Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform

Ren Chen, Viktor Prasanna

Computer Engineering Technical Report Number CENG-2015-11

Ming Hsieh Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562

01 2016

Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform

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*—Accelerating database applications using FPGAs has recently been an area of growing interest in both academia and industry. Equi-join is one of the key database operations whose performance highly depends on the performance of sorting. However, as the data sets grow in scale, the database primitive sorting exhibits high memory usage on FPGA. A fully pipelined N-key merge sorter consists of $\log N$ sorting stages using $O(N)$ memory totally. For sorting large data sets, external memory has to be employed to perform data buffering between the sorting stages after exhausting FPGA memory resource. This introduces pipeline stalls as well as several data communication iterations between FPGA and external memory, thus causing significant performance decline. In this paper, we speed-up equi-join using a hybrid CPU-FPGA heterogeneous platform. To alleviate the burden of memory usage for sorting, we propose a merge sort based hybrid design where the first few sorting stages in the merge sort tree are replaced with "folded" bitonic sorting networks. These "folded" bitonic sorting networks work in parallel on the FPGA. The partial results are then merged on the CPU to produce the final sorted result. Based on this hybrid sorting design, we develop two streaming join algorithms, which are tailored to the target hybrid platform by optimizing the classic CPU-based nested-loop join and sort-merge join algorithms. On a range of data set sizes, our design achieves high throughput and performs 3.1x better than software-only and 1.9x better than accelerator only implementations. Our design sustains 21.6% of the peak bandwidth, which is 3.9x utilization obtained by the state-of-the-art FPGA equi-join implementation.**

Keywords*—*Database Operation, Heterogeneous Platform, Hardware Acceleration, CPU-FPGA, Sorting, Join, Selection

I. Introduction

To meet the demand for analysis of ever-increasing data set brought by the Big Data trend, accelerating database operations in memory with high-performance is of vital importance [1], [2]. With potential for breakthrough performance, FPGAs have recently become an attractive option to accelerate database applications [1], [2], [3], [4], [5]. State-of-the-art FPGAs offer high operating frequency, massive parallelsim, unprecedented logic density and a host of other features [6]. Several recent work for accelerating in-memory database operations have been proposed on FPGA platforms [3], [4], [7], [8]. Their results show that reconfigurable logic for accelerating database applications is performance-competitive with multi-core CPUs and GPGPUs. However, as the data set size scales up, the hardware implementation of database primitives such as sorting easily drains the FPGA memory resource. A fully pipelined N-key merge sorter consists of $\log N$ cascaded merge sort stages requiring a(few) data buffer(s) between each two consecutive merge sort stages, thus consuming $O(\log N)$ independent memory blocks with $O(N)$ memory totally. For sorting large data set up to the GB range, this memory requirement easily exceeds the capability of most of the state-of-the-art single FPGA devices [6]. Fig. 1 shows the throughput performance of FPGA-only sorting approach in [3] with growing input size. To process large data set, as FPGA fails to hold intermediate data on-chip, more data loading and offloading iterations (passes) have to be performed between FPGA and external memory. This causes pipeline stalls as well as significant throughput performance decline, as shown in Fig. 1.

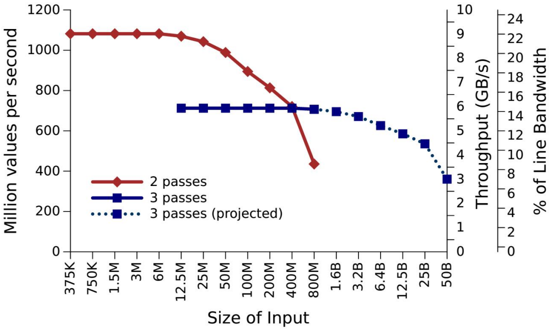


Fig. 1: Throughput of FPGA-only sorting in prior work [3]

As accelerators continue to raise the bar for both performance and energy efficiency, a recent emerging hardware trend is to incorporate dedicated hardware such as FPGA into high performance computing system [5], [9], [10], [11]. These emerging heterogeneous architectures promise massive parallelism by offering continuing advances in hardware acceleration through FPGA technology. The customized advancing high bandwidth and low latency interconnections also make the communication between CPU and FPGA more efficient [9], [10]. Most previous database operation accelerating solutions are developed for a single computational resource [3], [7], [8], [12], [13], [14]. In this paper, we show how to accelerate database operations on a heterogeneous platform using both CPU and FPGA. A key idea in this paper is to alleviate the memory burden of sorting on FPGA by developing a hybrid CPU-FPGA based sorting design. Parallel bitonic sorting network based accelerators with flexible data parallelism are developed to exploit the massive parallelism on FPGA. Merge sort tree based design with less computation load is employed on the CPU. A decomposition-based task partition approach is proposed to partition the input data set into several sub data sets sorted by FPGA accelerators in parallel, and then the partial results are merged on the CPU. Based on the hybrid sorting design, we develop two streaming join algorithms by optimizing the classic CPU-based nested loop join algorithm and sort-merge join algorithm. Experimental results show that our proposed join algorithms achieve significant performance improvement compared with CPU-only and FPGA-only join implementations for large data sets. Contributions of this work are:

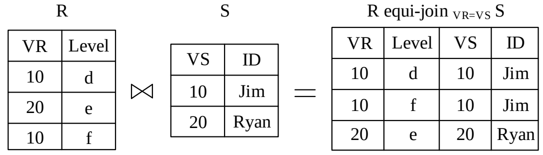
This work has been funded by US NSF under grant CCF-1018801. Equipment grant from Xilinx, Inc. is gratefully acknowledged.

- A hybrid sorting design avoiding the repeated FPGA-DRAM data loading and offloading iterations.
- A decomposition based task partition approach exploiting parallel FPGA accelerators and CPU.
- Streaming join algorithms on a CPU-FPGA heterogeneous platform delivering high throughput performance for large data sets.
- Demonstrate significant improvement in throughput performance of the proposed join design compared with CPU-only design and FPGA-only design.
- Detailed system implementation achieving optimized DRAM bandwidth utilization compared with state-of-the-art FPGA equi-join implementation.

II. Background and Related Work

A. Join

Join is one of the most fundamental database operator for relational database management system [15]. The execution of join operation is potentially very expensive, and yet it is almost required in all practical queries. In join operation, cross product of payload values needs to be generated when there are duplicate matching keys. In this paper, we are focused on equi-join, which is a specific type of comparator-based join using only equality comparisons in the join-predicate. Figure 2 depicts the essence of equi-join operation. Tuples in R and S are joined to form a new tuple if the attribute value VR in R is equivalent to the attribute value VS in S. The well known algorithms for join include sort-merge join, nested-loop join and hash join [15]. The sort-merge join algorithm can be realized by the sequential execution of sorting, merge-join, and selection operations, described as below: 1) *Sorting:* given an unsorted data sequence, rearrange the data elements so that the output sequence is in either increasing or decreasing order. 2) *Merge-join:* given two sorted sequences of fixed-width keys with associated payload values, obtain an output sequence including all the keys that the two sequences have in common, with the payload values. 3) *Selection:* given a column of data elements stored as an array of equal data width and bit masks of selected elements, the data output are selected elements based on the bit masks.



R

VR	Level
10	d
20	e
10	f

⋈

S

VS	ID
10	Jim
20	Ryan

=

R equi-join $_{VR=VS}$ S

VR	Level	VS	ID
10	d	10	Jim
10	f	10	Jim
20	e	20	Ryan

Fig. 2: Equi-join operation

Without performing the sorting, the nested-loop join algorithm joins two data columns by using two nested loops for scanning and merge-join. Block nested-loop join is an improved version of the nested-loop algorithm reducing the memory access cost [15]. Hash join is similar with nested-loop but uses join attributes as hash keys in both R and S.

B. Sorting

The most time consuming part ($\geq 90\%$ in software implementation) of sort-merge join is the sorting [16]. Therefore, the sort-merge join performance highly depends on the performance of sorting. Implementing sorting using hardware or software has been and will continue to be an active research area [3], [7], [8]. The well known sort merge tree based algorithm sorts an n-key data sequence in $\log n$ steps using $O(n \log n)$ operations. The software implementation of merge sort tree algorithm has been proved efficient [1]. However, at every merge stage, for each data element it will be either kept at current stage or spit out depending on the comparison result, this control-intensive process prevents this algorithm being parallelized in hardware. Another disadvantage of sort merge tree based hardware accelerator is that the throughput performance highly depends on the values of the input data. Thus a high throughput is not always sustained. Compared with sort merge tree, bitonic sorting network can be built with much higher data parallelism and lower control overhead, thus widely employed in hardware implementations [7], [8]. Bitonic sorting network is well known as a parallel comparison-based sorting network. It can be built using $(\log n)(\log n + 1)/2$ stages of parallel comparators, each stage contains $n/2$ comparators, for sorting n-key data sequence [17]. Recent research work shows this parallel sorting network can be employed in hardware implementation using FPGA to better utilize memory bandwidth [7]. In this paper, we propose a hybrid sorting design using bitonic sorting network based hardware accelerators and sort merge tree based software implementation.

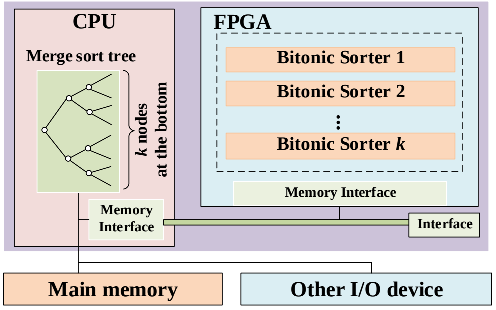
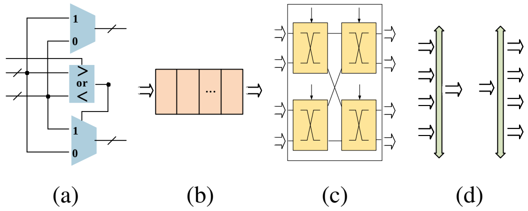


Fig. 3: Hybrid Sorting Design



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

Fig. 4: (a) Compare-and-switch (CAS) unit, (b) Data buffer, (c) Connection network, (d) Parallel-to-serial/serial-to-parallel MUX (PS/SP)

III. Hybrid Design for Sorting

Fig 3 shows the overview of our proposed merge sort based hybrid sorting design where the first few sorting stages in the merge sort tree are replaced with “folded” bitonic sorting networks, each is implemented as an FPGA accelerator named as a bitonic sorter. k such bitonic sorters work in parallel on FPGA, the partial results from FPGA are then merged on CPU using merge sort tree based implementation.

A. High Throughput Bitonic Sorter on FPGA

The bitonic sorter consists of four building blocks (Fig.4): compare-and-switch (CAS) unit, data buffer, connection network, and parallel-to-serial/serial-to-parallel (PS/SP) multiplexer. A complete design is obtained by a combination of the basic blocks.

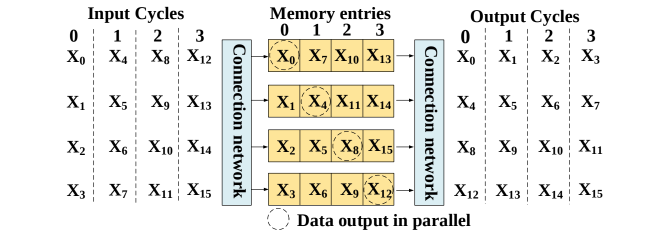


Fig. 5: Data permutation in the data buffers for 16-key sorting

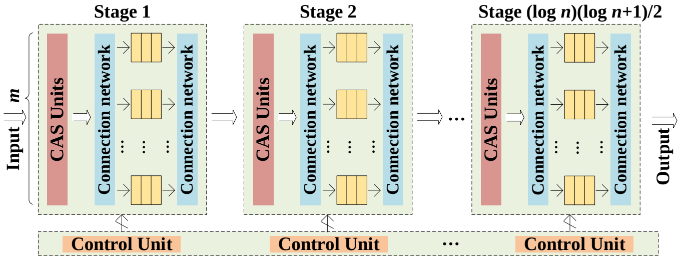


Fig. 6: A fully pipelined high throughput bitonic sorter

1) CAS unit: This module compares two input values and switch the values either in ascending or descending order depending on the control bit value. Each CAS unit is pipelined using flip-flops. To implement an n-input "folded" bitonic sorting network, $\log n(\log n + 1)$ cascaded stages of CAS units are required. Each stage consists of $m/2$ CAS units. m is the data parallelism denoted as the number of parallel inputs/outputs per cycle. The data permutation between adjacent subsequent stages of CAS units is performed through the modules including the connection network and data buffers.

2) Data buffer: Each data buffer consists of a dual-port RAM having n/m entries. Data is written into one port and read from the other port simultaneously. Fig. 5 shows the data buffering process for sorting 16 keys. In four cycles, 16 permuted data inputs are fed into the data buffers. In each cycle, with alternating locations, four data outputs are read in parallel. For different n value, the read and write addresses are generated with different strides. In Fig. 5, X_0, X_4, X_8, X_{12} are written in input cycle 0, 1, 2, 3 respectively. Then they are output simultaneously in output cycle 0.

3) Connection network: Parallel input data are required to be permuted before being processed by the subsequent modules. The connection network is implemented based on our prior work on data permutation in [18]. As shown in Fig. 5, in input cycle 0, (X_0, X_1, X_2, X_3) are fed into the first entry of each data buffer without permutation. In the next cycle, another four data inputs are written into the second entry of each data buffer with one location permuted. The parallel output data $(X_i, X_{i+4}, X_{i+8}, X_{(i+12)mod16}, i = 0, 1, 2, 3)$ are stored in different RAMs after four cycles.

4) PS/SP module: This module is used to multiplex serial/parallel input data to output in parallel/serial respectively. For example, when the number of I/Os is limited to one, but the CAS units operate on four data inputs in parallel, thus the PS/SP module is employed to match the data rate both before and after the CAS units.

Fig. 6 shows a fully pipelined high throughput sorting architecture built using the architectural building blocks introduced above. In the figure, n is the input size; m determines the number of parallel inputs/outputs. The input data sequences can be fed into the sorter continuously in a streaming manner at a fixed rate. After a specific delay, the sorted data sequences are output at the same rate. As a bitonic sorter has $(\log n)(\log n + 1)/2$ stages of data buffers, the latency introduced by all the data buffers can be calculated by

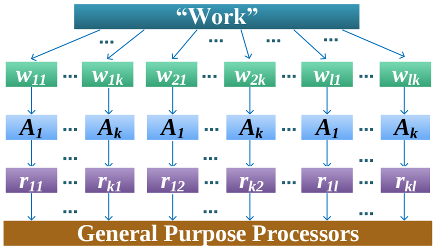


Fig. 7: Decomposition based task partition approach

$$T(n,m) = \sum_{i=\log m}^{\log n-1} \left(\sum_{j=\log m}^{i} \frac{2^{j+1}}{m} + \frac{2^{i+1}}{m} \right) \qquad (1)$$

which is $(6(n - m) - 2\log(n/m))/m$. The factor 2^{j+1} or 2^{i+1} indicates the size of a data buffer. As the total latency introduced by all the CAS units and connection networks is $O((\log m)\log^2 n)$, the entire latency of the bitonic sorter is $O(n/m)$ $(1 \leq m \leq n)$.

B. Decomposition-based Task Partition Approach

In this section, we present a decomposition-based approach for task-partition in our hybrid design for sorting. Assuming k fully pipelined bitonic sorters are implemented on FPGA for sorting, our task partition approach is described as follows:

1) Decompose: partition the N-key data set ("work") into $\frac{N}{nk}$ groups of subtasks, each group has k subtasks. Let l denotes $\frac{N}{nk}$. Each subtask is to sort n keys using a bitonic sorter. Each subtask is denoted using w_{ij} $(1{\leq}i{\leq}l,\ 1{\leq}j{\leq}k)$.

2) Accelerate: distribute the subtasks w_{ij} to the bitonic sorters denoted as $A_1, ..., A_k$. A bitonic sorter A_p $(1{\leq}p{\leq}k)$ handles l subtasks including w_{ip} $(1{\leq}i{\leq}l)$. All the bitonic sorters work on the subtasks in parallel. As each bitonic sorter is fully pipelined, all its assigned l subtasks are processed continuously without any stalls.

3) Merge: For bitonic sorter A_i, its data results are represented as $r_{i1}, ..., r_{il}$, which are produced sequentially in a streaming manner. These sorted data sequences are then transferred from the FPGA to external memory. The rest work to obtain a complete sorted data sequence will be handled by the CPU based on sort merge tree algorithm.

Figure 7 shows the basic idea of the proposed task partitioning approach. To sort N (divisible by n) keys using our hybrid sorting design, each bitonic sorter sorts $l = N/(nk)$ n-key data sequences in a streaming manner. Theoretically, the throughput of each bitonic sorter can be calculated as:

$$Th = \frac{nl}{2nl/m + 6n/m} = \frac{m}{2 + 6/l} \qquad (2)$$

where $2nl/m$ is the number of input and output cycles, $6n/m$ is the number of cycles to fill the pipeline, obtained through approximation of Equation 1. We fix n in our hybrid sorting design. n is chosen based on the amount of available memory on FPGA. As a result, theoretically, as N is increased with fixed values of n and k, Th finally approximates $m/2$. This indicates that a high throughput can always be sustained by the parallel bitonic sorters with increasing data set size. After FPGA acceleration, the rest of the computation task

Algorithm 1 Streaming Sort-Merge Join Algorithm

```
procedure SSMJ
input: $L.r_{ij}$,$R.r_{ij}(1 \le i \le k, 1 \le j \le l_L(l_R)), keysel$
output: $L.r_{ij} \bowtie R.r_{ij}$
Initialize: $s_L \leftarrow$ size of L, $s_R \leftarrow$ size of R, $l_L = s_L/(kn)$ , $l_R = s_R/(kn)$, $j = 0$
    while $j < l_L$ do                                        ▷ sorting Phase
        if receive $L.r_{1j}, L.r_{2j}, ..., L.r_{kj}$ from FPGA then
            then merge sort $L.r_{1j}, L.r_{2j}, ..., L.r_{kj}$
            $L.r(:,j) \leftarrow L.r_{1j}, L.r_{2j}, ..., L.r_{kj}$ and $j$++
        end if
    end while
    $j = 0$
    while $j < l_R$ do
        if receive $R.r_{1j}, R.r_{2j}, ..., R.r_{kj}$ from FPGA then
            merge sort $R.r_{1j}, R.r_{2j}, ..., R.r_{kj}$
            $R.r(:,j) \leftarrow R.r_{1j}, R.r_{2j}, ..., R.r_{kj}$ and $j$++
        end if
    end while
    merge sort $L.r(:,1), L.r(:,2), ..., L.r(:,l_L)$
    merge sort $R.r(:,1), R.r(:,2), ..., R.r(:,l_R)$
    for $i = 1$ to $s_L/T$ do                              ▷ merge-join and select
        for $j = 1$ to $s_R/T$ do
            call MJS($L.r(:,i)$,$R.r(:,j)$,$keysel$)
        end for
    end for
end procedure
```

is shifted to CPU which performs $O(N \log \frac{N}{n})$ operations using merge sort tree algorithm. In FPGA-only approach, to complete the final $\log \frac{N}{n}$ sorting stages, FPGA accelerator has to visit external memory for O($\log \frac{N}{n}$) iterations, each iteration loading $2n$ keys and offloading $2n$ merged keys. Without high performance memory hierarchy in CPU platform, these repeated iterations significantly lower the throughput performance of the FPGA-only approach. However, as N/n increases, the CPU execution time may become the performance bottleneck in our hybrid design, especially considering a lower memory bandwidth utilization for CPU. To resolve this issue, we propose two streaming join algorithms in Section IV by optimizing the classic CPU join algorithms to overlap the CPU and FPGA computation. Experimental results of our join design in Section VI-C4 show that about an average of 40% of the execution time of FPGA is overlapped with the CPU execution time.

IV. Streaming Join Algorithms

We develop two streaming join algorithms: streaming sort-merge join (SSMJ) algorithm and streaming block nested loop join (SBNL) algorithm. Both the two algorithms are valuable in practical: SSMJ is applicable if the client query requires two data columns to be joined and sorted; SBNL algorithm has less computation workload if the client query is a join-only request.

Algorithm 2 Merge-Join and Selection (MJS)

```
procedure MJS
input: $x, y, keysel$
output: $x \bowtie y$
    if ($x$.min $>$ $y$.max) || ($x$.max $<$ $y$.min) then
        return
    end if
    for each item $u$ in $x$ do
        for each item $v$ in $y$ do
            if $u$.key $==$ $v$.key then
                output $u \bowtie v$ if $u$.key $\in keysel$
            end if
        end for
    end for
end procedure
```

4) Streaming Sort-Merge Join (SSMJ): We assume two input table columns with equal size need to be joined. The data values of the table columns are represented using vectors L and R. Sub-vectors of L and R are first sorted by the bitonic sorters sequentially. The partial results produced by FPGA will then be merged by the CPU, which also performs the merge-join and selection operations on the sorted L and R. Algorithm 1 shows our proposed algorithm. Notations in Section III-B are reused to illustrate our algorithm. The sorted data sequences from FPGA are denoted as $L.r_{ij}$ and $R.r_{ij}$. The size of $L(R)$ is denoted as $s_L(s_R)$. Assumes that the size of each $L.r_{ij}$ or $R.r_{ij}$ is n. The k bitonic sorters will produce k sorted data sequences in parallel, each of size n, after every some specific delay. Once k sorted data sequences have been sorted, the bitonic sorters will notify CPU so that it can starts merging the k sorted data sequences immediately. This merge sort process will firstly be performed on $L.r_{ij}$ and then $R.r_{ij}$ as L and R are sorted by the bitonic sorters sequentially. After that, the processor needs to further merge the sorted subvectors including $L.r_{:,1}, L.r_{:,1}, ..., L.r_{:,s_L/k}$ or $R.r_{:,1}, R.r_{:,1}, ..., R.r_{:,s_L/k}$. Until now, both L and R have been sorted based on the key values. After that, merge-join and selection operations shown in Algorithm 2 are performed on the CPU. We still use L and R to represent the sorted inputs. $L(R)$ is divided into $s_L/T(s_R/T)$ sub-vectors denoted as $L(:,i)(R(:,j))$. T is empirically selected in our experiments depending on the cache size. In each loop iteration, two sub-vectors are fetched, each having T data elements. If the two sub-vectors have no key value overlap, the next iteration will be executed. Otherwise, compare the key values of the two sub-vectors and output the join result if a key is selected.

Algorithm 3 Streaming Blocked-Nested-Loop Algorithm

```
procedure SBNL
input: $L.r_{ij}$,$R.r_{ij}(1 \le i \le k, 1 \le j \le l_L(l_R)), keysel$
output: $L.r_{ij} \bowtie R.r_{ij}$
constants: $s_L \leftarrow$ size of L,$s_R \leftarrow$ size of R
    while !(finished all $L.r_{ij}$⋈ all $R.r_{ij}$) do
        if interrupt received then
            for $j = 1$ to $l_L$ do
                for $i = 1$ to $k$ do
                    if $L.r_{ij}$ not received then
                        continue
                    else if finished $L.r_{ij}$⋈ all $R.r_{ij}$ then
                        continue
                    else
                        for $j' = 1$ to $l_R$ do
                            for $i' = 1$ to $k$ do
                                if done $L.r_{ij}$⋈ $R.r_{i'j'}$ then
                                    continue
                                else
                                    $x \leftarrow L.r_{ij}, y \leftarrow R.r_{i'j'}$
                                    call MJS($x$,$y$,$keysel$)
                                end if
                            end for
                        end for
                    end if
                end for
            end for
        end if
    end while
end procedure
```

5) Streaming Blocked-Nested-Loop (SBNL): In this algorithm, instead of completing sorting phase on CPU after receiving intermediate results from FPGA in SSMJ algorithm, we perform merge-join and selection operations immediately.

TABLE I: Key features of the ZedBoard

CPU core	Dual ARM Cortex-A9, 666 MHz
CPU cache	32 KB L1D+L1I, 512 KB L2
DRAM and bandwidth	512 MB DDR3, ∼3.2 GB/s[1]
FPGA logic resource	85000 logic cells, 53200 slice LUTs
FPGA on-chip RAM	560 KB (BRAM)

[1] Considering a 75% of memory controller efficiency [9]

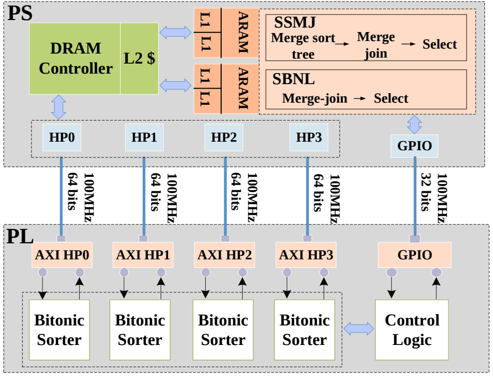


Fig. 8: BLock diagram of the complete system design on Zynq

We use the same notations as in Algorithm 1. We evenly distribute the sorting tasks to the k bitonic sorters; L and R are sorted in parallel, each handled by $k/2$ bitonic sorters. We assume L and R have equal size. Algorithm 3 shows our proposed SBNL solution. Similarly, let the inputs of the CPU be $L.r_{ij}$,$R.r_{ij}(1 \leq i \leq k, 1 \leq j \leq l_L(l_R))$, which are produced in a streaming manner by the FPGA accelerators. Once k n-key data sequences have been sorted in parallel, the bitonic sorters send an interrupt signal to the processor. After the sorted k data sequences have been transferred to the memory, the software checks if each $L.r_{ij}$ has been received or not using a table of size kl_L. For specific values of i and j, if $L.r_{ij}$ has been received, it will further check whether the join operation has been performed between the $L.r_{ij}$ and all $R.r_{i'j'}$ using a flag, thus totally kl_L flag bits for all $L.r_{ij}$. If the flag bit is zero, the MJS procedure introduced in Algorithm 2 will be called using $L.r_{ij}$ and $R.r_{i'j'}$ as the input if this procedure has not been performed on the two previously. The benefit of using this algorithm is that, as $L.r_{ij}$ and $R.r_{i'j'}$ have been sorted, we can check if the key value ranges of the two input data vectors are not overlapped. If so, the merge-join phase can be avoided thus saving time. Furthermore, the computation for sorting using the bitonic sorters and the computation for merge-join on the CPU can be further overlapped. As a result, the overall computation latency can be reduced.

V. CPU-FPGA System Implementation

We target the ZedBoard platform with Xilinx Zynq Z7020 as our experimental platform to implement our proposed join design. Xilinx Zynq processor is a high performance low power SoC architecture integrating general purpose CPU and FPGA [9]. The key features of Zynq processor is shown in Table I. The Zynq processor consists of two parts: programmable logic (PL) and programmable system (PS) integrating ARM CPU, on-chip interconnection and various peripherals [9]. A set of advanced extensible interface (AXI) interfaces are available for the communication between PS and the PL logic. Each AXI interface supports 32/64 bit full-duplex transaction, such that total 8/16 bytes of data can be transferred through AXI read and AXI write channels simultaneously. Fig 8 shows the block diagram of our design for equi-join on Zynq.

A. Parallelization and throughput-balancing

We consider data feeding rate for throughput-balancing purpose. The data feeding rate of a bitonic sorter is determined by the clock frequency, data width and data parallelism. The data feeding rate S of each bitonic sorter can be calculated as:

$$S = m \times w \times F_{clock} \tag{3}$$

where m is the data parallelism defined in Section III-A, w is the data width per data element, F_{clock} is the clock operating frequency on FPGA. Throughput balancing can be easily achieved once obtaining S. Assuming F_{clock} is 100 MHz, we attach one bitonic sorter to each of the four AXI HP ports as shown in Fig 8, then the resulted total data feeding rate is 3.125 GB/s. This ensures throughput balancing between the DRAM and the bitonic sorters considering a total of 3.2 GB/s peak bandwidth achieved by the DDR3 DRAM on Zynq if running at 1066 MHz [9]. The current DDR3 device has a data bus width of 32-bit, and we can expect a higher bandwidth when it is replaced with a 64-bit DDR device [19]. In this way, more data parallelism on FPGA can be explored.

B. System control and data flow

The system starts from input phase by feeding data inputs to the bitonic sorter continuously in a streaming manner. The input data set is evenly partitioned to ensure the workloads of the four bitonic sorters are balanced. The entire input data set is read from DRAM by all bitonic sorters during the input phase through AXI HP. Assuming data set size is N and each bitonic sorter is capable of sorting n inputs, we have $k = 4$ and each bitonic sorter handles $N/4n$ sorting subtasks in a streaming manner, each subtask is to sort n inputs, as shown in Fig. 7. An AXI GP interface is enabled and configured so that the processor can send control information to or receive updates from the FPGA accelerators. To track the current status of the $N/4n$ sorting tasks assigned to a bitonic sorter, a status bit vector of size $N/4n$ is employed. For each bitonic sorter, it updates its status bit vector through GP AXI after finishing the current sorting subtask and completing the corresponding data transfer process. The software engine on the CPU always checks the values of the status bit vector stored in the DRAM and starts either the merge sort tree operation in SSMJ algorithm or the merge-join operation in SBNL algorithm if any two new sorting subtasks have been completed. Some other AXI related control interfaces such as the central interconnect and memory switch are also employed to ensure the correct system dataflow [9].

C. System implementation

We used the generated firmware by Xilinx Vivado toolset for our target board. To avoid feedback loop between the CPU and the FPGA, we implement the merge-join operation and the selection operation in software, especially considering the fact that the most time consuming part of join is sorting. We implemented both the SSMJ and SBNL algorithms introduced in Section IV on the platform. Detailed experimental results of the two algorithms are presented in Section VI.

VI. EXPERIMENTAL RESULTS

A. Experimental Setup

All our designs were implemented on the Zedboard with Xilinx Zynq SoC XC7Z020-CLG484-1 using Xilinx Vivado 14.4 [9]. To illustrate the benefit of our proposed design approach, we report the performance of the hybrid sorting

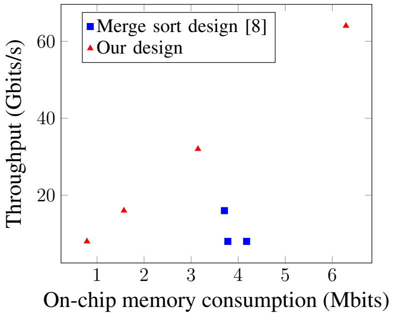


Fig. 9: Performance comparison to merge-sort design.

designs as well as the performance of the system equi-join design on the Zedboard. Each bitonic sorter can be optimized to run at a maximum frequency of 180 MHz. We clocked the bitonic sorters on the FPGA fabric at 100 MHz for the sake of throughput balancing in our system implementation. We use the Logic Analyzer in the Xilinx Vivado tool set to measure the throughput and latency of our design. To illustrate the advantage of using both the CPU and the FPGA for a single query, we present experimental results comparing our hybrid equi-join design with CPU only and FPGA only designs. We also provide performance comparison with prior works.

B. FPGA Accelerator Performance

To illustrate the benefit of using our proposed bitonic sorter, we compare the performance of our design with the state-of-the-art merge-sort based design. We separately implement the bitonic sorter on a Xilinx Virtex-7 FPGA (XC7VX690T) [6] to ensure a fair comparison with other work. Figure 9 shows the throughput performance comparison for sorting 16K-key 32-bit data sequence. The top right triangle indicates the throughput of our fully pipelined bitonic sorter. Other triangles in red represent compact designs by folding the fully pipelined bitonic sorter horizontally to save logic and memory resource, at the expense of reduced throughput. As shown in Figure 9, compared with the merge-sort based design, all our designs are dominating designs: one of our designs offers superior throughput or uses less on-chip memory or achieves both. This indicates that our proposed bitonic sorter achieves a higher memory efficiency compared to the merge-sort design, i.e., the fully pipelined bitonic sorter always outperforms in throughput performance providing the same amount of on-chip memory resource. The fully pipelined bitonic sorter can handle 4 64-bit values per clock cycle (250 MHz), providing a throughput of up to 7.9 GB/s, which almost fully utilizes the peak memory bandwidth (around 10 GB/s) of a 64-bit DDR3 DRAM [19]. As the proposed bitonic sorter has a flexible data parallelism, theoretically a high DRAM bandwidth utilization can always be achieved. There are two reasons why the proposed bitonic sorter outperforms: first, the throughput of merge sort based design depends on the values of the inputs; second, the merge sort based design is difficult to be parallelized to sustain a high throughput due to its inherent control complexity.

C. CPU-FPGA System Performance

In our system implementation, we employ four bitonic sorters running in parallel with a total data parallelism of four. All the bitonic sorters are fully pipelined by inserting registers between cascaded sorting stages. Each bitonic sorter is able to handle 64 KBytes data set and produces a 64-bit output result per clock cycle. The supported problem size is chosen to be 64 KBytes based on the available on-chip memory resource on Zynq PL. Each output is a combination of a 32-bit key and two 16-bit values. We measure the processing throughput which is the number of data values in Bytes produced per second when performing the equi-join.

TABLE II: Resource Consumption of the PL section on Zynq

Modules	# LUTs (MB/s)	LUT utilization	# of BRAMs	BRAM utilization	# of Register	Register utilization
Bitonic sorters	34686	65%	72	51.4%	18693	60%
AXI Interconnects	4048	7.6%	0	0%	4960	4.7%
AXI BRAM Controller	1756	3.3%	0	0%	1712	1.6%
Other AXI Interfaces	2119	3.9%	0	0%	2473	2.3%
BRAM for I/O buffering	0	0%	32	22.8%	0	0%

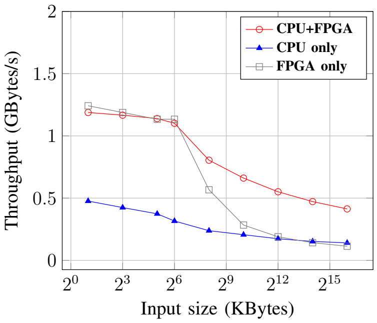
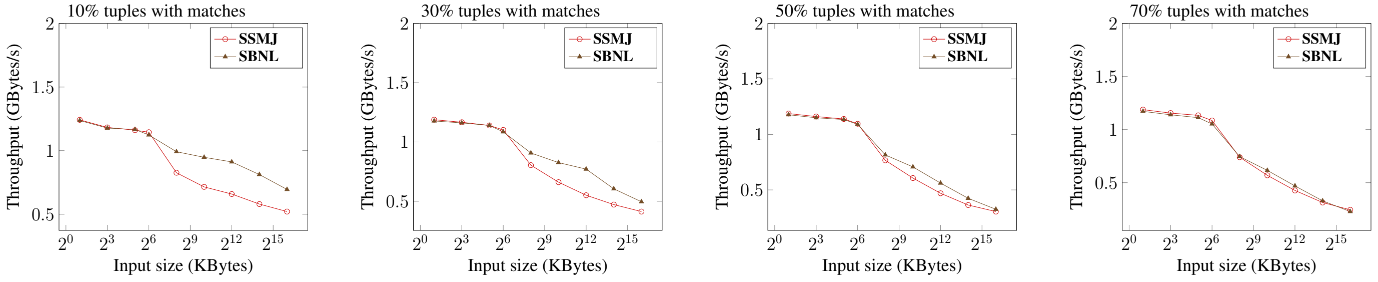


Fig. 10: Throughput comparison for various input sizes

1) Resource consumption: Table II summarizes the resource consumption of all the logic modules on the Zynq PL. The on-chip communication interfaces including the AXI interconnects, AXI BRAM controller, and other AXI related control interfaces consume 14.8% LUT of the programmable logic. The four bitonic sorters consume 65% LUT logic and 51.4% BRAM blocks. An additional 32 BRAM blocks are employed for input/output data buffering. All the communication interfaces are implemented using Xilinx provided IP cores [9]. As these IP cores can be memory mapped in the PS address space, the data communication between the FPGA and the processor can be easily handled at the software level.

2) Comparing Software, Hardware and Hybrid Designs: In this section, we compare the performance of SSMJ algorithm based accelerator with the sort-merge join algorithm based CPU-only and FPGA-only implementations. The SSMJ based CPU+FPGA approach uses the same experimental setup introduced in Section VI-C1. The CPU-only design runs on a single Cortex-A9 core inside the Zynq system with caches enabled. The FPGA-only design is implemented on the Zynq system PL section. The percentage of matching tuples for all the input sizes is varied from 10% to 70% and average throughput performance is reported. We assume the keys of the 50% input data sets have distinct values, i.e. the cardinality is 50%. In FPGA-only design, for input sizes greater than 64 KBytes, four fully pipelined bitonic sorters for sorting 64 KBytes data set are first employed to rearrange the entire input data set into sorted 64 KBytes sub data sets. Then a compact bitonic sorter with one single sorting stage by folding the fully pipelined bitonic sorter is used to merge the sorted 64 KBytes sub data sets into a single sorted data set. For input sizes smaller than 64 KBytes, the compact bitonic sorter is not required. Accelerators for merge-join operation and selection operation in the FPGA-only design are implemented based on prior work [3]. The modules in the FPGA-only design has a total data parallelism



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

Fig. 11: Throughput performance of the SBNL-based design and the SSMJ-based design

of four, which is same as the total data parallelism of the bitonic sorters in the system implementation of the SSMJ algorithm. We vary the data set (a data column) size from 2 KBytes to 64 MBytes for performance evaluation. The overall throughput (GBytes/s) for the three different design approaches are shown in Figure 10. Our proposed hybrid design achieves an average of 3.1x throughput improvement compared with the CPU only approach. This is because the CPU only approach usually achieves a low memory bandwidth utilization [20], [16]. The proposed hybrid design is 1.9x as fast on average as the FPGA-only approach. We can see that for input size greater than 64 KBytes, the throughput performance of the FPGA-only design declines significantly, and switching to the CPU for the merge-join and selection phases in the hybrid approach gives faster execution than FPGA-only approach. This implies that for the FPGA-only implementation the benefits from the massive data parallelism on FPGA is offset by the cost of increasing data loading and offloading iterations between the FPGA accelerator and the DRAM. We also observe that a large portion of the CPU computation is overlapped with the FPGA computation in our hybrid design. This in turn justifies the efficiency of our proposed hybrid design approach. More related results about the execution time breakdown are presented in Section VI-C3.

3) Comparing SBNL and SSMJ: In this section, we present experimental results when performing equi-join using the SBNL and the SSMJ algorithms. We vary the input size from 2 KBytes to 64 MBytes to evaluate the scalability of the two hybrid designs. As the bitonic sorters on FPGAs are fixed to sort 64 KBytes data set, we just need to modify the software implementations to process data sets with various sizes. As introduced in Section IV-5, in SBNL algorithm, merge-join operation is performed only if two sorted data sequences have overlapped key values, thus the execution time for merge-join also depends on the number of matching tuples. Figure 11 shows the effect of the percentage of matching tuples on the performance of our proposed streaming join algorithms. We vary the percentage from 10% to 70% for all the input sizes. We set the cardinality as 50%. We notice that as there are less tuples with matches, the SBNL algorithm improves the overall join performance compared with the SSMJ algorithm, especially for large data sets. This is because less number of merge-join operations need be performed in SBNL if the percentage of tuples with matches decreases. As shown in Figure 11, compared with the SSMJ based approach, SBNL based approach improves the throughput by up to 38% when 10% tuples match. For both the SSMJ and SBNL algorithms, the throughput decreases with the input size. The reason for this is that as the maximum problem size supported by the FPGA accelerator is fixed, more computations need to be handled by the CPU with the growing data set size, thus the overall impact of FPGA acceleration becomes less. A larger FPGA device providing more on-chip memory resource can further speed up the performance.

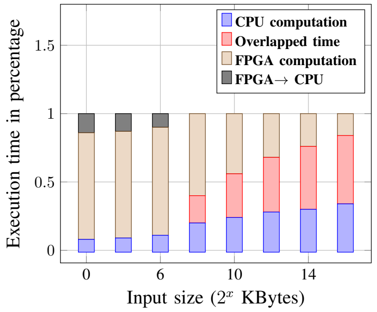


Fig. 12: Execution time breakdown of the SSMJ-based design

4) Hybrid Execution time breakdown: Figure 12 provides a breakdown of the execution time for the SSMJ algorithm for various input sizes. The FPGA→CPU time indicates the latency overhead for switching from FPGA accelerators to the CPU. We observe that this latency is easily hidden after CPU execution time and FPGA execution time overlap for input sizes beyond 64 KBytes. Data transfer overhead has been included in both the CPU computation time and the FPGA computation time. On the average, 40% of the execution time of FPGA is overlapped with the CPU execution time for input sizes greater than 64 KBytes. We observe that the execution time of FPGA almost increases linearly with the input size. This observation matches well with our theoretical analysis on the throughput of the bitonic sorters in Equation 2. The execution time of CPU increases significantly as the input size grows beyond 64 KBytes. This is consistent with the throughput performance declining in Figure 10. When the input size is smaller than 64 KBytes, the execution time of CPU accounts for 10% of the total on the average. For the input sizes beyond 64 KBytes, the CPU computation time shown in blue accounts for 37% of the total on the average, and eventually becomes a performance bottleneck. More performance improvement can be achieved by using a faster CPU with more cache resources.

D. Comparison with prior works

The available memory bandwidth on the hardware platform sets a hard limit on achievable throughput performance. As the ZedBoard offers much less DRAM bandwidth than platforms in prior work, our experimental results turns out to be comparatively slow with regarding to throughput. However, as indicated by our results in Sections VI-C2 and VI-C3, the performance of our hybrid design scales well as more bandwidth becomes available. Low throughput is not an inherent problem with our hybrid solution. We believe more powerful versions of CPU-FPGA platforms, such as the Xilinx UltraScale+ Zynq and Altera Stratix 10 SoCs will obtain improved performance by implementing our proposed algorithms. To make a more fair comparison with prior works, we use number of data values in Bytes per second (throughput) per unit bandwidth as a metric considering the memory-bandwidth-bound nature of the equi-join operation. Table III shows detailed comparison with several state-of-the-art works on reported join performance. In [16], the authors achieve a throughput of 128 million 64-bit tuples per second (1GB/s) with 25.6 GB/s available bandwidth on CPU platform. In [20], 4.6 GB/s of aggregate throughput is achieved using GPU accelerator, while the peak memory bandwidth is a total of 192.4 GB/s. In most recent work [3], the authors propose a hardware implementation for join on a multiple-FPGA platform achieving a throughput of 6.45 GB/s, while with a total of 115.2 GB/s of peak memory bandwidth, thus only 5.6% of memory bandwidth utilization. Our implementation provides a 3.9x increase on average over the reported bandwidth utilization of the state-of-the-art design [3].

TABLE III: Comparison with prior works

Work	Platform	Clock freq	Throughput (GB/s)	BW (GB/s)	Throughput/BW (%)
[3]	Multiple Xilinx Virtex-6 FPGAs	200 MHz	6.45	115.2	5.6%
[16]	Intel Core i7 965 System	3.2 GHz	1	25.6	3.8%
[20]	Nvidia GTX 580 GPU	1.5 GHz	4.6	192.4	2.3%
This work	Zedboard	100 MHz	0.69	3.2	21.6%

VII. Related Work

Using dedicated logic design to accelerate database operations especially on the advancing FPGA platform has become popular recently in both academia and industry [1], [5], [3], [7]. Researchers at Microsoft developed a reconfigurable fabric to accelerate large-scale data center services [5]. Their work presented a prototype system consisting of eight FPGAs devices cascaded using high performance interconnects. A portion of tasks for Microsoft Bing Search's ranking are accelerated using this prototype system. In [3], hardware designs to perform primitive database operations including selection, merge-join and sort are presented. High throughput performance is achieved by implementing their proposed design on an FPGA-based system. However, the memory bandwidth utilization of their design is relatively low. A system called glacier which compiles queries directly to a high level hardware description has been proposed in [2]. The team developed a streaming median operator by utilizing sorting networks in [13]. However, their design is targeted at much smaller data sets and throughput performance optimization is not presented. There are also some work focusing on accelerating sorting on FPGAs targeting database related applications. Several existing sorting architectures on FPGAs are implemented and evaluated in [8]. 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 reduce resource consumption. In [7], 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. However, the presented results are limited data set sizes. Other than FPGA, there are also some techniques for high performance join operation based on general purpose platforms [16], [14]. However, it is not clear how to apply these techniques on a heterogeneous CPU-FPGA platform.

VIII. Conclusion

In this paper, we developed streaming join algorithms customized for CPU-FPGA platform by optimizing the classic CPU-based nested-loop join and sort-merge join algorithms. A hybrid sorting design is proposed to alleviate the burden of memory usage on FPGA. As a result, our designs improve the average sustained throughput of equi-join implementation compared with FPGA-only and CPU-only designs, especially for large data sets. We reported the performance and identified the effect of the percentage of matching tuples on the throughput performance of the two streaming join algorithms. Our implementation on the Zedboard achieves significant improvement in DRAM bandwidth utilization compared with the state-of-the-art designs. We believe our proposed hybrid design is also applicable to other heterogeneous systems such as CPU+GPU platforms and can motivate the acceleration using heterogeneous system for many other data intensive applications.

References

[1] B. Sukhwani, H. Min, and et.al., "Database analytics acceleration using fpgas," in *Proc. of PACT*. ACM, 2012, pp. 411–420.

[2] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: A query compiler for fpgas," *Proc. VLDB Endow.*, pp. 229–240, Aug. 2009.

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

[4] 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.

[5] A. Putnam, A. Caulfield, E. Chung, and et.al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. of ACM/IEEE ISCA*, June 2014, pp. 13–24.

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

[7] R. Chen and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proc. of ACM/SIGDA FPGA*, 2015.

[8] 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.

[9] "Xilinx Zynq-7000 All Programmable SoC Technical Reference Manual," http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[10] "Intel Xeon+FPGA Platform for the Data Center," http://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf.

[11] "The Convey HC-2 Computer," http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC-2_Architectual_Overview.pdf.

[12] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *Computer Architecture Letters*, vol. 13, no. 2, pp. 57–60, July 2014.

[13] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *International Journal on VLDB*, vol. 21, no. 1, pp. 1–23, 2012.

[14] S. Blanas and J. M. Patel, "Memory footprint matters: Efficient equi-join algorithms for main memory data processing," in *Proceedings of the SOCC*. ACM, 2013, pp. 19:1–19:16.

[15] M. W. Blasgen and K. P. Eswaran, "Storage and access in relational data bases," *IBM Syst. J.*, vol. 16, no. 4, pp. 363–377, Dec. 1977.

[16] C. Kim, T. Kaldewey, and et.al., "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009.

[17] K. E. Batcher, "Sorting networks and their applications," in *Proc. of AFIPS*. ACM, 1968, pp. 307–314.

[18] in *Removed for blind review*.

[19] "Micron DDR3 and DDR4 SDRAM," http://www.micron.com/products/dram/.

[20] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *Proc. of the International Workshop on Data Management on New Hardware*. ACM, 2012, pp. 55–62.