OPTIMIZING INTERCONNECTION COMPLEXITY FOR REALIZING FIXED PERMUTATION IN DATA AND SIGNAL PROCESSING ALGORITHMS*

Ren Chen and Viktor K. Prasanna

Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, USA 90089
{renchen, prasanna}@usc.edu

ABSTRACT

In hardware implementation of several widely used data and signal processing algorithms, data permutations need to be performed between the consecutive computation stages consisting of parallel computational units. Recently, some highly data parallel streaming architectures for data permutation have been proposed to achieve high throughput. However, the interconnection complexity of these designs increases dramatically with the problem size and data parallelism. In this paper, we develop a hardware structure to perform data permutation with optimized interconnection complexity, defined as the interconnection area per throughput. We propose a novel design technique such that the required interconnection logic is highly reduced for realizing a fixed permutation on streaming data. Our experimental results show that the proposed design technique reduces interconnection complexity by 27.3% to 75.8%, and improves the throughput by 5.3%∼129% and the energy efficiency by 1.2x∼3.5x compared with the state-of-the-art.

1. INTRODUCTION

Intermediate data need to be permuted between subsequent computational stages in several widely used data or signal processing algorithms such as FFT, Viterbi coding, Bitonic sorting network, etc [1, 2, 3, 4, 5]. Efficient hardware solutions for performing data permutations are important for parallel hardware implementations of these data intensive algorithms. Data permutation can be simply realized by hardware wires if all the data inputs are available concurrently. However, such an approach is not desirable for large input size due to high routing area and complexity.

Previous hardware implementations for data permutation can be classified into memory-based designs and register-based designs [6]. In the traditional VLSI implementation for FFT, register-based delay feedback and delay commutator are proposed to perform stride permutation or bit reversal permutation [3, 7]. In these designs, a single input data sequence is broken into several parallel data streams flowing forward with proper delays. High computational performance per unit area is achieved by using such a pipelined design. However, throughput is limited by data rate of the single input [3, 7]. In [6], memory-based and register-based stride permutation networks for array processors are presented. These networks support any power-of-two stride and achieve high area efficiency. To realize a specific family of permutations called bit index permutations, a memory-based 3-stage structure is developed in [8]. These approaches only provide design methods for a particular class of permutations. In [9], the authors develop a memory-based technique that automatically generates a streaming datapath for arbitrary fixed permutation. Nevertheless, such a design consumes a lot of memories.

In this paper, we develop a hardware design for data permutation with optimized interconnection complexity, defined as the interconnection area per throughput in this paper. Our design is based on the idea of our preliminary work [10] on sorting by vertically folding a static network topology obtained by configuring a multi-stage Benes network [11] for realizing permutations in sorting networks. Similar to the designs proposed in [8, 9], our proposed design supports processing parallel data inputs with a fixed data parallelism. Based on a memory-based datapath structure, our design is constructed with optimized interconnection complexity, thus more practical for actual implementation. Specifically, our main contributions are:

- A memory-based hardware design for data permutation parameterizable with respect to data parallelism, problem size, and data width.
- A mapping approach to obtain a design with minimum interconnection area to perform an arbitrary fixed permutation compared with the state-of-the-arts.
- A heuristic algorithm for datapath and control unit generation, such that the interconnection area can be minimized in our design.
- Post place-and-route results on FPGA show that our design reduces interconnection complexity by 27.3%

*This work has been funded by US NSF under grants CCF-1320211 and ACI-1339756. Equipment grant from Xilinx, Inc. is gratefully acknowledged.

to 75.8% and improves throughput by 5.3%∼129% compared with the state-of-the-art designs.

2. BACKGROUND AND RELATED WORK

2.1. Classic Permutation Patterns

Permutations can be represented using permutation matrices. A *permutation matrix* denoted as P_n is an $n \times n$ matrix with all elements either 0 or 1, with exactly one at each row and column. For example, a stride permutation can be defined using matrix representation. Given an n-element data vector x and a stride d $(0 \leq d \leq n-1)$, the data vector y which is produced by the stride permutation over x can be given as $y = P_{n,d}x$, where $P_{n,d}$ is a permutation matrix. $P_{n,d}$ is an $n \times n$ bit matrix such that

$$P_{n,d}[i][j] = \begin{cases} 1 & \text{if } j = (d \times i) \bmod n + (\lfloor d \times i/n \rfloor); \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

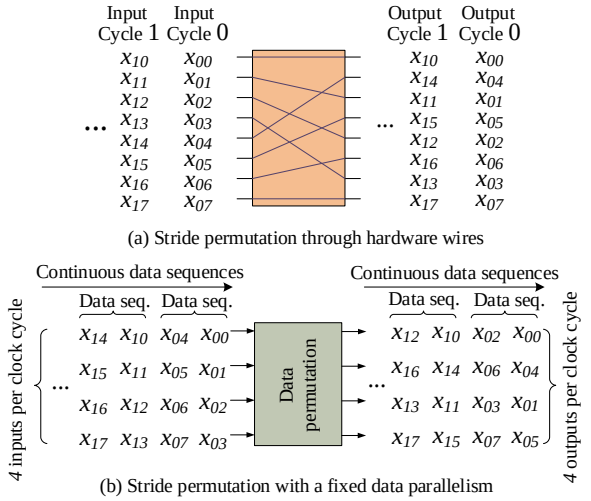
where mod is the modulus operation and $\lfloor\ \rfloor$ is the floor function. For example, a stride-by-2 permutation over a data vector of four data elements can be represented as:

$$\begin{pmatrix} y_0 & y_1 & y_2 & y_3 \end{pmatrix}^T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \end{pmatrix}^T$$

Another permutation pattern well known as bit-index permutation was studied in [8]. In this permutation, given a data set 0,1,...,n-1, the element at index i is swapped with the element at index $j = i$ XOR k, k is a given parameter. n must be divisible by the smallest power of 2 which is greater than k. These classic permutation patterns are widely used in data and signal processing algorithms [8, 10, 12].

2.2. Related Work

Existing works on hardware implementation of data permutation in the literature are mostly focused on a specific family of permutations such as stride permutations, bit-reversal, and bit-index permutations [1, 2, 9, 13]. In [7], delay commutators are proposed to perform stride permutation in the pipelined design for FFT. The folded FFT architecture achieves high computational performance per unit area. In [1], streaming permutation structures for a subset of stride permutations are proposed to obtain high performance hardware implementations for the fast Fourier transform. In [6], a design approach to achieve streaming implementations of stride permutations is developed. Memory-based permutation networks are considered to realize stride permutations for large problem sizes. Their proposed design achieves high data parallelism, however, requiring a large amount of independent dual-port memory blocks. By expanding this work, a streaming structure is proposed to perform any fixed permutation which is a linear mapping on the bit representation of the data locations [8]. Their design supports problem size



(a) Stride permutation through hardware wires

(b) Stride permutation with a fixed data parallelism

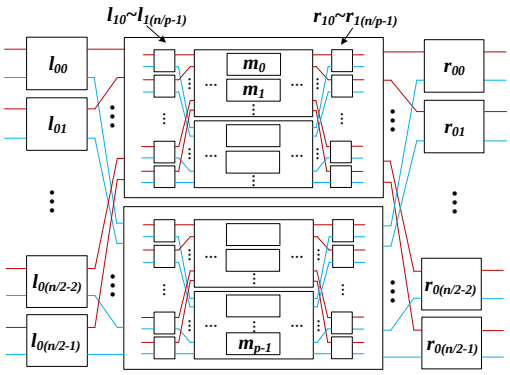
Fig. 1: Data permutation with a fixed data parallelism

and data parallelism that are powers of two. An extension of this approach is presented in [9]. This design is capable of performing arbitrary fixed permutation for a given data parallelism. However, hardware implementation of their design is expensive with respect to memory. Compared with the design in [9], our proposed design supports arbitrary fixed permutation with half memory consumption, furthermore, we propose a design technique such that the interconnection logic can be highly reduced. Detailed experimental results will be introduced in Section 5.

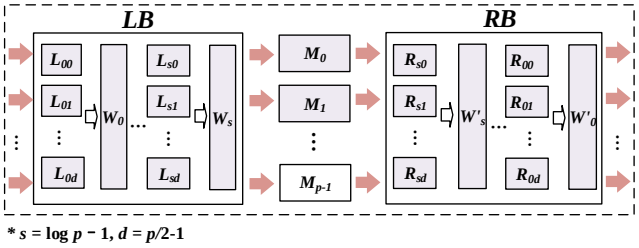
3. PROBLEM DEFINITION AND NOTATIONS

Designing a specialized hardware block for permuting streaming data is challenging as data elements need to moved across the temporal bound, i.e., a data element previously input in input cycle 0 needs to be output in output cycle 4. This feature requires register files or memories to be employed so that a data element can be output after a specific amount of delay. Fig. 1(a) shows an example of stride permutation with $n = 8, d = 4$. This permutation is simply implemented by a reordering of hardware wires. Fig. 1(b) shows an example of permutation on streaming data with a fixed data parallelism, where $n = 8$ and $p = 4$.

To formulate this problem, we define the ***streaming permutation*** as: to design a hardware structure with an available data parallelism of p $(1 \leq p \leq n)$, defined as the number of input data elements processed per cycle in parallel, such that an n-element data sequence enters the hardware structure over n/p consecutive cycles, after a specific amount of delay t, the sequence is reordered as specified by the given fixed permutation and output over n/p consecutive cycles. With continuous input data sequences, the throughput is p output elements per cycle and the latency is t. In this paper, without loss of generality, we assume that both p and n are powers of two.



(a) An n-to-n Benes Network $B_{n,p}$ decomposed $\log p$ times



(b) Generated datapath of our design denoted as streaming permutation network $SPN_{n,p}$

Fig. 2: An n-to-n Benes Network can be decomposed into two $n/2$-input Benes subnetworks, the Benes network in (a) has been recursively decomposed $\log p$ times, (b) shows the generated datapath by vertically folding the Benes network in (a).

4. DESIGN METHODOLOGY

4.1. Datapath Generation

Our proposed design for streaming permutation is constructed by utilizing the classic Benes network to generate its datapath logic. A Benes network is a multi-stage parallel routing network which is able to realize arbitrary fixed permutation between its input and output [11]. To illustrate our datapath generation idea, we have the following notations used in Figure 2:

- $A.P$: The permutation matrix representing the permutation performed by A (a switch, a switch box, an interconnection network, or an SPN).
- l_{ij}: The 2-input switch at the ith input stage of an n-input Benes network (see Fig. 2a), $i = 0, 1, ..., s$, $s = \log p - 1$, $j = 0, 1, ..., n/2 - 1$.
- r_{ij}: The 2-input switch at the stage lying i stages before the last stage of an n-input Benes network, $i = 0, 1, ..., s$, $j = 0, 1, ..., n/2 - 1$.
- m_i: The n/p-input subnetwork at the middle stage of an n-input Benes network, $i = 0, 1, ..., p - 1$.
- LB: The p-input switch box consisting of 2-input switches L_{ij} at the input stage of our design, $i = 0, 1, ..., s$, $j = 0, 1, ..., p - 1$.
- $W_i/W_i^{'}$: The p-to-p interconnection through wires, $i = 0, 1, ..., s$.
- RB: The p-input switch box consisting of 2-input switches R_{ij} at the output stage of our design, $i = 0, 1, ..., s$, $j = 0, 1, ..., p - 1$.
- M_i: The n/p-entry memory block at the middle stage of our design.

Given a one dimensional data vector V of size n, V enters/exits the $SPN_{n,p}$ in n/p cycles and the resulted two dimensional data vector is denoted as $V^{'}$. The dimensions of $V^{'}$ indicate spatial order and temporal order of a data element in $V^{'}$. We use $V^{'}$=stream(V) to denote this transformation. Let X/Y be the input/output vector of the Benes network $B_{n,p}$, then $Y = B_{n,p}.P \cdot X$. Assuming $X^{'}$ is the input of $SPN_{n,p}$ and $X^{'}$=stream(X) then we have:

Theorem 4.1. *For a given fixed p and any permutation $B_{n,p}.P$ performed by $B_{n,p}$, $SPN_{n,p}$ can be configured such that $Y^{'} = SPN_{n,p}.P \cdot X^{'}$ and $Y^{'}$=stream(Y) when $W_i.P = I_{2^i} \otimes P_{p/2^i,2}$ and $W_i^{'}.P = I_{2^i} \otimes P_{p/2^i,p/2^{i+1}}$ ($\log p - 1 \geq i \geq 0$)* [1].

Proof: For $p = 2$, $SPN_{n,2}$ is composed of L_{00}, R_{00}, M_0 and M_1; $W_0 = W_0^{'} = I_2$. $B_{n,2}$ has two sub-networks including m_0 and m_1. Let $Xa^T = PI \cdot X^T$, and

$$PI = \begin{pmatrix} l_{00}.P & & & \\ & l_{01}.P & & \\ & & ... & \\ & & & l_{0(n/2-1).P} \end{pmatrix} \quad (2)$$

Let $Xa = \{Xa_0, Xa_1, .., Xa_{n/2-1}\}$, where $Xa_i = (Xa[2i], Xa[2i+1])$. When feeding X into $SPN_{n,2}$ over $n/2$ cycles, configure L_{00} such that $L_{00}.P = l_{0j}.P$ in cycle j ($n/2 - 1 \geq j \geq 0$). Then the output of LB in cycle j will be Xa_i. Let $Xb = (Xa[0], Xa[2], ..., , Xa[n/2-2])$ and $Xc = (Xa[1], Xa[3], ..., , Xa[n/2-1])$ be the input vector of m_0 and m_1, respectively. To prove Theorem 4.1, we have the definition below:

Definition 1. *Let $Y = P \cdot X$, after writing an n-entry memory M such that $X[i]$ is stored in ith location ($n-1 \geq i \geq 0$) of M , if then reading M with data Y_k ($n-1 \geq k \geq 0$) in cycle k, we say the permutation P is performed by M on X temporally.*

As $W_0 = I_2$, in cycle j, the data written into M_0 will be $Xb[j]$, and the data written into M_1 will be $Xc[j]$. Let $Yb = m_0.P \cdot Xb$ and $Yc = m_1.P \cdot Xc$. Based on Definition 1, after writing M_0 with Xb, if reading M_0 with data $Yb[k]$ ($n/2 - 1 \geq k \geq 0$) in cycle k, the permutation $m_0.P$ can be performed by M_0 on Xb temporally. Similarly, the

[1] I_{2^i} is the identity matrix and $\otimes$ is the tensor (or Kronecker) product [14].

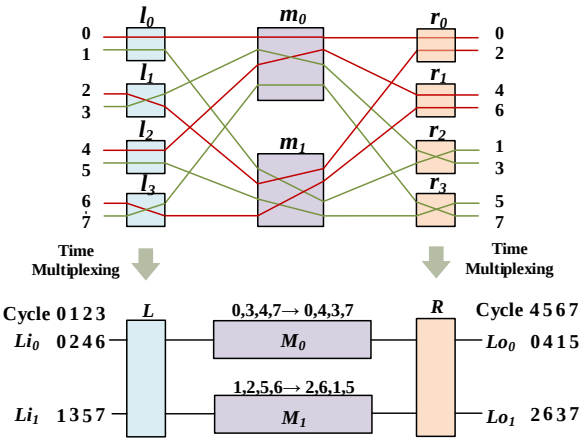


Fig. 3: An example of generating $SPN_{n,2}$

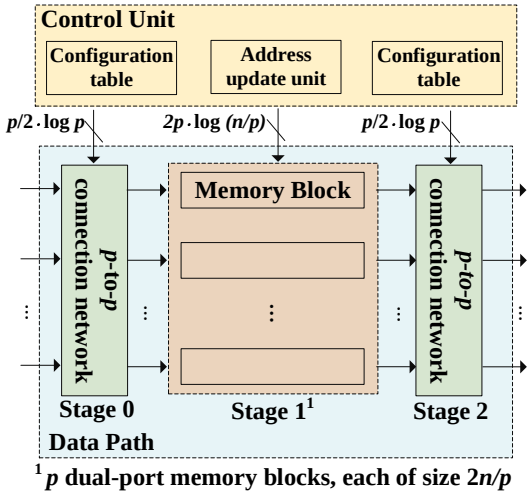


Fig. 4: Proposed permutation network

permutation $m_1.P$ can be performed by M_1 on Xc temporally. Thus, in jth $(n/2-1 \geq j \geq 0)$ cycle after RB starts to receive input data, the input vector of RB would be $(Yb[j],Yc[j])$ denoted as Ya_j which is also the input vector of the switch r_{0j}. Let $Ya = \{Ya_0, Ya_1, ..., Ya_{n/2-1}\}$ and

$$PO = \begin{pmatrix} r_{00}.P & & & \\ & r_{01}.P & & \\ & & ... & \\ & & & r_{0(n/2-1).P} \end{pmatrix} \quad (3)$$

then $Y^T = PO \cdot Ya^T$. Let $Y = \{Y_0, Y_1, .., Y_{n/2-1}\}$, where $Y_i = (Y[2i], Y[2i+1])$ $(n/2-1 \geq i \geq 0)$. When RB output data results in $n/2$ cycles, we can configure R_{00} such that $R_{00}.P = r_{0j}.P$ in cycle j $(n/2-1 \geq j \geq 0)$, thus the output data vector of RB in cycle j is Y_j. As a result, $Y' = \{Y_0, Y_1, .., Y_{n/2-1}\} = Y$. Thus, Theorem 4.1 is proved for $p = 2$. To prove Theorem 4.1 for any p, we have the lemma below:

Lemma 4.2. *For given fixed p and n, if $SPN_{n/2,p/2}$ can be configured to realize arbitrary $B_{n/2,p/2}.P$, then $SPN_{n,p}$ can also be configured to realize arbitrary $B_{n,p}.P$.*

Proof: Let m_0 and m_1 be the upper subnetworks in $B_{n,2}$. Then Benes network $B_{n/2,p/2}$ can be configured to perform $m_0.P$ or $m_1.P$. Based on the assumptions in Lemma 4.2, $SPN0_{n/2,p/2}$ and $SPN1_{n/2,p/2}$ can be configured to perform $m_0.P$ and $m_1.P$ on streaming data, respectively. Again, we can add the SPNs with switches including L_{0k} and R_{0k} $(p/2-1 \geq k \geq 0)$ to perform the permutation patterns of l_{0j} and r_{0j} in $B_{n,2}$. We still use PI/PO to denote the permutation performed by l_{0j}/r_{0j}, $Xa = PI \cdot X$, and $Y = PO \cdot Ya$. We can add wire interconnections $WI = P_{p,2}$ and $WO = P_{p,p/2}$ such that the output vector of L_{0k} to be the input of WI, and the output vector of WO to be the input of R_{0k}. Connect WI with $SPN0_{n/2,p/2}$ and $SPN1_{n/2,p/2}$ such that the first half output vector of WI to be the input of $SPN0_{n/2,p/2}$ and the second half output vector of WI to be the input of $SPN1_{n/2,p/2}$. Connect WO with $SPN0_{n/2,p/2}$ and $SPN1_{n/2,p/2}$ such that the first half input vector of WO to be the output of $SPN0_{n/2,p/2}$ and the second half input vector of WO to be the output of $SPN1_{n/2,p/2}$. As a result, in each cycle, $p/2$ data elements are fed into $SPN0_{n/2,p/2}$ or $SPN1_{n/2,p/2}$. Such a design consisting of L_{0k}, R_{0k}, WI, WO, $SPN0_{n/2,p/2}$ and $SPN1_{n/2,p/2}$ is able to realize any permutation $B_{n,p}.P$. Therefore, Lemma 4.2 is justified.

Based on the proof for Theorem 4.1 for $p = 2$ and Lemma 4.2, Theorem 4.1 holds for any n when given a fixed p. Without losing generality, we assume n and p to be powers of 2.

Lemma 4.3. *[11] An n-to-n Benes network $B_{n,p}$ can rearrangebly perform arbitrary fixed permutation.*

Theorem 4.4. *For a given fixed p, $SPN_{n,p}$ can be configured to realize any given permutation on streaming input of an n-input data vector without any memory conflicts using p memory blocks, each of size n/p.*

Proof: Based on Lemma 4.3, $B_{n,p}$ is able to perform any fixed permutation between its input and output. Based on Theorem 4.1, any permutation $B_{n,p}.P$ performed by $B_{n,p}$ can be realized by $SPN_{n,p}$ on streaming data. Based on the definition of $SPN_{n,p}$, in each cycle, only one data is written into or read from the memory block M_i. Thus, Theorem 4.4 is proved.

Fig. 3 shows how a permutation network with $p = 2$ is generated using a 8-to-8 Benes network. The Benes network is routed to perform stride permutation $P_{8,2}$. To realize $P_{8,2}$, the permutation network needs two 2-to-2 switches and two 4-entry memory blocks.

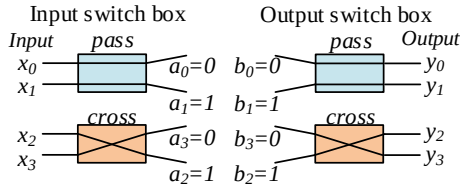


Fig. 5: Configuration bits of switch box in different states

4.2. Optimizing Interconnection Complexity

Fig. 4 shows the our complete design having a three-stage structure, including two stages of p-to-p connection networks (LB and RB) and one memory stage of p independent memory blocks (M_i, $p-1 \geq i \geq 0$). Each connection network is composed of $\log p$ stages, each stage consists of $p/2$ parallel 2-to-2 switches. Therefore, $(p \log p)/2$ control bits are required to determine the connection pattern of each connection network. These control bits are updated using a configuration table storing all the control information generated during design time. An address update unit is employed to update the memory addresses of the p memory blocks. Each memory block has one read port and one write port. To support processing continuous data inputs, each memory block needs to double the capacity (n/p) to be $2n/p$ to enable simultaneous read and write. Thus, $2p$ memory addresses (each of width $\log n/p$) are needed each clock cycle.

The interconnection complexity is defined as the interconnection area per throughput. The interconnection area of the proposed permutation network is mainly determined by the area of the two p-to-p connection networks. The **key observation** is that in a p-to-p connection network LB or RB, it is possible that a 2-to-2 switch realize the same permutation in each cycle. Therefore, assuming the permutation $L_{ij}.P$ or $R_{ij}.P$ is fixed in every cycle, hardware wires can be used instead for L_{ij} or R_{ij}. In this way, the logic consumption of a 2-to-2 switch can be saved. Such a switch is called as a *null switch*. As introduced in Section 4.1, the proposed SPN is generated using a routed Benes network. Thus, we propose a heuristic routing algorithm for the Benes network such that in the SPN for realizing a fixed n-to-n permutation P, a maximum number of *null switches* can be obtained. The configuration bits of the routed Benes network are then taken as input to generate the configuration tables and the address update unit in the control unit of SPN shown in Fig. 4.

A 2-to-2 switch in a Benes network will be either in *pass* state or *cross* state, which can be represented by a Boolean variable. Let x and y be the set of inputs and outputs of an n-to-n Benes network respectively. Fig. 5 shows the values of the configuration bits of a switch in different states, where a_i is the configuration bit of x_i, and b_i is the configuration bit of y_i. As shown in Fig. 5, if x_i is forwarded to the first output (upper side) of the switch, then $a_i = 0$, otherwise $a_i = 1$. Similarly, if y_i is forwarded from the first input of the switch, then $b_i = 0$, otherwise $b_i = 1$. Let π: $x \rightarrow y$ be an one-to-one input-output mapping specifying the given fixed permutation, such that $y_{\pi(i)} = x_i$. For example, for the stride permutation shown in Fig. 1, $\pi(0) = 0$, $\pi(1) = 2$, $\pi(2) = 4$, ..., etc.

Algorithm 1 End-to-End Routing Algorithm (EER)

```
1: procedure EER(x, y, i)
2:     a_i ← 0
3:     while !(routing complete) do
4:         b_π(i) ← a_i
5:         j ← 0
6:         if π_i%2 = 0 then
7:             b_π(i)+1 ← ā_i
8:             select j such that π(j) = π(i) + 1
9:             a_j ← b_π(i)+1
10:        else
11:            b_π(i)−1 ← ā_i
12:            select j such that π(j) = π(i) − 1
13:            a_j ← b_π(i)−1
14:        end if
15:        if x_i and x_j belong to the same switch box then
16:            randomly select a new i and initialize a_i if a_i = NULL
17:        else
18:            i ← j
19:        end if
20:    end while
21:    x' ← forward x using a for configuration
22:    y' ← forward y reversely using b for configuration
23:    return a, b, x', y'
24: end procedure
```

Algorithm 2 shows the heuristic algorithm for routing the Benes network. This algorithm takes as input the data parallelism p, the input and output vector x and y (both of size n). It first calls the procedure EER shown in Algorithm 1, which routes the first and last stages of an n-to-n Benes network for realizing the connection pattern specifying π: $x \rightarrow y$. The EER procedure returns four vectors including a, b, x', y'. a and b are data vectors storing the configuration bits of 2-to-2 switch boxes in the first and last stages of Benes network, respectively. x' is the resulting data vector after x bypasses the first stage of the Benes network using a for configuration. y' is the resulting data vector by forwarding y reversely in the last stage of the Benes network using b for configuration. x' and y' determine the connection patterns to be realized by the subnetworks in the middle stage. The RT procedure will be repeatedly called to decompose the problem into smaller subproblems. After running algorithm 2, we obtain A_k and B_k ($0 \leq k \leq 2p-3$) including all the required configuration bits which determine the connection patterns to be realized by the two p-to-p connection networks in the SPN. $2p$ data vectors, including X_k and Y_k for some values of k, are obtained in the recursive calls where $p = 1$ (not the original given data parallelism). These vectors are then used to generate the memory addresses of the p memory blocks. Assuming for some X_k and Y_k, X_k = $0, 1, 2, ..., n/p$, let π_k : $X_k \rightarrow Y_k$ represents one-to-one input-output mapping indicating connection request. A memory block to be written with a word using address $X_k[j]$ in cycle $j(0 \leq j \leq n/p-1)$, will then be read using address $v_k[j]$ in cycle $j + n/p$. v_k is

Algorithm 2 Heuristic Routing Algorithm (HR)

Global variables: $k \leftarrow 0, A_i, B_i, X_i, Y_i \leftarrow Null, i = 0, 1, ..., 2p-3$
procedure RT(x, y, p, i)
  $(a, b, x, y) \leftarrow$EER(x, y, i);
  $X_k \leftarrow x, Y_k \leftarrow y, A_k \leftarrow a, B_k \leftarrow b$
  $k \leftarrow k + 1$;
  if $p > 1$ **then**
    $s \leftarrow$ size of X_k
    $x \leftarrow (X_k[0], X_k[1], ..., X_k[s/2-1])$
    $y \leftarrow (Y_k[0], Y_k[1], ..., Y_k[s/2-1])$
    $i \leftarrow$ rand()$\%s$
    RT$(x, y, p/2, i)$;
    $x \leftarrow (X_k[s/2], X_k[s/2+1], ..., X_k[s-1])$
    $y \leftarrow (Y_k[s/2], Y_k[s/2+1], ..., Y_k[s-1])$
    $i \leftarrow$ rand()$\%s$
    RT$(x, y, p/2, i)$;
  else
    return
  end if
end procedure
procedure HR(x, y, p)
  $r \leftarrow 0, u \leftarrow 0, umax \leftarrow 0, result \leftarrow Null$
  while $r < runtimes$ **do**
    Obtain configuration bits for l_{ij} and r_{ij} through RT$(x, y, p, r{+}{+}\%n)$;
    for $i = 0$ to $p/2-1$ **do**
      for $j = 0$ to $\log p - 1$ **do**
        Check if L_{ij} is a $null$ switch
        Check if R_{ij} is a $null$ switch
        Update the number of $null$ switches u
      end for
    end for
    if $umax ==$ 0 **then**
      $umax \leftarrow u$
    else if $umax < u$ **then**
      $umax \leftarrow u$
      $result \leftarrow$ Configuration bits for L_{ij} and R_{ij} and memory address vectors v_k for memory M_k, k = 0, 1, ..., $p-1$
    end if
  end while
  return $result$
end procedure

called as the *memory address vector* including n/p memory addresses, each having a bit width of $\log(n/p)$. For $0 \leq i \leq n/p-1$, $\pi(v_k[i]) = X_k[i]$. If assuming we store all the memory addresses using on-chip memory, then the total memory consumption of a *memory address vector* is $n/p \log(n/p)$ bits. The *memory address vectors* are stored in the address update unit. For stride permutation or bit reversal, instead of storing the memory addresses using on-chip memory, the address update unit will dynamically update the memory address based on some initial memory address to save memory resource consumption. RT procedure will be called by $runtimes$ times to search for the optimal solution of RT(x, y, p, i) such that the maximum number of *null switches* recorded by $umax$ can be obtained using the mapping approach introduced in Section 4.1.

5. EXPERIMENTAL RESULTS

5.1. Experimental Setup

By varying the parameters n and p, we performed detailed experiments for our proposed SPN design. We implemented all our design on Virtex-7 FPGA (XC7VX980T) using Xilinx Tool Set Vivado 15.2. We choose 32-bit fixed point data vectors as input. In the experiments, for power evaluation, the input test vectors were randomly generated with an average toggle rate of 25% (pessimistic estimation) We used the VCD (value change dump) file as input to Vivado Power Analyzer to obtain accurate power dissipation estimation [15]. Implementations developed in [8] and [9] are employed as baseline. The Verilog implementations of the baselines were available through [16]. In the experiments, we randomly choose permutation patterns used in the signal and data processing algorithms for various n and p. The same permutations are also employed for evaluating our baselines.

Table 1: Resource consumption summary

Designs	Amount of memory	Memory type	Size of each memory	# of mux[1]	Supported permutation
This paper	p	Dual-port	$2N/p$	$0 \sim 2p \log p$	Any fixed
[9]	$2p$	Dual-port	$2N/p$	$2p \log p - 2p + 2$	Any fixed
[8]	p	Dual-port	$2N/p$	$2p \log p$	BIP[2]
[1]	p	Single-port	N/p	$2p \log p$	Stride permutation

[1] 2-to-1 mux, [2] Bit-index permutation [8]

5.2. Resource Consumption Evaluation

Table 1 summarizes the resource consumption of various designs for realizing permutations with a fixed data parallelism. Memory size refers to the size of each independent memory block storing 32-bit words. The design in [9] can realize any fixed permutation, however, it requires 2x total memory compared with our design. A single-port memory based technique is proposed in [1] for improving memory efficiency, however, their design only supports stride permutation. Matrix manipulations are employed to reach a solution of a hardware structure for data permutation in [8], however, the interconnection complexity is not considered. Multiplexers are the logic resource consumed by the connection networks. In our proposed permutation network, the number of required multiplexers can be reduced to zero theoretically. In this way, an improvement on throughput can be obtained due to higher maximum achievable design operating frequency. Furthermore, the energy efficiency is also improved as a result of less dynamic power consumption.

Fig. 6a and Fig. 6b show the BRAM consumption for various p for $n = 1024$ and $n = 8192$. Each BRAM denotes a simple dual-port 18kb BRAM (BRAM18E). The design in [9] consumes less BRAMs than our design as small size memory blocks in [9] are implemented using distributed RAMs (LUTs). Fig. 6b shows that the number of BRAMs is not affected by p when $p \leq 32$ for our designs. Note that regardless of how small the required memory capacity is, a BRAM has to be assigned. Therefore, in our design, 16 BRAMs are able to meet the memory requirement for $n = 8192$ when $p \leq 32$. Then the number of BRAMs doubles when $p = 64$. Similarly, in Fig. 6c, the number of BRAMs is not affected by n until the amount of BRAMs currently used fails to meet the memory resource requirements of the design. Fig. 6d shows that the number of

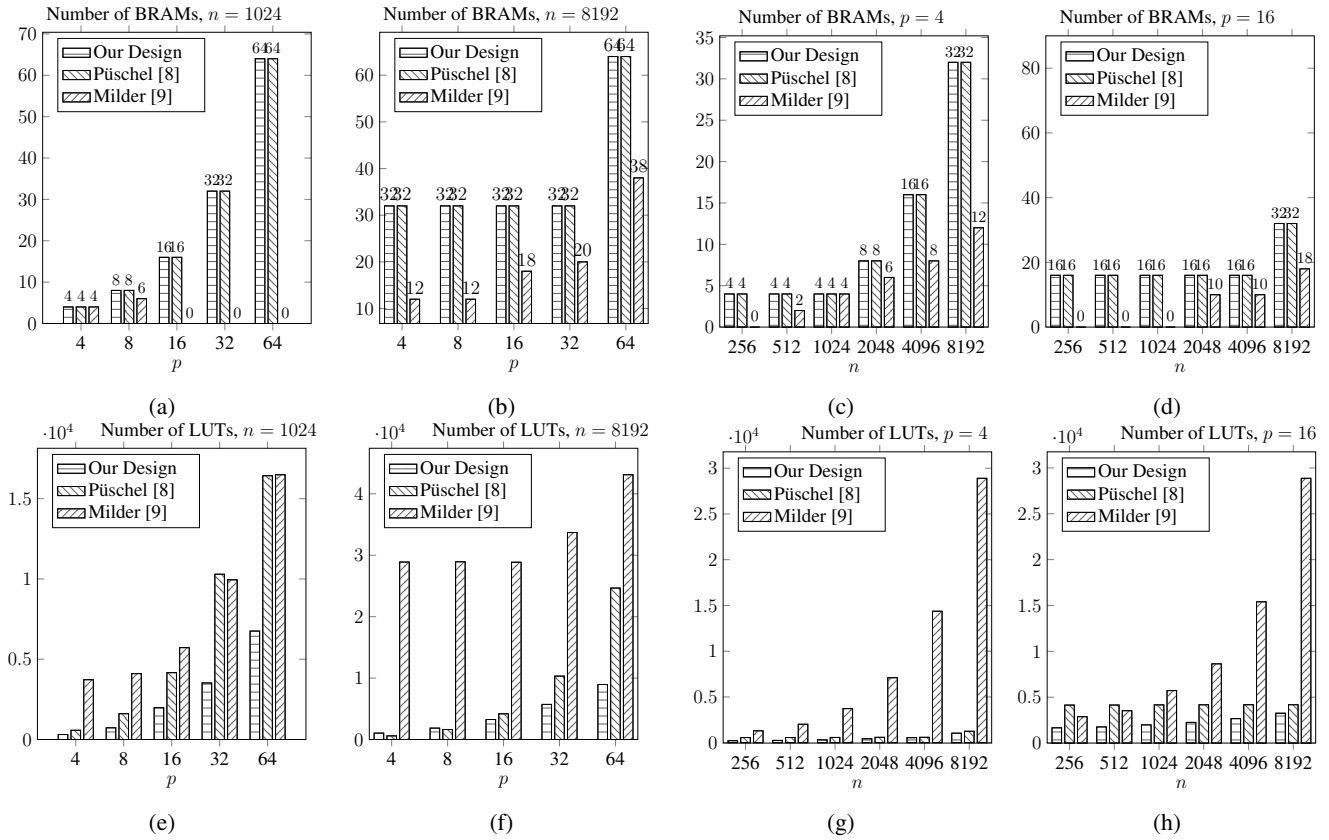


Fig. 6: Resource consumption for various n and p

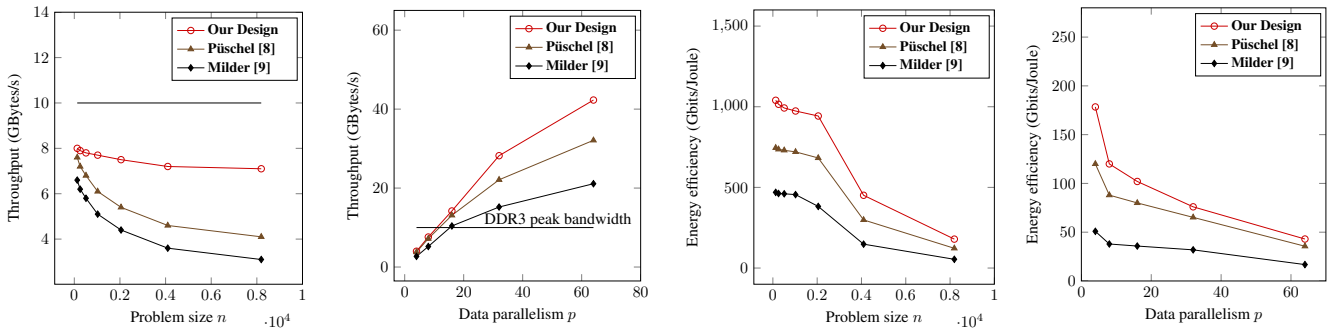
BRAMs is not affected by n for our design when $n < 8192$. This is due to the fact that p independent memory blocks are always required for a specific p. For example, 16 memory blocks are needed in our design for $p = 16$, a single BRAM can meet the memory requirement of each memory block for $256 \leq n < 8192$. Although the experimental results for other values of p and n are not presented here, similar BRAM consumption increasing rate is expected. Note that the maximum problem size n supported by our design is determined by the available on-chip memory resource on FPGA and can be more than 8192.

Fig. 6e and Fig. 6f present the LUT consumption, including LUTs consumed for memory and logic, for various p with $n = 1024$ and $n = 8192$, respectively. A significant amount of LUTs are consumed in [9] as distributed RAMs have been employed. In our design and [8], LUTs are mainly consumed by the connection networks. Therefore, the figures actually demonstrate how n and p affect the LUT consumption of the interconnection logic. The figures show that the LUT consumption increases significantly with p. This matches with the theoretical result in Table 1; the number of required multiplexers is $O(2p \log p)$ (Table. 1). For various p, our design reduces the LUT consumption by 22.1%∼65.7% and 59.0%∼96.4%, compared with the design in [8] and the design in [9], respectively. Fig. 6g and Fig. 6h show the LUT consumption for various n with $p = 4$ and $p = 16$, respectively. Our design reduces the LUT consumption by 27.3%∼75.8% and 42.2%∼92.3%, compared with the design in [8] and the design in [9], respectively. The above results show the optimized interconnection complexity in our design than the technique used in [8]. However, it is difficult to quantitatively compare the interconnection complexity between our design and the design in [9] which employs different memory implementation approach. Therefore, we further present experimental results of throughput and energy efficiency for performance comparison.

5.3. Performance Evaluation

To evaluate the throughput of our designs on FPGA, we assume *data memory* for storing original data input can be either BRAM or DRAM depending on the data set size. We assume the data memory bandwidth can be fully utilized as the data memory access behavior is known to be sequential. The throughput of our design is determined by p and the maximum operating frequency reported by the post place-and-route results.

Fig. 7a demonstrates the design throughput for various n with $p = 4$. The throughput decreases due to the lower achievable operating frequency of the design as n increases.



(a) (b) (c) (d)

Fig. 7: Performance results for various n and p

Our design is able to sustain 71%∼80% of the DDR3 peak bandwidth, (approximately 10 GBytes/s [17]) for various n with $p = 4$. Fig. 7a shows that our design improves the throughput by 5.3%∼73.2% compared with [8] and 27.4%∼129% compared with [9], with more throughput improvement for large problem sizes. Such performance improvement can be expected considering less memory consumption (than [9]) and reduced interconnection complexity in our design. Fig. 7b demonstrates the design throughput for various p with $n = 8192$. The throughput of our design increases almost linearly with p when $p < 64$. Such an observation is consistent with the lower growing rate of LUT consumption in our design shown in Fig. 6f. Fig. 7b shows that our design improves the throughput by 5.2%∼31.7% and 4.6%∼100%, compared with the design in [8] and [9], respectively. Fig. 7b also shows that the DDR3 peak bandwidth can be easily saturated when $p > 16$. In such cases, we can only employ the on-chip BRAMs with extremely high bandwidth as the data memory.

We also evaluate the energy efficiency of our design. focusing on the dynamic power consumption. We employed a balanced pipelining approach for the purpose of trade off between energy efficiency and throughput. Fig. 7c demonstrates the energy efficiency for various n (128∼8192) with $p = 4$. It can be observed that the energy efficiency drops significantly for $n = 2048$. This is due to the fact that the number of BRAMs used starts to increase with n when $n \geq 2048$. The result also shows that as n is varied, our design achieves 2.4x∼3.5x (1.2x∼1.5x) improvement in energy efficiency compared with the design in [9] ([8]). Fig. 7d demonstrates the energy efficiency when varying p for $n = 8192$. The energy efficiency decreases significantly with p. For various p, our design improves energy efficiency by 2.1x∼3.3x compared with [9] and 1.4x∼1.5x compared with [8].

6. CONCLUSION

In this paper, we presented a memory-based technique to realize arbitrary permutation with a fixed data parallelism. The novelty of our work is that our design is built by vertically folding the classic Benes network, thus maintaining the minimal connection cost feature of Benes network. We further develop a heuristic routing algorithm for Benes network so as to optimize the interconnection complexity of our proposed streaming permutation network. Detailed experimental results show that by reducing the interconnection complexity, our design outperforms the baselines with respect to both throughput and energy efficiency. Our future plan is to work on a design automation framework targeting automatic generation of high performance designs for data and signal processing kernels such as sorting, convolution neural network and FFT on FPGA.

7. REFERENCES

[1] R. Chen, H. Le, and V. K. Prasanna, "Energy efficient architecture for stride permutation on streaming data," in *Proc. of IEEE International Conference on ReConFig*, Dec 2013.

[2] H. Moussa, A. Baghdadi, and M. Jézéquel, "Binary de bruijn on-chip network for a flexible multiprocessor ldpc decoder," in *Proc. of ACM Design Automation Conference (DAC)*, 2008.

[3] C.-W. J. Wen-Chang Yeh, "High-speed and low-power split-radix FFT," *IEEE Transactions on Signal Processing*, vol. 51, no. 3, pp. 864–874, 2003.

[4] S. R. Kuppannagari, R. Chen, A. Sanny, S. G. Singapura, G. P. C. Tran, S. Zhou, Y. Hu, S. P. Crago, and V. K. Prasanna, "Energy performance of fpgas on perfect suite kernels," in *Proc. of IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[5] A. Srivastava, R. Chen, V. Prasanna, and Chelmis, "A hybrid design for high performance large-scale sorting on fpga," in *Proc. of International Conference on*

ReConFigurable Computing and FPGAs (ReConFig), 2015, pp. 1–6.

[6] T. Järvinen, *Systematic Methods for Designing Stride Permutation Interconnections*. Tampere University of Technology, 2004.

[7] S. He and M. Torkelson, "A new approach to pipeline FFT processor," in *Proc. of Parallel Processing Symposium (IPPS) '96*, 1996, pp. 766–770.

[8] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using rams," *Journal of the ACM*, vol. 56, no. 2, pp. 10:1–10:34, 2009.

[9] P. Milder, J. Hoe, and M. Puschel, "Automatic generation of streaming datapaths for arbitrary fixed permutations," in *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*, April 2009, pp. 1118–1123.

[10] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proc. of ACM/SIGDA International Symposium on FPGA*, pp. 240–249.

[11] V. E. Beneš, "Optimal rearrangeable multistage connecting networks," *Bell System Technical Journal*, vol. 43, no. 4, pp. 1641–1656, 1964.

[12] R. Chen and V. Prasanna, "Energy efficient parameterized FFT architecture," in *Proc. of IEEE International Conference on Field Programmable Logic and Applications (FPL)*.

[13] T. Jarvinen, P. Salmela, H. Sorokin, and J. Takala, "Stride permutation networks for array processors," in *Proc. of IEEE Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2004, pp. 376–386.

[14] T. K. Moon and W. C. Stirling, *Mathematical methods and algorithms for signal processing*. Prentice Hall New York, 2000, vol. 1.

[15] "XST user guide for Virtex-6, Spartan-6, and 7 series devices," http://www.xilinx.com/support/documentation.

[16] "SPIRAL Hardware Generator," http://www.spiral.net/hardware.html.

[17] "DDR3 SDRAM System-Power Calculator," http://www.micron.com/\-/media/Documents/Products/Power\%20Calculator/DDR3\textunderscorePower\textunderscoreCalc.\\XLSM.