A Hybrid Design for High Performance Large-scale Sorting on FPGA

Ajitesh Srivastava
Department of Computer Science
University of Southern California
Los Angeles, USA 90089
Email: ajiteshs@usc.edu

Ren Chen, Viktor K. Prasanna and Charalampos Chelmis
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, USA 90089
Email: {renchen, prasanna, chelmis}@usc.edu

Abstract—Sorting is a key kernel in numerous big data application including database operations, graphs and text analytics. Due to low control overhead, parallel bitonic sorting networks are usually employed for hardware implementations to accelerate sorting. Although a typical implementation of merge sort network can lead to low latency and small memory usage, it suffers from low throughput due to the lack of parallelism in the final stage. We analyze a pipelined merge sort network, showing its theoretical limits in terms of latency, memory and, throughput. To increase the throughput, we propose a merge sort based hybrid design where the final few stages in the merge sort network are replaced with "folded" bitonic merge networks. In these "folded" networks, all the interconnection patterns are realized by streaming permutation networks (SPN). We present a theoretical analysis to quantify latency, memory and throughput of our proposed design. Performance evaluations are performed by experiments on Xilinx Virtex-7 FPGA with post place-and-route results. We demonstrate that our implementation achieves a throughput close to 10 GBps, outperforming state-of-the-art implementation of sorting on the same hardware by 1.2x, while preserving lower latency and higher memory efficiency.

I. INTRODUCTION

A fundamental kernel in many applications in database operations, graphs, text analytics, signal processing, and biological computing is sorting [1], [2], [3], [4], [5]. Merge sort and bitonic sort [2] have both been implemented on parallel architectures. Merge sort can sort a list of N elements using a merge tree of depth $\log N$, however at the root of the tree no parallelism can be exploited due to serial nature of the algorithm. Bitonic sort, however, is a more practical solution to exploit high parallelism [6], [7], [8], which can sort N elements using $\log^2 N$ stages in $O(\log^2 N)$ time using $O(N \log^2 N)$ comparators. FPGAs offer a desirable platform for implementation of sorting architectures due to effective trade-off between energy and performance. The easy reconfiguration capabilities of FPGAs has inspired several sorting network architectures to be embedded in hardware [3], [9], [6], [10], [11], [8] . However, as the problem size N becomes large it becomes more difficult to realize the entire network on hardware due to limited amount of resources on FPGA. In the era of Big Data, due to ever-growing problem size, streaming architectures have come into picture. Therefore stream-oriented approach becomes necessary, which are typically characterized by small stream payload, access to

small local memories and high data rate flow. Achieving a high throughput in these architectures, and hence, high bandwidth utilization is a primary goal along with low-latency, memory-efficiency and energy-efficiency.

An implementation of merge sort on FPGA is done in [12], where the acceleration of sorting is aimed to speed up join operations in databases. It consists of pipelined stages, where each of the earlier stages have access to two buffers. A merge unit in each stage is responsible for reading p elements from each buffer and output the first p sorted elements of $2p$ elements. The throughput of a typical pipelined merge sort implementation is bottlenecked by the throughput of of final stage in the pipeline. To deal with this limitation, they use a high bandwidth node at the final stages. They have shown 5.7x improvement over software in terms of bandwidth utilization. Although, high throughput is reported, there is no theoretical guarantee of getting highest possible throughput. Recently, an FPGA implementation of bitonic sort [13] was shown to achieve a throughput of p for data parallelism p and problem size N, while requiring memory $6N$ and latency $6N/p$. The design was built upon a fully pipelined streaming permutation network folded to accommodate sorting problem size that is larger than data parallelism. Inspired by [12] and [13], we set forth to find a provably optimal architecture that will optimize throughput, and provide theoretical guarantees of the efficiency of our proposed solution.

Given N elements to be sorted coming in a streaming fashion, and a data parallelism of p, we propose to sort these elements using a hybrid design. Without loss of generality, we assume both N and p are powers of 2. In this design the earlier stages consist of serial merge units and final few stages consist of bitonic merge unit. The hybrid design enables high throughput due to the bitonic merge units while preserving the low logic and memory consumption of serial merge. The contributions of the paper are the following:

- We prove some fundamental limits of pipelined implementation of serial merge sort.
- We present an analysis of the proposed high throughput hybrid design and find the optimal number of serial merge and bitonic merge stages.
- We show that our design outperforms the state-of-the-art [13] for all practically achievable data-parallelism in

terms of latency and memory consumption, and produces the optimal throughput.

- We demonstrate through experiments that our design produces a higher throughput, while utilizing fewer BRAMs than state-of-the-art design, thus providing a more memory-efficient solution at a lower latency.

II. ARCHITECTURE

Problem definition: The sorting problem consists of re-ordering an N-element 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.

Before we propose our architecture, we explore the theoretical limitations of a simple merge sort network, that would later assist us in making certain design decisions to engineer lower latency, higher throughput, and memory-efficient design.

A. Serial Merge: Fundamental Limits

We start by presenting an analysis of an architecture for serial merge sort for sorting n elements (for sorting the entire sequence, $n = N$) with data parallelism p. The logical view of the merge sort architecture is shown in Figure 1. Every stage has access to two memory buffers from where the elements are read, sorted and written to the buffers at the lower stage. We assume that a merge unit in each stage can read two sorted lists and output the first p sorted elements of the merged list in $f(p)$ cycles. For the low throughput design in [12], $f(p) = 1 + \log p$ [1]. Also to start merging, "stage-0" provides p sorted elements to the input buffers of stage-1. Assume that this is done in $g(p)$ clock cycles. For instance, if bitonic sort is used to sort the p elements, then $g(p) = (\log p)^2$. We build our analysis on the following lemma.

Lemma 1: The smallest p elements of two non-empty sorted lists of which the first x and y elements are known, can be found correctly if $\min\{x, y\} \geq p$.
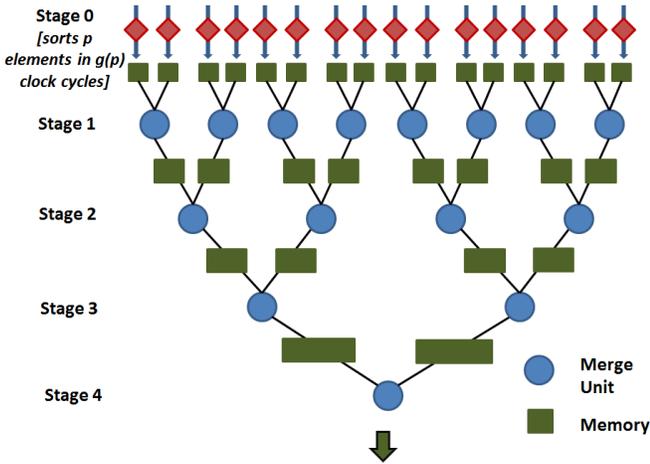

Fig. 1. Overview of the merge sort tree, with each merge unit capable of reading and writing p elements at a time.

[1]All logarithms in this paper are to the base 2.

In a hypothetical implementation, suppose we have a processing element (PE) at every stage that can operate on one merge unit at a time, traveling from left to right in the tree reading and spitting sorted elements to be read by the next stage. Consider a merge unit that is responsible for merging two lists of size $n/2$ into a single list of size n. Since, the PE in the previous stage is going from left to right, the merge unit in the current stage cannot start while the left incoming neighbor is not finished, as the right list would be empty (See Lemma 1). It follows from Lemma 1 that this merge unit can start as soon as the right incoming unit outputs p elements. Therefore main observation is that this merge unit starts to output the first p elements $f(p)$ cycles after the right incoming unit spits its first p elements. Let $T_S(k, i)$ be the latency to output first p elements by the ith merge unit from the left in the kth stage. Then, it follows that $T_S(k, i) = T_S(k - 1, 2i) + f(p)$.

Let K be the final stage. It is easy to see that $K = \log \frac{n}{p}$. We are interested in finding $T_S(K, 1)$.

$$T_S(K, 1) = T_S(K - 1, 2) + f(p)$$
$$= T_S(K - 2, 4) + 2f(p) = T_S(0, 2^K) + Kf(p). \quad (1)$$

Note that at stage-0, i^{th} unit outputs p elements after $i - 1$ units are finished, and it has also sorted p elements. Since, at this stage the sorting takes $g(p)$ cycles for every set of p elements, $T_S(0, i) = ig(p)$. Hence,

$$T_S(0, 2^K) = 2^K g(p) = \frac{n}{p} g(p). \quad (2)$$

From Equations 1 and 2, we get

$$T_S(K, 1) = \frac{n}{p} g(p) + f(p) \log \frac{n}{p}. \quad (3)$$

In fact, we can prove that the latency in Equation 3 is the minimum in the merge sort architecture under our assumptions. Again, we invoke Lemma 1, and state that for any given order of processing:

$$T_S(k, i) = \max\{T_S(k - 1, 2i - 1), T_S(k - 1, 2i)\} + f(p). \quad (4)$$

Therefore,

$$T_S(K, 1) = \max\{T_S(K - 1, 2i - 1), T_S(K - 1, 2i)\} + f(p)$$
$$= \max\{T_S(K - 2, 4i - 3), T(K - 2, 4i - 2),$$
$$T_S(K - 2, 4i - 1), T_S(K - 2, 4i)\} + 2f(p)$$
$$= \max_{j-1}^{2^K}\{T_S(0, j)\} + Kf(p). \quad (5)$$

$\max_{j-1}^{2^K}\{T_S(0, j)\}$ is the latency of that node where the merge is performed last in stage-0, which is equivalent to Equation 2. Hence, the latency given by Equation 3 is in fact the optimal.

For the memory requirement, we do not need as many memory buffers for implementation in each stage as shown in Figure 1. We note that a merge unit responsible of merging two lists of size $n/2$ would require a memory of $n/2 + p$ (from Lemma 1). From the data streaming through the tree, as soon as the merge unit observes these many elements it

can consume p, making room for p elements coming from the previous stage. Therefore, stage-1 requires $p+p$ memory, stage-2 requires $2p+p$, and so on, until stage-K which requires $2^{K-1}p + p$. Therefore, total memory requirement

$$
\begin{aligned}
M_S(n,p) &= (p+p) + (2p+p) + \ldots (2^{K-1}p+p) \\
&= p(1 + 2 + 4 + \cdots + 2^{K-1}) + Kp \\
&= p(2^K - 1) + Kp \\
&= p(\frac{n}{p} - 1) + p\log\frac{n}{p} = n + p\left(\log\frac{n}{p} - 1\right) \quad (6)
\end{aligned}
$$

Since each merge unit outputs p elements in every $f(p)$ clock cycles, the throughput is given by $\frac{p}{f(p)}$.

Now, if we utilize this merge-sort architecture to sort N elements using data parallelism of p, the latency, memory and throughput are approximately $Ng(p)/p$, N, and $p/f(p)$, respectively. While the design is memory efficient, the throughput is low. To improve the throughput, we propose a hybrid design in the following subsection.
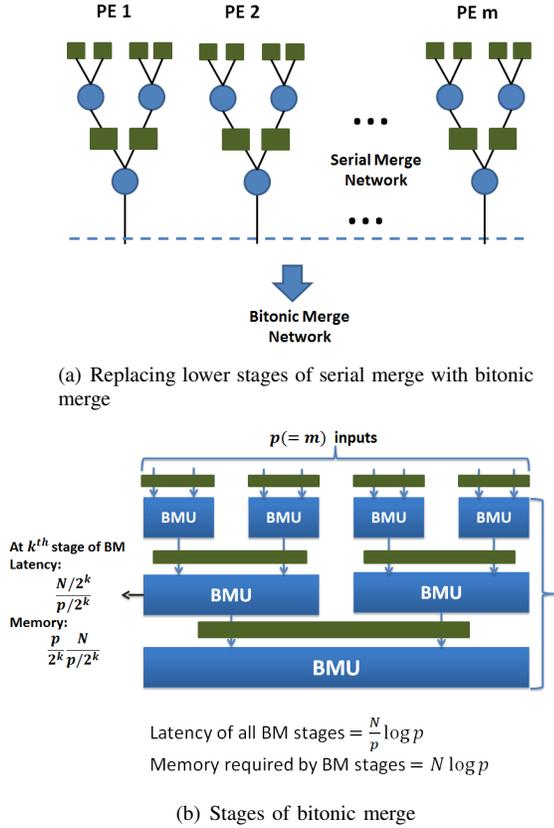

(a) Replacing lower stages of serial merge with bitonic merge

(b) Stages of bitonic merge

Fig. 2. The merge sort tree with a bitonic merge network in the lower stages.

B. Hybrid Network Design

The merge network has a low throughput (Equation **??**), compared to the data-parallelism p. To improve the throughput, we propose to split the sequence in m parts and sort them in parallel. This would require m PEs per stage. The outputs of the merge subtrees can be fed to a bitonic merge network. As p is the total data-parallelism, each of the subtrees has a data-parallelism of p/m. Since each subtree delivers p/m
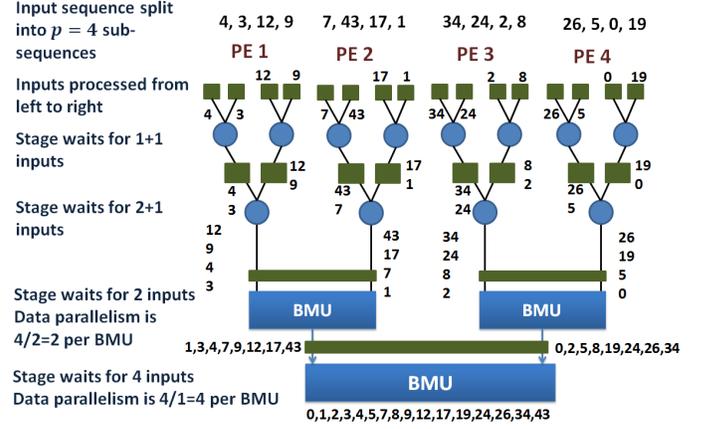

Fig. 3. An example of sorting a sequence of size $N = 16$ with data parallelism $p = 4$ through Hybrid Design.

elements every $f(p/m)$ clock cycles, the input to the bitonic merge network is received at $p/f(p/m)$ elements per cycles. The input rate is maximized when $p/m = 1$ (and therefore $f(p/m) = 1$ as it takes 1 cycle to find the maximum of two elements). Therefore $m = p$. Each subtree requires approximately $N/m = N/p$ memory and has a latency of $\frac{N/m}{p/m}g(p/m) = N/p$ cycles. Memory required by p subtrees is $pN/p = N$.

A bitonic merge unit that is responsible for merging two sorted sequences of 2^k has $k+1$ sub-stages, which for a naive implementation would require $(k+1)2^{k+1}/p'$, where p' is the data-parallelism available to that unit. However, with proper pipe-lining, the latency can be reduced to $2^{k+1}/p'$ [13]. We build the lower part of the network using these bitonic merge units. The bitonic merge (BM) network has $\log p$ stages, of which the first stage has $p/2$ BM units, second stage has $p/4$, and the final stage has $p/2^{\log p} = 1$ unit. One BM unit which has parallelism $p/2^k$ requires a memory of $\frac{N}{p/2^k}$ and has a latency of $\frac{N/2^k}{p/2^k} = N/p$.

Therefore the total latency and memory consumption of the BM network is given by

$$
T_B = \sum_{k=1}^{\log p} \frac{N}{p} = \frac{N}{p}\log p \quad (7)
$$

And $$M_B = \sum_{k=1}^{\log p} p/2^k \frac{N}{p/2^k} = N\log p. \quad (8)$$

As a result the total latency of our design is $T_S + T_B = (1 + \log p)N/p$ (from Equations 3 and 7) and memory consumption is $M_S + M_B = (1+\log p)N$ (from Equations 6 and 8). Figure 3 demonstrates the sorting through this design for a sequence of size $N = 16$ with data parallelism $p = 4$. The sequence is split into four sub-sequences and each sub-sequence is fed into one serial merge tree. The outputs of these serial merge trees are received by lower stages of bitonic merge units (BMUs). The data parallelism for each serial merge tree is 1, i.e., 1 per sub-sequence. The penultimate stage has two BMUs each with
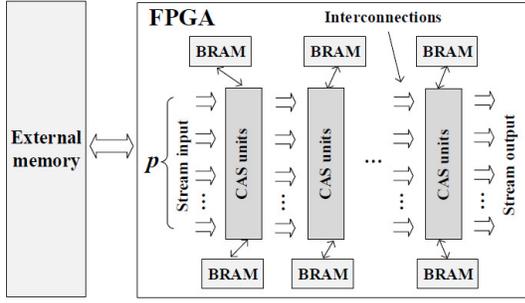
Fig. 4. Implementation of the streaming pipelined architecture on FPGA

data-parallelism 2 and the last stage has data-parallelism 4. Once the inputs are available, the serial merge units have a latency of 1 cycle, as they only compare two elements and and output the larger one. Latency of penultimate stage is $8/2 = 4$ cycles and that of the final stage is $16/4 = 4$ cycles. We emphasize again that for implementation, the whole tree is not synthesized in hardware. The tree is folded so that every stage has at most p merge units.

Table I compares our hybrid design with serial merge and *HT Design*. Hybrid design is superior to the *HT Design* [13] when

$$1 + \log p \leq 6 \implies p \leq 32. \tag{9}$$

In fact $p \geq 32$ may be impractical. Assuming 32 64-bit elements being delivered every clock cycle at a clock rate of 200 MHz would require a memory bandwidth of 47.68 GBps, which is very high for existing architectures.

TABLE I
COMPARISON OF DESIGNS FOR SORTING N ELEMENTS WITH
DATA-PARALLELISM p

Design	Serial Merge	HT Design	Hybrid
Latency	$Ng(p)/p$	$6N/p$	$(1 + \log p)N/p$
Memory	N	$6N$	$(1 + \log p)N$
Throughput	$p/f(p)$ to p	p	p

C. Architecture Implementation

The mapping of our architecture is shown in Figure 4. We assume that the sequence of size N to be sorted resides in the external memory. The merge tree in Fig 2 is folded to fit the available data parallelism. This is possible because, in a given time, at most p merge units are active with a combined data parallelism of p. The sequence is fed to the FPGA in a streaming fashion, p elements in a clock cycle. The period of a clock cycle is determined by the implementation of the design. Data parallelism p is limited by the number of pins on the FPGA device. The actual throughput then depends on the p, operating frequency and width of one element. The stages are realized using "compare-and-swap" (CAS) units [13], which are implemented to act as merge units using LUTs, and stages are pipelined using flip-flops. Each of the stages has access to a block in BRAM. All the interconnection patterns are realized by streaming permutation networks (SPN) [13].

III. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experimental Setup

We implemented our architecture on Virtex-7 FPGA (XC7VX690T, speed grade -2L). This device has 2940 BRAMs (each 18 $Kbits$) and 108300 slices. The designs were synthesized and place-and-routed by Vivado 2014.2 [14]. The data parallelism was set to $p = 4, 8$ and 16 respectively. The problem size ranges from 2^7 to 2^{17}. The following performance metrics were used

Throughput: The number of byte sorted per second (GBps). Computed as the product of number of elements sorted per second and data-width per element.
Memory consumption: Measure of number of BRAMs used.
Latency: Time until the first p sorted element are output.
Memory efficiency: The throughput achieved divided by the amount of on-chip memory used by the design (in bits). Evaluated using a plot of throughput vs memory consumption. A point on the top left of memory-efficiency plot would be considered favorable as it provides high throughput with low memory consumption.
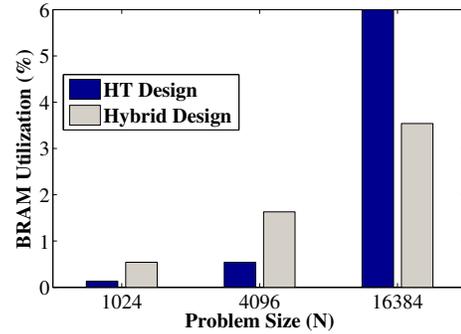

Fig. 5. Comparison of our hybrid design with the baseline and *HT Design* in terms of BRAM utilization.

B. Results

We compared the performance of our design with *HT Design*, which implements bitonic sort network on the same FPGA device. *HT Design* has been shown to be superior to the previous state-of-the-art methods in terms of throughput and several other performance metrics. Like our design, *HT Design* also provides a guaranteed throughput of p elements per clock cycle. It also claims to be a memory-efficient design. We do not present any comparison with [12] as they have performed their experiments on a more powerful platform, and have not provided any theoretical analysis for a fair comparison. For comparing our design with *HT Design* the data parallelism was set to 4 and problem size was set to 1024, 4096 and 16384. Figure 5 shows the comparison in terms of memory consumption. For $N = 1024$ and 4096, *HT Design* is better. However, for $N = 16384$, hybrid design significantly outperforms other designs. It can also be seen that BRAM utilization of our design grows much slower than *HT Design* making it more scalable for larger problem size N.

Figure 6 shows the memory-efficiency plot for the two designs. The red triangles denote the points corresponding to
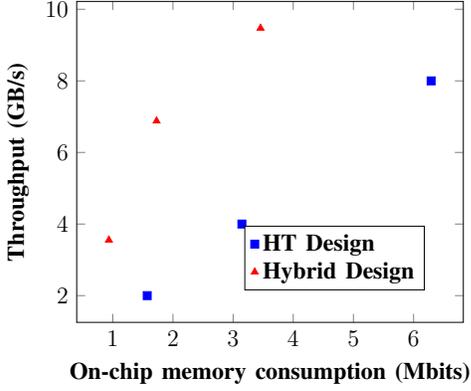
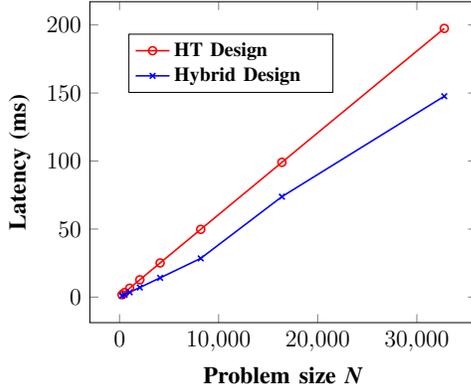Fig. 6. Memory efficiency comparison of various designs

Fig. 8. Latency for various N ($p = 8$)
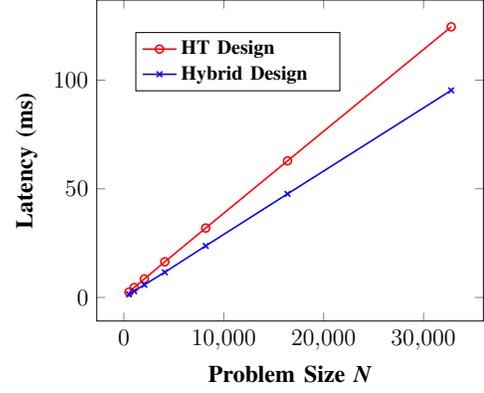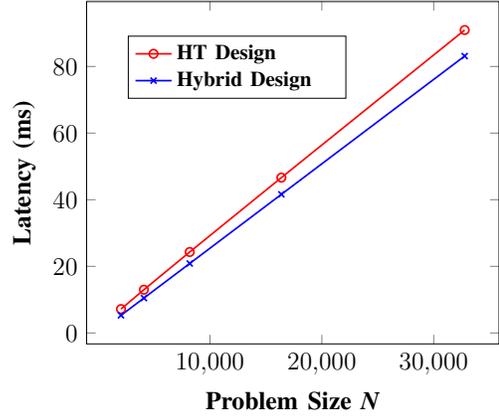

Fig. 7. Latency for various N ($p = 4$)

Fig. 9. Latency for various N ($p = 16$)

our design while the blue squares are for *HT Design* for different data-parallelism. Notice that points corresponding to our design lie entirely on the top left, suggesting higher memory-efficiency. Also notice that the best throughput obtained by our design is higher than the one reported by *HT Design*.

Figure 7, 8, and 9 show the comparison in terms of latency for data parallelism 4, 8, and 16, respectively. For all problem sizes and data parallelism, the hybrid design achieves lower latency than *HT Design*. The difference between their latency diminishes as we increase p, which is expected from our theoretical analysis (See Table I). While the dependence of *HT Design* on p is approximately given by the factor $6/p$, for our design it is $(1 + \log p)/p$. Therefore, ratio of their latencies is $6 : (1 + \log p)$. Hence, increasing p from 4 to 16 decreases the ratio. Since, we report the latency in ms rather than *cycles*, this ratio can be different from the theoretical ratio due to different clock rates of the designs.

We performed several experiments to study the throughput obtained by our design with varying data parallelism. Figure 10(a) shows the results. The horizontal dotted line at 12.8 GBps represents the upper bound of throughput determined by assuming 200 MHz clock rate for $p = 16$ 32-bit elements. Note that the optimal throughput is also restricted by the I/O pins on the FPGA device bringing the upper bound lower than 12.8. Observe that the throughput remains approximately same over all problem sizes for a given p. The slight decrement

in throughput is due to the decrease in clock rate when the resource utilization goes above a certain value. At $p = 16$ throughput is 9.54 GBps, which is 1.2x of the best throughput obtained by *HT Design* (8 GBps).

We also measured the utilization of BRAMs in our design. The results are shown in Figure 10(b). As expected, BRAM utilization increases with problem size, however it remains significantly low. We also explored the utilization of LUTs and Registers in our design. It can be observed that the number of LUTs and number of registers first drops and then increases. The initial drop is due to the fact that when problem size is small no BRAMs are used and more LUTs and registers are utilized for memory. After the drop, the increment happens very slowly with respect to the problem size. This suggests that for a large sequence, the availability of LUTs and registers is not the bottleneck, and the availability of BRAMs constraints the size of the problem that can be sorted without making any intermediate access to external memory.

IV. CONCLUSION

Sorting is a fundamental kernel of many Big Data applications. FPGAs provide energy-efficient solutions for implementing such parallel pipelined architecture. Although merge sort network is memory efficient, it poses a fundamental limit on throughput because of the final node being the bottleneck. Bitonic sort, on the other hand has the ability to provide high
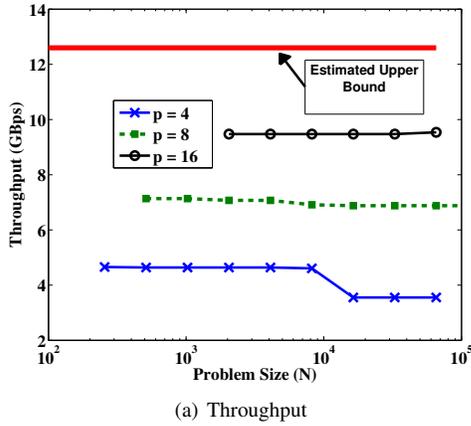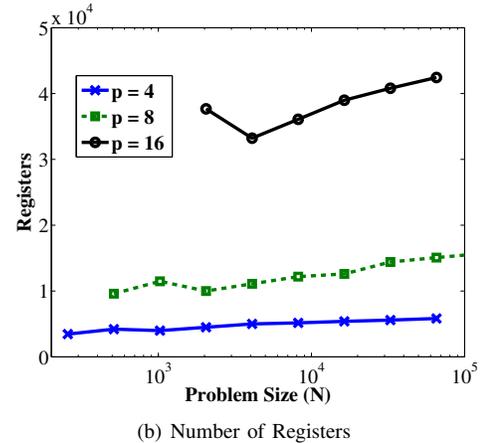
(a) Throughput
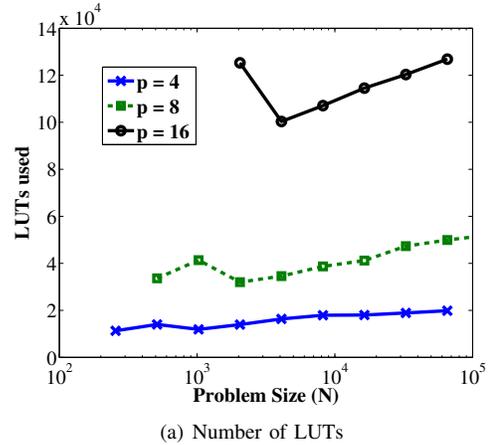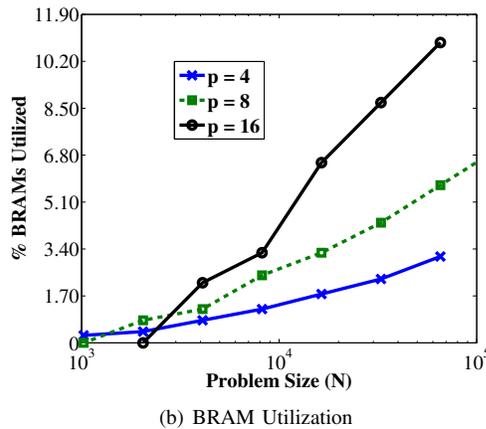
(b) BRAM Utilization

Fig. 10. Experimental results for varying data-parallelism.

(a) Number of LUTs

(b) Number of Registers

Fig. 11. Number of (a) LUTs and (b) registers used.

throughput at the cost of more resources. We presented a detailed analysis of merge network with pipelined stages, in term of latency, memory, and throughput. Further, we show that the throughput can be improved by replacing the final $\log p$ stages with bitonic merge stages, and hence utilizing the full data parallelism. The proposed design can be used to achieve optimal throughput, and yet achieve high performance in terms of latency and memory compared to state-of-the-art design for all practical values of data parallelism. The experiments demonstrate that our design outperforms the state-of-the-art implementation of bitonic sorting on the same device, in terms of throughput, memory consumption, latency, and memory efficiency. In the future, we plan to improve the implementation to incorporate other objectives such as interconnect complexity, logic consumption, and energy efficiency.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "An $O(n \log n)$ sorting network," in *Proc. of ACM STOC*. ACM, 1983, pp. 1–9.
[2] K. E. Batcher, "Sorting networks and their applications," in *Proc. of AFIPS*. ACM, 1968, pp. 307–314.
[3] A. Farmahini-Farahani, H. Duwe, M. Schulte, and K. Compton, "Modular design of high-throughput, low-latency sorting units," *IEEE TC*, vol. 62, no. 7, pp. 1389–1402, July 2013.
[4] T. Leighton, "Tight bounds on the complexity of parallel sorting," in *Proc. of ACM STOC*, 1984, pp. 71–80.
[5] 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*, vol. 31, no. 1, p. 3, 2013.
[6] 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*, May 2007, pp. 405–408.
[7] C. Thompson, "The VLSI complexity of sorting," *IEEE TC*, vol. C-32, no. 12, pp. 1171–1184, Dec 1983.
[8] M. Zuluaga, P. Milder, and M. Puschel, "Computer generation of streaming sorting networks," in *Proc. of ACM/EDAC/IEEE DAC*, June 2012, pp. 1241–1249.
[9] 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*, 2011, pp. 45–54.
[10] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *International Journal on VLDB*, vol. 21, no. 1, pp. 1–23, 2012.
[11] V. Sklyarov, I. Skliarova, D. Mihhailov, and A. Sudnitson, "Implementation in FPGA of address-based data sorting," in *Proc. of IEEE FPL*. IEEE, 2011, pp. 405–410.
[12] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 2014, pp. 151–160.
[13] 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*. ACM, 2015, pp. 240–249.
[14] "Vivado design suite user guide: design flows overview," http://www.xilinx.com/support/documentation/.