# AUTOMATIC GENERATION OF HIGH THROUGHPUT ENERGY EFFICIENT STREAMING ARCHITECTURES FOR ARBITRARY FIXED PERMUTATIONS*

*Ren Chen and Viktor K. Prasanna*

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

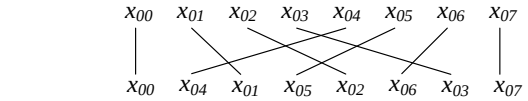
## ABSTRACT

Due to their high data-rate and simple control, streaming architectures have become popular for hardware implementation of data intensive applications. A key problem in designing such architectures is to permute streaming data. In this paper, we present a technique to realize arbitrary fixed permutation on streaming data. We develop a parameterized architecture which accepts data streams as input and generates the permuted data after a certain amount of delay. Our design accepts continuous input at a fixed rate of $p$ per cycle, where $p$ is the data parallelism of the architecture. To construct the streaming architecture for a given fixed permutation, we develop a mapping approach by configuring the classic Benes network to obtain the datapath and the control logic. We demonstrate a complete design automation tool which takes as input design parameters including the permutation pattern and the data parallelism $p$, and produces register-transfer level Verilog description of the design. We evaluate the generated designs on Xilinx Virtex-7 FPGA using post place-and-route results.

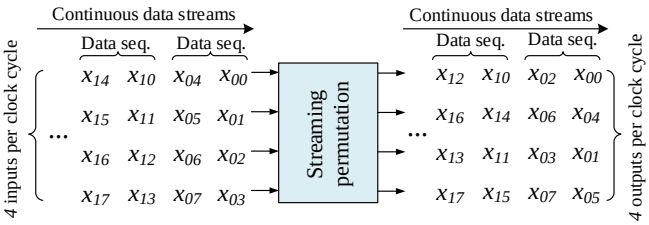
## 1. INTRODUCTION

Data permutations are needed in a wide variety of applications such as multi-dimensional signal processing and parallel sorting networks [1, 2, 3, 4]. Efficient hardware solution for performing data permutations is critical for parallel hardware implementations of these data intensive applications.

A data permutation is a fixed reordering of a given number of data elements. For example, given a stride $t$, a stride permutation reorders an $N$-element data vector, such that data elements with an index distance of $t$ are moved into adjacent locations. Fig. 1(a) shows an example of stride permutation with $N$=8 and $t$=4. This permutation can be simply implemented by a reordering of hardware wires if all the input elements are available concurrently. However, for large data sets, this simple approach is not technically feasible due to limited data bandwidth, complex routing and high area consumption. Instead, a more feasible approach is to feed and process the input data in a streaming manner: assuming a hardware block has an available data parallelism of $p$ ($1 \leq p \leq N$), an $N$-element input (output) sequence enters (leaves) the hardware block over $N/p$ consecutive clock cycles. Data parallelism $p$ is the number of input elements processed in parallel in each clock cycle. Fig. 1(b) shows an example of permutation on streaming data, where data sequence size $N = 8$ and $p = 4$.




(a) Stride permutation with a stride of four




(b) Stride permutation on streaming data

**Fig. 1**: Permutation on streaming data

Previous work on realizing permutations on hardware can be divided into RAM-based designs and register-based designs [4]. Well known register-based designs are delay feedback or delay commutator modules performing stride permutations widely used in FFT architectures [5]. However, this technique requires the inputs to be fed into the design in a particular order, thus limiting the data parallelism and throughput. In [4], the authors present RAM-based and register-based stride permutation networks for array processors. In [1], the authors develop a RAM-based structure to realize a specific family of permutations called bit index permutations. These approaches only provide methods to design streaming implementations for a particular family of permutations. In [6], a RAM-based technique that can be

*This work was partially supported by the DARPA under grant number HR0011-12-2-0023, and in the past supported by the U.S. NSF under grant number CCF-1018801.

used to automatically design a streaming datapath for any given permutation is developed. However, the resulting design are expensive with regarding to memory.

In this paper, we present a technique for realizing *any* given fixed permutation on streaming data. With a data parallelism $p$, continuous data streams can be permuted with a throughput of $p$ output elements per cycle. For problem size $N$, our approach supports using $p$ single port memory blocks with total memory size of $N$. For the same throughput, the method in [6] requires $2p$ dual-port memory blocks with total memory size of $4N$. Our contributions include:

- We develop a RAM-based technique to realize any given fixed permutation on streaming data.
- We propose a resource efficient mapping method using the classic Benes network to generate the datapath and the control logic of our design.
- Our approach minimizes the memory usage compared with the state-of-the-art, thus achieving higher energy efficiency while sustaining the same throughput.
- We develop a design automation tool which takes as input the permutation pattern and data parallelism, and generates register-transfer level Verilog description of the design.

## 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 where each element is either 0 or 1, with exactly one 1 in each row and each column. The product of permutation matrices is another permutation matrix [3].

A stride permutation can be defined using matrix representation. Given an $N$-element data vector $x$ and a stride $t$ $(0 \leq t \leq N-1)$, the data vector $y$ which is produced by the stride-by-$t$ permutation over $x$ can be given as $y = P_{N,t}x$, where $P_{N,t}$ is a permutation matrix. $P_{N,t}$ is an invertible $N \times N$ bit matrix such that

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

where *mod* is the modulus operation and $\lfloor\ \rfloor$ is the floor function. E.g., $P_{4,2}$ performs $x_0, x_1, x_2, x_3 \rightarrow x_0, x_2, x_1, x_3$. Another well known permutation matrix $Q_N$ of order $N$ is defined as $Q_N = (I_2 \otimes P_{\frac{N}{2},\frac{N}{4}}) \cdot P_{N,2}$, where $I_2$ is the identity matrix and $\otimes$ is the tensor (or Kronecker) product [3].

Bit-index permutations were first defined in [7]. 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 $k$. Important permutations including bit reversal and bit shuffle are special cases of Bit-index permutations.

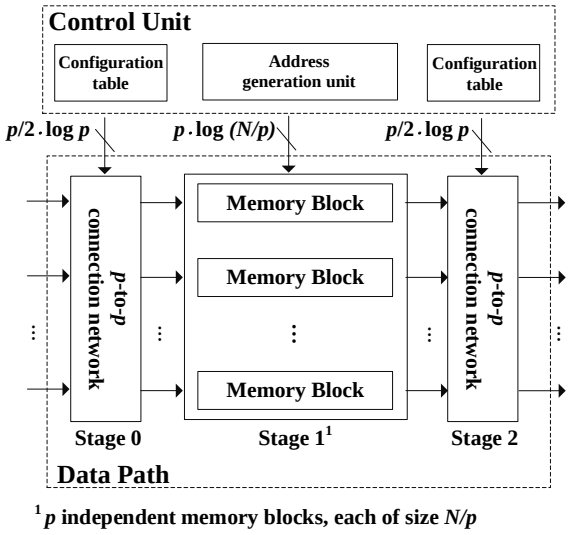



**Fig. 2**: Streaming implementation proposed in this paper

### 2.2. Related Work

Permutation on streaming data using hardware have been well studied. Existing work in the literature are mostly focused on a specific family of permutations such as stride permutations, perfect shuffle, and bit-index permutations. In [8], streaming permutation structures for a subset of stride permutations are proposed to obtain high performance hardware implementations for the fast Fourier transform. In [4], 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 resource efficiency, however, requiring many independent dual-port memory blocks. By expanding this work, a streaming structure is proposed to perform any permutation which is a linear mapping on the bit representation of the data locations [1]. Their design supports problem size and data parallelism that are powers of two. An extension of this approach is presented in [6]. This design is capable of performing arbitrary fixed permutation for a given data parallelism. Hardware implementations of their design is expensive with respect to memory. Compared with all the above related work, our proposed technique supports arbitrary fixed permutation on streaming data with high energy efficiency, and minimizes memory consumption while sustaining the same throughput.

## 3. ARCHITECTURE AND IMPLEMENTATIONS

### 3.1. Problem Definition

We use $p$ to denote the data parallelism defined as the number of input data elements processed per cycle. We define

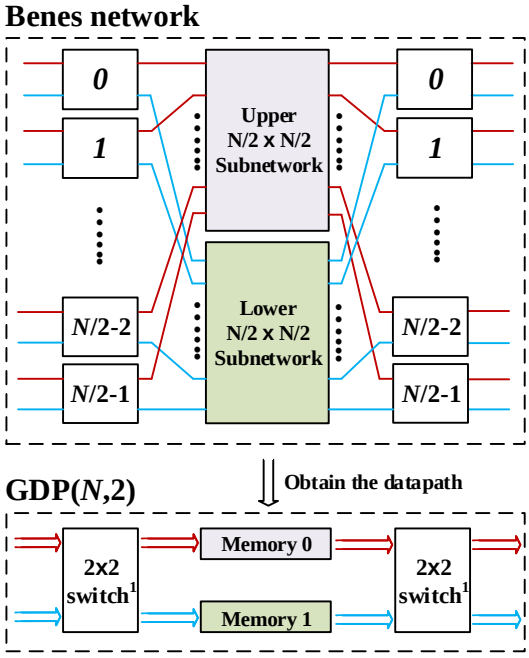



**Fig. 3**: Generating datapath for realizing a fixed permutation using GDP$(N, 2)$

the *streaming permutation* problem as: assuming a hardware block has an available data parallelism of $p$ ($1 \leq p \leq N$), an $N$-element sequence enters the hardware block over $N/p$ consecutive cycles, after a certain amount of delay $T$, the sequence is reordered as specified by the required permutation and output over $N/p$ consecutive cycles. With continuous 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.

### 3.2. Parameterized Streaming Architecture

Fig. 2 shows the overall architecture of the streaming design. Design parameters include problem size $N$ and data parallelism $p$. The datapath consists of two $p$-to-$p$ connection networks, one memory array including $p$ independent memory blocks, each of size $N/p$. The control unit has two configuration tables to reconfigure the connection networks dynamically, and one address generation unit (AGU) for memory access. Each connection network has $\log p$ stages of $2 \times 2$ switches. Each stage has $p/2$ switches. Thus, a connection network utilizes $(p/2) \log p$ $2 \times 2$ switches, which is asymptotically optimal compared with state-of-the-art [1, 8, 6]. We consider using two different memory architectures: each memory has one read port and one write port, or each memory has a concurrent read-before-write port (two data ports sharing one address port).

The connection network is run time configured each clock cycle based on the configuration table. Each table has $N/p$ configurations, each having $(p \log p)/2$ bits. The configurations are statically generated for a given fixed permutation. The latency of connection network is determined by the pipeline depth chosen for implementation. As the data flows through the streaming design shown in Fig. 2, they are first permuted by the connection network at the input. Then the permuted data flows are written into the $p$ memory blocks in $N/p$ clock cycles, and read out of the memory blocks in the subsequent $N/p$ clock cycles. All the memory addresses are generated dynamically by the AGU. Finally, data flows are permuted by the connection network at the output to obtain the results.

**Algorithm 1** MERGE

```
1: procedure MERGE(A, B)
2:     k ← data parallelism of A or B;
3:     P_{2k,2} ← stride-by-2 permutaion;
4:     S_in ← 2k 2 × 2 switches;
5:     Connect output of S_in and input of {A, B}[1] with fixed connection
       realizing P_{2k,2};
6:     S_out ← 2k 2 × 2 switches;
7:     Connect input of S_out and ouput of {A, B}[1] with fixed connection
       realizing P_{2k,2};
8:     Return the resulting datapath;
9: end procedure
```

[1] $A$ is the upper part, $B$ is the lower part

**Algorithm 2** Generating Datapath (GDP)

```
1: Initialize N and p
2: procedure GDP(N, p)
3:     p ← p/2;
4:     if p > 1 then
5:         A ← GDP(N/2, p/2);
6:         B ← GDP(N/2, p/2);
7:         return MERGE(A, B);
8:     else
9:         return GDP(N, 2);
10:    end if
11: end procedure
```

### 3.3. Generating the data path

In this section, we present our algorithm of automatic generating the datapath of the proposed streaming design. We use GDP$(N, 2)$ to denote the algorithm for generating the datapath of the streaming design with $p = 2$. Instead of giving the algorithm description of this procedure, we use Fig. 3 to illustrate the algorithm implementation. As shown in this figure, the algorithm takes as input the Benes network [9] of size $N$. A Benes network is a multi-stage routing network, where all $N/2$ modules in the first stage and $N/2$ modules in the last stage are $2 \times 2$ switches. The two modules in the middle stage are $N/2 \times N/2$ subnetworks and each can be decomposed into three-stage Benes network in a recursive manner. As shown in Fig. 3, each $2 \times 2$ switch has a unique link connecting to the upper subnetwork and a unique link to the lower subnetwork. The resulting datapath of our design also has a three-stage structure, including two stages of

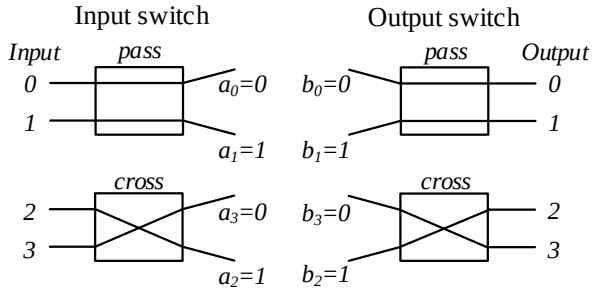



**Fig. 4**: Configuration bits of switch in different states

switches and one memory stage. The $2 \times 2$ switch in the first/last stage is run time reconfigured to realize the inside connection in switch $0, 1, ..., N/2 - 1$ in the first/last stage of Benes network. Each subnetwork in the middle stage of Benes network is replaced with a memory block in the datapath. The permutation performed by the subnetwork is realized by its corresponding memory, and input data are permuted temporally instead of spatially.

Algorithm 2 shows our divide-and-conquer based algorithm for generating the datapath of the streaming design with any $p$. When $p > 2$, the problem is divided into two subproblems taking as input $N/2$ and $p/2$. Each subproblem is further decomposed similarly. A merge algorithm shown in Algorithm 1 is used to merge the datapaths denoted using block $A$ and block $B$. $\{A,B\}$ represents the datapath by vertically combining block $A$ and block $B$, however, without any interconnections between $A$ and $B$. Then they are connected with block $S_{in}$ and block $S_{out}$ to obtain the final merged data path.

### 3.4. Generating the control logic

In this section, we present our algorithm for generating the control logic. The key idea is to configure the Benes network for a given fixed permutation. The resulting configuration bits are then taken as input to generate the configuration tables and the AGU shown in Fig. 2.

A $2 \times 2$ switch of 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 a Benes network respectively, Let $a : X \rightarrow \{0, 1\}$ and $b : Y \rightarrow \{0, 1\}$, where $a_i$ is the configuration bit of $X_i$, and $b_i$ is the configuration bit of $Y_i$. Fig. 4 shows the values of the configuration bits of switch in different states. We assume $X = Y = 0, 1, ..., N - 1$. Let $\pi$: $X \rightarrow Y$ be an one-to-one input-output mapping specifying the given fixed permutation. For stride-by-4 permutation shown in Fig. 1, $\pi(0) = 0, \pi(1) = 2, \pi(2) = 4, ...$, etc.

Algorithm 4 shows the divide-and-conquer based algorithm for generating the control logic. This algorithm takes as input the data parallelism $p$, the input $X$, and the output $Y$. It first calls the procedure SSR shown in Algorithm 3. The SSR procedure returns four vectors including $a, b, X', Y'$. $a(b)$ includes the configuration bits of $2 \times 2$ switches in the first(last) stage of Benes network. $X'$ and $Y'$ determine the connection patterns to be realized by the subnetworks in the middle stage. After running algorithm 4, we obtain $a_i$ and $b_i$ $(0 \leq i \leq \log p - 1)$ including all the required configuration bits. $a_i$ $(0 \leq i \leq \log p - 1)$ contain $N \log p$ configuration bits in total. Instead of storing $a_i$, we store the state bits of the $2 \times 2$ switches calculated based on $a_i$ in the configuration table at the input. According, the state bits calculated based on $b_i$ will be stored in the configuration tables at the output. Thus, each configuration table needs to store $(N/2) \log p$ bits.

**Algorithm 3** Single Stage Routing (SSR)

```
1:  procedure SSR(X, Y)
2:      i ← 0
3:      a_i ← 0
4:      while !(routing complete) do
5:          b_{π_i} ← a_i
6:          j ← 0
7:          if π_i%2 = 0 then
8:              b_{π_i+1} ← ā_i
9:              select j such that π_j = π_i + 1
10:             a_j ← b_{π_i+1}
11:         else
12:             b_{π_i-1} ← ā_i
13:             select j such that π_j = π_i − 1
14:             a_j ← b_{π_i-1}
15:         end if
16:         // Update i
17:         if X_i and X_j belong to the same switch box then
18:             select a new i and initilize a_i if a_i = NULL
19:         else
20:             i ← j
21:         end if
22:     end while
23:     X' ← forward X based on a
24:     Y' ← send Y backward based on b
25:     return a, b, X', Y'
26: end procedure
```

**Algorithm 4** Multiple Stage Routing (MSR)

```
1:  i ← 0
2:  procedure MSR(X, Y, p)
3:      (a_i, b_i, X_i, Y_i) = SSR(X, Y);
4:      i ← i + 1;
5:      if p > 1 then
6:          k ← size of X_i
7:          X_{i+1} ← {X_i[j], 0 ≤ j ≤ (k/2 − 1)}
8:          Y_{i+1} ← {Y_i[j], 0 ≤ j ≤ (k/2 − 1)}}
9:          MSR(X_{i+1}, Y_{i+1}, p/2);
10:         X'_{i+1} ← {X_i[j], k/2 ≤ j ≤ (k − 1)}
11:         Y'_{i+1} ← {Y_i[j], k/2 ≤ j ≤ (k − 1)}
12:         MSR(X'_{i+1}, Y'_{i+1}, p/2);
13:     else
14:         return (a_i, b_i, X_i, Y_i)
15:     end if
16: end procedure
```

In Algorithm 4, we finally need to apply SSR procedure to $p$ subnetworks, each of size $N/p$. We use $X_k$ and $Y_k$ $(0 \leq$

$k \leq p-1$) to denote the input and output vectors for these $p$ subnetworks. Again assuming $X_k = 0, 1, 2, ..., N/p$, let $\pi_k : X_k \rightarrow Y_k$ represents one-to-one input-output mapping indicating connection request. Then the vector $\pi_k$ will be used to access the memory block $k$ in our proposed streaming design. This memory address vector includes $N/p$ memory addresses, each having a bit width of $\log N/p$. If assuming we store all the memory addresses using on-chip memory, then the total memory consumption is $N \log N/p$ bits. For stride permutation or bit reversal, instead of storing the memory addresses using on-chip memory, the AGU will dynamically update the memory addresses based some initial memory addresses, such that the resource consumption for control can be significantly reduced. We also employ an algorithm proposed in our previous work [10] for memory optimization purpose. This algorithm enables the use of single-port memory to process continuous data streams. This single-port memory needs to support simultaneous read-write operation (read first) to the same memory location [11]. In the state-of-the-art [6], dual-port memory is required to perform permutation in time on continuous data streams.

**Theorem 3.1.** *For a given data parallelism $p$, the resulting design based on Algorithm 2 and Algorithm 4 can realize any fixed permutation on $N$-word streaming input using $p$ memory blocks, each of size $N/p$.*

**Proof:** Based on [9], the Benes network shown in Fig. 3 can realize any permutation by configuring its $2 \times 2$ switches. For the design generated based on GPD($N, 2$), by run time configuring its $2 \times 2$ switch at the input/out, the connections in switches in the first/last stage of Benes network can be realized. Each memory block in the second stage of the generated design has a memory size of $N/2$ and can be written with an output of the $2 \times 2$ switch at the input. In $N/2$ clock cycles, the two memory blocks in the second stage of the generated design realize the connections of the two subnetworks in the middle of the Benes network. Therefore, the design generated by GPD($N, 2$) simulates the Benes network with $p = 2$. The above deduction can be also applied for $p > 2$, hence proving the theorem.

Fig. 5 shows a mapping example where the Benes network is configured to perform stride permutation $P_{8,2}$. In this example, $N = 8$ and $p = 2$. To execute $P_{8,2}$, the streaming design needs two $2 \times 2$ switches and two 4-entry memories. The values of vectors $a$, $b$, $X_0$, $X_1$, $Y_0$, $Y_1$ are obtained using Algorithm 4.

## 4. EXPERIMENTAL RESULTS AND ANALYSIS

### 4.1. Experimental Setup

In this section, we present a detailed experimental evaluation of several implementations by varying the parameters $N$ and $p$. Based on the techniques we discussed in Section 3, we have built a design automation tool. All the designs were automatically generated by this tool in Verilog and implemented on Virtex-7 FPGA (XC7VX980T) using Xilinx Vivado 14.2. All the designs assume 32-bit fixed point data as input. The input test vectors for simulation were randomly generated with an average toggle rate of 25% (pessimistic estimation). We used the VCD file (value change dump file) as input to Vivado Power Analyzer to produce accurate power dissipation estimation [11]. We employ designs developed in [1] and [6] as baseline. The Verilog implementations of the baselines were obtained from [12]. As the cost of our design datapath does not depend on the specific permutation to be performed, we choose one randomly for various $N$ and $p$ in experiments. The same permutations are also chosen for our baselines.

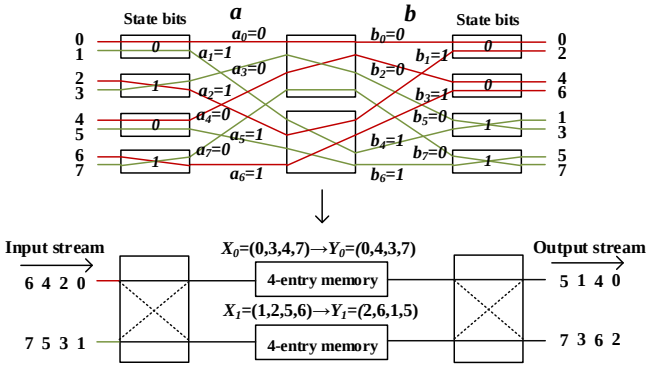



**Fig. 5**: Example: Realizing $P_{8,2}$ with $p = 2$

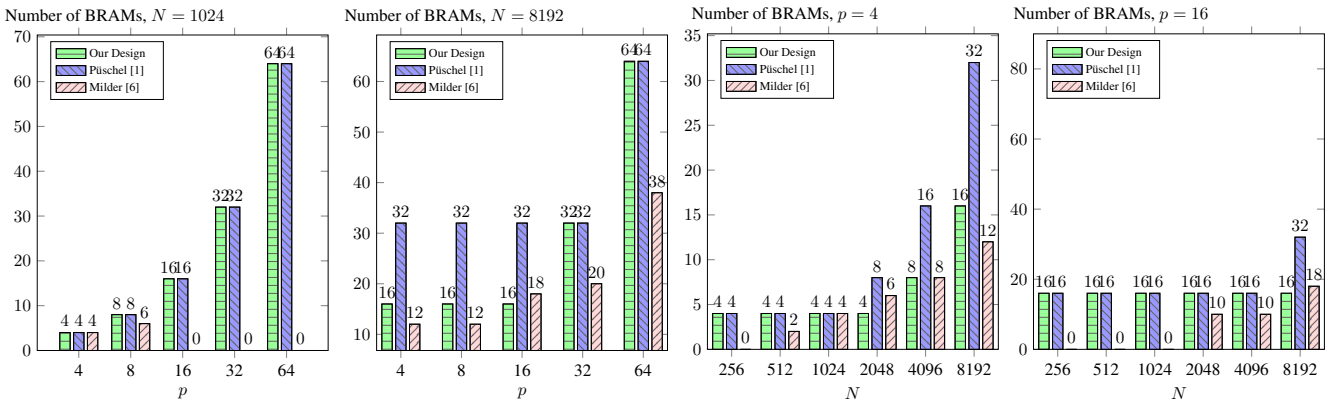
**Table 1**: Resource consumption summary

| Designs | Memory # | Memory type | Memory size | Mux[1] # | Supported permutation |
|---|---|---|---|---|---|
| This paper | $p$ | Single or Dual-port[2] | $N/p$ or $2N/p^{(2)}$ | $2p \log p$ | Any fixed |
| [6] | $2p$ | Dual-port | $2N/p$ | $2p \log p - 2p + 2$ | Any fixed |
| [1] | $p$ | Dual-port | $2N/p$ | $2p \log p$ | BIP[3] |
| [8] | $p$ | Single-port | $N/p$ | $2p \log p$ | Stride permutation |

[1] 2-to-1 mux, [2] Single-port ram based approach is an option in our tool, [3] Bit-index permutation [1]

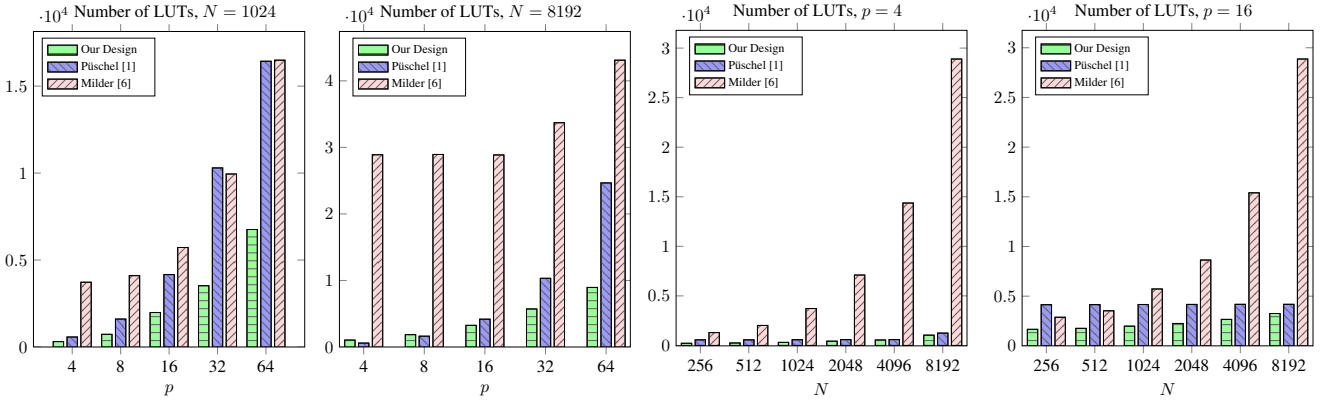
### 4.2. Design Automation Tool

We have built a design automation tool based on the techniques and algorithms discussed in this paper. The tool takes as input a fixed permutation with problem size $N$ and data parallelism $p$, and outputs a register-transfer level Verilog description of the design. The tool first generates an instance of the parameterized datapath based on Algorithm 2, and then uses Algorithm 4 to generate the configuration tables and the AGU required for control. Although the target platform in this paper is FPGA, the IP core generated by our




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

**Fig. 6**: BRAM consumption in various designs




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

**Fig. 7**: LUT consumption in various designs

tool can also be used for ASIC design. We plan to post our complete tool online in the near future.

### 4.3. Resource consumption evaluation

Table 1 summarizes theoretical results of the resource consumption of various designs for realizing fixed permutations on streaming data. Memory size refers to the size of each independent memory block storing 32-bit words. [6] is able to realize any fixed permutation, however, it requires 4x total memory consumption compared with our design. Besides, our design is up to 2x memory efficient than [1] and [8]. Note that designs in [1] and [8] only support a specific family of permutations. Our design also supports single-port memory for processing streaming data by employing the technique we proposed in [10]. Note that BRAM18E on FPGA configured in single-port mode has less active address ports than in simple dual-port mode [11]. Thus, this single-port RAM based approach can be employed to reduce dynamic power consumption and applicable to arbitrary permutation. In this paper, single-port RAM based approach is employed by all our designs.

Fig. 6a and Fig. 6b show the BRAM consumption for various $p$ with $N = 1024$ and $N = 8192$. Each BRAM denotes an 18kb BRAM (BRAM18E). In our design, BRAM18E is configured in single-port mode. In [1] and [6], BRAM18E is configured in simple dual-port mode. Most of the memory blocks in [6] are implemented using distributed RAMs, thus consuming more LUTs. Note that regardless of how small the required memory capacity is, a BRAM has to be assigned. This explains why our design consumes the same amount of BRAMs as [1] consumes in Fig. 6a and Fig. 6d. In the future, we plan to replace BRAM blocks with distributed RAMs for small size memories. In this way, we can expect 50% less distributed RAM consumption in our design compared with [1] for experiments shown in Fig. 6a and Fig. 6d. Similarly, Fig. 6b shows that the number of BRAMs is not affected by $p$ when $p \leq 16$ for our design. 16 BRAMs are able to meet the memory requirement for $N = 8192$ until $p > 16$. When $p > 32$, the number of BRAMs increases linearly with $p$. 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

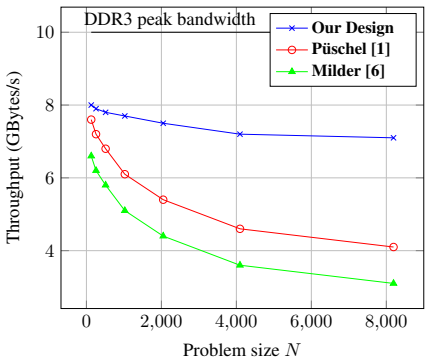



**Fig. 8**: Throughput for various $N$

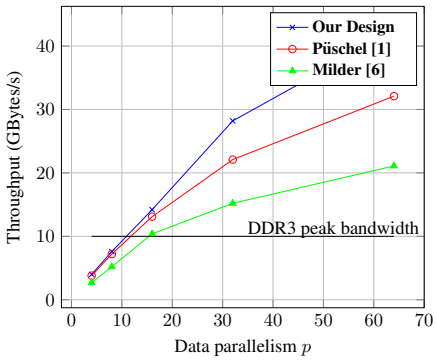



**Fig. 9**: Throughput for various $p$

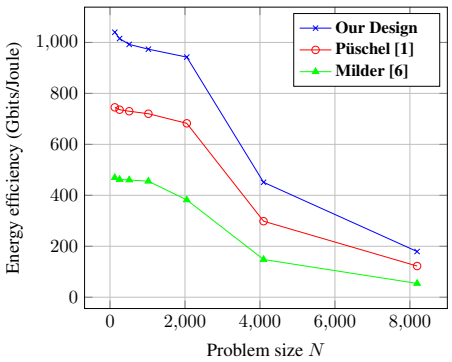



**Fig. 10**: Energy efficiency for various $N$

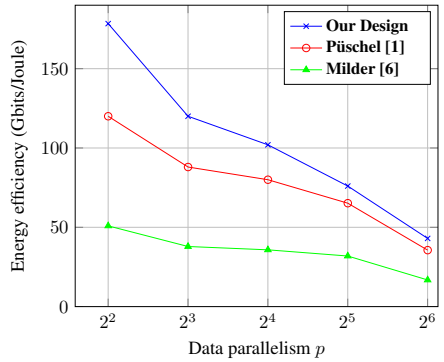



**Fig. 11**: Energy efficiency for various $p$

requirements of the design. It is shown that the number of BRAMs is not affected by $N$ for our design in Fig. 6d. This is due to the fact that a fixed number of independent memory blocks are always required for $p = 16$. For example, 16 memory blocks are needed in our design for $p = 16$, a single BRAM can meet the memory requirement of a memory block when $256 \leq N \leq 8192$. Although the experimental results for various $p$ and $N$ are not presented here, we can expect the BRAM consumption to vary similarly when increasing $N$ or $p$. The difference would be the $N$ value at the turning point where the BRAM number begins to increase with $N$. Note that the maximum problem size $N$ supported by our design is determined by the maximum memory size supported by the device.

Fig. 7a and Fig. 7b present the LUT consumption for various $p$ with $N = 1024$ and $N = 8192$, respectively. A significant amount of LUTs are consumed in [6] as distributed RAMs have been employed. In our design and [1], LUTs are mainly consumed by the connection networks, configuration tables, and the AGU. Therefore, the figures actually demonstrate how $N$ and $p$ affect the LUT consumption of the control logic and the mux. The figures show that the total amount of LUTs consumed increases significantly with $p$. This is because each $2 \times 2$ switch in the connection networks needs to process 32-bit numbers. It also matches with the theoretical result that the number of required mux is $2p \log p$ (see Table. 1). Fig. 7c and Fig. 7d show the LUT consumption for various $N$ with $p = 4$ and $p = 16$, respectively. The two figures show that our design reduces the LUT consumption by 22.1%∼67.2% compared with designs in [1] for various $N$. This indicates the control logic generation approach in our design is more efficient than the technique used in [1]. As a result, our later experimental results show significant performance improvement compared with designs in [1] and [6], although it is difficult to quantitatively compare the resource efficiency between our design and [6] as [6] employs different memory implementation approach.

### 4.4. Performance evaluation

We first evaluate the throughput of our designs implemented on FPGA Virtex-7. We assume either BRAM or DRAM can be employed as data memory for storing input data depending on the data set size. As access behavior of data memory is statically known, we assume the data memory bandwidth can be fully utilized. We assume the design throughput is only determined by the data parallelism and the maximum operating frequency reported by the post place-and-route results.

Fig. 8 demonstrates the design throughput for various $N$ with $p = 4$. As the problem size increases, the throughput decreases due to the lower achievable maximum operating frequency by the design. As shown in the figure, our de-

sign is able to sustain 71%∼80% of the DDR3 peak bandwidth, (approximately 10 GBytes/s [13]) for various $N$ with $p = 4$. We employed a balanced pipelining approach for the purpose of trade off between energy efficiency and throughput. Fig. 8 shows that compared with the baselines, the throughput is improved significantly (up to 78%) in our design, especially for large problem sizes. Such performance improvement can be expected considering that less BRAM blocks and LUTs are used in our design. Note that the connection networks can be optimized through pipelining, however, BRAM blocks can not be pipelined and are only available in specific regions of the device.

Similarly, Fig. 9 demonstrates the design throughput for various $p$ with $N = 8192$. As shown in Fig. 6b, the amount of BRAM used starts to increase only when $p \geq 32$. Accordingly, we observe that the throughput almost increases linearly with $p$ when $p < 32$ as shown in Fig. 9. Again, although experimental results for various fixed $N$ are not presented, the main difference is the value of $p$ where the slope starts to decline. Fig. 9 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. This also implies that a potential high memory bandwidth utilization may be achieved if using the emerging 3D stacked memory providing up to 16x more throughput than DDR3 [13].

We also evaluate the energy efficiency of our design. We only focus on the dynamic power consumption of our proposed permutation design and the baselines. Fig. 10 demonstrates the energy efficiency for various $N$ from 128 to 8192 with $p = 4$. It can be observed that the energy efficiency drops significantly when $N = 2048$. 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 the problem size is varied, our design achieves 2.1x∼3.3x improvement in energy efficiency compared with design in [6]. Fig. 11 demonstrates the energy efficiency when varying $p$ with $N = 8192$. As noted in Fig. 7b, the LUT consumption dramatically increases with $p$, the energy efficiency decreases significantly accordingly. As shown in Fig. 11, for various $p$, our design improves energy efficiency by up to 3.1x compared with the baselines.

## 5. CONCLUSION

In this paper, we presented a RAM-based technique to realize any given fixed permutation on streaming data. The novelty of our work is that we develop a resource efficient mapping method using the classic Benes network to obtain the datapath and the control logic of our design. As a result, our proposed approach reduces the memory consumption by up to 4x, while achieving the same throughput compared with the state-of-the-art. We implemented a design automation tool which takes as input the permutation pattern and data parallelism to generate the design in Verilog. Detailed experimental results show that by reducing the BRAM and LUT consumption on FPGA, our design outperforms the baselines with respect to both throughput and energy efficiency. In the future, we plan to work on a design automation framework targeting automatic generation of high throughput energy-efficient designs for data and signal processing kernels such as FFT, sorting, and convolution network on FPGA.

## 6. REFERENCES


[1] 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.

[2] R. Chen, H. Le, and V. K. Prasanna, "Energy efficient parameterized FFT architecture," in *Proc. of IEEE International Conference on FPL*, Sept. 2013, pp. 1–7.

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

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

[5] B. Baas, "A low-power, high-performance, 1024-point FFT processor," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 3, pp. 380–387, 1999.

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

[7] D. Nassimi and S. Sahni, "An optimal routing algorithm for mesh-connected parallel computers," *Journal of ACM*, vol. 27, no. 1, pp. 6–29, Jan. 1980.

[8] R. Chen and V. K. Prasanna, "Energy efficient architecture for stride permutation on streaming data," in *Proc. of IEEE International Conference on ReConFig*, Dec. 2013, pp. 1–7.

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

[10] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 240–249.

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

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

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