Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA*

Ren Chen, Sruja Siriyal, Viktor Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, USA 90089
{renchen, siriyal, prasanna}@usc.edu

ABSTRACT

Parallel sorting networks are widely employed in hardware implementations for sorting due to their high data parallelism and low control overhead. In this paper, we propose an energy and memory efficient mapping methodology for implementing bitonic sorting network on FPGA. Using this methodology, the proposed sorting architecture can be built for a given data parallelism while supporting continuous data streams. We propose a streaming permutation network (*SPN*) by "folding" the classic Clos network. We prove that the *SPN* is programmable to realize all the interconnection patterns in the bitonic sorting network. A low cost design for sorting with minimal resource usage is obtained by reusing one *SPN*. We also demonstrate a high throughput design by trading off area for performance. With a data parallelism of p ($2 \leq p \leq N/\log^2 N$), the high throughput design sorts an N-key sequence with latency $O(N/p)$, throughput (# of keys sorted per cycle) $O(p)$ and uses $O(N)$ memory. This achieves optimal memory efficiency (defined as the ratio of throughput to the amount of on-chip memory used by the design) of $O(p/N)$. Another noteworthy feature of the high throughput design is that only single-port memory rather than dual-port memory is required for processing continuous data streams. This results in 50% reduction in memory consumption. Post place-and-route results show that our architecture demonstrates 1.3x∼1.6x improvment in energy efficiency and 1.5x∼5.3x better memory efficiency compared with the state-of-the-art designs.

Categories and Subject Descriptors

B.0 [**Hardware**]: GENERAL

General Terms

Parallel algorithm, Energy, Performance, Memory

Keywords

Sorting, Bitonic sorting network, Clos network, FPGA acceleration, energy efficiency, memory efficiency

1. INTRODUCTION

Sorting is one of the most fundamental computing problems. It has been widely used in many applications including digital signal processing, biological computing and large-scale scientific computing [4, 6, 15, 19, 24]. With the advent of Big Data, there is tremendous interest in speeding up solutions for sorting either using software or hardware. Parallel comparison-based sorting networks have been widely utilized in practice for hardware implementation. Benefiting from their straight-forward data flow graphs and simple control schemes, these networks are highly desirable for realizing high speed and parallel sorting architectures.

Bitonic sorting is a parallel comparison-based sorting network [6]. Using $O(N \log^2 N)$ comparators it sorts an arbitrary sequence of N inputs in $O(\log^2 N)$ time. This network has been widely employed in hardware implementations for sorting [17, 26, 28]. Although many other parallel sorting networks with $O(\log N)$ depth have been introduced [4, 19], the bitonic sorting network is still one of the most practical solutions. Many parallel software implementations for sorting have been proposed based on bitonic sorting network [13, 21, 22]. These works either improve the sorting algorithms in terms of throughput and latency, or adapt the algorithms to a variety of general purpose parallel architectures such as SIMD or MIMD machines. However, when considering both energy and performance as the key metrics, hardware-based sorting solutions are preferred. A number of VLSI implementations of sorters have been proposed or implemented in hardware [13, 26]. These VLSI sorters are usually evaluated using area×time^2 performance as one of the key metrics. High performance with low I/O bandwidth requirement can be achieved by these designs, while throughput was not considered. Recently, as a trade-off solution between energy and performance, FPGA-based systems with re-programmability have become popular for realizing sorting [15, 16, 17, 20, 25, 28]. Latency, area, and throughput have been used as the key metrics for performance evaluation. High performance implementations have been realized by exploiting the key features of FPGAs.

In this paper, our focus is to map bitonic sorting network onto FPGA, considering performance, energy and memory

*This work was partially supported by the US NSF under grant CCF-1018801 and by the DARPA PERFECT program. Equipment grant from Xilinx, Inc. is gratefully acknowledged.

FPGA'15, February 22–24, 2015, Monterey, California, USA.

http://dx.doi.org/10.1145/2684746.2689068.

efficiency (defined as the ratio of throughput to the amount of on-chip memory used by the design). We develop a mapping methodology utilizing the classic Clos network [14] to perform data movement between the adjacent sorting stages. It is straight forward to map bitonic sorting network onto hardware using cascaded comparators if all the input data are available concurrently. However, for large data sets, this simple approach is not technically feasible due to high routing complexity, area consumption, as well as limited I/O bandwidth. For large data sets, we show how to "fold" the bitonic sorting network as well as the Clos network to construct a sorting architecture with available data parallelism and I/O bandwidth. Our contributions in this work are:

- We propose a mapping approach to obtain a parallel sorting architecture by utilizing Clos network for inter-stage communnication. The sorting architecture is parameterizable with respect to data parallelism, problem size, and data width.
- We prove that the constructed sorting architecture can process continuous data streams without any memory conflicts (concurrent read or write access to more than one word in a single-port memory).
- We propose a fully pipelined streaming permutation network (*SPN*) by mapping the Clos network with a data parallelism smaller than the input problem size. We develop an in-place permutation in time algorithm for *SPN* to process continuous data streams. This algorithm enables the use of single-port memory rather than dual-port (a read port and a write port) memory and reduces the memory consumption by 50%.
- We show that for a given data parallelism of p (a divisor of N), the *SPN* is programmable to realize all the interconnection patterns in the bitonic sorting network, with a low logic overhead.
- We demonstrate the trade-off among throughput, latency and area using two illustrative designs including a high throughput design and a low cost design. Both designs are parameterizable.
- We perform detailed performance analysis, showing that for the high throughput design, assuming the available data parallelism is p ($2 \leq p \leq N/\log^2 N$), the latency for sorting N-key sequence is $O(N/p)$, the throughput (# of keys sorted per cycle) is $O(p)$, and the memory consumption is $O(N)$.
- We conduct detailed experiments on a state-of-the-art FPGA device. Post place-and-route results show that our architecture demonstrates 1.3x∼1.6x improvement in energy efficiency and 1.5x∼5.3x better memory efficiency compared with the state-of-the-art designs.

2. BACKGROUND AND RELATED WORK

2.1 Bitonic Sorting Algorithm

Hardware implementation of bitonic sorting has been extensively studied in the literature, especially in the VLSI area [13, 21, 26]. The key building blocks of a bitonic sorting network are the bitonic merge (BM) networks which rearrange bitonic sequences to be ordered. A bitonic sequence is a sequence with $n_0 \leq ... \leq n_k \geq ... \geq n_{N-1}$ for some k ($0 \leq k \leq N-1$), or a circular shift of such a sequence [6]. Throughout this paper, we use N to denote the size of a data sequence to be sorted. Without losing generality, we assume N is a power of two. For any given bitonic sequence of N keys, we can use a column of $N/2$ comparators to split it into two bitonic sequences of $N/2$ keys each [6]. Fig. 1b shows the first stage of BM network for splitting a 8-key bitonic sequence. By iteratively splitting the sequence to be sorted, a BM network can sort an N-key bitonic sequence into sorted order in $\log N$ stages[1], where each stage consists of $N/2$ comparators. Fig. 1a shows the BM network for sorting a 8-key bitonic sequence. A bitonic sorting network for N-key sequence can be built using two bitonic sorting networks for $\frac{N}{2}$-key sequences and a BM network for N-key bitonic sequence. After recursively applying this rule, a bitonic sorting network built using BM networks needs $(\log N)(\log N + 1)/2$ stages of comparators.

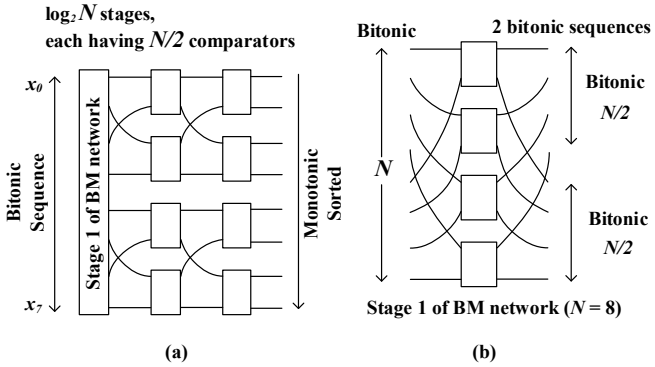

Figure 1: Constructing a bitonic merge network: a) Bitonic merge network for $N = 8$, b) Splitting a bitonic sequence into two bitonic sequences

2.2 Clos Network

Clos network is a multi-stage interconnection network first developed in the 1950s for telephone switching systems [14]. Now Clos network is still widely used in the design of switching systems such as IP routers, data center network, and VLSI interconnection network [14]. Fig. 2a shows the basic structure of a Clos network. r is the number of crossbars in the first and third stages. The number of crossbars in the middle stage is denoted as d. The number of inputs (outputs) of the first (third) stage crossbars is s. $N = sr$ is the network input size. A network is *rearrangeably non-blocking* if an unused input can always be connected to an unused output with the need to rearrange the existing connections [14]. The Clos network is *rearrangeably non-blocking* so long as $d \geq s$ [14]. We will use this result in Section 4.2 to prove the correctness of our proposed mapping approach. Fig. 2b shows a routing example of using the Clos network to perform the permutation $i \rightarrow (i+3) \bmod 9$.

2.3 Hardware-Based Sorting Architectures

A hardware algorithm for sorting N elements with a fixed number of I/O ports is presented in [22]. They extend the column sort algorithm to sort data elements in row-major order. They also develop a multi-way merge algorithm to obtain the sorted sequence. It sorts N elements in $\Theta(\frac{N \log N}{p \log p})$ time using a sorting network of fixed I/O size p and depth $O(\log^2 p)$. In [18], algorithms to reduce communication cost

[1]In this paper, all logarithms are to the base 2.

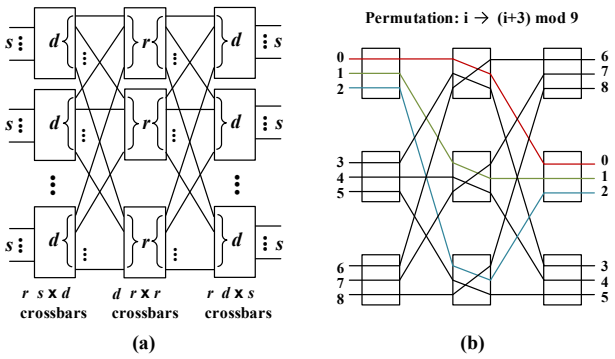

Figure 2: a) Clos network, b) A routing example

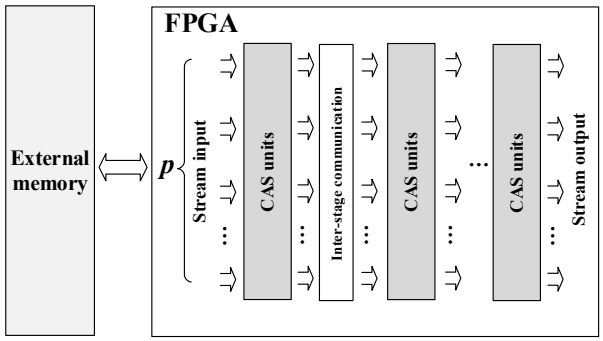

Figure 3: Architectural framework

for bitonic sorting on SIMD and MIMD processors are introduced. They reduce the communication between processors and shared-memory by almost one half when compared with the straight-forward bitonic sorting algorithms.

The area$\times$ time^2 performance of various designs for VLSI sorters is investigated in [26]. Three bitonic sort based VLSI designs are discussed. These designs use $\log N$, $\log^2 N$, and $\sqrt{N \log N}$ processors, and achieve time performance of $O(N \log^2 N)$, $O(N \log N)$ and $O(N/\log N)$, respectively. In [15], the authors present a modular design technique to obtain a high throughput and low latency sorting unit using 65-nm TSMC technology. The proposed sorter is applicable for cases when m largest numbers need to be selected from N-key sequences. In [28], the SPIRAL project develop DSL (Domain Specific Language) to enable mapping sorting algorithms with flexible design choices. Their design supports processing continuous data streams, while energy and memory efficiency are not considered. Several existing sorting architectures on FPGAs are implemented and evaluated in [16]. FIFO or tree based merge sorter as well as bucket sorter are selected as target designs for implementation. They also discuss how to use partial run-time reconfiguration to obtain minimal resource consumption. In [17], a parameterized sorting architecture using bitonic merge network is presented. Their key idea is to build a recurrent architecture of bitonic sorting network to achieve throughput area trade-offs. Hardware designs to perform primitive database operations including selection, merge join and sorting are presented in [9]. High memory bandwidth utilization is achieved by implementing their proposed design on an FPGA-based system. Performance comparison between our design and some of the related work is detailed in Section 6. Some other high performance sorting architectures are developed for platforms other than FPGA [5, 24]. However, it is not clear how to apply their techniques on FPGAs.

3. ARCHITECTURE FRAMEWORK

Problem definition: The sorting problem consists of reordering an N-key sequence. The input sequences are stored in the external memory. With an available data parallelism of p ($2 \leq p \leq N$), p keys are fed into the design in each clock cycle. After a certain amount of time T, the first batch of sorted keys are output, p keys in each clock cycle. With continuous data streams, the throughput is p keys per clock cycle and the latency is T.

Fig. 3 shows the architectural framework within which we propose our mapping approach using FPGA as the target platform. The details of our architectural framework are:

- Data memory: We assume the input consists of several data sequences, each has length of N. External memory is employed to store the inputs.
- Input/output: The input data sequences are fed into the FPGA continuously in a streaming manner. The input data sequences enter the on-chip design at a fixed rate. After a specific delay, the sorted data sequences are output at the same rate.
- CAS units: Compare-and-swap (CAS) units are used for executing the basic operations required by the sorting algorithm. Each CAS unit can be implemented using LUTs and can be pipelined using flip-flops. Fig. 3 shows several stages of CAS units. Each stage consists of $p/2$ CAS units.
- Inter-stage communication: A sorting architecture is composed of cascaded comparison stages. The communication between adjacent stages (inter-stage communication) is performed by permuting data using interconnections and on-chip memories.
- Data parallelism: It is the number of keys processed in parallel each clock cycle in a comparison stage. p (a divisor of N) is used to denote the data parallelism.

4. MEMORY EFFICIENT MAPPING

4.1 Interconnection Patterns

The interconnection patterns in a bitonic sorting network can be represented using stride permutations. A stride permutation can be defined using matrix representation. Given an m-element data vector x and a stride t ($1 \leq t \leq m-1$), the data vector y produced by the stride-by-t permutation over x is given as $y = P_{m,t}x$, where $P_{m,t}$ is a permutation matrix. $P_{m,t}$ is an invertible $m \times m$ bit matrix such that

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

where mod is the modulus operation and $\lfloor\ \rfloor$ is the floor function. For example, $P_{4,2}$ performs $x_0, x_1, x_2, x_3 \rightarrow x_0, x_2, x_1, x_3$; $P_{4,3}$ performs $x_0, x_1, x_2, x_3 \rightarrow x_0, x_3, x_1, x_4$.

Fig. 1a shows that at the first stage of the BM network, to sort N-key sequences, $P_{N,\frac{N}{2}}$ and $P_{N,2}$ are performed at the input and the output respectively. At the output, two bitonic sequences are generated and then permuted using $P_{\frac{N}{2},\frac{N}{4}}$. Therefore, the interconnection pattern at the output can be represented as $(I_2 \otimes P_{\frac{N}{2},\frac{N}{4}}) \cdot P_{N,2} = Q_N$, where

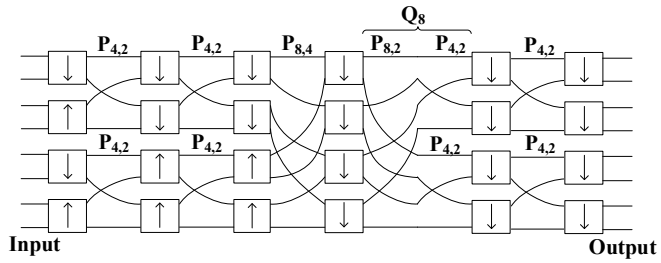

Figure 4: Interconnection patterns in 8-input bitonic sorting network (arrows show the sorting order)

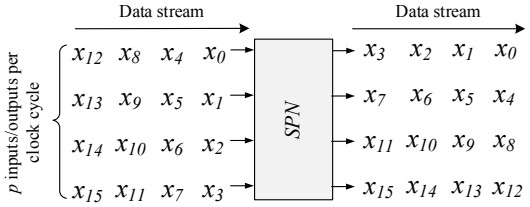

Figure 5: Example: data permutation on streaming data ($N = 16, t = 4, p = 4$)

I_2 is the identity matrix and $\otimes$ is the tensor (or Kronecker) product. By using the divide-and-conquer method discussed in Section 2.1, we therefore obtain a total of $(\log N)(\log N + 1)/2$ interconnection patterns. Note that there are $2 \log N$ unique patterns. All the interconnection patterns can be realized using Clos network. Fig. 4 shows the interconnection patterns in an 8-input bitonic sorting network.

4.2 Mapping the Clos Network

In this section, we will introduce how to map the Clos network into a streaming permutation network (*SPN*) to perform the *permutation on streaming data*, which is defined as: given a hardware block with a data parallelism of p, the N-key input flows into the hardware block over N/p consecutive cycles, after a certain amount of delay, the input is reordered as specified by the required permutation and flows out over N/p consecutive cycles. The object of our mapping approach is to obtain such a hardware block, where multiple single-port memories rather than a p-port memory are employed. Fig. 5 illustrates a streaming version of the stride permutation $P_{16,4}$.

A well known hardware solution to perform stride permutation in bitonic sorting is the delay feedback or delay commutator module widely used in FFT designs [10, 11]. However, using the delay feedback or delay commutator for sorting needs the inputs to be fed in with some particular temporal order. This requirement is easily met for FFT applications as input data is sampled in time. But for sorting this constraint is not necessarily to be met. With limited data parallelism, using the delay feedback or commutator modules cannot fully utilize the high bandwidth provided by the state-of-the-art memory devices [1]. Also the interconnection pattern Q_N composed of three stride permutations makes delay feedback or commutator inefficient.

As shown in Fig. 6, we propose an *SPN* by "folding" the Clos network to perform the data permutations in bitonic sorting. Data permutation on streaming input has been recently studied [23]. Compared with this work, not only we

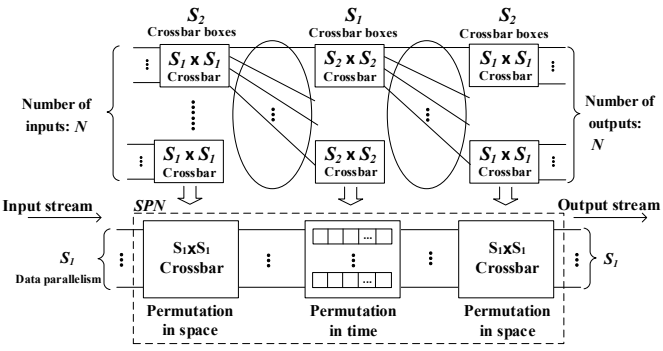

Figure 6: Folding the Clos network into an *SPN*

build the connection between the classic permutation network and the *SPN*, but also our hardware design is memory efficient and uses only single-port memory blocks. Fig. 6 shows the key idea for mapping the Clos network where $d = s = S_1$, $r = S_2$. Assuming the input size is N, and $N = S_1 \times S_2$, a Clos network can be built to be *rearrangeably non-blocking* if $S_2 \geq S_1$ [14]. We can map the Clos network onto an *SPN* with a data parallelism of S_1. We use permutation in space to represent data permutation performed by the crossbar interconnections while permutation in time is defined as permuting temporal order of data elements in a given data sequence. As shown in Fig. 6, in the proposed *SPN*, permutation in space is performed in the first and third stage by two $S_1 \times S_1$ crossbars, and permutation in time is executed in the second stage by S_1 independent single-port memory blocks, each having a memory size of S_2. In the *SPN*, a *memory conflict* is said to occur if concurrent read or write access to more than one word in a single-port memory block is performed in a clock cycle.

THEOREM 4.1. *With a data parallelism of S_1, the proposed SPN can realize any given permutation on streaming input of an N-key data sequence without any memory conflicts using S_1 single-port memory blocks, each of size S_2, where $S_2 \geq S_1$ and $N = S_1 \times S_2$.*

Proof: In the Clos network shown in Fig. 6, where $N = S_1 \times S_2$, each $S_1 \times S_1$ crossbar in the first or third stage has exactly one connection to each of the S_1 $S_2 \times S_2$ crossbars in the second stage. Similarly, there is exactly one connection between each middle stage crossbar and each first or third stage crossbar. This network is *rearrangeably non-blocking* and thus can realize arbitrary data permutation on an N-key data sequence [14]. Clearly, a routing on this network can also be realized by the proposed *SPN* shown in Fig. 6. By reusing the $S_1 \times S_1$ crossbar in the first stage of *SPN* S_2 times, the routing of the first stage in the Clos network can be realized. Likewise, the crossbar in the third stage of *SPN* can also be used to implement the routing of the third stage of the Clos network. Each memory block in the second stage of *SPN* has a memory size of S_2 and can be written by an output of the $S_1 \times S_1$ crossbar in the first stage. In S_2 steps, the S_1 memory blocks in the second stage of *SPN* realize the routing of the second stage of the Clos network. Therefore, the proposed *SPN* simulates the Clos network in Fig. 6, hence proving the theorem. ■

With a data parallelism $p = S_1$, an *SPN* having the control information to perform all the interconnection patterns in bitonic sort is programmable with respect to m, t, and v. m

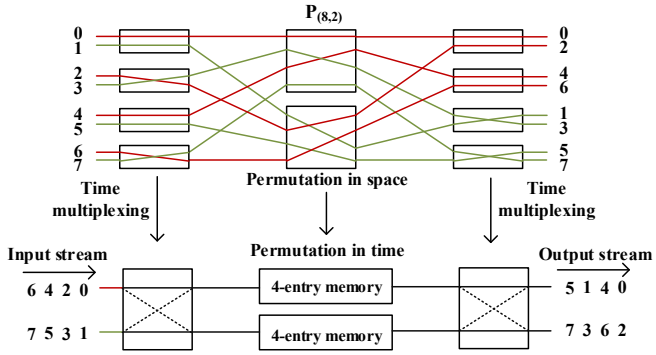

Figure 7: Example: 3-stage mapping for $P_{(8,2)}$ and $p = 2$

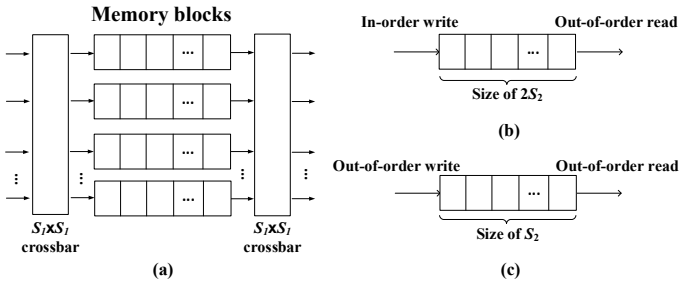

Figure 8: a) *SPN* unit, b) Permutation in time on dual-port (a read port and a write port) memory before data remapping, c) Permutation in time on single-port memory after data remapping

and t represent length of data sequence to be permuted and stride value, respectively. v is used to differentiate $P_{m,t}$ and Q_m. Here programming refers to switching the context of control information to configure m, t, and v during run-time. We denote programmable *SPN* as *SPN*(p, m, t, v). It can be programmed to perform either $P_{m,t}(v = 0)$ or $Q_m(v = 1)$ arising in the bitonic sorting network. Theorem 4.1 states that *SPN(p,m,t,v)* is programmable with respect to m, t and v if $S_1 \times S_2 \geq m$ and $m/S_1 \geq S_1$. Also the latency is m/S_1. In Section 5.3, we will show how we achieve a low cost design for sorting, by programming the *SPN*(p, m, t, v). Fig. 7 shows a mapping example where the Clos network is configured to perform stride permutation $P_{8,2}$. In this example, $N = 8$, $S_1 = 2$ and $S_2 = 4$. To execute $P_{8,2}$, the *SPN* needs two 2×2 crossbars and two 4-entry memories. The control information for the *SPN* can be easily obtained using a routing algorithm for Clos network [14]. This observation is non-trivial as any of the previous optimizations on routing algorithms for Clos network can be reused for realizing *SPN*. In this paper, we adopt a well known routing algorithm for Clos network to obtain all the control information for *SPN* [7]. In the second stage of *SPN*, each memory block can be implemented with single-port memory to permute a single data sequence. However, when processing continuous data streams, dual-port memory is required as concurrent read and write access to different memory locations need to be performed. An algorithm which enables the use of single-port memory for processing continuous data streams is introduced next.

4.3 In-Place Permutation in Time

In this section, we develop an algorithm to perform the permutation in time in the second stage of *SPN*. This algorithm enables the use of single-port memory to process continuous data streams, i.e., processing of p keys from a new data sequence starts immediately after the final set of the previous data sequence enters the sorter. A single-port memory needs to support simultaneous read-write operation (read first) to the same memory location [3]. In the state-of-the-art [23], dual-port memory is required to perform permutation in time on continuous data streams.

4.3.1 In-place algorithm

Fig. 8a shows the overall design of the *SPN* network which consists of two $S_1 \times S_1$ crossbars and S_1 memory blocks (each having S_2 locations). Fig. 8b shows how permutation in time is realized in [23]. A data sequence of length S_2 is written into a memory block over S_2 memory access cycles. To perform in-order write, the data sequence is written with an address sequence $e = (0, 1, ..., S_2 - 1)$, address $e[i]$ is used in the ith $(0 \leq i \leq S_2 - 1)$ clock cycle. Similarly, out-of-order write can be performed using an address sequence resulting from permuting the address sequence e. In-order or out-of-order read operation can be defined likewise. Using this approach, when processing continuous data streams, each memory block is read from and written into simultaneously possibly at different memory locations in each memory access cycle. An S_2-key sequence can be permuted in time either using a dual-port memory of size $2S_2$ or two single-port memories, each of size S_2.

Algorithm 1 In-place permutation in time in the *SPN*

Input: $data_{i,k}$, $addr_{i,j}$
Constants: n_1, n_2

```
{Initialization}
for i = 0 to S1 − 1 do
    Initialize addr_{i,j}
end for
{Permutation in time}
j = 0
for k = 0 to n2 − 1 do
    for i = 0 to S1 − 1, in parallel do
        for h = 0 to S2 − 1 do
            Simultaneous read-write operation:
            Read data_{i,k−1,h} from the memory block i
             with address addr_{i,j,h};
            Write data_{i,k,h} into the memory block i with
             address addr_{i,j,h}.
        end for
        if (j == n1 − 1) then
            j = 0
        else
            j = j + 1
        end if
    end for
end for
{End of the algorithm}
```

To reduce the memory consumption, we remap the data when writing to each memory block so that the read and write operations can be always performed at the same memory location, and the input data sequence is reordered correctly as specified by the permutation in time. Fig. 8c shows the key idea of our proposed *in-place algorithm* for permutation in time. Each memory block is written out-of-order (with a dynamically updated address sequence) rather than in-order. In this way, each memory location is written with

a new value once the old value is read out, thus the proposed permutation in time algorithm is an *in-place algorithm*.

Algorithm 1 shows the details of the proposed *in-place algorithm* in the second stage of *SPN*. We use index i ($0 \leq i \leq S_1 - 1$) to represent the S_1 single-port memory blocks. P_i denotes the permutation (in time) to be performed on the input stream to memory block i. $addr_{i,j,h}$ and $data_{i,k,h}$ represent the addresses and data for accessing the memory block i respectively. j ($0 \leq j \leq n_1 - 1$) is the index of the address sequences, k ($0 \leq k \leq n_2 - 1$) is the index of the data sequences, and h ($0 \leq h \leq S_2 - 1$) is the index of an element in a data/address sequence. $addr_{i,j}$ is calculated based on P_i. n_1 is the number of address sequences required for data remapping. n_2 is the number of data sequences to be permuted. Both n_1 and n_2 are constants (the range of values are discussed next). The above analysis leads to:

THEOREM 4.2. *Any permutation in time on continuous data streams consisting of S_2-key data sequences can be realized using a single-port memory block of size S_2.*

4.3.2 Data remapping overhead

To implement the proposed *in-place algorithm*, address sequences need to be pre-computed for data remapping. Algorithm 1 requires $addr_{i,j}$ to be computed in advance. In hardware implementation, we can either use LUTs on FPGA to store $addr_{i,j}$ or dynamically update the memory address using a customized logic unit. To estimate the data remapping cost, we need to evaluate the range of n_1. Again let P_i represent the permutation to be performed on the memory block i. Assuming $addr_{i,0} = [0, 1, 2, ..., S_2 - 1]^T$, to implement the *in-place algorithm*, we need to iteratively compute $addr_{i,j}$ ($0 \leq j \leq n_1 - 1$) using the following equation:

$$addr_{i,j+1} = P_i \cdot addr_{i,j},\ 0 \leq j \leq n_1 - 2 \qquad (2)$$

such that finally $addr_{i,0} = P_i \cdot addr_{i,n_1-1}$. Our key observation is that n_1 always exsits to be a constant determined by P_i. This leads to:

THEOREM 4.3. *For any given permutation in time, the proposed in-place algorithm requires a constant number of address sequences for data remapping.*

Proof: Consider permutation P_i to be performed on memory block i ($0 \leq i \leq S_1 - 1$). Based on Equation 2, $P_i \cdot addr_{i,n_1-1} = P_i^2 \cdot addr_{i,n_1-2} = ... = P_i^{n_1} \cdot addr_{i,0}$. Thus, we have $P_i \cdot addr_{i,n_1-1} = addr_{i,0}$ when $P_i^{n_1} = I$. Note that some power (a constant) of a permutation matrix is the identity matrix [8]. Therefore, we can always find a constant n_1 such that $P_i^{n_1} = I$. The value of n_1 depends on P_i. The address sequences $addr_{i,0}, addr_{i,1}, ..., addr_{i,n_1-1}$ are the required address sequences for accessing memory block i. ■

As each memory block has a size of S_2, the memory address width is $\log S_2$. According to [8], stride permutation is periodic, i.e., $P_{S_2,t}^{\log S_2} = I$. Thus, when P_i is a stride permutation, the number of address sequences n_1 for memory block i is $\log S_2$. The routing results show that P_i is a stride permutation, a cyclic shift, or a combination of both for all the *SPN*s. For all the above cases, n_1 is $O(\log S_2)$. Based on the analysis above, for $N = 1024$, using $S_1 = 8$ and $S_2 = 128$ in *SPN*, the number of bits required for storing all the address sequences is $\leq \log S_2 \times \log S_2 \times S_2 \times S_1 = 49$ *kbits*. In actual implementation, it is not required to store the entire address sequence as we can update the addresses dynamically using an initial address. Using this approach, the memory needed for storing all the address sequences in an *SPN* for bitonic sort is $\leq \log S_2 \times \log S_2 \times S_1 = 0.38$ *kbits*. Note that, on the state-of-the-art FPGA, each block RAM (BRAM) has a memory capacity of 18 *kbits* [3]. In our implementations, we decide whether to apply the in-place algorithm or not based on the value of S_2 of an *SPN*.

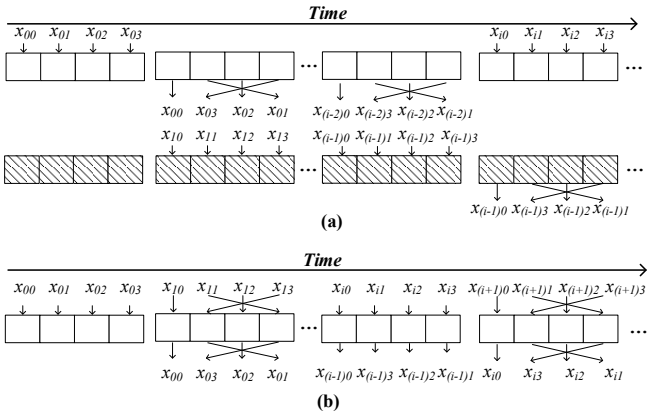

Figure 9: Permutation in time on 4-key sequences: a) using two single-port memories b) using one single-port memory

Fig. 9 illustrates how permutation on continuous data streams is performed. Continuous input data sequences including x_0,x_1,...,x_i, ... (each of length four) are successively permuted temporally. For each data sequence x_j ($j \geq 0$), a permutation of $[x_{j0}, x_{j1}, x_{j2}, x_{j3}] \rightarrow [x_{j0}, x_{j3}, x_{j2}, x_{j1}]$ is performed. Fig. 9a shows the permutation process using two single port memories. As shown in this figure, the two single port memories are alternately read and written during consecutive time periods. For each data element x_{jk}, k represents the index of its time sequence when x_j is written into a memory block. We can see that the data elements in each data sequence are reordered in time. In each cycle, read and write operations are executed concurrently using different memory addresses. Fig. 9b shows performing the permutation using one single port memory using our proposed in-place algorithm. Read and write operations are performed simultaneously using one address in each cycle. When permuting a data sequence, if address sequence $\{0, 1, 2, 3\}$ is used for memory access, then for the next data sequence, address sequence $\{0, 3, 2, 1\}$ will be used. These two address sequences are used alternately for permuting continuous data streams. To dynamically generate the two address sequences, each of size 4, we can employ a 2-bit up counter and a 2-bit down counter. Note that the single-port memory should support concurrent read-write operation (read first) in a memory access cycle.

5. ARCHITECTURE IMPLEMENTATION

In this section, we develop a high throughput design and a low cost design to realize bitonic sorting on FPGA. The high throughput design is fully pipelined and takes advantage of the on-chip distributed (dist.) RAM or BRAM to achieve high performance. With a high data parallelism, it supports processing continuous input data streams without constraints on input temporal order. The low cost design consumes minimal hardware resources by reusing the architectural components. It achieves high resource efficiency by

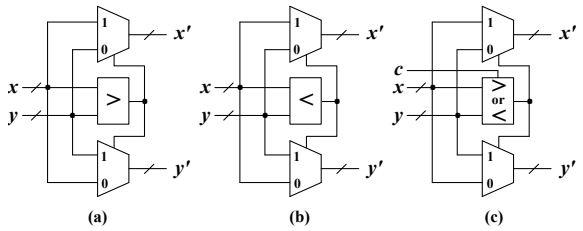

Figure 10: CAS units: for generating a) ascending order, b) for descending order, c) either order

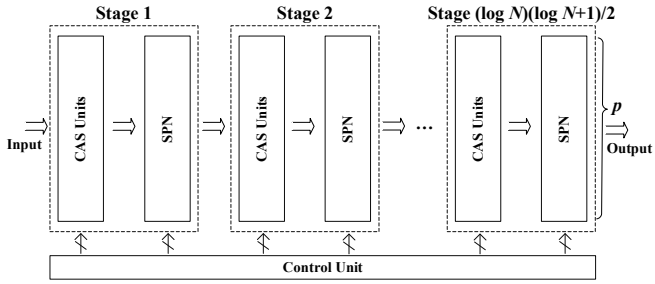

Figure 11: Overall architecture of high throughput design

dynamically reusing one *SPN* to perform all the inter-stage communication. In our designs, we use two basic building blocks: compare-and-swap (CAS) unit and *SPN* unit.

5.1 Architectural Components

5.1.1 CAS unit

A CAS unit is used to compare two input values and swap the values either in ascending or descending order. Three different designs of CAS unit are shown in Fig. 10. Fig. 10a shows the CAS unit used to swap the input values to output in ascending order. Similarly, the design in Fig. 10b is used to produce output values in descending order. Besides, we also design a configurable CAS unit shown in Fig. 10c for resource sharing purpose.

5.1.2 SPN unit

An *SPN* unit is composed of memory blocks and crossbars. As introduced in Section 4.2, data parallelism needs to be fixed before implementing an *SPN* unit. For a given p, we use $SPN(p, m, t, v)$ to denote an *SPN* supporting permutations including $P_{m,t}(v = 0)$ and $Q_m(v = 1)$. In the *SPN*, each memory block can be bound to dist. RAM or BRAM on FPGA. According to [3], BRAM is more power and area efficient than dist. RAM when used for implementing large size memories. However, for a small size memory, it is more power efficient to implement the memory with dist. RAM. Therefore, we empirically perform the memory binding to optimize the memory power consumption based on related experimental results in [10, 12]. Besides, using the proposed in-place permutation in time algorithm, each memory block is implemented using a single-port BRAM which is configured to be in the read-before-write mode [3]; thus data previously stored at the write address appears at the outputs while input data is being stored in the same memory location. Dist. RAM inherently supports this feature. Several LUT-based multiplexers are used to realize a crossbar switch. A logic unit for address generation is designed to dynamically update the memory address for accessing each memory block.

5.2 High Throughput Design

The high throughput design consists of a series of SPN units and CAS units. This design supports sorting of continuous data streams. During sorting, each *SPN* unit or CAS unit keeps on processing continually incoming data streams without interrupts. To build a high throughput design, after the data parallelism ($p = S_1$) is fixed, we need at most $(\log N)(\log N + 1)/2$ stages of *SPN*s, each *SPN* is responsible for executing a specific permutation at that stage. When $m \leq p$, *SPN* is replaced with an m-to-m crossbar. For example, a high throughput design (p=2) for sorting an 8-key sequence needs five *SPN*s and three of them are designed for realizing $P_{4,2}$, other two perform $P_{8,4}$ and Q_8 respectively. Similarly, $(\log N)(\log N + 1)/2$ stages of comparators are required, and each stage needs $p/2$ comparators. Fig. 11 shows the overall architecture of high throughput design.

In the high throughput design, there are $(\log N)(\log N + 1)/2$ stages of *SPN*. Each *SPN* denoted as $SPN(p, m, t, v)$ has its own parameter values for m, t and v. The latency introduced by all the *SPN*s denoted as $T_{SPNs}(N, p)$ can be calculated by

$$T_{SPNs}(N,p) = \sum_{i=\log p}^{\log N-1} \left(\sum_{j=\log p}^{i} \frac{2^{j+1}}{p} + \frac{2^{i+1}}{p} \right) \qquad (3)$$

which is $(6(N - p) - 2\log(N/p))/p$. As the latency introduced by CAS units is $O((\log p)\log^2 N)$, the entire latency of the high throughput design is $O(N/p)$ $(2 \leq p \leq N/\log^2 N)$. Similarly, we can show that the memory consumed by all the *SPN*s is $O(N)$. Furthermore, the high throughput design can be pipelined to process continuous data streams, resulting in a throughput of p. To achieve a high throughput, a pipeline stage can be inserted after each of the $(\log N)(\log N+1)/2$ comparison stages. The area consumption is $O(p\log^2 N)$ and the interconnect complexity is $O(p)$. When using external memory as data memory, the required number of I/Os is $O(p)$. Each stage consists of $p/2$ CAS units. No control bits are needed for CAS units shown in Fig. 10a and Fig. 10b. Each CAS unit shown in Fig. 10c requires only one control bit. The total number of control bits for all the CAS units is $O(\log^2 N)$. In $SPN(p, m, t, v)$, two p-to-p crossbars require $O(p \log p)$ control bits.

5.3 Low Cost Design

Fig. 12 shows the architecture of the low cost design. The low cost design cannot support processing continuous data streams, thus the proposed in-place permutation in time algorithm is not applicable. For a given data parallelism p, the low cost design requires an $SPN(p, m, t, v)$ and $p/2$ CAS units. During sorting, $SPN(p, m, t, v)$ is reused by programming m, t and v so that all the permutations $P_{m,t}$ and Q_m for the sorting problem can be realized. In this way, this design achieves the highest resource efficiency at the expense of throughput. Based on the introduction in Section 4.1, only $2\log N$ different interconnection patterns exist between the $\log^2 N$ comparison stages in the bitonic sorting network. Thus $2\log N$ states are introduced for control.

To complete execution of one comparison stage in the bitonic sorting network, $p/2$ CAS units are reused N/p times. Since the total number of comparison stages is $(\log N)(\log N+$

Table 1: Performance comparison of sorting architectures

Design	Latency	Logic	Memory	Memory type	Throughput[1]	Memory throughput ratio[4]	Support for continuous data streams
[9]	$o((N \log p)/p)$	$o(p \log N)$	$o(Np)$	Dual-port and p-port	$o(p)$	$2N + o(N)$	No
[22]	$o(\frac{N \log N}{p \log p})$	$o(p \log^2 p)$	$o(N)$	Dual-port	$o(\frac{p \log p}{\log N})$	$o(\frac{N \log N}{p \log p})$	No
[16]	$o(N)$	$o(\log N)$	$2N + o(N)$	Dual-port	$o(1)$	$2N + o(N)$	Yes
SPIRAL [28][2]	$6N/p + o(N/p)$	$o(p \log^2 N)$	$12N + o(N)$	Dual-port	$o(p)$	$12N/p + o(N)$	Yes
[26][3]	$3N + o(N)$	$o(\log^2 N)$	$6N + o(N)$	Dual-port	o(1)	$6N + o(N)$	Yes
High throughput design	$6N/p + o(N/p)$	$o(p \log^2 N)$	$6N + o(N)$	Single-port	$o(p)$	$6N/p + o(N)$	Yes
Low cost design	$o(N \log^2 N/p)$	$o(p)$	$o(N)$	Dual-port	$o(p/\log^2 N)$	$o(N \log^2 N/p)$	No

[1] Throughput $\neq$ 1/Latency as all N data elements are sorted concurrently or when processing continuous data streams
[2] The highest throughput design in their work, [3] $O(\log^2 N)$-processor bitonic sort, [4] The reciprocal of the memory efficiency (used for asymptotic analysis)

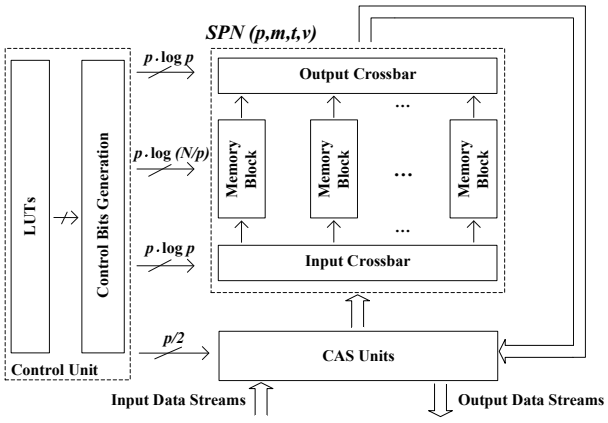

Figure 12: Overall architecture of low cost design

1)/2, the low cost design has a latency of $O((\log^2 N)N/p)$ for sorting. Before the execution of next stage, the intermediate results of the current stage needs to be stored. Thus, the required on-chip memory size is exactly N. The CAS units are programmed to generate an ascending order or a descending order of the inputs based on the comparison results. To complete each comparison stage, the CAS units needs to be programmed N/p times. Benefiting from the symmetric property of bitonic sorting network, only one state machine with $\log N$ states for updating $p/2$ bits is required to program all the CAS units. Therefore, the programming overhead with respect to memory bits for CAS units is $O((p/2) \log N)$. Similarly, $SPN(p, m, t, v)$ needs to be reused $(\log N)(\log N + 1)/2$ times to perform all the comparison stages. For accessing each memory block in the $SPN(p, m, t, v)$, a $\log(N/p)$-bit counter is required for read, and p state machines (each having $2 \log N$ states) for updating $p \log(N/p)$ control bits are realized for write. Similarly, to program the crossbars in the *SPN*, two state machines (each having $2 \log N$ states) for updating $p \log p$ control bits are required. As a result, the programming overhead of the $SPN(p, m, t, v)$ is $O((p \log N) \log(N/p))$.

6. EXPERIMENTS

6.1 Experimental Setup

Both the high throughput design and the low cost design were implemented on Virtex-7 FPGA (XC7VX690T, speed grade -2L). This device has 2940 BRAMs (each 18 *kbits*) and 108300 slices. The designs were synthesized and placed-and-routed by Vivado 2014.2 [2]. Post place-and-route simulations were conducted for behavior and timing verification. We created input test vectors having an average toggle rate of 50% for simulation. We used SAIF (switching activity interchange format) files as inputs to Vivado power analysis tool to produce accurate power dissipation estimation [3].

6.2 Performance Metrics

We consider the following metrics:

- Throughput: is defined as the number of bits sorted per second (Gbits/s). The *throughput* is computed as the product of number of keys sorted per second and data width per key.
- Energy efficiency (or power efficiency): is defined as the number of bits sorted per unit energy dissipated (Gbits/Joule) by the design and is calculated as the throughput divided by the average power consumed by the design.
- Memory efficiency: measured as the throughput achieved divided by the amount of on-chip memory used by the design (in bits).

6.3 Performance Evaluation

In this section, we use HT Design to denote the high throughput design. LUT-L and LUT-M represent LUTs in logic and LUTs in memory, respectively. We configure the data parallelism p as four in all our implementations.

6.3.1 Asymptotic analysis

Table 1 presents an asymptotic analysis of the performance of various sorting architectures. The details of some of the prior designs are introduced in Section 2.3. The proposed HT Design is one of the designs achieving a linear time complexity (latency) which decreases with the available data parallelism p. We also show the constants with little o notation in the asymptotic expressions for the sake of comparison [27]. The table shows that the memory throughput ratio (the reciprocal of memory efficiency) of the HT Design is $6N/p + o(N)$. When $p \geq 4$ and $N \geq 128$, the HT Design outperforms all the other designs with respect to memory efficiency. Moreover, benefiting from the proposed in-place permutation in time algorithm, the HT Design uses only single-port memory to process continuous data streams.

6.3.2 Results of design points

We employ a baseline architecture implemented using the HT Design without applying the proposed in-place permutation in time algorithm discussed in Section 4.3. We fix

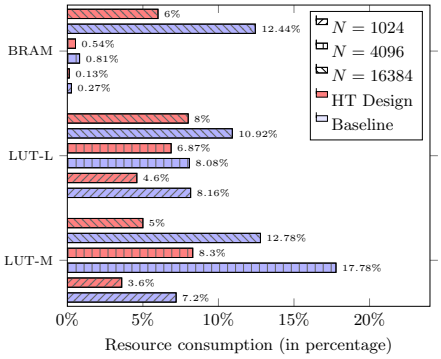

Figure 13: Memory and logic resource used by the HT Design and the baseline

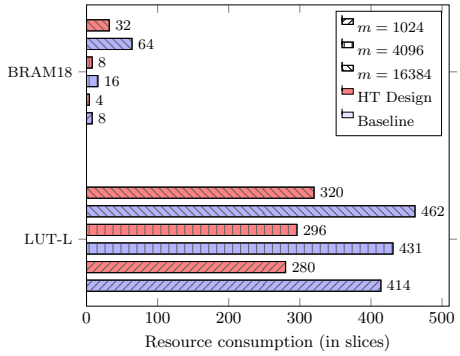

Figure 14: Memory and logic resource used by *SPN*

the data width as 32-bit for the HT Design and the baseline architecture. For both of them, we evaluate the amount of BRAM, LUT-L, LUT-M consumed for problem sizes $N = 1024, 4096$, and 16384. The results are shown in Fig. 13. In this plot, the available amount of BRAMs or LUTs is normalized to one on the x-axis. The red bars and blue bars show the resource consumption of the HT Design and the baseline, respectively. The problem size is differentiated by the bar fill pattern. The figure shows that the consumption of both BRAM and LUT-M nearly doubles for the baseline for all the problem sizes. The reduction in memory usage is especially significant for $N = 16K$. Moreover, the figure shows that the utilization of LUT-L is also reduced in the HT Design for the selected problem sizes. This shows that as dual-port memory is eliminated and the total memory size is halved, LUT-L needed for implementing memories is reduced significantly. Thus, it implies that the logic overhead for implementing the proposed *in-place algorithm* is almost negligible, and it demonstrates the superiority of our proposed in-place algorithm. The figure shows that the LUT-M consumption for $N = 4096$ is more than that for $N = 16384$. The reason is that most of the memory blocks are implemented using BRAMs when $N = 16384$.

We also evaluate resource consumption of $SPN(p, m, t, v)$ alone, for both the HT Design and the baseline. The results are presented in Fig. 14. p, t, v are fixed as 4, 2, and 0 respectively. We selected m as 1024, 4096, and 16384 in the experiments for illustration. The number of BRAM18 (18kb BRAM) and LUT-L are shown using bars on y-axis. Considering the selected values of m, the memory blocks in *SPN* are all implemented using BRAMs, thus there is no LUT-M used. Clearly it shows that the the number of

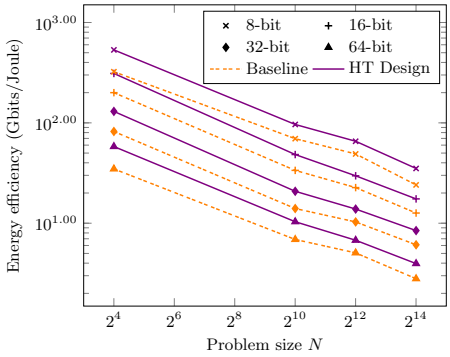

Figure 15: Energy efficiency for various problem sizes

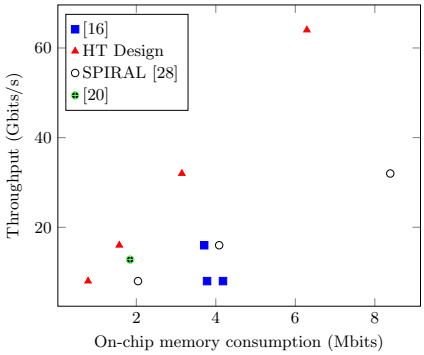

Figure 16: Memory efficiency comparison of various designs

BRAM18 is exactly halved for all the selected problem sizes. Besides, the elimination of dual-port memory also results in reduction in the amount of LUT-L used; up to 32% logic reduction is achieved. The above results show that using our proposed mapping methodology, the data permutations in bitonic sorting network can be performed efficiently with respect to memory and logic consumption.

We further evaluate the energy efficiency of HT Design and the baseline for $N = 16$, 1024, 4096, and 16384. The operating frequency is fixed at 250 MHz for the sake of power evaluation. All our designs were pipelined to achieve this clock rate. The data width is varied from 8-bit to 64-bit. The experimental results are presented to demonstrate the benefit of the proposed in-place permutation in time algorithm for sorting from a power point of view. Fig. 15 shows that the energy efficiency of both designs are sensitive to data width. This is because we create testbench with 50% toggle rate for each design. When the data width increases, the switching activity of the designs increases significantly. The results also show that as the data width and problem size are varied, the HT design achieves 33%∼67% improvement in energy efficiency. In the experiments, we aggressively optimized speed for both designs by inserting pipeline stages after each comparison stage. We believe more energy efficiency can be gained by trading off the speed.

6.3.3 *Performance comparison*

Fig. 16 presents a scatter plot comparing the design points of our work with several prior work. The design points labeled [16] are developed for sorting $43K$-key or $21.5K$-key data sequence. The design points labeled HT Design and SPIRAL [28] are realized for sorting $16K$-element data sequence. The design in [20] can process up to 250 MB data

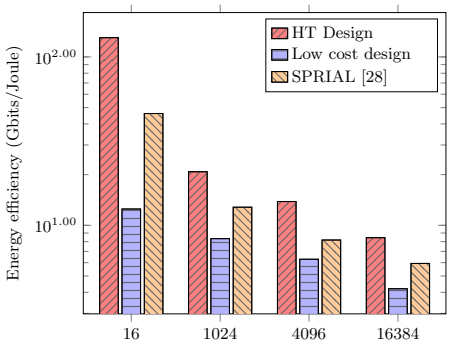

Figure 17: Energy efficiency comparison

sets consisting of 8-key data sequences. An embedded system based sorting solution is presented in [20]. The x-axis represents the on-chip memory consumption (in Mbits) of a design point, and the y-axis represents the throughput achieved by the design. Design points closer to the upper left corner of the plot achieve higher throughput with less on-chip memory. In Fig. 16, all our designs are dominating designs: for every design in the literature considered in this evaluation, one of our designs offers superior throughput or memory efficiency or both. Our designs achieve 2.3x∼5.3x better memory efficiency compared with [16]. Our best design provides 2.6x and 1.5x better memory efficiency compared with SPIRAL and [20], respectively.

We also compare both the HT Design and the low cost design with the designs developed by the SPIRAL project [28] with respect to energy efficiency. Their tool automatically generates customized sorting soft IP cores in synthesizable RTL Verilog with user specified parameters, including problem size, data parallelism, data precision, etc. For the sake of illustration, the operating frequency is set to 250 MHz for power evaluation and the data parallelism was set to four. Energy efficiency of our design is compared against that of the soft IP cores having the same data parallelism. The problem size is chosen to be 16, 1024, 4096 and 16384. As shown in Fig. 17, for various N, the HT Design improves energy efficiency by up to 1.6x. The results also show that the low cost architecture consumes the most amount of energy. The reason is that a considerable amount of energy is consumed by the path connecting the design and I/O ports. We believe the low cost design can achieve a much higher energy efficiency if implemented using VLSI technologies.

7. CONCLUSIONS

In this work, we presented an energy and memory efficient mapping of bitonic sorting on FPGA. We proposed a streaming permutation network which is programmable to perform all the data permutations in the bitonic sorting network. We constructed two illustrative designs: a high throughput design and a low cost design. We developed an in-place permutation in time algorithm so that the high throughput design is able to employ single-port memory to process continuous data streams. Experimental results show that our design dominates all other designs in the literature with regard to memory efficiency. The proposed mapping approach can be used to generate high performance designs to optimize latency, throughput and energy efficiency. In the future, we plan to work on an accurate performance model for energy-efficiency estimation, which can be used for design space exploration to obtain power optimized sorting architectures with various constraints.

8. REFERENCES

[1] DDR4 SDRAM. `http://www.micron.com/products/dram/ddr4-sdram`.

[2] Vivado design suite user guide: design flows overview. `http://www.xilinx.com/support/documentation/`.

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

[4] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. of ACM STOC*, pages 1–9. ACM, 1983.

[5] K. Andryc, M. Merchant, and R. Tessier. FlexGrip: A soft GPGPU for FPGAs. In *Proc. of IEEE FPT*, pages 230–237, Dec 2013.

[6] K. E. Batcher. Sorting networks and their applications. In *Proc. of AFIPS*, pages 307–314. ACM, 1968.

[7] V. E. Benes. Permutation groups, complexes, and rearrangeable connecting networks. *Bell System Technical Journal*, 43(4):1619–1640, 1964.

[8] R. A. Brualdi. *Combinatorial matrix classes*, volume 13. Cambridge University Press, 2006.

[9] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *Proc. of ACM/SIGDA FPGA*, 2014.

[10] R. Chen, H. Le, and V. K. Prasanna. Energy efficient parameterized FFT architecture. *In Proc. of IEEE International Conference on FPL*, 2013.

[11] R. Chen, N. Park, and V. K. Prasanna. High throughput energy efficient parallel FFT architecture on FPGAs. In *Proc. of IEEE International Conference on HPEC*, pages 1–6, 2013.

[12] R. Chen and V. K. Prasanna. Energy-efficient architecture for stride permutation on streaming data. In *Proc. of IEEE International Conference on ReConFig*, pages 1–7, Dec 2013.

[13] M. V. Chien and A. Y. Oruc. Adaptive binary sorting schemes and associated interconnection networks. *IEEE TPDS*, 5(6):561–572, 1994.

[14] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[15] A. Farmahini-Farahani, H. Duwe, M. Schulte, and K. Compton. Modular design of high-throughput, low-latency sorting units. *IEEE TC*, 62(7):1389–1402, July 2013.

[16] D. Koch and J. Torresen. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proc. of ACM/SIGDA FPGA*, pages 45–54, 2011.

[17] C. Layer, D. Schaupp, and H.-J. Pfleiderer. Area and throughput aware comparator networks optimization for parallel data processing on FPGA. In *Proc. of IEEE ISCAS*, pages 405–408, May 2007.

[18] J.-D. Lee and K. Batcher. Minimizing communication in the bitonic sort. *IEEE TPDS*, 11(5):459–474, May 2000.

[19] T. Leighton. Tight bounds on the complexity of parallel sorting. In *Proc. of ACM STOC*, pages 71–80, 1984.

[20] R. Mueller, J. Teubner, and G. Alonso. Sorting networks on FPGAs. *International Journal on VLDB*, 21(1):1–23, 2012.

[21] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE TC*, 100(1):2–7, 1979.

[22] S. Olarlu, M. C. Pinotti, and S. Q. Zheng. An optimal hardware-algorithm for sorting using a fixed-size parallel sorting device. *IEEE TC*, 49(12):1310–1324, 2000.

[23] M. Püschel, P. A. Milder, and J. C. Hoe. Permuting streaming data using rams. *Journal of the ACM*, 56(2):10:1–10:34, 2009.

[24] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced and energy-efficient large-scale sorting system. *ACM TOCS*, 31(1):3, 2013.

[25] V. Sklyarov, I. Skliarova, D. Mihhailov, and A. Sudnitson. Implementation in FPGA of address-based data sorting. In *Proc. of IEEE FPL*, pages 405–410. IEEE, 2011.

[26] C. Thompson. The VLSI complexity of sorting. *IEEE TC*, C-32(12):1171–1184, Dec 1983.

[27] J. D. Ullman, A. V. Aho, and J. E. Hopcroft. The design and analysis of computer algorithms. *Addison-Wesley, Reading*, 4:1–2, 1974.

[28] M. Zuluaga, P. Milder, and M. Puschel. Computer generation of streaming sorting networks. In *Proc. of ACM/EDAC/IEEE DAC*, pages 1241–1249, June 2012.