algorithms

MDPI

Article

A Novel Reduction Circuit Based on Binary Tree Path Partition on FPGAs

Linhuai Tang [1,2], Zhihong Huang [1,2], Gang Cai [1,2,*], Yong Zheng [1,2] and Jiamin Chen [1,2,*]

[1] Aerospace Information Research Institute, Chinese Academy of Sciences, Beijing 100094, China; tanglinhuai16@mails.ucas.ac.cn (L.T.); huangzhihong@mail.ie.ac.cn (Z.H.); zhengyong17@mails.ucas.ac.cn (Y.Z.)

[2] School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100049, China

* Correspondence: caigang@aircas.ac.cn (G.C.); chenjm@aircas.ac.cn (J.C.)

Abstract: Due to high parallelism, field-programmable gate arrays are widely used as accelerators in engineering and scientific fields, which involve a large number of operations of vector and matrix. High-performance accumulation circuits are the key to large-scale matrix operations. By selecting the adder as the reduction operator, the reduction circuit can implement the accumulation function. However, the pipelined adder will bring challenges to the design of the reduction circuit. To solve this problem, we propose a novel reduction circuit based on binary tree path partition, which can simultaneously handle multiple data sets with arbitrary lengths. It divides the input data into multiple groups and sends them to different iterations for calculation. The elements belonging to the same data set in each group are added to obtain a partial result, and the partial results of the same data set are added to achieve the final result. Compared with other reduction methods, it has the least area-time product.

Keywords: pipeline; vector reduction; accumulator; FPGAs

Citation: Tang, L.; Huang, Z.; Cai, G.; Zheng, Y.; Chen, J. A Novel Reduction Circuit Based on Binary Tree Path Partition on FPGAs. *Algorithms* **2021**, *14*, 30. https://doi.org/10.3390/a14020030

Received: 14 December 2020
Accepted: 18 January 2021
Published: 20 January 2021

1. Introduction

In field-programmable gate array (FPGA) applications, reduction circuits can be applied to reduce vectors to scalars. In recent years, advances in integrated circuit technology have brought significant improvements to FPGA performance. Coupled with the inherently high hardware parallelism, FPGAs are used as hardware accelerators in more and more fields, such as signal processing [1,2], scientific computing [3–5], machine learning [6–8], and data centers [9–11]. In these applications, some algorithms include a large number of operations of vector and matrix, which belongs to the category of reduction problem. Therefore, a high-performance reduction circuit is key to matrix operations.

The combination of a high-performance reduction circuit and a small-sized binary tree can effectively solve the trade-off between area and performance. The computing task in Figure 1a contains m data sets $Set_0 - Set_{m-1}$, and the data set Set_i contains n_i elements. It can be executed by a variety of hardware solutions. Figure 1b shows the use of a reduction circuit as a solution. The data are serially input, one data per cycle. Its advantage lies in its small area, usually only a reduction operator is needed. Figure 1c is the diagram of the binary tree. Data are input in parallel, and all elements of a data set are input at a time. Compared with the reduction circuit, the performance of the binary tree is higher. However, it requires more operators, which consumes a lot of area. A compromise is to combine the reduction circuit with a small-sized binary tree, as shown in Figure 1d. It can not only ensure that the area is maintained in a reasonable range, but also has higher performance. Therefore, a high-performance, small-area reduction circuit is required.

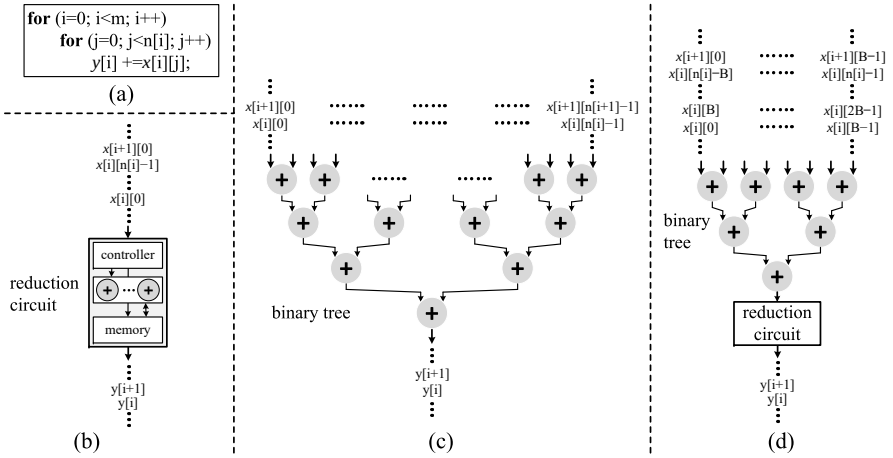


Figure 1. Reduction of multiple data sets: different solutions. (**a**) computing tasks; (**b**) reduction circuit; (**c**) binary tree; (**d**) a compromise solution: reduction circuit + binary tree.

The design of the reduction circuit faces many challenges. The first is the number of reduction operators. Reduction operators are the primary area source of reduction circuit, so it is necessary to reduce the number of reduction operators as much as possible. The second is data hazards. When the reduction operator of the multi-stage pipeline is used, some partial results cannot be consumed immediately. It needs to be stored in the buffer, waiting for the arrival of another operand. In order to reduce the size of the buffer, the partial results of different data sets are stored in the same buffer. For a unit in the buffer, only data can be written after it is used; otherwise, write after read hazards will occur. Therefore, the data in the buffer need to be properly managed. Third, the circuit needs to be capable of processing any number of data sets, and each data set contains any number of elements. If the reduction circuit can only handle a certain amount of data, it will greatly limit its application. Finally, there are necessities for the performance and complexity of the reduction method.

In this paper, a novel reduction circuit based on binary tree path partition (BTPP) is proposed. It can handle multiple data sets, and each data set can have any number of elements. The proposed method is implemented on FPGAs, which has the smallest area-time product compared with other designs.

The remainder of this paper is organized as follows. Section 2 describes the background of the reduction circuit, and Section 3 gives some previous works. Then, Section 4 details the principle and working mechanism of BTPP. After that, the hardware design of BTPP is discussed in Section 5, and the evaluation and comparison results are given in Section 6. Finally, Section 7 summarizes the paper.

2. Background

The function of the reduction circuit is to reduce vector to a scalar by using associative and commutative reduction operators [12]. For the data set $X = \{x_0, x_1, \cdots, x_{n-1}\}$, reduction circuit performs the following operations:

$$y = x_0 \circledast x_1 \circledast \cdots \circledast x_{n-1} \tag{1}$$

where $\circledast$ is the reduction operator. The reduction operator is a binary operator, such as adder, multiplier, max/min function, etc. If the reduction operator is an adder, then $y = \sum_{i=0}^{n-1} x_i$. If the reduction operator is a multiplier, the output of the reduction circuit is $\prod_{i=0}^{n-1} x_i$. The reduction circuit is only related to the number of pipeline stages and bit width of the reduction operator, and does not care about the hardware structure of the reduction operator. This means that the reduction circuit can use the existing IP core as the reduction operator, which greatly reduces the design difficulty and shortens the design cycle. In addition, it also helps the reduction circuit change its function. For example, the

reduction operator of a reduction circuit is an adder. By changing the adder to a multiplier, as long as their pipeline stages and bit widths are equal, the reduction circuit can work normally without changing the control circuit. In this paper, the reduction operator takes the adder as an example to analyze the reduction circuit.

The simplest reduction circuit can be realized by feeding back the output of the reduction operator to the input, as shown in Figure 2a. It only requires a reduction operator and can handle any amount of data. However, the increase in the frequency of the reduction operator does not bring about an increase in throughput. The reduction operator used in Figure 2b is combinational logic, and calculating $y[0] = \sum_{j=0}^{2} x[0][j]$ requires c_c clock cycles, while Figure 2c uses a reduction operator of a p-stage pipeline ($p = 2$), and calculating $y[0] = \sum_{j=0}^{2} x[0][j]$ requires c_s clock cycles, where c_c, c_s satisfy

$$\frac{c_s}{c_c} = p + 1 \tag{2}$$

There is a feedback loop in Figure 2a, assuming that the reduction operator is combinational logic and its critical path delay is D. Since there is a register in the loop, its frequency f_c is

$$f_c = \frac{1}{T_c} = \frac{1}{D/1} = \frac{1}{D} \tag{3}$$

where T_c is the time of one cycle. For a p-stage pipeline adder, there are $p + 1$ registers in the loop, then its frequency f_s becomes

$$f_s = \frac{1}{T_s} = \frac{1}{D/(p+1)} = \frac{p+1}{D} \tag{4}$$

where T_s has the same properties as T_c. The frequency of the latter is increased by $p + 1$ times. Therefore, the calculation time required in Figure 2b is

$$t_c = \frac{c_c}{f_c} = c_c \times D, \tag{5}$$

and the calculation time required in Figure 2c is

$$t_s = \frac{c_s}{f_s} = \frac{(p+1) \times c_c}{f_s} = c_c \times D \tag{6}$$

This means that, when handling the same amount of data, the time required for these two solutions is the same. Figure 2d shows an example of performing one operation per clock cycle. It can only reduce the input data to $p + 1$ partial results, and the reduction calculation cannot be completed. Therefore, it is necessary to propose a reduction circuit to complete the calculation correctly and give full play to the advantages of the pipelined reduction operator.

There are some differences between the reduction circuit and accumulator [13–16], which are listed as follows:

- The accumulator has a single function, while the reduction circuit can choose different types of reduction operators to achieve different functions, such as accumulation and max function.
- The accumulator needs to implement the addition function in the circuit, and the circuit structure is complicated. While the reduction circuit uses the existing adder or multiplier as the reduction operator and then builds the peripheral circuit. The reduction circuit focuses more on data control and management.
- The accumulator is sensitive to the data type. When changing the data type, the accumulator may need to modify some circuits. For example, if the data type is changed from 16-bit fixed-point to 32-bit single-precision floating-point, the accumulator needs to add circuits for processing exponent and fraction according to the data type of the

floating-point number. As for the reduction circuit, it only needs to replace a reduction operator that supports single-precision floating point numbers, and then change the data width of the control circuit and buffer to 32-bit.

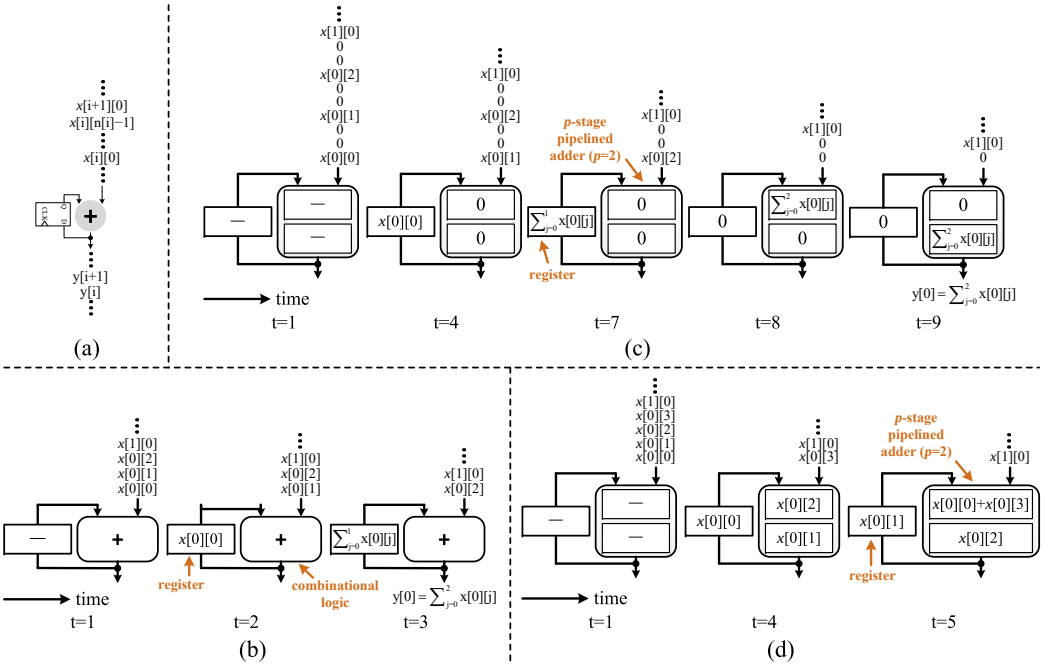


Figure 2. (**a**) A simple reduction circuit example; (**b**) use a combinational logic reduction operator to calculate $y[0] = \sum_{j=0}^{2} x[0][j]$; (**c**) use a p-stage pipeline reduction operator to calculate $y[0] = \sum_{j=0}^{2} x[0][j]$, where $p = 2$; (**d**) if an operation is required to be performed once in a clock cycle, the solution (**c**) can only reduce the input data to $p + 1$ partial results, where $p = 2$.

3. Related Works

The number of adders used in some reduction circuits is affected by other factors. For example, the number of adders in the partially compacted binary tree (PCBT) [17] is related to the number of input elements, and the number of adders in modular fully pipelined architecture (MFPA) [18] is affected by the number of stages of the adder pipeline. They tend to use more adders than designs with fixed adders. The symmetric method (SM) and the asymmetric method (AM) proposed in [12] use only one adder, but they can only handle one data set at the same time. Some reduction methods can handle multiple sets simultaneously. Zhuo et al. also proposed the fully-compressed binary tree (FCBT), dual strided adder (DSA), and single strided adder (SSA) in [17], and the number of adders they used is 2, 2, and 1, respectively. Huang et al. also proposed two reduction methods, area-efficient modular fully pipelined architecture (AeMFPA) and the alternative design of AeMFPA (A^2eMFPA) in [18], which focus on reuse and portability, and the number of adders they use is 2. The delayed buffering (DB) proposed by Tai et al. in [19] uses only one adder, and it has good performance. Reference [20] proposed a reduction circuit based on a binary tree. However, the use of seven floating-point adders greatly reduces its appeal. Finally, a tag-based random order vector reduction circuit reported in [21] also uses only one adder and has the ability to process multiple data sets at the same time. However, the buffer size becomes an irritating issue since it depends on the number of data set. We have previously conducted related research on reduction circuit in [22,23], and proposed the state-based method (SBM). In this paper, we propose a simpler method than SBM.

4. Binary Tree Path Partition

4.1. An Intuition

For m accumulation tasks $y_a = \sum_{i=0}^{n_0-1} a_i$, $y_b = \sum_{i=0}^{n_1-1} b_i$, $\cdots$, the idea of the BTPP is to divide the total $\sum_{i=0}^{m-1} n_i$ elements in units of p (p is the pipeline stage of the adder). As shown in Figure 3a, the calculation process of y_a is divided into k iterations, where each iteration processes p data ($p = 8$). When the data processed by an iteration is less than p, 0 needs to be complemented, such as iteration $k-1$. However, if p is large, the way of complementing 0 will cause performance degradation. To further improve performance, we use path partition, as shown in Figure 3b. According to the input data, the system realizes path partition by changing the connections between nodes. This means that, when the number of elements n_i in a single set is not an integer multiple of p, the system can still provide high performance. For p operations in an iteration, if we arrange their execution time reasonably, it can be implemented using a p-stage pipeline adder. That is to say, the binary tree included in the iteration is the way BTPP processes data, but there is no binary tree in the hardware implementation of BTPP. BTPP only uses one reduction operator, which can effectively reduce area consumption. In the initial situation, the structure of different iterations is the same, so we only need to use the circuit to achieve the function of one iteration, as shown in Figure 3c.

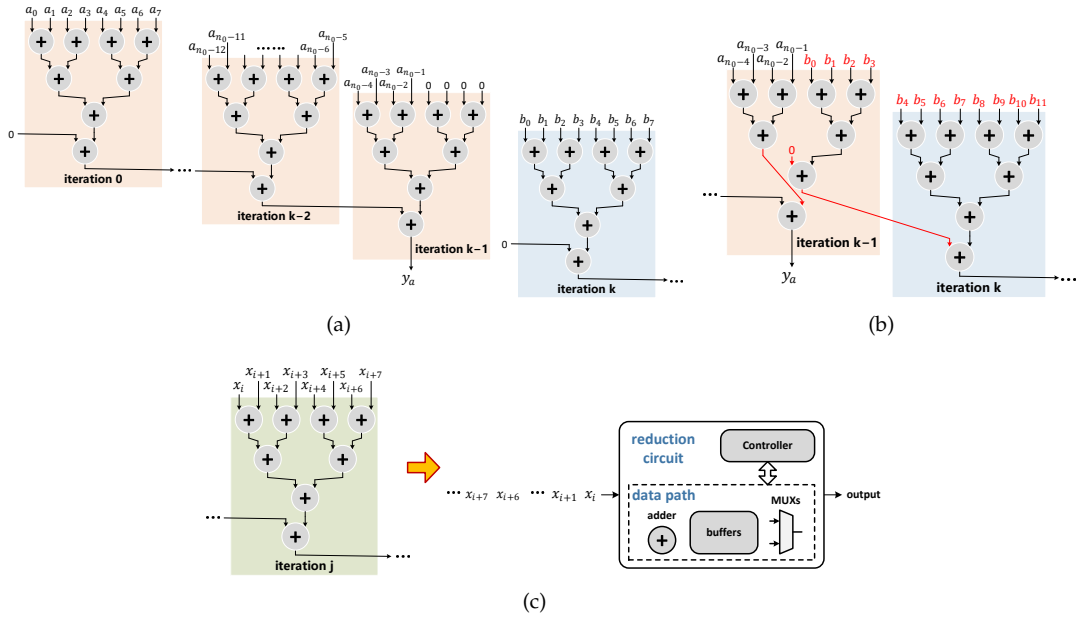


Figure 3. The design idea of BTPP. (**a**) structured calculation pattern combining binary tree and adder chain; (**b**) apply path partition to improve system performance; (**c**) BTPP hardware design: use only one reduction operator.

It can be seen from Figure 3 that the reduction circuit designed with this method has the ability to handle multiple data sets. The latter data sets have no effect on the previous data sets. For example, as shown in Figure 3b, regardless of the existence of data set B, the path partition result of data set A does not change. Therefore, it does not matter whether a single data set or multiple data sets are handled. In addition, it can handle data sets with any number of elements. When the number of elements is greater than p, these elements are calculated through multiple iterations. When the number of elements is less than p, the path partition allows elements of different data sets to exist in the same iteration.

4.2. The Proposed Reduction Method

Algorithms 1 and 2 show the pseudo-code of BTPP. BTPP is divided into two parts. The first part whose input is the pipeline stages p of the reduction operator is to obtain the initial path table that contains information such as data flow and initial scheduling.

Then, the initial path table is partitioned according to the input elements. The pipeline stages p of the reduction circuit needs to satisfy $p \geq 2$. When $p = 1$, BTPP becomes serial execution, similar to Figure 2a, and BTPP is not required at this time. When $p \geq 2$, the method of Figure 2a fails to take advantage of the pipeline, which is a problem that BTPP needs to solve. Our main contribution is embodied in Algorithm 2, which is the core of the entire BTPP.

Algorithm 1 BTPP Part 1: Binary tree path initialization

Input: pipeline stages p of the reduction operator

Output: initial path table

1. create a binary tree of p input;
2. add a node to add the result of the binary tree to the result of the last iteration;
3. get the data flow graph $G = (N, E)$;
4. limiting the initial interval to p and the number of adders to 1, perform software pipeline on G to obtain scheduling scheme S;
5. get the initial path table according to G and S;
6. **return** initial path table.

Algorithm 2 BTPP Part 2: Path partition

Input: initial path table

Output: path table

1. **while** there is element x_i input **do**
2. **if** x_i is the last element in the data set X **then**
3. */* for paths belonging to X in iteration n: */*
4. **if** do not need to be added to the results of the previous iteration **then**
5. delete the node used for addition between iterations;
6. **end if**
7. delete all nodes in paths belonging to data set X that only use operand 1;
8. */* for paths that do not belong to X in iteration n: */*
9. delete the nodes occupied by X;
10. **return** path table.
11. **end if**
12. **end while**

The reduction task in Figure 1a has m data sets, and the i-th data set contains $n[i]$ elements. For this reduction task, Figure 4 shows the workflow executed on different hardware. Figure 4a is the workflow executed on the processor. First, the reduction task is statically compiled by the compiler to obtain instructions. Then, these instructions are sent to the processor. Figure 4b is the workflow of the reduction task executed on the BTPP hardware. BTPP part 1 is used to generate the initial path table. The initial path table is only related to the pipeline stages p of the reduction operator, and it does not contain any information about the reduction task. In other words, if the circuit directly uses the initial path table to process the reduction task, it will get the wrong result. The information obtained after the path partition of the reduction task is included in the path table, which is generated by BTPP part 2. These path tables are sent to the controller and data-path for correct calculation. Figure 4a already knows the number of data sets and the number of elements in each data set at the compilation phase. Unlike Figure 4a, the initial path table generated during the compilation phase does not contain any information related to

the reduction task. The reduction task is to input data into the BTPP hardware during the hardware execution phase, and then BTPP part 2 performs path partition on the hardware in real time to obtain the path table. In other words, the BTPP hardware neither knows how many data sets are input nor how many elements each data set contains before the hardware execution phase. If the reduction circuit is designed according to the process shown in Figure 4a, the circuit can only be used for the specified number of data sets and elements. When the number of data sets is changed or the number of elements in each data set is changed, the scheme in Figure 4a needs to be recompiled to generate new instructions. However, BTPP does not have this problem because it can handle any number of data sets, and each data set can have any number of elements. Only when the pipeline stages p of the reduction operator is changed does BTPP part 1 need to be re-executed to generate a new initial path table.

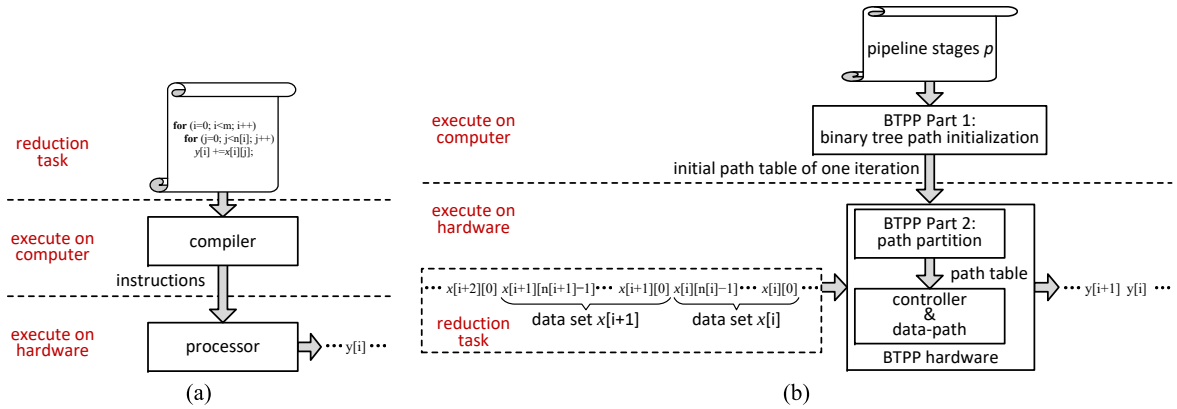


Figure 4. The workflow of the reduction task in Figure 1a on different hardware. (**a**) execute on the processor; (**b**) execute on BTPP hardware.

Figure 5 is an example of Algorithm 1, where the adder pipeline stage $p = 4$. First, we construct a binary tree with p input, and then add an operation to add the results of the previous iteration, as shown in Figure 5a,b. Then, we execute the software pipeline based on Figure 5b, and the result is shown in Figure 5c. Its initial interval is p, and only one addition operation is performed per clock cycle. Due to the use of a software pipeline, the p addition operations in one iteration can be implemented by an adder with a p-stage pipeline in a time-division multiplexing manner. We mark the clock cycle of each addition operation in the data flow graph to get Figure 5d. After using the software pipeline, the execution time of nodes in different iterations shows a certain regularity, so we only need to analyze a single iteration. As shown in Figure 5e, the input data of iteration k is x_i–x_{i+3}, and they start to be input at time $k \times p$, and one piece of data are input every clock cycle. The execution time of each node in iteration k can also be obtained by k and p. Thus, we get the initial path table of iteration k. For data x_i, the nodes it passes are $t_1 \rightarrow t_3 \rightarrow t_4$. The nodes that the data x_{i+1} pass through is the same as the data x_i, which is also $t_1 \rightarrow t_3 \rightarrow t_4$. However, the situation of node t_1 is different from that of nodes t_3 and t_4. For node t_1, x_i is used as the first operand input, and x_{i+1} is used as the second operand input. For nodes t_3 and t_4, x_i and x_{i+1} go through the same operand. To further distinguish the path of each data, we added the operand information of each node in the initial path table. Since an addition operation requires two operands, we use "1" to represent the data as the first operand of the node and use "2" to represent the data as the second operand of the node. The path of the data x_{i+2} and x_{i+3} is similar to that of x_i and x_{i+1}. TNI (to next iteration) in the path table is used to indicate whether the result produced at the end of the data path is a partial result or the final result. For example, for x_i, its path ends at node t_4. If TNI is 1, it means that the result produced by node t_4 is a partial result, and it needs to be sent to the next iteration. Otherwise, the result produced by node t_4 is the final result. In the initial

path table, the default value of TNI is 1. Path partition is carried out on the basis of the initial path table.

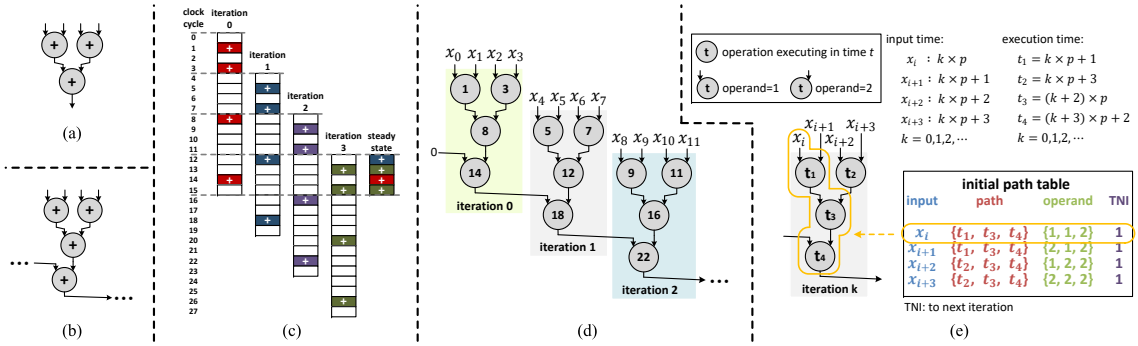


Figure 5. An example of the Algorithm 1, where $p = 4$. (**a**) binary tree; (**b**) binary tree with an addition used to add the previous partial result; (**c**) software pipeline; (**d**) data flow graph marked with the clock cycle of each operation; (**e**) initial path table of iteration k.

Figure 6 shows an example of path partition, in which the pipeline stage p of the adder is 4. For data set $A = \{a_0, a_1, \cdots, a_8\}$, solving $\sum_{i=0}^{8} a_i$ requires $\lceil 9/4 \rceil = 3$ iterations to complete. For iteration 0, the input data are a_0–a_3. Since none of the input data of iteration 0 are the last elements of the data set, iteration 0 does not require path partition. Paths of iteration 0 in the path table are the same as paths of iteration 0 in the initial path table. Since iteration 0 does not need to be added to the result of the previous iteration, the operand 1 of node 14 is 0. The output of iteration 0 is a partial result, not the final result, so the partial result needs to be sent to iteration 1. The situation of iteration 1 is similar to iteration 0. The input data of iteration 1 are not the last elements of the data set, so iteration 1 does not require path partition. Paths of iteration 1 in the path table are the same as the paths of iteration 1 in the initial path table. Node 18 adds the partial result of iteration 0 and the result of the addition of a_4–a_7, and the generated partial result is sent to iteration 2. For iteration 2, the input data are a_8, b_0, b_1, and b_2. Since a_8 is the last element of data set A, iteration 2 requires path partition. The path passed by a_8 in the initial path table is $9 \rightarrow 16 \rightarrow 22$, and the corresponding operand is $1 \rightarrow 1 \rightarrow 2$. Since a_8 needs to be added to the partial result generated by iteration 1, node 22 cannot be deleted. In iteration 2, data set A contains only one element a_8, so it is necessary to delete the nodes that only use operand 1 from the path nodes that a_8 passes. Nodes 9, 16 are deleted, and a_8 only passes through node 22. Node 22 is occupied by the data set A, so node 22 in the path of b_0, b_1, and b_2 needs to be deleted. Since the result produced by node 22 is the final result, the TNI corresponding to a_8 is 0. The path table obtained after path partitioning is shown in Figure 6.

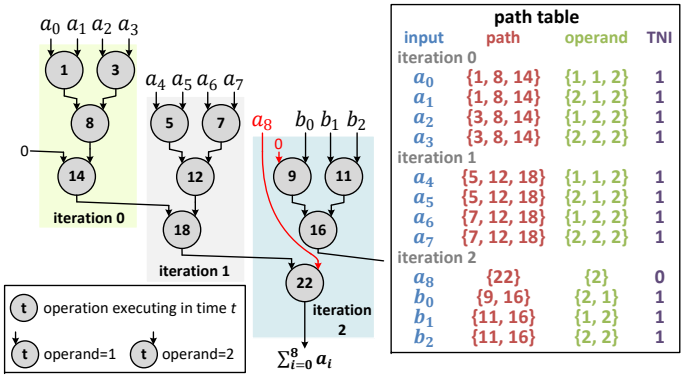


Figure 6. Example of path partition, where $p = 4$, and $\sum_{i=0}^{8} a_i$ needs to be solved.

5. Hardware Design

The hardware structure of BTPP is shown in Figure 7. It is mainly composed of the controller, path table pipeline, and data-path. The controller includes modules *system_time_gen*, *path_table_gen*, *buffer_management*, *MUX_sel_gen*, and *dout_valid_gen*. Different signals are marked with different colored lines. Data are input sequentially through the port *din*, and the signal *din_valid* is used to indicate whether the current input data are valid. If the signal *set_end* is valid, it means that the current input data are the last element of the set. Module *system_time_gen* is used to generate time information, which indicates that the *i*-th operation of the *k*-th iteration is being executed. Module *path_table_gen* generates path table based on input signals *din_valid* and *set_end*. In addition, the path table is input into the path table pipeline. The path table pipeline has a *p*-stage, the same as the reduction operator. Module *buffer_management* and *MUX_sel_gen* respectively generate buffer read and write signals and MUX control signals. These signals will be sent to the data-path to control data reading and writing and computations. When all the paths in the path table have been executed, and it does not need to be added to the result of the next iteration, the signal *dout_valid* generated by the module *dout_valid_gen* becomes valid, which indicates that the signal *dout* at the current moment is the final result of the set.

Data-path contains a *p*-stage pipeline adder. The data input from *din* will be stored in buffer 1 first, and it will be delayed by *p* clock cycles. Therefore, the size of buffer 1 is *p*. During these *p* clock cycles, a path table is generated. If $x_{i,j}$ is the last element of the data set, and it is not consumed at the current moment, it will be stored in buffer 4. The data in buffer 4 will be sent to the reduction operator as operand 2. If *dout* is not the final result of the set, it will be sent to the reduction operator or stored in the buffer. Buffer 2 and buffer 3 are used to store operand 1 and operand 2, respectively. The two operands are stored in two different buffers, which can simplify the control logic of the buffer and improve the speed of reading and writing. The path tables of the data in buffer 2 are stored in buffer 5, so the two buffers use the same control signals. The path tables of the data in buffer 3 and buffer 4 are stored in buffer 6 and buffer 7, respectively.

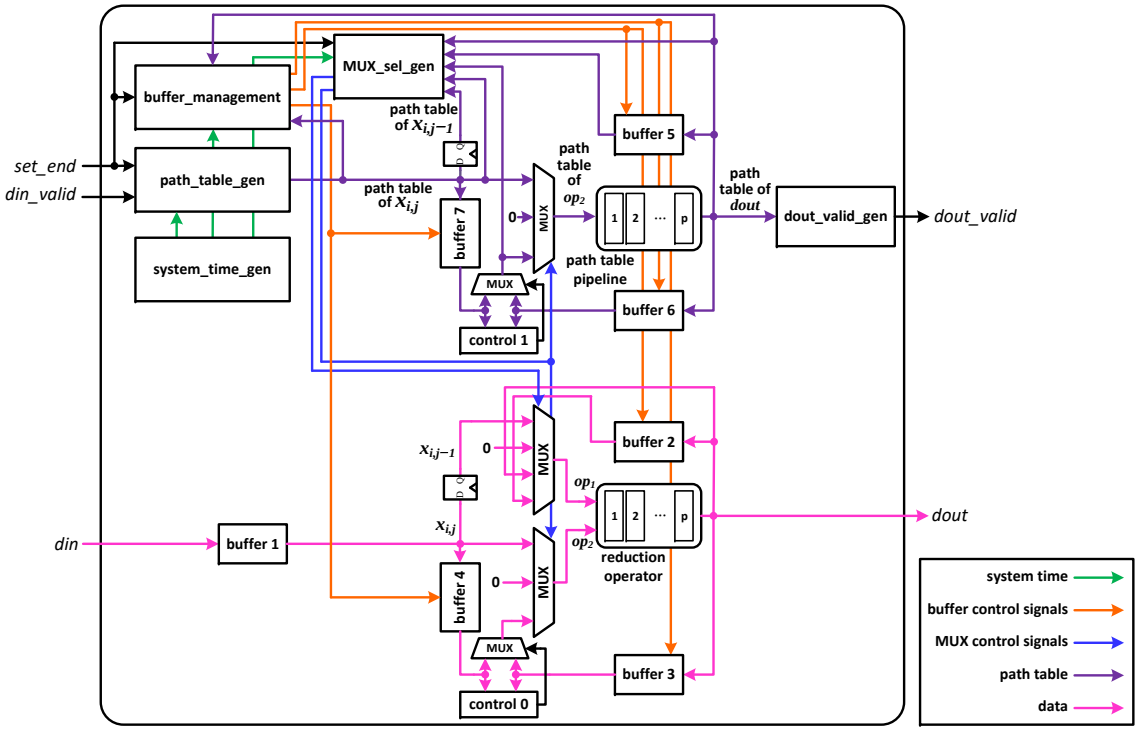


Figure 7. The hardware structure of BTPP.

When $p = 1$, BTPP executes serially, and there is no need to use a buffer. When $p \geq 2$, buffers 2–7 have the same size, and their sizes are all $(\log_2 p + 2) \times \lceil p/2 \rceil$. The maximum number of iterations executed simultaneously is $(\log_2 p + 2)$, where $p \geq 2$. As shown in Figure 5c, there are four iterations at the same time ($p = 4$). After iteration 0 is completed, iteration 4 is executed. When $p \geq 2$, there are $\lceil p/2 \rceil$ nodes in an iteration that need to be allocated storage units. Taking Figure 5e as an example, there are only two possible cases for node t_1 and node t_2. For node t_1, it will execute $x_i + x_{i+1}$ or $0 + x_{i+1}$. In both cases, there is no need to read operands from the buffer. Only the operands of node t_3 and node t_4 are stored in the buffer, so $p/2$ nodes need to be allocated storage units. If p is an odd number, it is $p/2 + 1$. Therefore, the sizes are $(\log_2 p + 2) \times \lceil p/2 \rceil$ when $p \geq 2$.

BTPP has a simple buffer management mechanism. From the previous description, when $p \geq 2$, there are at most $(\log_2 p + 2)$ iterations at the same time, and only $\lceil p/2 \rceil$ nodes in each iteration need to be allocated storage units. For these $(\log_2 p + 2) \times \lceil p/2 \rceil$ nodes, their operand 1 is stored in buffer 4, and operand 2 is stored in buffer 2 or buffer 3. The sizes of buffers are $(\log_2 p + 2) \times \lceil p/2 \rceil$, which means that each node has a dedicated storage unit. When the execution of the node is completed, the storage unit will be released. For example, in Figure 5c, after iteration 0 is executed, the storage units it occupied are released and used to store the data of iteration 4. Therefore, when a node is executed, the module *buffer_management* generates the address to read the dedicated storage unit of the node. The most significant bits of the data in buffers 5–7 are valid bits. According to these valid bits, the circuit can determine whether the data in buffers 2–4 are valid. After reading the data, the circuit writes "0" to the valid bit in buffer 5–7, which means that the data are invalid. When writing data, module *buffer_management* generates a write enable signal and an address to write data to the dedicated storage unit of the node.

Path partition is the core of the BTPP. Complicated hardware circuits will cause a large delay, leading to performance degradation. If the configuration codes corresponding to all possible situations of the path partition are stored in the memory, a large amount of memory resources will be consumed. Therefore, we need to use simple hardware to achieve path partition. In BTPP hardware, the module *path_table_gen* is used to achieve path partition and generate path tables. It is mainly implemented by the initial path table, path generation matrix (PGM), and path table flag (PTF).

Figure 8 shows an example of obtaining the path generation matrix. We rewrite the initial path table of iteration k in Figure 5e into the form in Figure 8. For operations, when the second operand comes, we can confirm its execution. Therefore, we use this feature to determine the priority of nodes in the iteration. When x_i is input, by judging whether it needs to be added to the result of the previous iteration, it can be determined whether the node at the time t_4 needs to be passed. In the same way, when $x_{i+1}, x_{i+2}, x_{i+3}$ are input, it can be determined whether it is necessary to pass the nodes at t_1, t_3, and t_2. This essentially reorders the nodes in the initial path table. We use the element "1" to indicate that the path passes through the node corresponding to the element, and the element "0" indicates that it does not pass, so we get the PGM. The first column in the PGM stores the address of the initial path table. For example, the data in the second row of the initial path table are $[t_4, t_4, t_4, t_4]$, which is stored in the 0th row of the PGM together with the location address to get the data $[2, t_4, t_4, t_4, t_4]$. Use "1" to represent the node passing t_4, so the 0th row of PGM is $[2, 1, 1, 1, 1]$. It can be seen from the PGM that the nodes that the data $x_i - x_{i+3}$ pass have not changed. Since the execution time of nodes in different iterations is different, this method can reduce the amount of data storage. The PGM can be reused in each iteration.

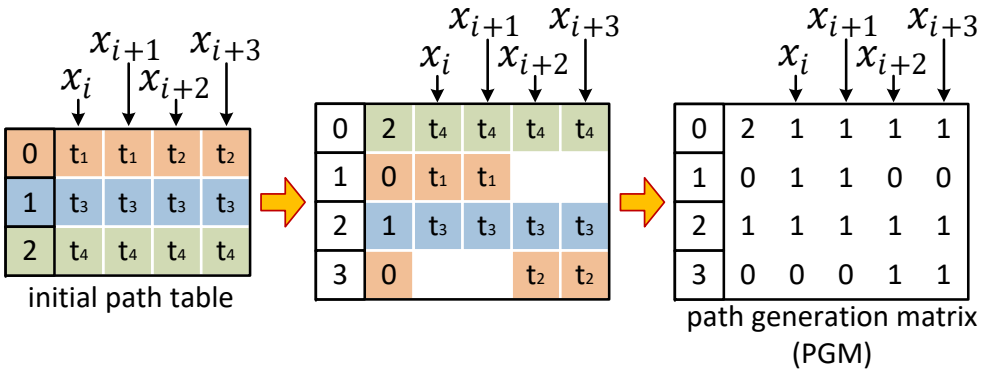


Figure 8. The generation mechanism of the path generation matrix.

Figure 9 gives an example of the path partition of iteration 2 in Figure 6. According to the information in Figure 5e, when $n = 2$, the data input time of iteration 2 is 8–11. The data corresponding to the data $x_i - x_{i+3}$ are respectively $a_8 - b_2$, so x_i belongs to set A, and $x_{i+1} - x_{i+3}$ belongs to set B. Path partition is obtained through multiple OR operations on elements in PGM and PTF. In an iteration, the elements belonging to the same set in the PGM and PTF will perform the OR operation. All elements of the PTF are zero in the initial situation, and *mod_cycle* is the current cycle modulo p. As shown in Figure 9a, when the cycle is 8, *mod_cycle* is 0. At this time, columns corresponding to only x_i in the PGM and initial PTF perform an OR operation to obtain PTF(0). As shown in Figure 9b–d, when the cycle is 9–11, *mod_cycle* is 1, 2, and 3, respectively. At this time, columns corresponding to $x_{i+1} - x_{i+3}$ in PGM and PTF perform OR operations to obtain PTF(1), PTF(2), and PTF(3), respectively. PTF(3) contains the information after path partition.

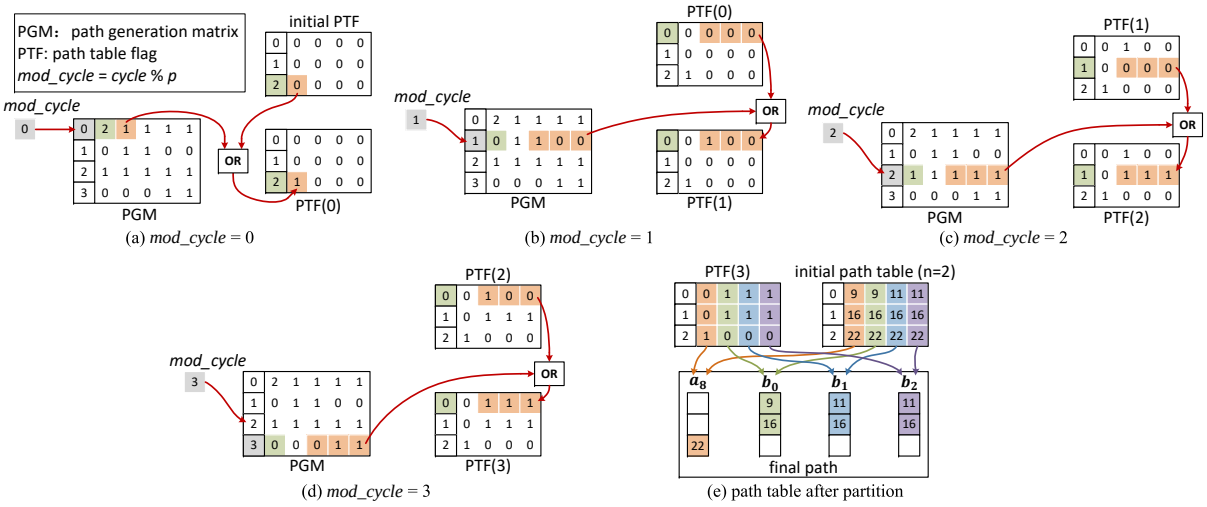


Figure 9. The example of path partition of iteration 2 in Figure 6. (**a**–**d**) the process of obtaining PTF(3); (**e**) PTF(3) combined with the initial path table to get the final path.

PTF combined with the initial path table can get the partitioned path. As shown in Figure 9e, the execution time $t_1 - t_4$ of the nodes in iteration 2 are 9, 11, 16, and 22, respectively, so we can get the initial path table. The element "0" in PTF(3) means that the path does not pass through the node corresponding to the element, and the element "1" means that the path passes through the node corresponding to the element. When $n = 2$, the default path of x_i in the initial path table is $\{t_1, t_3, t_4\}$, which is 9, 16, 22. However, after path partition, the first column of PTF(3) is {0, 0, 1}, which means that x_i does not pass through nodes 9 and 16, but only passes through node 22. Therefore, the path of a_8 is {22}. The paths corresponding to data $b_0 - b_2$ are similar.

A working example of BTPP hardware is given. To better understand the hardware working, we first give the overall situation of the calculation, as shown in Figure 10. Different from Figure 6, buffer 1 delays the input data by p clock cycles, so the execution time of all nodes is delayed by p. In this example, the reduction operator is an adder with a 4-stage pipeline ($p = 4$), and the tasks to be solved are $\sum_{i=0}^{4} a_i$ and $\sum_{i=0}^{3} b_i$. Figure 11 shows the hardware working details. Path table is generated at $x_{i,j}$ and updated at $dout$. At cycle 0, data a_0 is input from din. The signal din_valid is 1, which means that a_0 is valid data. Then, data $a_1 - a_3$ are input in sequence. At cycle 4, the signal set_end becomes valid, indicating that the data a_4 input at the current moment are the last data of the data set A. After four clock cycles are delayed by buffer 1, data a_0 appear at $x_{i,j}$, and its path table has been generated. In the path table, the "5" in $\{5, 12, 18\}$ and the first "1" in $\{1, 1, 2\}$ represent that a_0 is sent to the reduction operator as the first operand at cycle 5. Other data represent similar meanings. The value of TNI is 1, which means that the calculation result of cycle 18 will be sent to the next iteration for calculation. Data a_0 are not consumed at the current moment. Since it will be sent to the reduction operator as the first operand, there is no need to store it in the buffer. At cycle 5, $x_{i,j}$ is a_1. According to its path table, it is sent to the reduction operator as the second operand at this time. Therefore, $op2$ is a_1. The signal $x_{i,j-1}$ is a_0, which meets the demand of $op1$ at the current moment, so $op1$ is a_0. The situation is similar for cycles 6–7. At cycle 8, $x_{i,j}$ is a_4, and it is not consumed at the current moment. It is the second operand of the node 22, so it is stored in buffer 4. At cycle 9, $x_{i,j}$ is b_0. According to the path table, b_0 is sent to $op2$. The signal $x_{i,j-1}$ at this time does not meet the requirements of $op1$, so $op1$ is 0. The signal $dout$ outputs partial result $a_0 + a_1$, and its path table is $\{12, 18\}$ and $\{1, 2\}$. From the path table of a_0, a_1 and $a_0 + a_1$, it is easy to see that $a_0 + a_1$ is the calculation result of a_0 and a_1. Since $a_0 + a_1$ needs to be consumed as the first operand at cycle 12, it is stored in buffer 2. The situation at cycle 11 is similar, and the difference is that $a_2 + a_3$ is stored in buffer 3 because it is consumed as the second operand. At cycle 20, the path table of $b_0 + b_1 + b_2$ is empty except TNI is 1. From the information of din and iteration, the data $b_0 + b_1 + b_2$ are the result of iteration 1. Therefore, it needs to be sent to the last node of iteration 2, which means that $b_0 + b_1 + b_2$ will be consumed at cycle 26. The situation is similar for cycle 22. At cycle 26, the path table of $a_0 + a_1 + a_2 + a_3 + a_4$ is empty, which means it is the final result of data set A, so the signal $dout\_valid$ becomes valid.

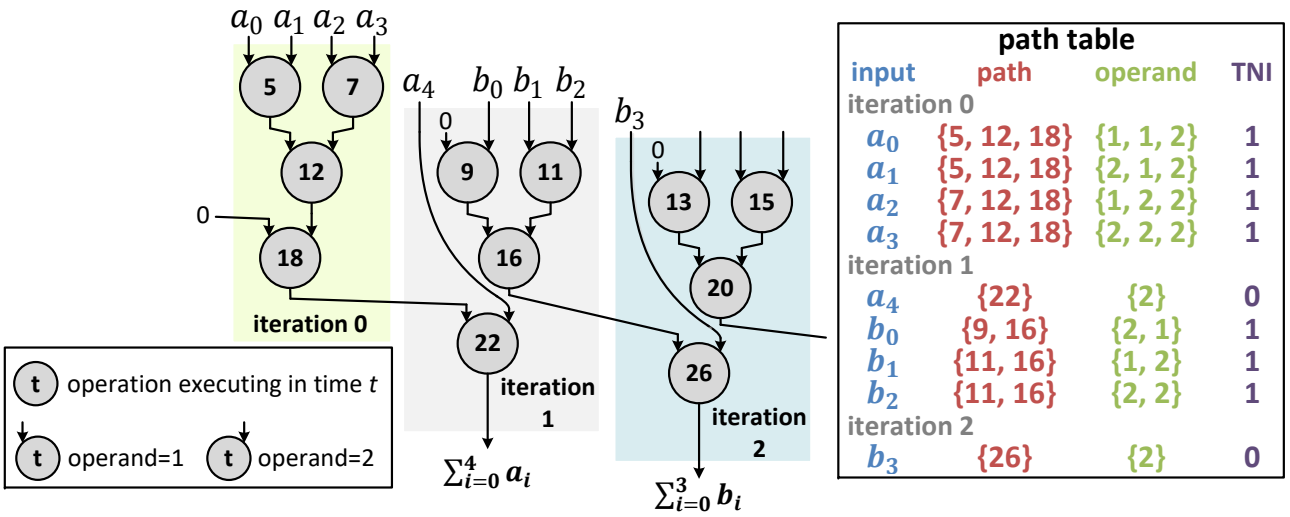


Figure 10. A calculation example, where $p = 4$. $\sum_{i=0}^{4} a_i$ and $\sum_{i=0}^{3} b_i$ needs to be solved.

cycle	iteration	din_valid	din	set_end	$x_{i,j}$ data	$x_{i,j}$ path_table path	$x_{i,j}$ path_table operand	$x_{i,j}$ path_table TNI	$x_{i,j-1}$	op1	op2	dout data	dout path_table path	dout path_table operand	dout path_table TNI	buffer 2	buffer 3	buffer 4	dout_valid
0	0	1	a_0																
1	0	1	a_1																
2	0	1	a_2																
3	0	1	a_3																
4	1	1	a_4	1	a_0	{5,12,18}	{1,1,2}	1											
5	1	1	b_0		a_1	{5,12,18}	{2,1,2}	1	a_0	a_0	a_1								
6	1	1	b_1		a_2	{7,12,18}	{1,2,2}	1	a_1										
7	1	1	b_2		a_3	{7,12,18}	{2,2,2}	1	a_2	a_2	a_3								
8	2	1	b_3	1	a_4	{22}	{2}	0	a_3										
9					b_0	{9,16}	{2,1}	1	a_4	0	b_0	a_0+a_1	{12,18}	{1,2}	1			a_4	
10					b_1	{11,16}	{1,2}	1	b_0							a_0+a_1		a_4	
11					b_2	{11,16}	{2,2}	1	b_1	b_1	b_2	a_2+a_3	{12,18}	{2,2}	1	a_0+a_1		a_4	
12					b_3	26	{2}	0	b_2	a_0+a_1	a_2+a_3					a_0+a_1	a_2+a_3	a_4	
13									b_3			b_0	{16}	{1}	1			a_4	b_3
14																b_0		a_4	b_3
15												b_1+b_2	{16}	{2}	1	b_0		a_4	b_3
16										b_0	b_1+b_2	$a_0+a_1+a_2+a_3$	{18}	{2}	1	b_0	b_1+b_2	a_4	b_3
17																	$a_0+a_1+a_2+a_3$	a_4	b_3
18										0	$a_0+a_1+a_2+a_3$							a_4	b_3
19																		a_4	b_3
20												$b_0+b_1+b_2$	{}	{}	1			a_4	b_3
21																$b_0+b_1+b_2$		a_4	b_3
22										$a_0+a_1+a_2+a_3$	a_4	$a_0+a_1+a_2+a_3$	{}	{}	1	$b_0+b_1+b_2$		a_4	b_3
23																$b_0+b_1+b_2$			b_3
24																$b_0+b_1+b_2$			b_3
25																$b_0+b_1+b_2$			b_3
26										$b_0+b_1+b_2$	b_3	$a_0+a_1+a_2+a_3+a_4$	{}	{}	0	$b_0+b_1+b_2$			b_3	1
27																			
28																			
29																			
30												$b_0+b_1+b_2+b_3$	{}	{}	0				1

reduction operator: adder with 4-stage pipeline

TNI: to_next_iteration

For the sake of simplicity, the registers used to delay one clock cycle are included in the buffer 2 and buffer 3, and the elements of the same column in the buffer 2 and buffer 3 does not mean that they are cached in the same physical unit.

Figure 11. An example to show how BTPP hardware works. The reduction operator used is an adder with 4-stage pipeline, and the calculation tasks are $\sum_{i=0}^{4} a_i$ and $\sum_{i=0}^{3} b_i$.

6. Evaluations and Comparison

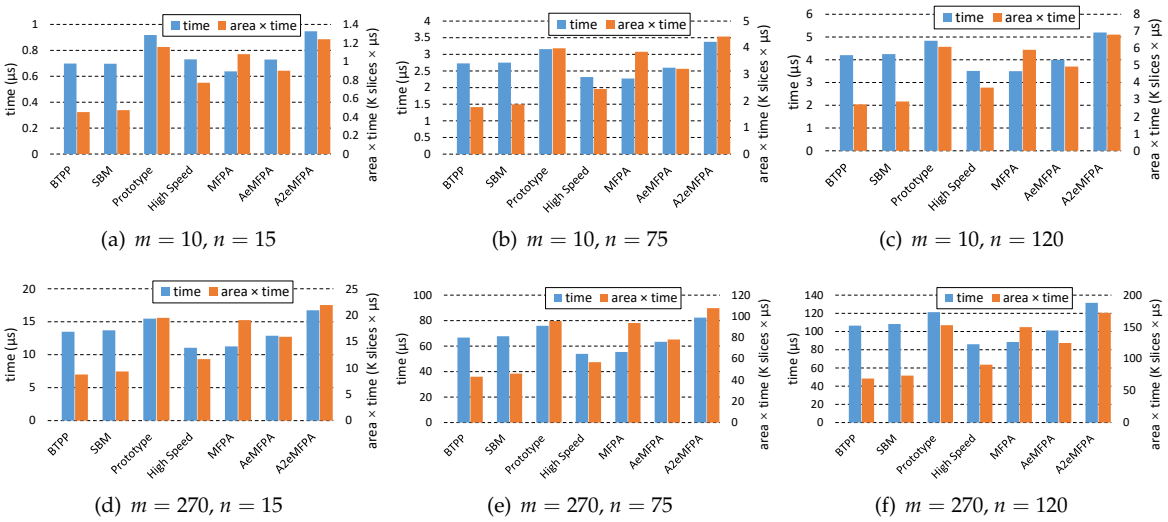
We implemented BTPP on Xilinx XC5VLX110T platform using Verilog HDL because most of the existing work is implemented on the same platform. Our circuit is designed and validated manually. To ensure that the reduction operator used by BTTP is the same as other works, a double-precision floating-point adder with 14-stage pipeline, which is synthesized by Xilinx ISE 10.1, is chosen as the reduction operator. The post-implementation results are shown in Table 1. MFPA, AeMFPA, and A^{2}eMFPA are designs of multiple reduction operators. They consume more slices than the design of a single reduction operator, but the two designs proposed by [21] are exceptions. Although they are designs of a single reduction operator, their frequency and slices are related to the number of data sets they can process simultaneously. As the number of data sets grows, the slices increase more, which will limit its application. The frequency and slices of BTPP and SBM are stable. It has nothing to do with the number of data sets and the number of elements contained in each data set. Their frequency and slices are more balanced in all designs. However, the circuit of SBM is more complicated, which is mainly reflected in the buffer management unit. It needs to identify which units in the buffer store data that are valid, and it also needs to detect elements in the buffer that belong to the same data set. Compared with SBM, the hardware structure of BTPP is simple, without particularly complex logic. At the same time, it can guarantee a better clock frequency and slice consumption than SBM. On Virtex-5, each slice includes four 6-input LUTs and four registers. BTPP uses 1554 slice registers. In addition, 1495 slice LUTs are used as logic, and only three slice LUTs are used as route-thru. The buffers in BTPP are all implemented with dual-port RAM, so 10 BRAMs are used. In addition, 3 DSP48Es are used in BTPP.

Table 1. Implementation results comparison between different designs on Xilinx XC5VLX110T.

	BTPP	SBM [23]	Prototype [21]			High Speed [21]			MPFA [18]	AeMFPA [18]	A^{2}eMFPA [18]
			$m^* = 512$	$m^* = 1024$	$m^* = 2048$	$m^* = 512$	$m^* = 1024$	$m^* = 2048$			
adders	1	1	1	1	1	1	1	1	5	2	2
Freq. (MHz)	305	300	268	245	201	378	359	356	367	321	247
Slices	648	680	1260	1848	3645	1055	1394	2176	1692	1234	1309

pipeline stages of adder: 14, data type: double-precision floating-point (64-bit), $^*m$ is the upper limit of the number of data sets processed simultaneously by Prototype [21] and High Speed [21].

We compared the execution time and area-time product of different designs in Table 1, and the results are shown in Figure 12. The number of pipeline stages of reduction operator is 14. The parameter m is the number of data sets, and n is the number of elements in each data set. The two designs of [21] in Figure 12 are two circuits that support a maximum of 512 data sets. Since the clock frequency of BTPP is lower than High Speed, MFPA and AeMFPA, BTPP does not have an advantage in execution time. However, it has a smaller consumption of slices, so it has the least area-time product of all designs, which makes BTPP extremely competitive.

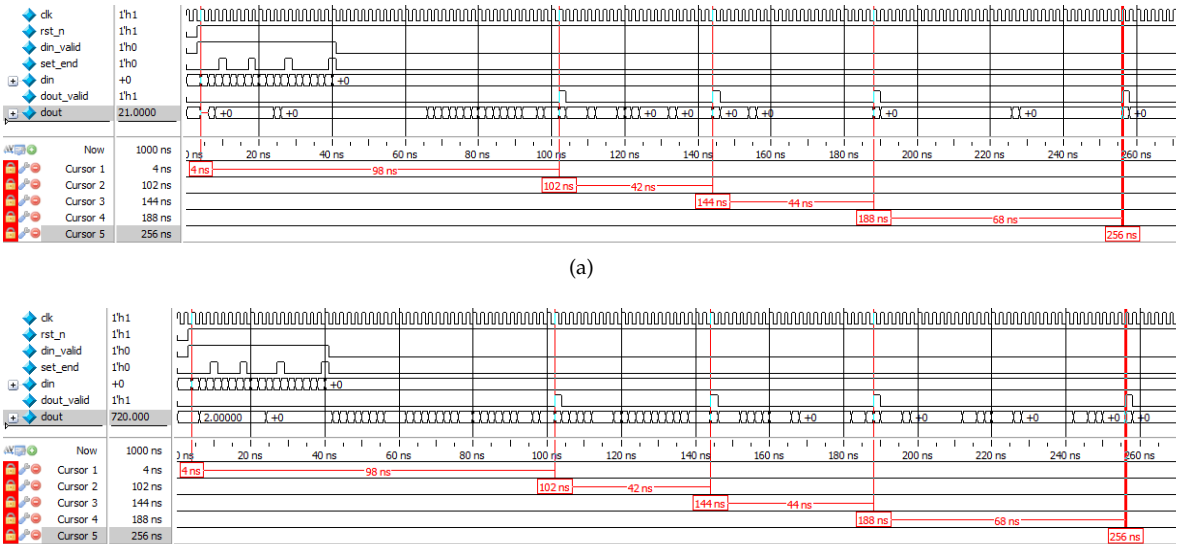


(a) $m = 10, n = 15$ (b) $m = 10, n = 75$ (c) $m = 10, n = 120$

(d) $m = 270, n = 15$ (e) $m = 270, n = 75$ (f) $m = 270, n = 120$

Figure 12. The comparison of execution time and area-time product between different designs, where $p = 14$.

To prove that the BTPP can be used for different reduction operators, we give pre-synthesis simulation examples of BTPP using an adder and a multiplier, respectively, in Figure 13. Since the most common problems are $\sum_{i=0}^{n-1} x_i$ and $\prod_{i=0}^{n-1} x_i$, which are the main problems solved by the reduction circuit, we chose adder and multiplier as the reduction operator. The adder and multiplier are IPs generated by Xilinx Vivado 2019.1 software, their data are double-precision floating-point numbers, and the number of pipeline stages is 14. The circuit is implemented in Verilog HDL, and is simulated by ModelSim called by Vivado. Data sets $\{1, 2, 3\}$, $\{1, 2, 3, 4\}$, $\{1, 2, 3, 4, 5\}$, and $\{1, 2, 3, 4, 5, 6\}$ are the test. It can be seen from the figure that the input signals of Figure 13a,b are the same, and the time for *dout_valid* to become valid is also the same. At 256ns, both circuits output the calculation results of the last data set. The output of Figure 13a is 21 ($= 1 + 2 + 3 + 4 + 5 + 6$), and the output of Figure 13b is 720 ($= 1 \times 2 \times 3 \times 4 \times 5 \times 6$).



(a)

(b)

Figure 13. Pre-synthesis simulation. (**a**) reduction operator is a double-precision floating-point adder with a 14-stage pipeline; (**b**) reduction operator is a double-precision floating-point multiplier with a 14-stage pipeline.

We have implemented the two designs in Figure 13 on Xilinx Artix-7 XC7A100T, and the post-implementation results are shown in Table 2. Since the multiplier uses more DSP, its frequency is much higher than that of the adder, and it uses fewer slice LUTs and slice registers than the adder. The BTPP that uses the adder as the reduction operator and the BTPP that uses the multiplier as the reduction operator are listed in the last two columns of Table 2. Through comparison, it can be seen that the control circuit of BTPP does not use DSP but uses 9.5 BRAMs. The resources used by the two BTPP control circuits are approximately equal.

Table 2. BTPP with different reduction operators are implemented on Xilinx Artix-7 XC7A100T.

	Adder	**Multiplier**	**BTPP (Adder)**	**BTPP (Multiplier)**
bit-width	64	64	64	64
pipeline stages	14	14	−−	−−
Freq. (MHz)	296	344	294	285
Slice LUTs	661	176	1372	926
Slice Reg.	995	557	1809	1324
DSPs	3	11	3	11
BRAMs	0	0	9.5	9.5

We have implemented two circuits for matrix-vector multiplication on Xilinx Artix-7 XC7A100T. The first one is composed of a multiplier and an adder. The accumulation function is realized by feeding back the output of the adder to the input. The second circuit is composed of a multiplier and BTPP that uses an adder as a reduction operator. The adder and multiplier used here are the same as those in Table 2, and the data type and number of pipeline stages are unchanged. The circuit is implemented in Verilog HDL, synthesized and placed and routed through Xilinx Vivado 2019.1 software. The post-implementation results are shown in Table 3. For the circuit that does not use BTPP, the slice LUTs and slice registers consumed are less than the design using BTPP.

Table 3. Two circuits for matrix-vector multiplication are implemented on Xilinx Artix-7 XC7A100T.

	Multiplier + Adder	Multiplier + BTPP
bit-width	64	64
Freq. (MHz)	294	285
Slice LUTs	835	1541
Slice Reg.	1744	2430
DSPs	14	14
BRAMs	0	9.5

The two circuits in Table 3 are applied to matrix–vector multiplication, and the results are shown in Figure 14. The matrix in Figure 14 has 15 rows and the number of columns is between 15 and 30. In other words, the reduction circuit needs to handle 15 data sets, and the number of elements in each data set is between 15 and 30. For the case of not using BTPP, it needs to consume a lot of execution time. This is because the deep pipelining of the adder does not bring about an increase in throughput. BTPP takes full advantage of the deep pipelining of the adder and greatly reduces the execution time. Compared with the design that does not use BTPP, it can achieve a 12× speed up. BTPP uses the less additional area in exchange for a great improvement in performance, which fully demonstrates the advantages of BTPP.

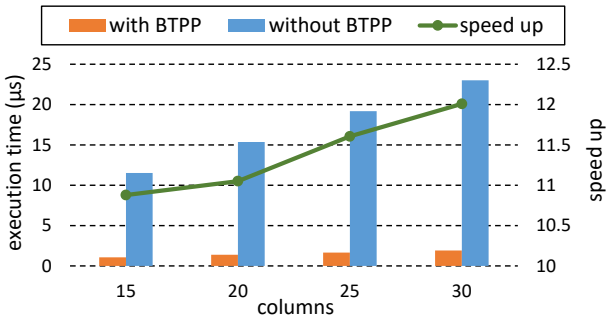


Figure 14. The comparison of execution time between the design using BTPP and the design not using BTPP. The number of rows of the matrix is 15 and the number of columns varies from 15 to 30.

7. Conclusions

In this paper, we propose a novel reduction circuit based on binary tree path partition. It can process any number of data sets, and each data set can contain any number of elements. We introduced BTPP by taking the adder as the reduction operator as an example. BTPP divides the input data into multiple data groups, and each data group contains p elements. The p elements are added through a binary tree to get the partial result, and the final result is obtained by adding these partial results. After all operations are software pipelined, the circuit only needs to use a p-stage pipeline adder to perform these operations. We implemented BTPP on FPGAs. Compared with other methods, BTPP has the least area-time product.

Author Contributions: Funding acquisition, Z.H. and J.C.; Investigation, L.T.; Methodology, L.T.; Project administration, G.C.; Writing—original draft, L.T.; Writing—review & editing, Z.H., G.C., Y.Z. and J.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (Grant Nos. 61704173, 61901440), the Beijing Municipal Natural Science Foundation (Grant No. 4202080), and the One Hundred Person Project of the Chinese Academy of Sciences.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Alawad, M.; Lin, M. FIR Filter Based on Stochastic Computing with Reconfigurable Digital Fabric. In Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, Vancouver, BC, Canada, 2–6 May 2015; pp. 92–95.
2. Mittal, R.; Prince, A.A.; Nalband, S.; Robert, F.; Fredo, A.R.J. Low-Power Hardware Accelerator for Detrending Measured Biopotential Data. *IEEE Trans. Instrum. Meas.* **2021**, *70*, 1–9. [CrossRef]
3. Dorrance, R.; Ren, F.; Marković, D. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 26–28 February 2014; pp. 161–170.
4. Sigurbergsson, B.; Hogervorst, T.; Qiu, T.D.; Nane, R. Sparstition: A Partitioning Scheme for Large-Scale Sparse Matrix Vector Multiplication on FPGA. In Proceedings of the 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), New York, NY, USA, 15–17 July 2019; pp. 51–58.
5. Jain, A.K.; Omidian, H.; Fraisse, H.; Benipal, M.; Liu, L.; Gaitonde, D. A Domain-Specific Architecture for Accelerating Sparse Matrix Vector Multiplication on FPGAs. In Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), Gothenburg, Sweden, 31 August–4 September 2020; pp. 127–132.
6. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015; pp. 161–170.
7. Wang, C.; Gong, L.; Li, X.; Zhou, X. A Ubiquitous Machine Learning Accelerator with Automatic Parallelization on FPGA. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2346–2359. [CrossRef]
8. Zeng, S.; Dai, G.; Sun, H.; Zhong, K.; Ge, G.; Guo, K.; Wang, Y.; Yang, H. Enabling Efficient and Flexible FPGA Virtualization for Deep Learning in the Cloud. In Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Fayetteville, AR, USA, 3–6 May 2020; pp. 102–110.
9. Ovtcharov, K.; Ruwase, O.; Kim, J.; Fowers, J.; Strauss, K.; Chung, E.S. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In Proceedings of the 2015 IEEE Hot Chips 27 Symposium (HCS), Cupertino, CA, USA, 22–25 August 2015; pp. 1–38.
10. Jian, O.; Wei, Q.; Yong, W.; Tu, Y.; Jing, W.; Bowen, J. SDA: Software-Defined Accelerator for general-purpose big data analysis system. In Proceedings of the 2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, 21–23 August 2016; pp. 1–23.
11. Yu, X.; Wang, Y.; Miao, J.; Wu, E.; Zhang, H.; Meng, Y.; Zhang, B.; Min, B.; Chen, D.; Gao, J. A Data-Center FPGA Acceleration Platform for Convolutional Neural Networks. In Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL), Barcelona, Spain, 8–12 September 2019; pp. 151–158.
12. Ni, L.M.; Kai, H. Vector-Reduction Techniques for Arithmetic Pipelines. *IEEE Trans. Comput.* **1985**, *C-34*, 404–411. [CrossRef]
13. Dinechin, F.D.; Pasca, B.; Cret, O.; Tudoran, R. An FPGA-specific approach to floating-point accumulation and sum-of-products. In Proceedings of the 2008 International Conference on Field-Programmable Technology, Taipei, Taiwan, 7–10 December 2008; pp. 33–40.
14. Sun, S.; Zambreno, J. A floating-point accumulator for FPGA-based high performance computing applications. In Proceedings of the 2009 International Conference on Field-Programmable Technology, Sydney, Australia, 9–11 December 2009; pp. 493–499.
15. Bachir, T.O.; David, J. Performing Floating-Point Accumulation on a Modern FPGA in Single and Double Precision. In Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, Charlotte, NC, USA, 2–4 May 2010; pp. 105–108.
16. Lahari, P.L.; Bharathi, M.; Shirur, Y.J.M. High Speed Floating Point Multiply Accumulate Unit using Offset Binary Coding. In Proceedings of the 2020 7th International Conference on Smart Structures and Systems (ICSSS), Chennai, India, 23–24 July 2020; pp. 1–5.
17. Zhuo, L.; Morris, G.R.; Prasanna, V.K. High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs. *IEEE Trans. Parallel Distrib. Syst.* **2007**, *18*, 1377–1392. [CrossRef]
18. Huang, M.; Andrews, D. Modular Design of Fully Pipelined Reduction Circuits on FPGAs. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *24*, 1818–1826. [CrossRef]
19. Tai, Y.; Lo, C.D.; Psarris, K. Accelerating Matrix Operations with Improved Deeply Pipelined Vector Reduction. *IEEE Trans. Parallel Distrib. Syst.* **2012**, *23*, 202–210. [CrossRef]
20. Kuhara, T.; Tsuruta, C.; Hanawa, T.; Amano, H. Reduction calculator in an FPGA based switching Hub for high performance clusters. In Proceedings of the 2015 25th International Conference on Field Programmable Logic and Applications (FPL), London, UK, 2–4 September 2015; pp. 1–4.
21. Huang, Y.; Huang, W.; Chen, R.; Wu, H.; Wei, M. A Tag Based Random Order Vector Reduction Circuit. *IEEE Access* **2020**, *8*, 41502–41515. [CrossRef]

22. Tang, L.; Cai, G.; Yin, T.; Zheng, Y.; Chen, J. A Resource Consumption and Performance Overhead Optimized Reduction Circuit on FPGAs. In Proceedings of the 2019 International Conference on Field-Programmable Technology (ICFPT), Tianjin, China, 9–13 December 2019; pp. 287–290.
23. Tang, L.; Cai, G.; Zheng, Y.; Chen, J. A Resource and Performance Optimization Reduction Circuit on FPGAs. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 355–366. [CrossRef]