ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume 27, Number 1, 2024, 1–18

Multi-Function Scan Circuit for Assisting the Parallel Computational Map Pattern

Mihai ANTONESCU* and Gheorghe M. STEFAN

University Politehnica of Bucharest, Romania, Faculty of Electronics, Telecommunications and Information Technology

Email: mihai.antonescu@upb.ro*, gheorghe.stefan@upb.ro

* Corresponding author

Abstract. Some parallel computing patterns can be accelerated using appropriate networks of simple circuits. We propose a solution, based on the Beneš-Waksman permutation network, which is adapted to efficiently accelerate not only permutation, but some of the most used parallel computational patterns such as: pack, prefix operations and reductions. The structural context considered for deploying our circuit is the map parallel pattern represented by an array of computational elements. The developed network receives a vector from the map array and outputs a vector for functions such as permute, pack, prefix sum (thus closing a first global loop over the map array of cells). For reduction functions (add, min, max) the network returns a scalar (thus closing a second global loop over the map array). With these improvements, this network adds circuit support for frequently used functions, in addition to map-type functions performed in the array of computing elements. While for reduction functions the frame of the permutation network can be easily adapted, for prefix functions and for the pack function new forms of implementation are proposed. The cells of the Beneš-Waksman network are redesigned to support the additional functionality. Some applications are then presented to emphasize the utility of our design.

Key-words: Benes network; General-Purpose Accelerator; Multi-Function Permutation circuit; Scan Circuit

1. Introduction

The most intuitive parallel computation pattern is the n-cell ***map*** pattern (see p. 21 in [1]). Besides this pattern there are other patterns that can be easily implemented in software, although

their procedures are time consuming. Thus the gains obtained using the map pattern are diminished if patterns such as reduction, prefixes, pack [1] don't benefit from appropriate hardware solutions. There are two main categories of patterns which we will consider in our approach: (1) input data is only rearranged, as for the permute and pack functions, and (2) input data is submitted to simple operations such as addition or comparison, as is the case for prefix and reduce functions. The solution we adopt is to improve the functionality of each cell of a Beneš-Waksman permutation network [2–4].

In the following, we use the term *scan function* as a generic name to specify functions whose input and output are both vectors. The functions we consider (see page 21 in [1]) for integration in the same circuit are:

- scan functions: permute, pack, prefix sum (a typical scan function).
- reduction functions: add, min, max.

The depth of these circuits is in $O(log\, n)$. The size for the generic scan functions is in $O(nlog\, n)$, while for the reduction functions the size is in $O(n)$. Taking into account the frequency of use for these functions, in the case of a general parallel computing system, the circuit implementation for any one as a distinct structure is not efficient. However, it may be effective to design a structure able to perform all these functions in $O(log\, n)$ time with a $O(nlog\, n)$ sized circuit. Such a circuit has numerous applications of interest, some of which will be described in Section 7.

The most commonly used functions are the reduction functions. They appear in linear algebra computation, in vector and matrix multiplications, with direct applications in machine learning algorithms used today. It is such an important operation, that most parallel computation frameworks include it as a primitive [5].

The prefix sum function can also be found as a primitive in some parallel frameworks (for example mpi [5]). It can also appear in algorithms such as counting, sort or random number generation in an arbitrary probability distribution, enumerations (counting the number of true elements up to a point), pack (as explained below), distribute (copy elements to multiple locations, although possibly needing multiple scans) and sparse matrix-vector multiplication [6].

The pack function is used to separate marked from unmarked values in a sequence of data. One common usage of this type of separation is selecting non-zero values (while preserving their order) from zero values. Another use is in Convolutional Deep Neural Network for the pooling (sub-sampling) stage with various strides.

Permutations are another data reorganizing function. Some notable permutations are: reversals, perfect shuffle, matrix transpose computing (when processed as a vector) [7].

The vast majority of currently used multi-core engines don't have circuit support for the previously listed functions. They are programmed [6] [8] [9] with performance limited mainly by the inter-core communication issues or due to the limited bandwidth with the system memory. Processing these functions in the software layer is relatively slow and current circuit approaches are based around single function circuits, tailored specifically to one application. The support offered by the circuit we propose improves the performance of a many-cell computing engine by removing the limitations imposed by the inter-communication in a map array of cells while also supporting multiple functions for various use cases in different applications. The term *remove* was used because skillful programming techniques combined with pipelining can negate the effects of the network latency. An example is shown in [10] and additional examples are shown in Chapter 7.

The backbone circuit which we chose for further development is the Beneš-Waksman permutation network. It offers the permutation scan-type function by design, and compared to other network topologies (as seen in [11]), its connections easily support the generic reduction shape with which we decided to start our design approach.

Therefore, the main contribution of the work presented in this paper is integrating, on the same network, more than one frequently used function for currently accelerated applications running on a linear array of computing elements.

In Section 2 we present the functions accelerated using our circuit. In Section 3 the pipeline implementation is presented. The structure of the multi-functional cell is described in Section 4 , while in Section 5 a sequential implementation is introduced. Section 6 contains synthesis results of our proposed architecture. Section 7 exemplifies the use of the proposed multi-function network.

2. The System

2.1. The Map Scan-Reduce architecture

The system configuration in which we integrate our multi-function circuit is represented in Fig. 1, where:

- CONTROL: is a mono-core processing element used to send in each cycle an instruction, through the *log*-depth net DISTRIBUTE, to be executed in each cell of the MAP section
- MAP: is a linear array of cells, each featured with an execution unit and a big register file which allow the distribution of a large number of vectors along itself. Each cell also contains a Boolean value indicating whether the cell is active or not.
- SCAN: is a *log*-depth programmable generic scan network whose structure and functions are described in this paper.

REDUCE output is the right most output of the SCAN net. In most implementations this output returns to CONTROL, starting form the vector provided by MAP, the sum of components or the minimum or maximum value.

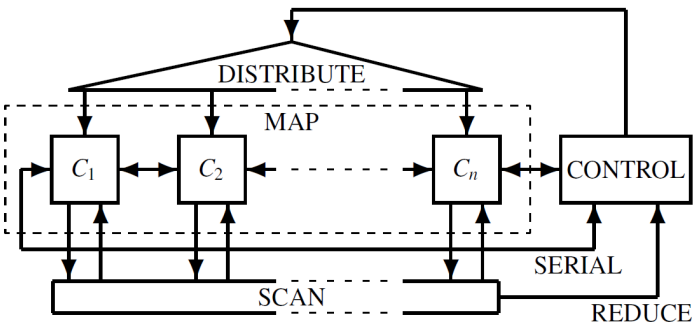

Fig. 1. The cellular system with two global loops: through the vector received back from SCAN and through the scalar received by the CONTROL processor from the REDUCE output.

The SCAN net receives from MAP a n-component vector and returns a vector of the same size, thus closing the first global loop over MAP. The output REDUCE returns a scalar to CONTROL. This scalar can be used by CONTROL to close through the DISTRIBUTE net a second loop over the MAP array.

2.2. Scan functions

2.2.1. The permute function

The generic shape of the circuit we propose is based on the permutation network introduced by Václav E. Beneš and optimized by Abraham Waksman, further called the Beneš-Waksman net. The recursive definition of the network in represented in Fig. 2. The cell used to build the network has two data inputs, x_{2i} and x_{2i-1}. The outputs, y_{2i} and y_{2i-1}, can be stored in registers allowing the pipeline connections in the net. In our implementation, each input is a pair of words: $x_j = \{v_j, a_j\}$, for $j = 1,2,\ldots,n$, where: v_j is the component of the vector to be permuted, while a_j is used to control which input arrives at which output specifically. In the first layer of the network a_j is a $(-1+2log_2 n)$-bit word. Each layer "consumes" one bit (the least significant), such that at each input of the last layer a_j we find a single bit. An example of an 8-input Beneš-Waksman permutation network is given in Section 3.1.

The elementary Permute cell has two inputs and two outputs and is comprised of two multiplexers. These multiplexer are connected such that the selection bit (least significant bit of a_j) decides if the outputs are identical to the inputs or if the inputs are switched.

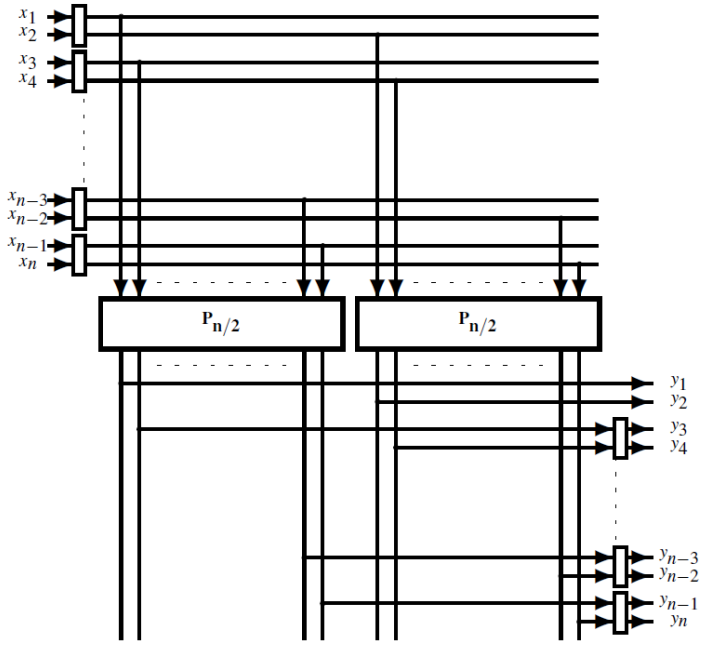

Fig. 2. The recursive definition of the n-input permutation network, P_n, introduced by Václav E. Beneš and optimized by Abraham Waksman.

The size and depth of a n-input permutation net are:

$$S_{perm}(n) = S_{perm}(2) \times (1 - n + nlog_2 n) \in O(nlog\,n)$$

$$D_{perm}(n) = D_{perm}(2) \times (-1 + 2log_2 n) \in O(log\,n).$$

Thus, both size and depth are kept within acceptable limits to be realized as circuits.

The current use of the Beneš-Waksman net is only for permutation. The cell is a simple switching circuit, further called `permute`, which can be programmed, with one bit, p_i, to perform the following function:

$$\{y_{2i}, y_{2i-1}\} = p_i \; ? \; \{x_{2i-1}, x_{2i}\} : \{x_{2i}, x_{2i-1}\}.$$

2.2.2. Pack function

In many vectorial computations we are faced with the occurrence of values of no interest distributed along a vector. The pack function could be considered a dynamically configured permutation where each component of the input vector X is accompanied by a predicate indicating if it is of interest or not. All the components of no interest are discarded to the right side of the vector, while the useful components are maintained in the left part of the vector in order of appearance.

Its recursive definition closely follows that of the Permute function (Fig. 2), with two main differences: (1) the inputs and outputs it has and (2) the last $(-1 + log_2 n)$ layers are `dummy` cells with $\{y_{2i}, y_{2i-1}\} = \{x_{2i}, x_{2i-1}\}$. On the first $log_2 n$ levels the `pack` cells receive:

$$x_j = \{v_j, enable, destination_j\}$$

If $enable = 1$, then $destination_j$ is used in computing each cell's permutation according to the position of v_j in the output vector (one bit per cell). The outputs of each cell is $y_j = \{v_j, enable, destination_j\}$ such that in the first $log_2 n$ layers the "destination" field is consumed.

2.2.3. Prefix functions

Prefix functions are a subcategory of scan functions whose outputs at each position can be seen as a reduction operation performed on all previous values. Applications will be discussed in Section 7. The proposed applications will require two types of prefix functions, one over the data vector (using the proposed network) and one over the Boolean/activation vector (using a separate or-prefix network).

Sum prefixes function is defined on n-component vectors, $X = [x_1, \ldots, x_n]$, and and provides $Y = [y_1, \ldots, y_n]$ where: $y_1 = x_1$, $y_i = y_{i-1} + x_i$, for $i = 2, \ldots, n$. An optimal recursive definition [12] for implementing the associated circuit is represented in Fig. 3, where the function performed by the elementary cell is:

$$\{y_{2i}, y_{2i-1}\} = \{(x_{2i-1} + x_{2i}), x_{2i-1}\}$$

for $i = 1, \ldots, n/2$.

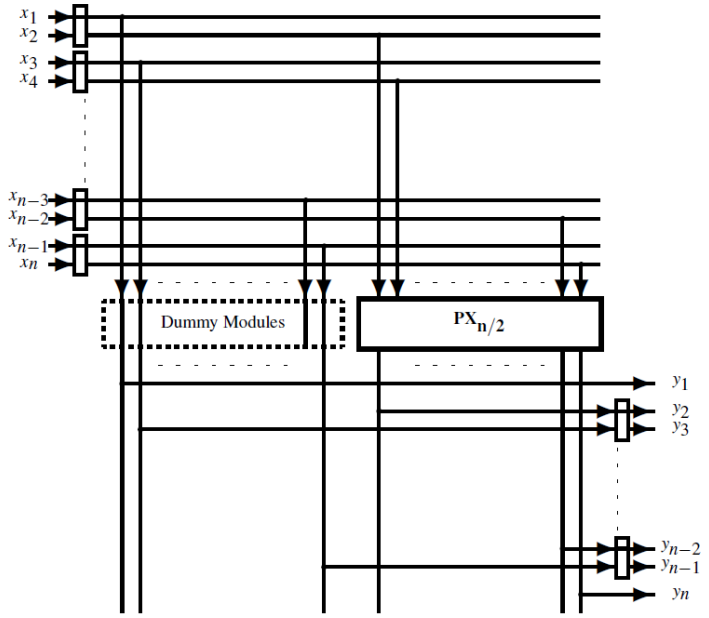

Fig. 3. The recursive definition of the n-input prefix network, PX_n.

2.3. Reduction functions

An operation applied over all values in the input vector resulting in a scalar value (or conversely, the right most output of a prefix function) represents the reduction function. Functions of interest contain: add ($y_n = \sum_1^n x_i$), min ($y_n = MIN_1^n x_i$) and max ($y_n = MAX_1^n x_i$).

3. *Log*-Depth Pipelined Version

The implementation of all scan functions, mentioned in the previous section, on the recursive shape of the permutation net requires some work for redefining network cells.

3.1. Permute network: the generic structure

For the particular case of $n = 8$ the permutation network is exemplified in Fig. 4. According to the definition provided in Fig. 2, three cells are dummy cells. They are simply registers to keep the pipe stages aligned.

3.2. Pack on the permute shape

For the pack function the network is used to perform a permutation whose configuration is specified by $\{enable, destination\}$. The *destination* label for each input is computed using the prefix add scan function applied to the Boolean activation vector. The resulting vector will contain the enabled scalars in positions specified by destination (a $(log_2 n)$-bit number).

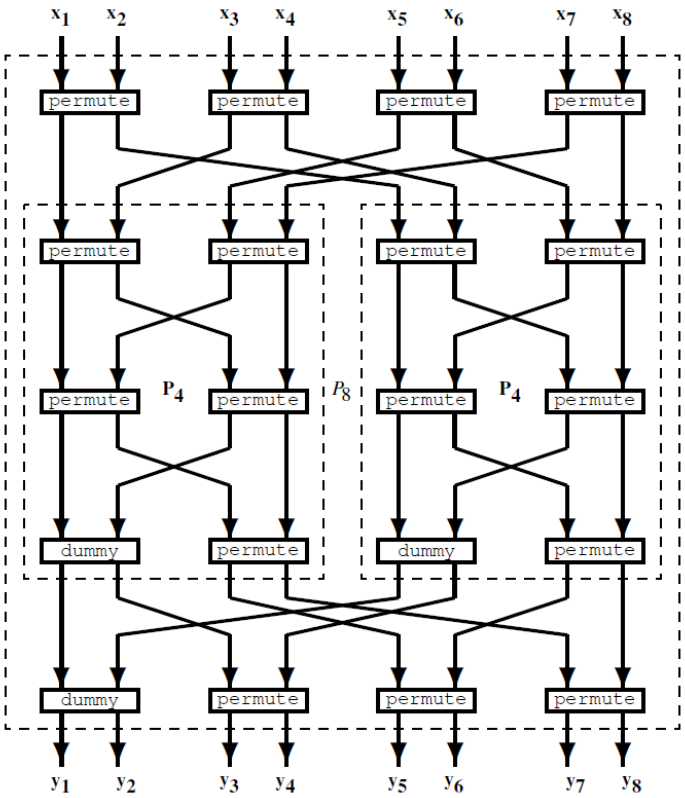

Fig. 4. The generic structure of the permute network for $n = 8$.

Subsection 7.3. exemplifies this concept in the context of the Pooling application. According to this information, each cell from the first $log_2 n$ stages of the net, make the following decision:

1. if both inputs are not enabled, then the scalars are propagated as they are.
2. if only one input is enabled, then the least significant bit of the enabled input decides the switch.
3. if both inputs are enabled, then the destination's least significant bit from the first input is used to decide the switch.

After each stage, *destination*s are shifted one position to right. The last $(-1 + log_2 n)$ stages of the net consist of `dummy` cells, each letting its inputs pass unswitched (as seen in Fig. 5).

3.3. Prefix sum implementation on the permute shape

The generic permute network, delivers at each lower half cell's input two numbers: a partial prefix that is greater than the actual needed value for that position and one of the original input numbers. These numbers are then subtracted in order to obtain the final values. The configuration

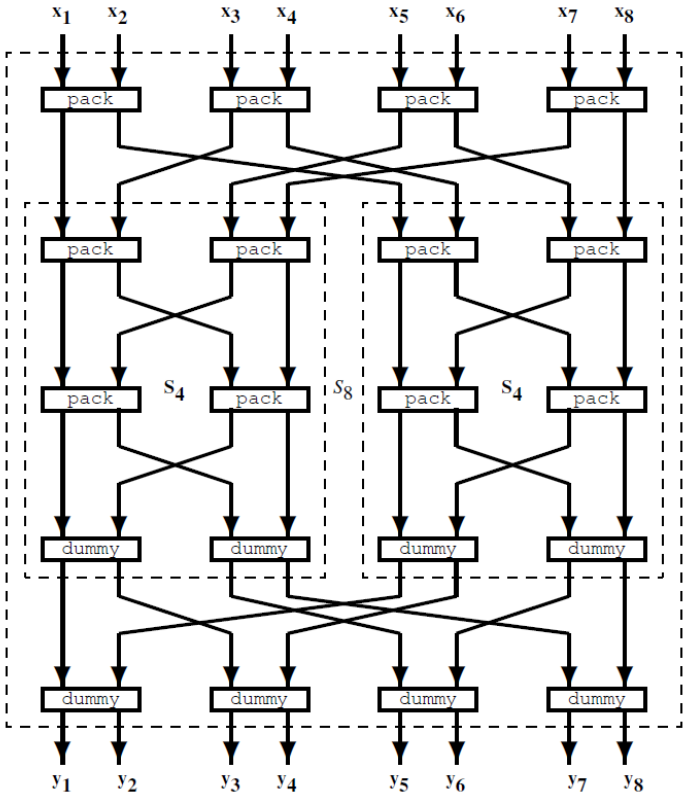

Fig. 5. Pack configuration.

seen in Fig. 6 emerges, containing 3 types of cells (in addition to dummy cells that propagate the input as is), labeled as follows:

- $[1 \quad 1+2] : \{y_1, y_2\} <= \{x_1, x_1+x_2\}$
- $[2 \quad 1+2] : \{y_1, y_2\} <= \{x_2, x_1+x_2\}$
- $[2-1 \quad 2] : \{y_1, y_2\} <= \{x_2-x_1, x_2\}$

Similarly, other 2 input variable, associative functions that support the inverse operation can be implemented using this network shape, although circuits may not be the most efficient implementation for all possible functions.

3.4. Reduction on the permute shape

The result of the reduction function can be taken over from the right most output of the $(log_2 n)$-th stage of the net.

Cell's functions for the reduction operation are represented in Fig. 7. The main computing resources needed for the three currently used reduction functions are an adder/subtracter and

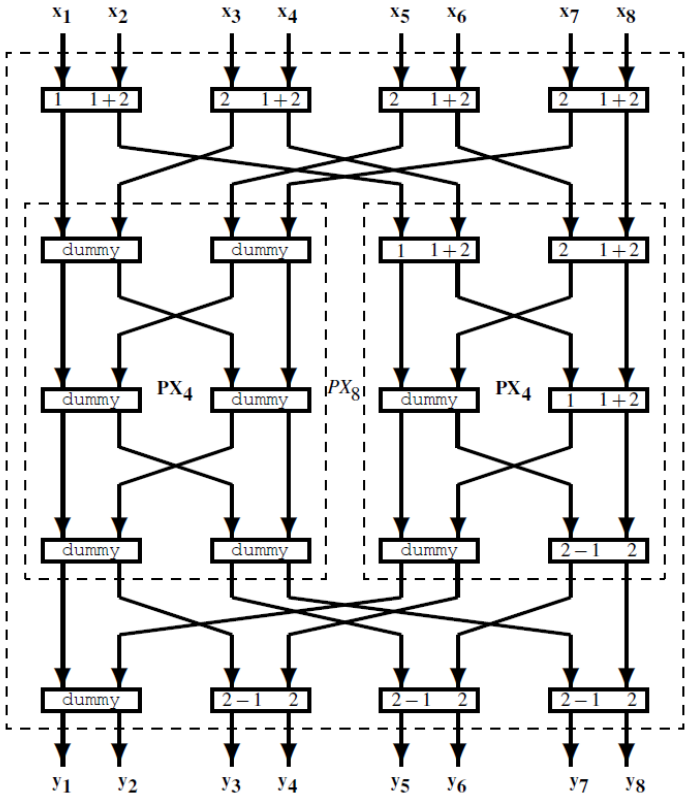

Fig. 6. Adapting the permute shape to sum prefix computation.

multiplexers to select the right or left output for the comparison functions min and max, or the sum for the add function.

Only $n-1$ cells are involved in computing the reduction functions, all the other cells, marked with `don't care` in Fig. 7, are not involved in this computation.

4. The Multi-Function Cell

The multifunctional elementary cell has, in addition to the connections specified above, connections that allow the propagation of the executed function's opcode. Thus, the input and output of each elementary cell (with position i,j) used to configure an n-input network contains three fields, as follows:

$$in_{ij} = \{x_{2i-1}, x_{2i}, funcIn\}$$
$$out_{ij} = \{y_{2i-1}, y_{2i}, funcOut\}$$

for $i = 1,\ldots,n/2$ and $j = 1,\ldots,(-1+2log_2 n)$, where:

$$x_{2i} = \{dataIn_{even}, enableIn_{even}, destIn_{even}\};\ \ x_{2i-1} = \{dataIn_{odd}, enableIn_{odd}, destIn_{odd}\}$$

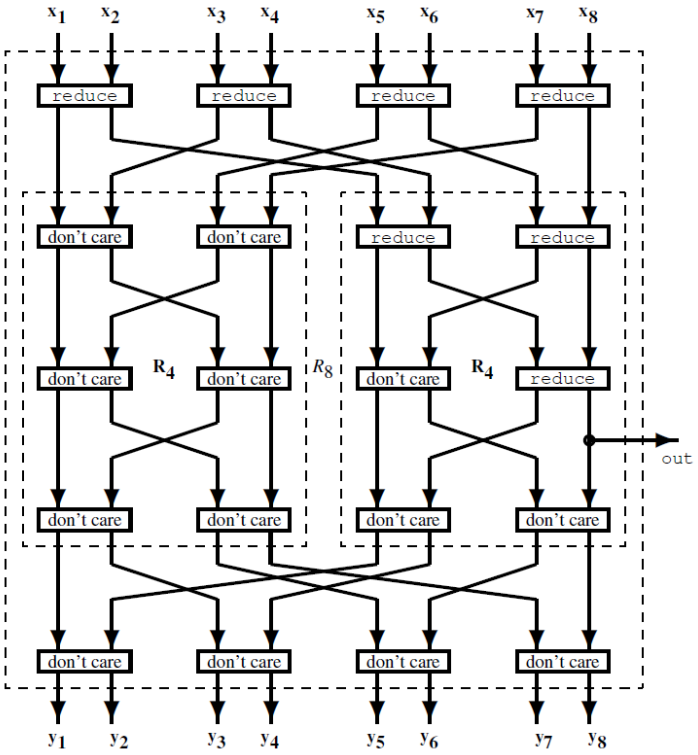

Fig. 7. Reduction configuration.

$$y_{2i} = \{out_{even}, enableOut_{even}, dout_{even}\};\ \ y_{2i-1} = \{out_{odd}, enableOut_{odd}, dout_{odd}\}$$

where:

- $dataIn_{even}$, $dataIn_{odd}$, $dataOut_{even}$ and $dataOut_{odd}$ are scalars
- $enableIn_{even}$, $enableIn_{odd}$, $enableOut_{even}$ and $enableOut_{odd}$ are enable bits controlling whether that input is processed using the selected function
- $destIn_{even}$, $destIn_{odd}$, $destOut_{even}$ and $destOut_{odd}$ are the destination pointers "consumed" bit by bit in each stage of the net, as follows:

$$destOut_{even} \Leftarrow destIn_{even}/2;\ \ destOut_{odd} \Leftarrow destIn_{odd}/2$$

The structure of the cell represented in Fig. 8 is the maximal version. The indexes i and j are constant synthesis parameters used to identify the position of the cell in the net. At the synthesis stage each cell can be synthesized according to its position in the net specified by the pair $\{i, j\}$ used to specify the line and the column in the net. The DECODE circuit takes into account the code for the function, *funcIn*, the enable bits and only the least significant bit of the destination codes. There is a weak dependency of the cell size on the number of inputs n as some cell inputs depend logarithmically on n. Cell size also depends linearly on the number of bits used for the n scalars submitted to the net.

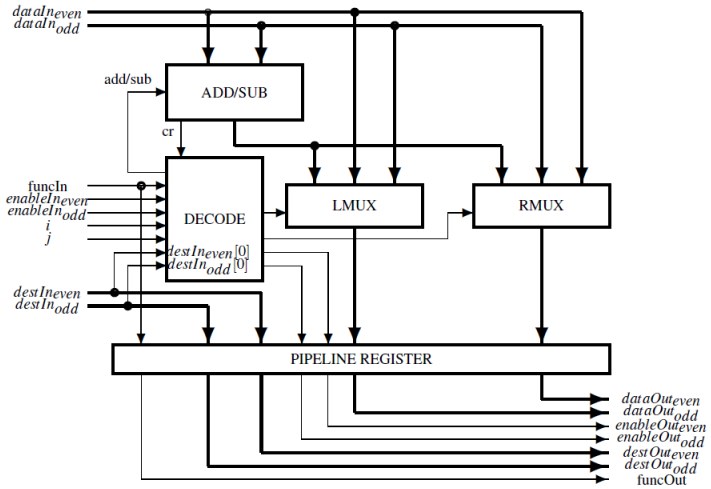

Fig. 8. The multi-function cell.

The types of cells, in descending order of complexity, are:

1. $(n-1)$ reduction cells: adder/subtracter, one 2-input multiplexer, one 3-input multiplexer, logic, pipeline registers
2. $(n-1-log_2 n)$ subtract cells: subtracter, one 2-input multiplexer, one 3-input multiplexer, logic, pipeline registers
3. $(1-n+0.5nlog_2 n)$ pack cells: two 2-input multiplexers, logic, pipeline registers
4. $(-2n+0.5nlog_2 n+log_2 n+2)$ permute cells: two 2-input multiplexers, pipeline registers
5. $(0.5n-1)$ dummy cells: pipeline registers

Based on this position dependant synthesis, Fig. 9 emerges, for an 8-input network.

5. *Log*-Step Sequential Version

The sequential version is made up of an one-dimensional array, of $n/2$ fully equipped cells, loop connected with multiplexers (see Fig. 10 for an 8-input example). The index j is now no longer a wired constant associated to each cell, but is instead used to specify the current stage for the $n/2$ fully equipped cells. Based on j, each multiplexer selects the proper input. The resulted circuit geometry connects multiple multiplexer inputs to the same data wire and as such, further size optimizations can be done. The *func* input is kept constant during the $(-1+2log_2 n)$ or $log_2 n$ (for reductions) cycles imposed by the computation of the function it specifies.

Execution time remains in $O(log\, n)$ but with no opportunity to involve this circuit in pipeline execution. The advantage we obtain is in reducing the size of the circuit to $O(n)$. With this method, depending on application needs, one can trade physical space for execution time or vice-versa.

Depending on the weight of the scan functions involved in the algorithms to be run and on throughput and size constraints, the pipelined or the sequential solution will be adopted.

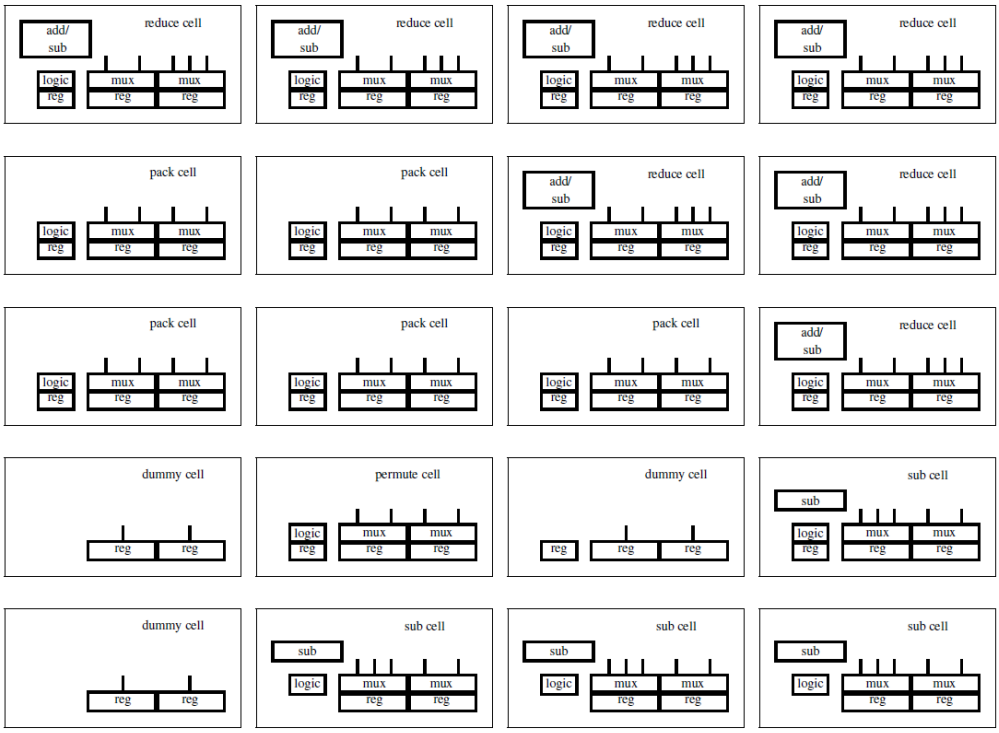

Fig. 9. The distribution of cells according to their type.

6. Simulation and Synthesis Results

This network was tested using a basic DUT vs Golden Model approach written in SystemVerilog. Randomly generated input data was fed to both the DUT and the Golden Model and their outputs were compared. All tests were passed for different sized networks, proving design functionality.

This network was synthesised targeting a ZYNQ7020 FPGA device, using a data size of 32 bits. Synthesis results are presented in Tables 1, 2 and 3. Table 3 shows the relative increase in the number of LUTs and REGs between the two networks: a standard Beneš-Waksman and a Multi-Function circuit.

No. Inputs	Number of Cells	LUTs	REGs	Average LUTs/cell	Average REGs/cell
8	20	592	1390	30	70
16	56	1867	3693	33	66
32	144	4911	9689	34	67
64	352	12291	24277	35	69
128	832	29715	58805	36	71
256	1920	70051	138885	36	72

Table 1. Network synthesis results: Permute-Only

The Multi-Function circuit scales slightly steeper that the Permute only network because of

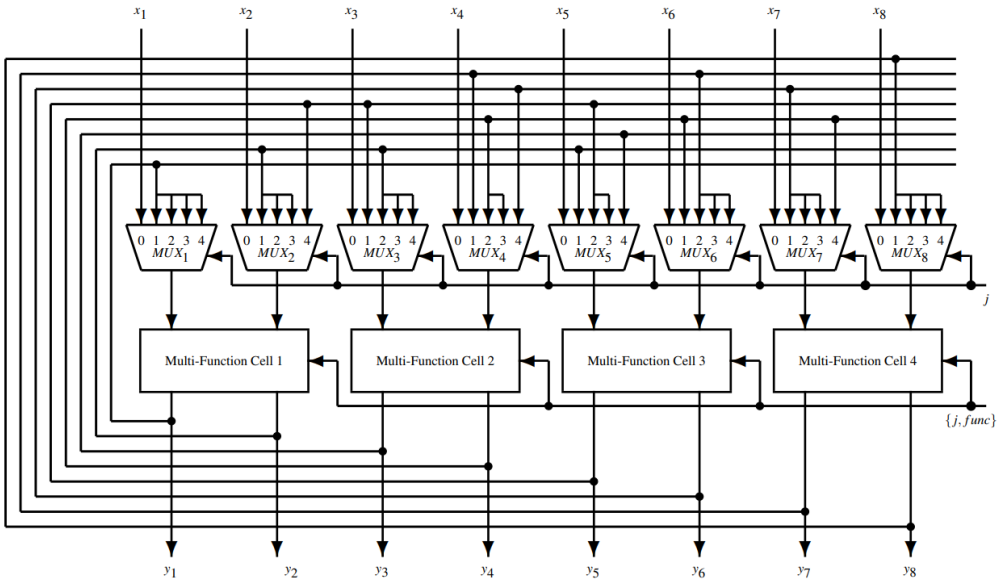

Fig. 10. The sequential version for 8 inputs.

No. Inputs	Number of Cells	LUTs	REGs	Average LUTs/cell	Average REGs/cell
8	20	1240	1459	62	73
16	56	3813	3921	68	70
32	144	10208	10368	71	72
64	352	25894	26252	74	75
128	832	63389	64146	76	77
256	1920	151180	152627	79	79

Table 2. Network synthesis results: Multi-Function

No. Inputs	% Increase in LUTs	% Increase in REGs
8	109.4	4.9
16	104.2	5.8
32	107.8	6.5
64	110.7	7.5
128	113.3	8.3
256	115.8	9.0

Table 3. Network synthesis results: Multi-Function size increase compared to permute-only Beneš-Waksman netowrk

the additional resources needed by the control circuitry.

7. Examples of Applications

For the next examples, we will consider this following additional resource, distributed along the cells of the system represented in Fig. 1 : Boolean vector, $B = [b_1, \ldots, b_n]$, used to specify the state of each cell. For $b_i = 1$ the cell C_i is considered active, else the cell is inactive. The elements of the vector B are used to enable the components of the data vector sent to the SCAN circuit. Data sent by a cell is considered enabled if the cell is active. Another vector involved in the following examples is $IX = [1, 2, \ldots, n]$ used to index cells in the MAP array.

7.1. Select function

The select function returns the i-th element from a vector V: $sel(i, V)$.

```
/*****************************************************************
The select scan function
Initial:
    V = [S1 S2 ... Sn] // n-component vector submitted to selection
    IX                 // desired location index
    i                  // the index of the searched element in V
Final: out = Si        // the i-th component of V
*****************************************************************/
where(IX = i)  // only the cell where IX=i remains active: bi = 1
SCAN(V, B, reduction add) // the scan net takes V for reduction add
wait(log_2 n) // due to the latency of the SCAN net to reduce out
out = REDUCE // REDUCE output is Si because only cell i is active
```

7.2. Matrix-vector multiplication

Matrix-vector multiplication is one of the most frequently accelerated computation in linear algebra applications. In this operation, the reduction net is involved with the function reduction add used for computing the inner product between each line of the matrix and the vector. While the first stage of the inner product operation, the scalar multiplication distributed along the MAP cells (Fig. 1), is obviously executed efficiently (it benefits of an acceleration in $O(n)$), the second stage, the addition, is performed in $O(log\, n)$ time by the reduction net, thus reducing the overall efficiency. The acceleration obtained waiting for the propagation through the reduction net for the reduction add function is in $O(n/log\, n)$.

One solution for avoiding the effect of latency introduced by the reduction network is to use a queue memory distributed along the MAP array, each cell harboring one level, starting with the left-most cell [13]. In each cycle the result of multiplying one matrix line with the vector is submitted to the reduction net, then, after $log_2 n$ cycles, the first inner product is generated at the output REDUCE (see Fig. 1) and it can be pushed in the queue. The process of multiplication continues n cycles. For the last inner product we must wait another $log_2 n$ cycles to do the last push in order to complete the result in the queue. The resulting execution time is $2n + log_2 n + const$ and the acceleration is in $O(n)$. Simulations provided $6n\times$ acceleration.

7.3. The pooling stage of convolutional stage of a CDNN

The pooling process receives as input a $m \times n$ matrix and provides as output a $m/p \times n/p$ matrix substituting each $p \times p$ sub-matrix with a value (sum of elements or maximal value). From a column point of view, this can easily be done at the map-array cell level. Compacting lines is done using the pack function. Prefix add is used to compute the position of each element maintained in the final vectors. The following algorithm describes the process for each line:

```
/*****************************************************************
Pack a vector according to a boolean vector
Initial:
        V = [S1 S2 ... Sn] // vector to pack
        B = [B1 B2 ... Bn] // Bi=1 selects Si to be left aligned
Final:
        S = [x1 x2 ... xq y1 y2 ... yr] with q + r = n
        xi, for i = 1,...,q, are the selected values
*****************************************************************/
SCAN(B, B, prefix add) // compute the positions of the active cels
wait(-1+2log_2 n) // provides a delay due to net's latency
DEST <= SCANout // the destinations for Si are loaded
DEST <= DEST - 1
SCAN(V, B, DEST, scan pack) // send V for pack to DEST
wait(-2+2log_2 n) // provides a delay due to net's latency
activate // activate all cells
S <= SCANout
```

For example:

```
V    = [3 6 1 8 3 5 6 3 2 6 7 4 9 3 5 2]
B    = [0 0 0 1 0 1 1 1 0 0 0 0 1 0 1 0]
DEST = [x x x 0 x 1 2 3 x x x x 4 x 5 x]
S    = [8 5 6 3 9 5 x x x x x x x x x x]
```

In the experimental environment, for n cells, the execution time is:

$$t_{pack}(m) = 2 \times 2log_2 n + const \in O(log_2 n)$$

Acceleration belongs to $O(log\ n)$. In the case of pooling multiple vectors, stored vertically (one after another) in each cell, with a proper algorithmical approach, the logarithm can be absorbed resulting in execution time proportional to the number of vectors treated. Acceleration now becomes $O(n)$.

7.4. Matrix transpose

Considering the matrix **V** of components v(i,j), whose scalar components must be transposed into the matrix **W** of scalar components w(i,j), for $i, j = 0, 1, \ldots, n-1$. We consider the following transpose algorithm:

```
/*****************************************************************
Algorithm for matrix transpose.
    Initial: v(i,j), for i,j = 0, 1, ... n-1
    Final:   w(i,j), for i,j = 0, 1, ... n-1
        vector: A = [a[0] a[1] ... a[n-1]]
    Add and sub operations are modulo n
*****************************************************************/
for(k=0; k<m; k=k+1) begin
 (1) A <= [v(0-k,0) v[1-k,1]...v[n-1-k,n-1]
 (2) A <= [a(0-k) a(1-k) ... a(n-1-k)]
 (3) for(j=0; j<n; j=j+1) w(j+k,j) <= a(j)
end
```

The steps (1) and (3) in the main loop of the previous algorithm are performed in the MAP section of the system (see Fig. 1). For the second step, the permutation function of our SCAN circuit is used. The previous algorithm is developed for a $n \times n$ matrix with n being the number of map-array cells, but can be relatively easily adapted to computing multiple smaller square matrices at a time. For better understanding of this algorithm, let us consider the small example of the transpose of a 4×4 matrix:

```
     Initial
                          step 1:  step 2:  step 3:  step 4:
V:   0 1 2 3
     4 5 6 7      (1) 0 5 A F  C 1 6 B  8 D 2 7  4 9 E 3
     8 9 A B
     C D E F      (2) 0 5 A F  1 6 B C  2 7 8 D  3 4 9 E

W:   x x x x      (3) 0 x x x  0 x x C  0 x 8 C  0 4 8 C
     x x x x          x 5 x x  1 5 x x  1 5 x D  1 5 9 D
     x x x x          x x A x  x 6 A x  2 6 A x  2 6 A E
     x x x x          x x x F  x x B F  x 7 B F  3 7 B F
```

The execution time on a system with n cells in the MAP section (see Fig. 1) belongs to $O(n)$ and acceleration belongs to $O(n)$.

7.5. FFT

Depending on the number of samples for which the FFT is calculated, our system can be used in several ways. For small values, all samples associated with a FFT can be loaded and processed into one cell. In this case the "butterfly" interconnections assumed by the algorithm do not affect the performance. In the case of a larger number of samples, when samples from each FFT will use memory belonging to several cells, cells will have to communicate with each other affecting performance. In this case, our permutation network will prove useful.

For the sake of simplicity, let us consider FFTs with the number of samples $p^2 = n$. Thus, a n-cell MAP will store p values in each of its p vectors. For this situation, there are two solutions:

7.5.1. Version 1: uses permute to exchange data between cells

The FFT calculation is done in two stages. In the first stage, the large distance "butterflies" are solved. Data involved in multiplications and additions performed in this stage are stored in the columns of the matrix that are stored in the same cell and thus calculation does not involve communication between cells in MAP. In the second stage small distance "butterflies" are solved. This time data is distributed in different cells at distances of maximum $n/2$ cells. Communication between cells is achieved through the permutation function. For this case, $log_2 n$ permutations are defined. Unfortunately, latency cannot be avoided since a "butterfly" cannot be calculated until the previous one has finished being processed.

7.5.2. Version 2: uses permute to perform the transpose of the matrix

The second version involves three stages. The first stage is identical to the previous version. In the second stage, the $p \times p$-component matrices are transposed, so that data involved in the small "butterflies" will be in the same cells, without the need for further inter-cellular communication. After the transpose is complete, data is properly arranged such that the third stage will be similar to the first. In this case, an acceleration of $\sim 1.5 \times n$ is achieved.

8. Concluding Remarks

In the context of frequently used patterns for many-core parallel computation, we have developed a multi-function programmable network that offers circuit support for multiple parallel patterns.

The novelty of our approach consists of adding additional functions to the consecrated shape of the Beneš-Waksman permutation network:

- the implementation of the prefix scan function for addition
- the implementation of a dynamically configured permutation for the pack function
- the implementation of reduction type functions for addition, min, max

These functions are used to enhance the computing capabilities of a MAP type array, as seen in Fig. 1. These new resources can be used in a wide range of algorithms as seen in Section 7.

The multi-function permutation circuit has the main advantage of efficiently implementing some of the most frequently used scan-type and reduction type functions on the same circuit. In this circuit, the resulting cells are more complex than the simplest permute-only cell for $(2n - 2 - log_2\, n)$ cells out of the total $n(-1 + 2log_2\, n)/2$ number of cells. Roughly 20% of the cells are fully equipped. The resulting multi-function network is roughly double the size of Beneš-Waksman permutation-only network . If size constraints are of vital importance and time is not, a sequential version of the network can also be chosen and deployed.

References

[1] M. McCOOL, A. D. ROBINSON and J. REINDERS, *Structured Parallel Programming. Patterns for Efficient Computation*, Morgan Kaufman, 2012.

[2] V. E. BENEŠ, *Optimal Rearrangeable Multistage Connecting Networks*, Bell System Technical Journal, Vol 43, No 4, pp 1646-1656, 1964.

[3] V. E. BENEŠ, *MathemaTIcal Theory of connecting networks and Telephone Traffic*, Academic Press New York, 1965.

[4] A. WAKSMAN, *A Permutation Network*, Journal of the ACM, v. 15. no. 1, pp. 159-163, 1968.

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0" June 2021*. Accessed June 15, 2023 [online]. Available: `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`,

[6] S. SENGUPTA, M, HARRIS, Y, ZHANG and J. D. OWENS, *Scan Primitives for GPU Computing*, Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, San Diego, California, pp. 97–106, 2007.

[7] D. NASSIMI and S. SAHNI, *A Self-Routing Benes Network and Parallel Permutation Algorithms*, IEEE Transactions on Computers, C-30(5), pp. 332-340, 1981.

[8] M, BADER, H.J. BUNGARTZ, D. MUDIGERE, S. NARASIMHAN and B. NARAYANAN, *Fast GPGPU Data Rearrangement Kernels using CUDA*, arXiv:1011.3583 [cs.DC], 2010.

[9] M. HARRIS, *Optimizing Parallel Reduction in CUDA*, [Online]. Available: `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

[10] V. DRAGOMIR and G. M. STEFAN, *All-Pair Shortest Path on a Hybrid Map-Reduce Based Architecture*, Proceeding of the Romanian Academy, Series A 20(4), pp. 411-417, 2019.

[11] W. J. DALLY and B. TOWLES, *Principles and practices of interconnection networks*, Elsevier, Morgan Kaufmann Publishers, 2004.

[12] R. E. LADNER and M. J. FISCHER, *Parallel prefix computation*, Journal of the ACM, Volume 27(4), pp 831–838, Oct. 1980.

[13] M MALITA, G.V. POPESCU and G. M. STEFAN, *Pseudo-Reconfigurable Heterogeneous Solution for Accelerating Spectral Clustering*, Proceedings of 2020 IEEE International Conference on Big Data, Atlanta, GA, USA, pp. 5138–5145, 2020.