position [4,5]. Li et al., propose a framework to accelerate QPSO algorithm on FPGA [6]. This framework reschedules the dataflow of QPSO 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 FPGA, this framework on FPGA achieves better throughput. In this work, we have further improved the abstraction of the QPSO algorithm framework on FPGA by analyzing the algorithm structure and simplifying the coding process using functions from our library, define more parametric interfaces. The results show that QPSO on Xilinx Kintex Ultrascale xcku040 achieves up to 123 times acceleration

ratio compared to the Intel Core i7-6700 CPU. Taking this as an example, our proposed library greatly reduces the coding difficulty, improves the efficiency of FPGA programming, and simplifies the process of algorithm reproduction while ensuring the high performance of algorithm implementation.

The results show that our templates contain predefined, schema-specific control logic that allows parametric parallel computing templates to greatly enhance programming flexibility, computational efficiency, code reusability, and for complex computational tasks, adaptability to more resource-efficient computational models with constant time complexity.

The contributions of this paper are as follows:

(1) Several standard functional operators suitable for hardware parallel computing are defined.
(2) A functional concurrent programming paradigm is abstracted based on C++ templates and Xilinx HLS.
(3) The efficiency of this programming paradigm is verified with two algorithms of different complexity.

## 2. Relate Work

HLS has existed for many years, and it is very active in academic circles due to its short development and verification time. Many HLS-related DSLs have been proposed: Lift [7], Chisel, Bluespec [8], SpinalHDL, Lava [9], and C$\lambda$ash [10] to implement functional programming, which improves the abstraction of hardware code, and has the following advantages: Automatic bit width inference deduction (even across module boundaries), error checking capability, Parameterization capability, a large number of basic components and reusable Intellectual Property core (IP). Although these methods take advantage of the characteristics of modern programming languages, they are essentially HDL, and the design of these languages is the only real correspondence with real circuits. The designed applications are mapped directly to the hardware without too much compiler optimization. However, in high-level synthesis, timing optimizations are crucial for achieving high-performance circuits, these tools need to add timing constraints manually when they are integrated. For some algorithms that need to explore the best performance structure, the lack of timing optimization further increases the workload, and it requires the programmer to understand the hardware design concept, which undoubtedly improves the threshold for software engineers to use FPGA. While xilinx HLS accepts design in high-level language (e.g., C, C++, and SystemC) and generates synthesizable cycle-accurate RTL through code transformations and synthesis optimizations. In standard, statically scheduled HLS, such optimizations are typically performed in conjunction with modulo scheduling [11–13]: the aim is to create pipelines with the best possible loop initiation intervals under the given clock and resource constraints. Vivado HLS [14] estimates the timing and area resources based on built-in libraries for each FPGA. When using logic synthesis to compile the RTL into a gate-level implementation, perform physical placement of the gates in the FPGA, and perform routing of the inter-connections between gates, logic synthesis might make additional optimizations that change the Vivado HLS estimates.

The latest generation of HLS technology, focusing on the C/C++ language, gaining design intent, realizing a hardware-software common model, achieving joint design and joint verification, and achieving success [15–17]. HLS has many advantages, such as completing FPGA design in a higher level of abstraction, requiring less hardware knowledge, exploring design space faster, making minor modifications to the program, richer, and more convenient verification and debugging methods. In addition, FPGA have a wide range of applications in the implementation and acceleration of various algorithms. The variability of the algorithm, easy iteration, easy debugging, and easy maintenance, it is very difficult to use RTL development, and the progress of the project is also affected. How to use high-level languages, such as C, C++, etc., for FPGA development has become a hot trend in EDA software [18].

Dataflow circuits are fundamentally different: their schedules are not predetermined at compile time but devised as the circuit runs. Moreover, Lana [19,20] investigates how to create timing-efficient, high-throughput pipelines, and their MILP model is based on the theory of marked graphs and allows for resource-optimal buffer placement and sizing, with the purpose of maximizing throughput at the desired clock frequency. However, they are purely theoretical optimizations of the computational model without abstracting a generalized computational template for the computational model, which still requires a complete understanding of the circuit structure and does not improve the user's coding efficiency.

Current HLS tools [21] do not always produce the most performance-optimized implementations: no matter how well optimized the HLS hardware is, it will not perform as well as a HDL design implementation. As reported in previous work [22], HLS generates efficient hardware when the input code is written in a specific coding style (adding pragma), which we call refactored code. Therefore, creating optimized hardware with HLS still requires a deep knowledge of the underlying hardware architecture and how to effectively utilize HLS tools. There are benefits of using HLS instead of HDL so that the entire application is in a high-level language: simulation speed is generally faster, debugging is less difficult.

A lot of work is implemented on HLS, such as the Genetic algorithm (GA) [23], which is one of most popular evolutionary search algorithms that simulates natural selection of genetic evolution for searching solution to arbitrary engineering problems. However, it is computationally intensive and will become a limiting factor for evolving solution to most of the real-life problems as it involves large number of parameters that needs to be determined. As for Neural Networks, Zhang et al. proposed Caffeine [24], a hardware/software co-designed library to accelerate convolutional neural networks on FPGA. And they propose to accelerate convolutional layers and fully connected layers with a uniformed representation. The authors of [25] proposed DeepBurning, an automation tool to generate FPGA-based accelerators for NN models. DeepBurning compiles DNNs described in a Caffe-like script and generates the corresponding RTL-level accelerator under user-specified constraints. Cong et al. [26] proposed an automated framework for mapping deep neural networks onto FPGA with RTL-HLS hybrid templates, which takes symbolic descriptions (in TensorFlow) of DNNs as input, and outputs implementations of the corresponding FPGA-based accelerators for model inference. They implement accelerators with RTL-HLS hybrid templates, and convert model inference into general-purpose computations like matrix multiplication. Several optimization kernels are developed and invoked to ensure the functionality, performance, and energy efficiency of the accelerator.

Haggai et al. [27] use HLS proposed methodology and design patterns that enable code reuse. Evaluate proposal by implementing two networking applications: a key-value store cache and a UDP-based firewall for FPGA-based SmartNICs, showing that their methodology can simplify the implementation of high-performance networking applications using HLS. However, their work (ntl) only achieves a pipelining level of II = 3 with no performance improvement over their experimental control group except for a reduction in code size, and our implementation of the library function operator easily achieves II = 1 performance while maintaining coding efficiency.

## 3. Functional Operators in Our Model

Functional languages are much more natural fit for high-level hardware generation as they have limited to no side effects and more naturally express a dataflow representation of applications which can be mapped directly to hardware pipelines. The core idea of functional programming is pure function: functions use (without modifying) only the results of calculations with the actual parameters passed to them. If a pure function is called several times with the same real parameters, it will give the same result without leaving any traces (no side effects). This all means that pure functions cannot modify the state of the program. It also means that pure functions cannot read from the standard

input and cannot write to the standard output. In a function language, as all functions are referentially transparent, even complex functions can be parallelized for each function without side effects. As a result, function expressions can be represented directly as data flow graphs and the data flow can be mapped directly to hardware functions. The difficulty with functional programming is the granularity of the functional units. Functional units mainly contain arithmetic functions and storage functions. First, the arithmetic and data storage part of the algorithm to be implemented has to be abstracted and decomposed into independent sub-functional units with the same arithmetic function. This process involves algorithm transformation and possibly even redesign of the algorithm. Second, the granularity of the functional units has to be carefully weighed. Too large a granularity can lead to state machine complexity, circuit redundancy, and reduced chip utilization. Too small a granularity may increase the burden on the IO interface. Therefore, the functional unit granularity is based on the principle of minimizing unnecessary generality [28]. Finally, due to the different resource ratios required for computing and memory functions, the utilization of each resource on the FPGA chip should also be taken into account when designing functional units to ensure maximum utilization of at least one resource in order to achieve improved performance at the algorithm scale. In this paper, C++ templates are used to implement several operators that are typical of functional ideas, taking into account the specific requirements of concurrent computing in hardware. C++ templates are instantiated during compilation as classes and functions related to the template parameters, and therefore they essentially implement code generation functions.

### *3.1. TreeOP*

For dealing with the problem of concurrent computational partitioning of large arrays, we propose a TreeOP template that iteratively expands the code using a binary tree based on the input array.

As shown in Algorithm 1, at each iteration, code lines 4 and 5 divide the array into left and right subtrees, and the subtrees are half the length of the original subtrees, and code line 6 put the set of subtrees and the length of the subtrees into TreeOp for the next iteration, until subtree length = 1 at code line 9. After division, the code is fully expanded and each leaf node is entered as a single operator for subsequent calculations. The fully expanded state essentially generates the corresponding code, and the leaf node after partitioning can continue to be the input of subsequent operators.

**Algorithm 1** Implementation of TreeOP.

```
1:  template<typename Result, typename Item,Result( *pairOp),int num, int idx = 0 >
2:  class TreeOp {
3:      static inline Result tree(const Item numbers[num]) {
4:          Result t1 = TreeOp<Result, Item, pairOp, num/2, idx>::tree(numbers);
5:          Result t2 = TreeOp<Result, Item, pairOp, num − num/2, idx +
    num/2>::tree(numbers);
6:          return pairOp(t1,t2);
7:      }
8:  };
9:  class TreeOp<Result, Item, pairOp, binOp, 1, idx> {
10:     static inline Result tree(const Item numbers[]) {
11:         return binOp( numbers, idx);
12:     }
13: }
```

### *3.2. MapOP*

A map function is a specified operation on each element of a conceptually organized list of independent elements (e.g., a list of test scores), all duplicates being independent, and when the number of iterations is known in advance, all calculations depend only on

the index value. Map transforms collections by functions. Specifically, it applies a function (hereafter referred to as an operator function) to all elements of the collection in parallel. Each operator function accesses a separate data element. Each parallel transformation of this operator function is called an instance of the operator function. The operator function uses Map to execute all Map instances in any order without any side effects. With this independence, the different elements of a Map can be synchronized with each other, thus achieving maximum parallelism.

The Map function we implement on HLS is in the form of map< DataType, Function>(), DataType is a custom data type that can be defined as Int, Double, Float, etc., while Function is a highly concurrent function that can be customized by the user. The code is shown in Algorithm 2.

**Algorithm 2** Implementation of MapOP.

```
template<typename Item,typename Result,Result (*binOp)(const Item& ),typename
UpStream>
class MapOp {
    MapOp<Result,MapResult,mapOp,MapOp<Item,Result,binOp,UpStream> >
        map() {
            return MapOp<Result, MapResult,mapOp
                                ,MapOp<Item,Result,binOp,UpStream> > (*this);
    }
    Result get() {
        return binOp( up.get() );
    }
    Result get(NumType idx) {
        return binOp( up.get(idx) );
    }
};
```

### 3.3. ZipwithOP

Zipwith has two input sets, and the element function outputs a new result from each of the two input pairs. Zipwith operates on two data structures and creates a new structure using a binary function. As the lambda functions in Map and Zipwith have no side effects, individual function calls for different input elements are independent of each other and can be executed in parallel. Calculations in computational models such as Map, Zip, and Reduce can operate on multi-element data structures without side effects, thus taking full advantage of available parallelism.

The Zipwith function we implement on HLS is in the form of zipWith (UpStream) where UpStream is another set of inputs. The code is shown in Algorithm 3.

**Algorithm 3** Implementation of ZipwithOP.

```
template<typename UpStream1, typename UpStream2>
    class ZipOp {
template<typename UpStream2>
    ZipOp<ATStream<Item,data>, UpStream2>
    zipWith(UpStream2& up2)
        return ZipOp<ATStream<Item,data>, UpStream2>(*this, up2);
    }
    ZipItem get(){
        return std::make_ pair(up1.get(), up2.get());
    }
    ZipItem get(NumType idx){
    return std::make_ pair(up1.get(idx), up2.get(idx));
    }
}
```

### *3.4. ReduceOP*

Reduce refers to the specific merging of elements of a list to form a smaller set of values. Usually, only a 0 or 1 output value is generated per Reduce. The intermediate values are provided to the user's reduce function via an iterator. Although not as parallel as the Map function, the Reduce function is useful in highly parallel environments because Reduce always has a simple answer, is relatively independent of large-scale operations, has no data dependencies, and supports commutative law.

The Reduce function we implement on HLS is in the form of reduce<DataType, Function, TotalData,CONC,PipeStep>(), where DataType and Function are the same as above, TotalData is the total amount of data to be computed, and CONC is the concurrency of your own design, PipeStep is a structure for solving some computations that take too long and require additional pipelining levels to break the concurrency impact of data dependencies, as described in detail later in StreamReduce. The code is shown in Algorithm 4.

**Algorithm 4** Implementation of ReduceOP.

```
template<typename Item, typename Result, Result (*pairOp)(const Item&,const Item&),int total, int parallel, int pipestep, typename UpStream>
class ReduceOp {
private:
    UpStream& up;
public:
    typedef Result ItemType;
    ReduceOp(UpStream& up):up(up){ }
    Item get(){
        const int round = total/parallel;
        Item roundReduce[pipestep];
# pragma HLS RESOURCE variable=roundReduce core=RAM_ S2P_ LUTRAM
ReduceFor:
    for (int r = 0; r < round; ++r){
    # pragma HLS PIPELINE
        Item pvalue[parallel];
        for (int i = 0; i < parallel; ++i){
            pvalue[i] = up.get(r * parallel + i);
        }
        Result reduceTmp = TreeOp<Result,Item, pairOp, parallel>::tree(pvalue);
        if (r < pipestep )
            roundReduce[r % pipestep] = reduceTmp;
        else
            roundReduce[r % pipestep] = pairOp(reduceTmp,roundReduce[r % pipestep]);
    }
}
```

## 4. GroupPipeReduce Model

### *4.1. GroupReduce*

The traditional sequential execution of Reduce does not optimize the code and the HLS tool will calculate the expansion in order of each clock cycle. For example, eight inputs will perform seven calculations with a time complexity of N, while TreeReduce is the fully concurrent version of Reduce, TreeReduce accepts a set of elements, merges the even-numbered bits of the set with the next odd-numbered bit in the set, and repeats the calculation. The time complexity of the operator is $\log_2 N$. We use C++ templates to implement TreeReduce. Instead of using the usual UNROLLFOR method, we use TreeOP to divide and expand the data for a set of inputs: we assume that the data coming in from the upstream is at level i. The Array[2n] and Array[2n + 1] bits at this layer

perform the user-required reduce operation to output a result as the input at layer i + 1. The reduce calculation is completed when the number of output results is 1 at a certain layer. Layer-to-layer data computation is pure pipelined, with one result guaranteed per clock cycle.

### *4.2. PipeReduce*

For high-precision floating-point calculations, the increasing complexity of the results may cause the current computation unit to take too long to compute, so we designed an alternative computation model, PipeReduce, for those computations that cannot be completed in a single clock cycle. This approach solves the problem of long operator computation time. The computation mode implements stepwise pipelining, dividing different levels of pipelining according to different clock cycles of the computation results, flexibly solving the problem that the critical path in the algorithm cannot be concurrent, breaking the data dependency, and optimizing the computation efficiency. For example, in traditional HDL development, we need to set different state machines when dividing different pipeline levels, and we need to process a lot of code to change the state machine. Our templates are optimized for this, and you can change the flow hierarchy flexibly by simply changing the PipeStep parameter. In addition, due to the advanced nature of HLS, the problem of handling different concurrent and different pipeline levels can be implemented automatically without manual adjustment.

As shown in the Figure 1, the left half of the computational model is GroupReduce, and the output results start as inputs to PipeReduce. We can set the degree of concurrency (CONC) to M (a set of M BRAM inputs). Assuming the number of clock cycles required to complete the calculation is N, we set the number of N PipeStep, the next step is a 2-step operation:

1. Input data with the current data in PipeStep to continue the reduce operation, and the result is stored in the current PipeStep.
2. The data input for the next cycle continues the reduce operation with the data in the next state PipeStep, and the results are stored in the next PipeStep.

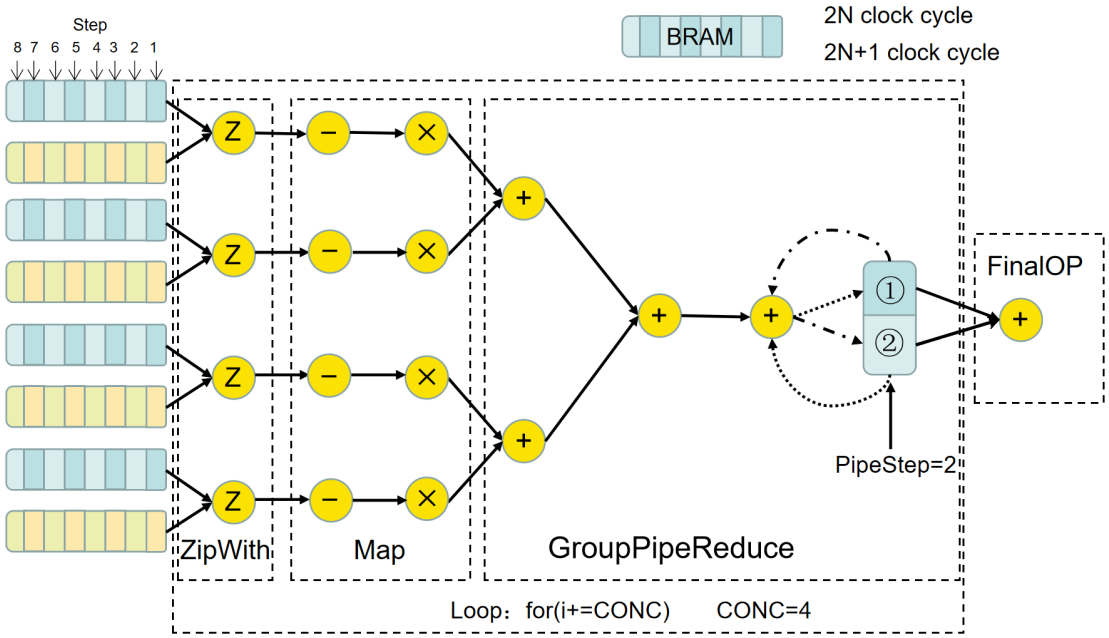


**Figure 1.** Demonstrates the execution logic of Map, Reduce, and Zipwith.

## 5. Experimental Design

### *5.1. Application of Custom Functions*

We designed several simple functions to prove the experiment, such as squared and accumulated sums. We need to define the parameters of the algorithm (input/output interfaces) and the function body to pass the function as parameters into our defined template.

Shown in Figure 1 is a data flow diagram for Algorithm 5. Assume that the total data volume is 32 and the concurrency is 4, the floating-point calculation requires N clock cycles, and the PipeStep state is set to N (In this example we assume that N is 2). Then, it is calculated as follows. Two data streams as input of the algorithm, then the algorithm performs a Map computation on its own square, and the output continues to be computed in GroupReduce mode. Next, the first and second output results are directly stored in ① and ② PipeStep, respectively. The input of the third clock cycle is reduced at position ①, and the result returns to position ①; the input of the next clock cycle is reduced at position ②, and the result returns to position ②. Then, the process is repeated until only bits ① and ② are left at the end of the input, and then the two are recreated one more Reduce (FinalOP) to output the final result. For the part of the overall calculation process where there is a data dependency, we have implemented II = 1 on HLS, which represents the highest level of pipelining. For concurrently executable processes, we have defined parametric interfaces that allow the user to assign their own level of concurrency. For sequential access input and output we use three types of interfaces to handle different kinds of data: ATStream for static arrays, APStream for dynamic arrays, and HLSAdpStream for streamed data. The output has three interfaces. Reduce ultimately generates a scalar result. For vectors, there is the toArray that places the result into an array and the user needs to define the array's expansion hierarchy, as shown in the pragma, and there is the toHLSStream that places the result into hls::stream<>, which behaves like a FIFO of infinite depth and therefore does not need to define the scale of hls::stream<>.

**Algorithm 5** Sum of vector distances squared

```
ATStream<DataType,v1> s1;
ATStream<DataType,v2> s2;
data_ out = s1
    .zipWith(s2)
    .map<DataType,diff>()
    .map<DataType,square>()
    .reduce<ResultType,sum,32,4,2>()
    .get();
```

#### 5.1.1. Timing Analysis

We name a set of experiments by the type of data computed and the parameters in Reduce such as Int ; 32, 8, 2 at 200 Mhz, the four operators in Algriothm 5 carry out the squared sum of the distances between two sets of vectors, which we label ReduceFor:

$$f(x, y) = (x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)^2 \quad (1)$$

The equation above performs a total of three operations: adding, subtracting, and squaring. We name each of them as sum, diff, and square, respectively, return results of x + x, x − x, and x × x.

As shown in Figure 2, the first clock cycle is concurrent load (read). In the second clock cycle, the two groups of inputs are each divided into eight concurrent groups according to the order of two ZipWith division, the same clock cycle on its own fusion of the data Map subtraction function, meanwhile Map Subtract function for self-fused data at the same clock cycle. The results of the eight outputs of the third clock cycle continue to execute the Map function that multiplies itself. Results are output in the fourth clock cycle but available in the fifth clock cycle. Therefore, the next step is to start GroupPipeReduce for a single

output. FinalOP operation for Reduce at the seventh clock cycle for pipestep output, which completes the calculation. FinalOP (Reduce) is performed on the output of pipestep in the seventh clock cycle, and the calculation is complete. In combination with the experimental results, the time required for a DSP48E to perform the same computation is fixed, the impact of higher frequency is to shorten the interval between different computation operations and thus improve performance, while higher frequency brings the problem of more complex state machines and data paths that consume more on-chip resources.

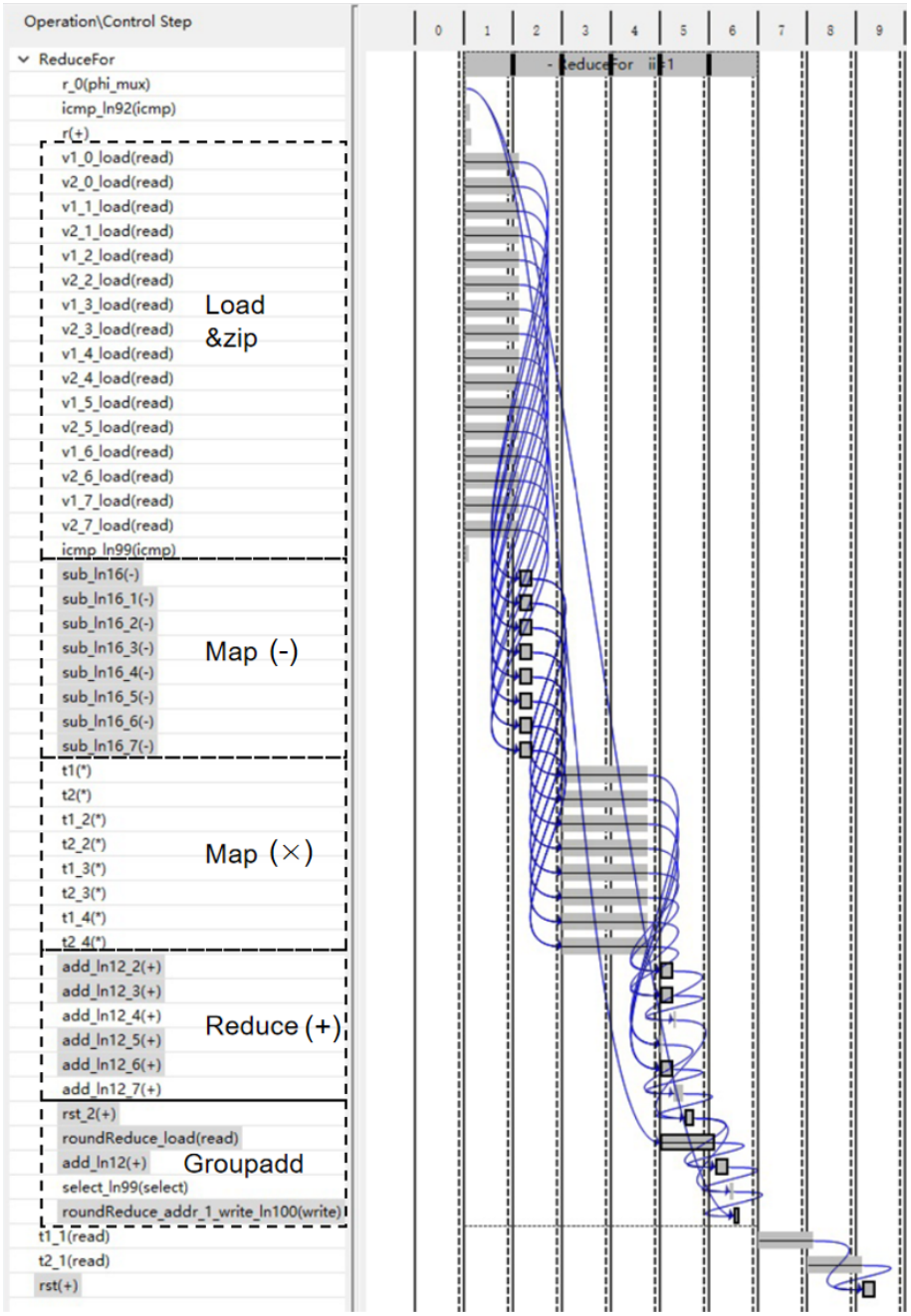


**Figure 2.** Demonstrates timing diagram with 16 groups of input Zipwith into 8 groups, followed by Map, Map, Reduce, and Grouppipeadd.

### 5.1.2. Resource Analysis

As mentioned earlier, you can check the performance metrics in the synthesis report generated by the HLS tool when determining whether the design meets your needs.

After analyzing the report, you can use specific optimization commands to optimize your implementation.

The synthesis report contains information on the following performance metrics: Area, Latency, Initiation interval (II), Loop iteration latency, Loop initiation interval, Loop latency represent resource utilization, computation delay, pipeline level, cycle delay, and other performance indicators; we conducted a resource performance analysis for each metric.

Two DaType Int (4 bytes) and Float (8 bytes) are used as input to the calculation, which is based on the code5, and set number of data inputs to 32 and record the DSP48E, FIFO, and LUT resources in resource utilization by frequency, concurrency, and pipeline levels. The results of the analysis of the impact on the overall cycle latency with the same concurrency and at different pipeline levels are as follows.

Figure 3a shows that changes in frequency, concurrency, and pipeline length have no effect on the DSP48E in the II = 1 full pipeline state. However, Figure 3b,c shows that LUT and FIFO resources are positively correlated as the frequency and concurrency increase and the pipeline becomes longer. Because the levels of tree merge used in Reduce are different at different concurrency levels and pipeline level, the loop control logic required varies significantly.

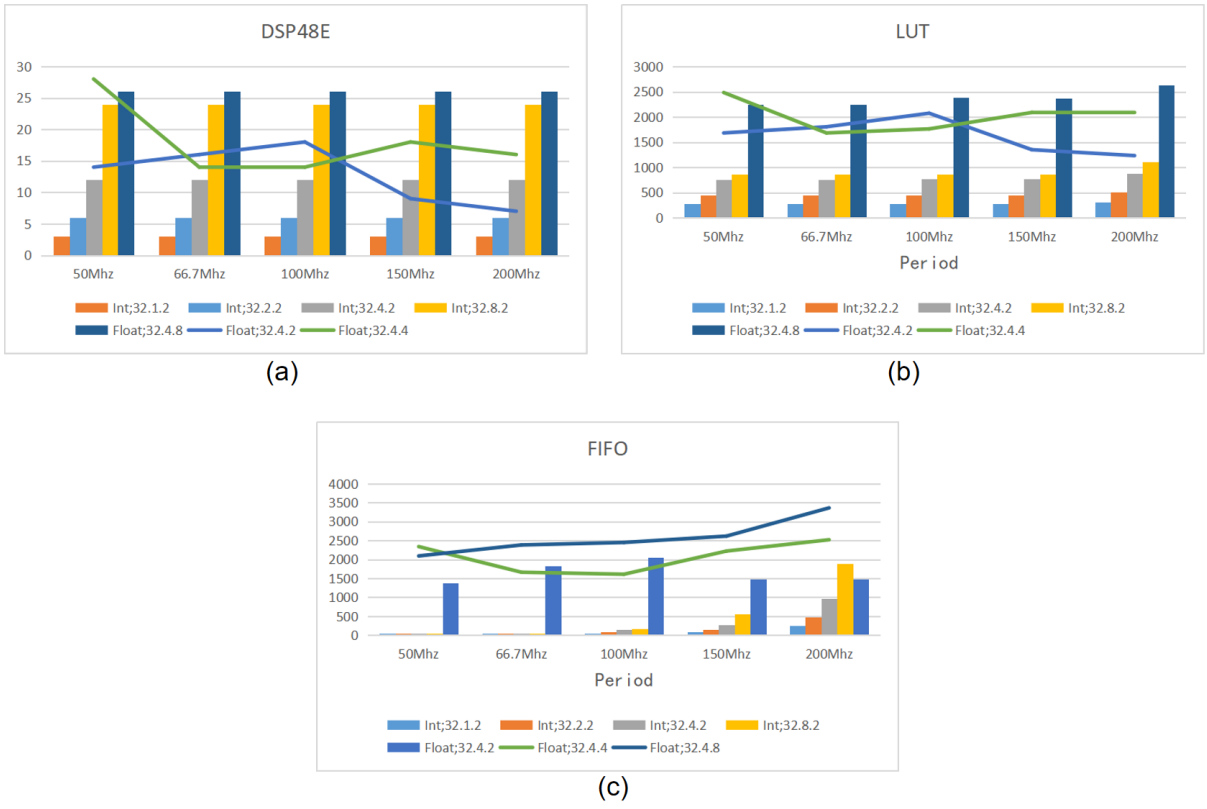


**Figure 3.** Shows the effect of parameter type, frequency on: (**a**) DSP48E; (**b**) LUT; (**C**) FIFO.

In particular, due to the impact of the data type on the calculation, Float types 32.4.2 and 32.4.4 will have a longer delay due to the longer calculation time which in turn affects the Initiation Interval, resulting in unpredictable resource consumption, which means that the HDL code logic implemented is quite different as shown in Figure 4a,b.

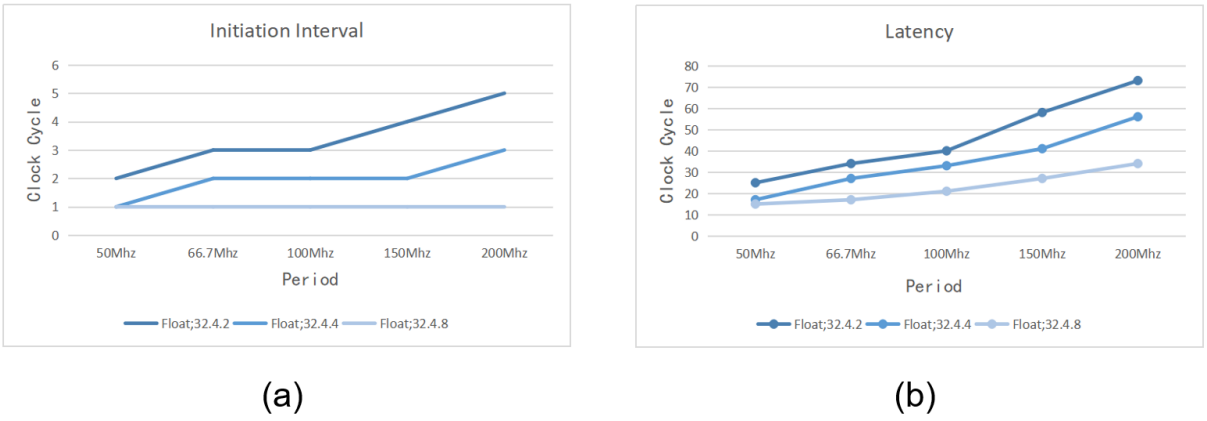


**Figure 4.** Demonstrates the effect of frequency, concurrency on: (**a**) Initiation Interval; (**b**) Latency.

The above experiments lead to the conclusion that small changes in the high-level abstraction, such as frequency, concurrency, operators, etc., can make a significant difference in the low-level implementation, and the state machine changes resulting from these changes are tedious and inefficient to describe in HDL code. Other HLS languages are not optimized for specific implementations, but merely generate a functional HDL code implementation that does not cope well with these implementations and changes.

We compare our work with Josipovic [29]: they propose Parallel Pattern Templates using a template generation strategy to implement hardware structures. Their work uses RTL-C hybrid templates to generate Map, Reduce, and ZipWith functions, and they design functions such as Nearest neighbor (which determines the distance of a point p to its nearest neighbor among the points in set S in a two-dimensional space) and other functions. Compared to their work, our functions are generated using only C++ templates, which undoubtedly reduces the design effort significantly, and we additionally implement a design that enables pipeline II = 1 in the Reduce function compared to it. In the experimental part, we use a single line of code in Algorithm 5 to complete an algorithm with similar functionality, which proves that our library has significant advantages in terms of ease of use.

### *5.2. High-Performance Implementation of QPSO on FPGA*

FPGA offer the most flexible hardware architecture, including gate-level parallelism, variable memory bandwidth, and a hierarchy-free structure of all programmable memories. By combining high concurrency with a variable deep pipeline, you can better exploit the advantages of FPGA and achieve higher throughput at full flow. If only parallel architecture is used, there will not be enough acceleration ratio.

On FPGA, the most efficient part of an algorithm is a full flow structure with a 1-clock cycle initialization interval(II). To construct a full flow structure of an algorithm, the data flow of the algorithm needs to be reconstructed as a continuous, uninterrupted operation from data input to result output.The information exchange is in the middle of the data flow before the data flow reconstruction of the group intelligence algorithm. When using group computation, one must wait for all particle data to arrive before solving for the global optimum, which will disrupt the flow of the pipeline. An alternative implementation is to implement the algorithm in two pipelines before and after the information exchange location, but this will also reduce the efficiency of the algorithm. Therefore, the use of the multi-stage pipelining "map-reduce" framework design pattern is a good way to improve the computational performance of the algorithm on the FPGA.

#### 5.2.1. Data Structures on FPGA

As shown in Figure 5, we define five key parameters for the algorithm: PARTICAL-GROUP, NUM_INGROUP, DIMENSION, ITERATION, and FitFunction, which represent

the number of groups of particles, the number of particles in the group, the number of dimensions of the particles, the number of iterations, and the function for evaluating the degree of adaptation, respectively. Where PARTICALGROUP times NUM_ INGROUP is the total number of particles, which we divide for high concurrency and streaming. The computationally intensive part of the algorithm is calculating the adaptation of each particle in each iteration and updating the pbest(Best fit in Group), gbest(Best fit in Global), and particle positions based on the adaptation. Our detailed flow for the implementation of this algorithm on FPGA is as follows: Dividing the total number of particles into PARTICGROUP groups, with NUM_INGROUP particles in each group, allows for full expansion within the group and II = 1 flow between the groups on HLS. We expand the examples in the group according to TreeOP and then: (1) update the particle position and (2) calculate the fitness according to the given evaluation FitFunction.

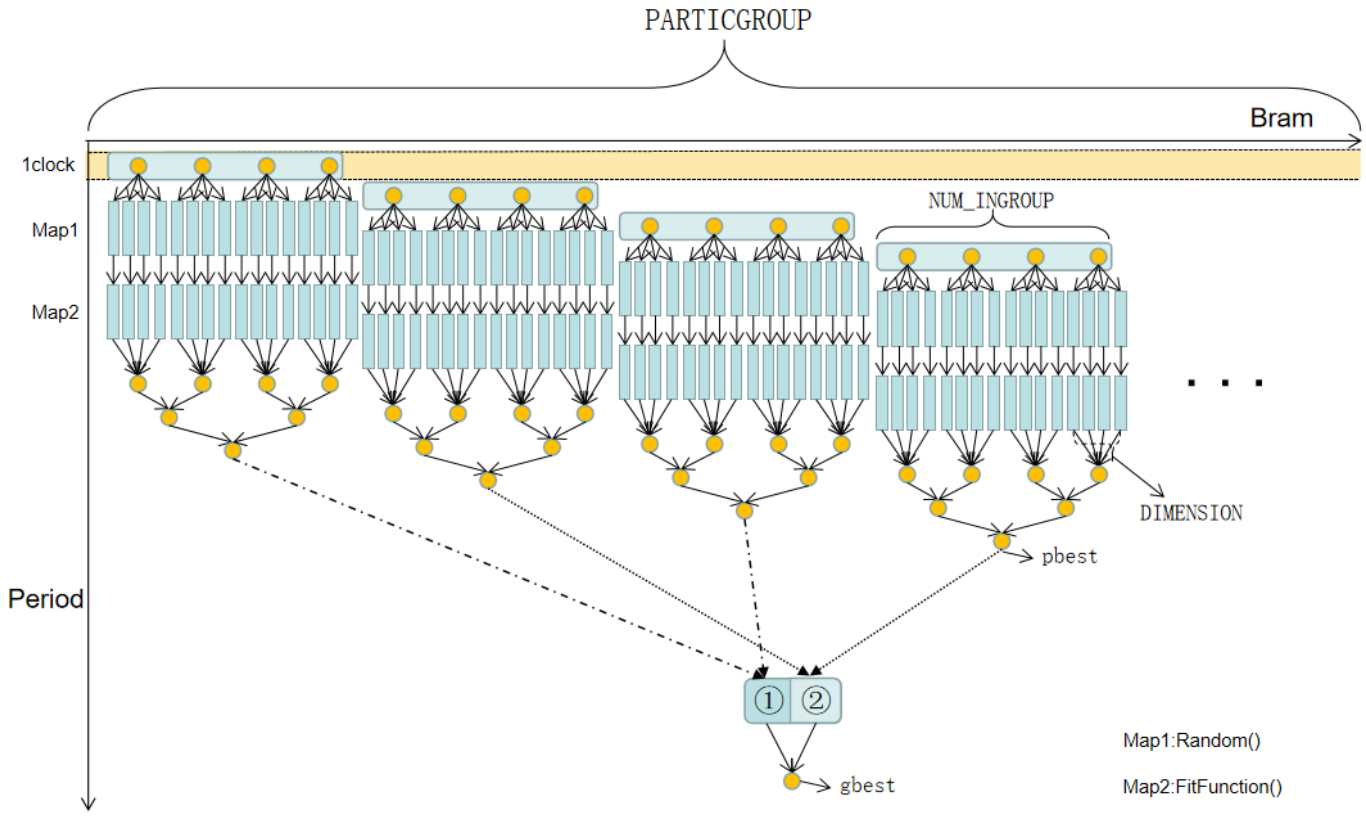


**Figure 5.** Demonstrates parallel expansion within a group in a QPSO flow and calculation of the flow per clock cycle between groups.

Both operations can use the Map function. Then, use the GroupReduce function to compare the best-fit particles in the group according to their fitness, take out the best-fit particles in the group, and return the index value. Groups are streamed in groups per clock cycle. For each group of results per clock cycle, we use PipeReduce for convergence. Set the appropriate PipeStep as needed for the current calculation. The PipeStep will finally return the best fit and index value for all particles in this iteration, thus completing an iteration. Then, go to the next iteration and repeat the above process until the iteration ends. The pseudocode for the implementation of QPSO on FPGA using our library functions can then be simplified as shown in Algorithm 6.

**Algorithm 6** Pseudocode of QPSO.

```
for t = 1 to Iteration IT do
    for p = 1 to ParticalGroup PG do
        for n = 1 to NUM_INGROUP NI do
            for d = 1 to Dimension Number DN do
                execute Dimension.map(random)
            end for
            execute Group.map(FitFunction)
            execute Group.reduce(GroupBestFit)
        end for
    end for
    execute Global.zipwith(GlobalBestFit)
end for
```

#### 5.2.2. The Speedup of FPGA to Multi-Core CPUs

In this section, to prove the validity of the code, we implemented high-performance versions of QPSO in FPGA and CPU, respectively, using eight benchmark functions for the performance evaluation and comparison as shown in Table 1. We take Xilinx Kintex Ultrascale xcku040 (Xilinx, San Jose, CA, USA) as our FPGA platform and Intel Core i7-6700 CPU (Intel, Santa Clara, CA, USA) as our CPU platform. We use the *Perf*[3] tool for analyzing the performance of QPSO 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. Compared with multi-core CPUs, FPGA reach a maximum of 123 times speedup on benchmark function Sphere as shown in Figure 6.

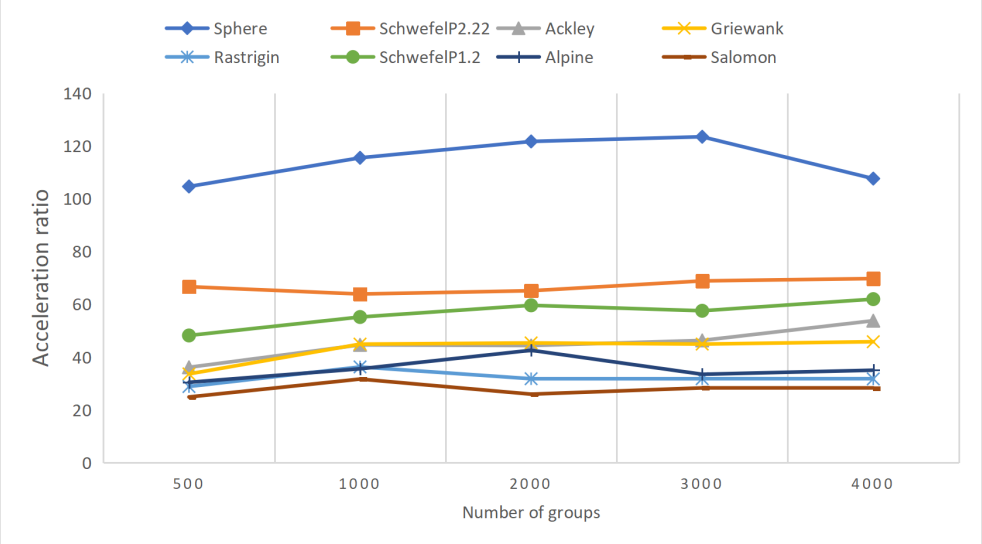


**Figure 6.** The speedup of FPGA to multi-core CPUs.

**Table 1.** 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 | $f_5(x) = \sum_{i=1}^{D} (x_i^2 - 10\cos(2\pi x_i) + 10)$ | $[-5.12, 5.12]$ | $\{0\}^D$ | 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 |

## 6. Discussion

In this paper, we propose high-performance computing pattern using a template-based hardware generation strategy through extending C++-based Xilinx Vivado HLS tools, which can quickly implement optimized algorithms on FPGA with specified simple parameters. The four operators are TreeOP, MapOP, ZipWithOP, and ReduceOP. TreeOP solves the problem of partitioning large arrays by concurrency, if we implement this function directly with HLS, we need to define several similar functions by ourselves, which is a relatively repetitive and troublesome workload. MapOP can take customized functions as parameters for concurrent execution of algorithms with no data dependencies. Zipwith has two input sets, and the element function outputs a new result from each of the two input pairs. Reduce refers to the specific merging of elements of a list to form a smaller set of values.

Except for the full concurrent execution of the Reduce function in a binary tree structure, we have made an additional change: to handle cases where some calculation time exceeds the current cycle, we have proposed a long pipeline structure. The advantage of increasing the pipeline level instead of using full concurrency is that it increases the frequency of computation: a new set of data is processed at the beginning of each clock cycle, and the results of the previous computations are stored in the buffer for backup. This structure greatly improves resource utilization and reuse rate.

Using the above computational model, we designed two experiments: (1) Sum of vector distances squared and (2) Algorithm QPSO. After that we show the code implementation, resource utilization of the former, and performance comparison with the latter. The results show that our computational model guarantees a high performance while improving the coding efficiency.

**Author Contributions:** Conceptualization, K.W. and D.L.; methodology, T.G.; software, K.W. and T.G.; validation, T.G., Z.W. and K.W.; formal analysis, L.H.; investigation, T.G.; resources, L.H.; data curation, T.G.; writing—original draft preparation, T.G.; writing—review and editing, T.G., D.L. and K.W.; project administration, L.H.; funding acquisition, L.H. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported in part by the Jilin Province Science and Technology Development Plan Project under Grant 20190201273JC, 20200201174JC, and in part by the Jilin Provincial Key Laboratory of Big Data Intelligent Computing.

**Data Availability Statement:** Data are contained within the article.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Coussy, P.; Adam, M. *High-Level Synthesis: From Algorithm to Digital Circuits*; Springer: Berlin/Heidelberg, Germany, 2008.
2. Bachrach, J. Chisel: Constructing hardware in a Scala embedded language. In Proceedings of the DAC Design Automation Conference, San Francisco, CA, USA, 3–7 June 2012.

3. SpinalHDL Documentation. Available online: https://github.com/SpinalHDL/SpinalHDL (accessed on 9 May 2016).
4. Sun, J.; Feng, B.; Xu, W. Particle swarm optimization with particles having quantum behavior. *Proc. Congr. Evol. Comput.* **2004**, *1*, 325–331.
5. Sun, J.; Xu, W.; Feng, B. A global search strategy of quantum-behaved particle swarm optimization. *Proc. IEEE Conf. Cybern. Intell. Syst.* **2004**, *1*, 111–116.
6. Li, D.; Huang, L.; Wang, K.; Pang, W.; Zhou, Y.; Zhang, R. A General Framework for Accelerating Swarm Intelligence Algorithms on FPGAs, GPUs and Multi-Core CPUs. *IEEE Access* **2018**, *6*, 72327–72344. [CrossRef]
7. Kristien, M.; Bodin, B.; Steuwer, M.; Dubach, C. High-Level Synthesis of Functional Patternswith Lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*; ACM: New York, NY, USA, 2019; pp. 35–45. [CrossRef]
8. Nikhil, R.S. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In *High-Level Synthesis*; Springer: Berlin/Heidelberg, Germany, 2008.
9. Bjesse, P.; Claessen, K.; Sheeran, M.; Singh, S. Lava: Hardware design in Haskell. In *ICFP*; ACM: New York, NY, USA, 1998.
10. Baaij, C.; Kooijman, M.; Kuper, J.; Boeijink, A.; Gerards, M. C$\lambda$aSH: Structural descriptions of synchronous hardware using Haskell. In Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Lille, France, 1–3 September 2010; pp. 714–721.
11. Rau, B.R. Iterative modulo scheduling. *Int. J. Parallel Program.* **1996**, *24*, 3–64. [CrossRef]
12. Canis, A.; Brown, S.D.; Anderson, J.H. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications, Munich, Germany, 2–4 September 2014; pp. 1–8.
13. Zhang, Z.; Liu, B. SDC-based modulo scheduling for pipeline synthesis. In Proceedings of the 32nd International Conference on Computer-Aided Design, San Jose, CA, USA, 18–21 November 2013; pp. 211–218.
14. Vivado Design Suite User Guide High-Level Synthesis UG902 (v2019.2). Available online: https://www.xilinx.com/support/documentation/sw_\manuals/xilinx2019_\2/ug902-vivado-high-level-synthesis.pdf (accessed on 22 May 2019).
15. Lei, J.; Li, Y.; Zhao, D.; Xie, J.; Chang, C.I.; Wu, L.; Li, X.; Zhang, J.; Li, W. A Deep Pipelined Implementation of Hyperspectral Target Detection Algorithm on FPGA Using HLS. *Remote Sens.* **2018**, *10*, 516. [CrossRef]
16. Coussy, P.; Gajski, D.D.; Meredith, M.; Takach, A. An introduction to high-level synthesis. *IEEE Des. Test Comput.* **2009**, *26*, 8–17. [CrossRef]
17. Martin, G.; Smith, G. High-level synthesis: Past, present, and future. *IEEE Des. Test Comput.* **2009**, *26*, 18–25. [CrossRef]
18. Andrade, J.; George, N.; Karras, K.; Novo, D.; Silva, V.; Ienne, P.; Falcao, G. Fast design space exploration using vivado HLS: Non-binary LDPC decoders. In Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Vancouver, BC, Canada, 2–6 May 2015.
19. Josipović, L.; Sheikhha, S.; Guerrieri, A.; Ienne, P.; Cortadella, J. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*; ACM: New York, NY, USA, 2020.
20. Josipović, L.; Ghosal, R.; Ienne, P. Dynamically scheduled high-level synthesis. In Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 127–136.
21. Vivado Design Suite User Guide High-Level Synthesis UG902 (v2018.3). Available online: https://www.xilinx.com/support/documentation/sw_\manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf (accessed on 11 December 2019).
22. Huang, L.; Li, D.L.; Wang, K.P.; Gao, T.; Tavares, A. A Survey on Performance Optimization of High-Level Synthesis Tools. *J. Comput. Sci. Technol.* **2020**, *35*, 697–720. [CrossRef]
23. Alqudah, E.; Jarrah, A. Parallel implementation of genetic algorithm on FPGA using Vivado high level synthesis. *Int. J. Bio-Inspired Comput.* **2020**, *15*, 90. [CrossRef]
24. Zhang, C.; Sun, G.; Fang, Z.; Zhou, P.; Pan, P.; Cong, J. Caffeine: Towards Uniformed Rep-resentation and Acceleration for Deep Convolutional Neural Networks. *Int. Conf. Comput. Aided Des.* **2018**, *38*, 2072–2085.
25. Wang, Y.; Xu, J.; Han, Y.; Li, H.; Li, X. DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family. In Proceedings of the IEEE/ACM Proceedings of Design Automation Conference, Austin, TX, USA, 5–9 June 2016.
26. Guan, Y.; Liang, H.; Xu, N.; Wang, W.; Shi, S.; Chen, X.; Sun, G.; Zhang, W.; Cong, J. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; pp. 152–159. [CrossRef]
27. Eran, H.; Zeno, L.; István, Z.; Silberstein, M. Design Patterns for Code Reuse in HLS Packet Processing Pipelines. In Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 28 April–1 May 2019.
28. Kos, A.; Rankovic, V.; Tomažic, S. Sorting Networks on Maxeler Dataflow Supercomputing Systems. *Adv. Comput.* **2015**, *96*, 139–186.
29. Josipovic, L.; George, N.; Ienne, P. Enriching C-based High-Level Synthesis with parallel pattern templates. In Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT), Xi'an, China, 7–9 December 2016.