2016 IEEE International Parallel and Distributed Processing Symposium Workshops

High Throughput Large Scale Sorting on a CPU-FPGA Heterogeneous Platform*

Chi Zhang, Ren Chen, Viktor Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, USA 90089
Email: {zhan527, renchen, prasanna}@usc.edu

***Abstract*—Recently accelerating sorting using FPGA has been of growing interest in both industry and academia. However, the supported size of data set is usually small for FPGA-only sorting designs due to limited on-chip memory. In this paper, we propose a design to speed-up large scale sorting using a CPU-FPGA heterogeneous platform. We first optimize a fully-pipelined merge sort based accelerator and employ several such designs working in parallel on FPGA. The partial results from the FPGA are then merged on the CPU. On the Intel QuickAssist QPI FPGA Platform, for a range of data set size, we improve the throughput by 2.9x and 1.9x compared with CPU-only and FPGA-only baselines, respectively. Compared with the state-of-the-art FPGA implementation for sorting, our design achieves 2.3x throughput improvement.**

***Keywords*-FPGA; Merge sort; Heterogeneous architecture;**

I. Introduction

With the advances in memory technology such as DDR4 and Hybrid Memory Cube [1], [2], the main memory now is capable of storing large data sets. As a result, in-memory database becomes feasible [3]. To fully utilize the memory bandwidth, accelerating in-memory database operations using dedicated hardware has been studied [3], [4], [5], [6]. Sorting is one of the most fundamental database primitive operations which requires efficient implementation and high performance in terms of latency, throughput and memory bandwidth utilization [7]. Several recent works on accelerating sorting have been proposed on FPGA platforms [4], [6], [8]. These results show that reconfigurable logic for accelerating sorting demonstrates competitive performance compared with multi-core CPUs and GPGPUs. However, the maximum data set size supported by the FPGA accelerator is usually small due to the limited available on-chip memory resource.

Heterogeneous architectures incorporating CPU with FPGA are becoming attractive for achieving large performance improvements as accelerators continue to raise the bar for both performance and energy efficiency. Emerging heterogeneous architectures such as Microsoft Catapult, Xilinx Zynq and Intel QuickAssist QPI FPGA Platform [9], [10], [11], [12] promise massive parallelism by offering continuing advances in hardware acceleration through FPGA technology. Advances in interconnection technology among heterogeneous devices also make data communication much more efficient. As the CPU and the FPGA are able to communicate through the shared memory in these platforms, cache hit rate increases and overall data communication latency improves.

In this paper, to achieve high throughput for sorting large data sets, we develop a hybrid design for sorting tailored to a CPU-FPGA heterogeneous platform. In our design, the CPU acts as the master computation and dispatching unit while the FPGA acts as an Accelerator Function Unit (AFU). We develop a Merge Sort Accelerator (MSA) which supports processing streaming data. Several MSAs are employed on the AFU and work in parallel to exploit massive data parallelism on FPGA. We carefully choose the data set size of each MSA and fully pipeline them to achieve high throughput. A complete input data set is partitioned into several sub data sets; each of them is first sorted by a MSA. Concurrently, CPU merges the sorted sub data sets from the FPGA. As a result, computation threads of the CPU and the FPGA are able to work in parallel by overlapping the CPU and FPGA computations. To store sorted sub data sets from the FPGA, a shared workspace is allocated in the main memory by the CPU. Our experimental results show that high throughput can be sustained using our proposed hybrid design to sort large data sets up to 2 GBytes. Besides, our design demonstrates significant performance improvement compared with FPGA-only and CPU-only baselines. Specific contributions of this paper are

- A high throughput merge sort based hybrid design on a CPU-FPGA heterogeneous platform.
- A divide and conquer based strategy to exploit thread level parallelism through concurrent processing on CPU and FPGA using shared memory.
- Detailed system design to map a large scale Merge Sort Accelerator onto a heterogeneous CPU-FPGA platform.
- Analysis of the performance improvement and resource utilization of the hybrid design compared with FPGA-only and CPU-only baselines.
- Throughput improvement by 2.9x, 1.9x and 2.3x compared with CPU-only, FPGA-only baselines and state-of-art FPGA design respectively.

II. Background and Related Work

Pure software sorting approaches are often limited by throughput and memory bandwidth. Recently, several customized hardware designs have been proposed to achieve high throughput sorting [4], [6], [8]. In this section, we

*This work was partially supported by U.S. NSF under grant CCF-1320211 and ACI-1339756. Equipment grant from Intel and Altera is gratefully acknowledged.

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.117

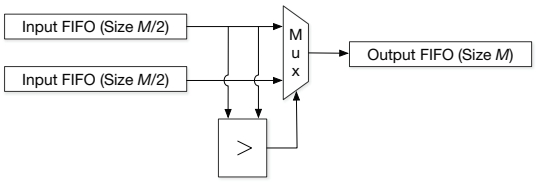


Figure 1. FIFO-based 2-to-1 merge unit [13]

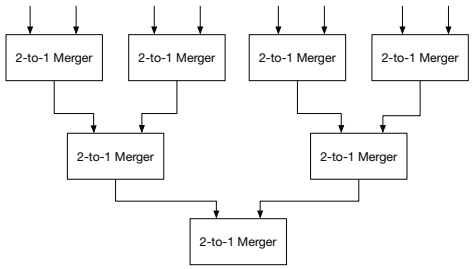


Figure 2. Logical Merge Sort Tree Structure (depth=3)

evaluate several merge sort implementations on hardware and give a brief introduction to our target CPU-FPGA heterogeneous platform.

A. FIFO-based Merge Sort Design

Figure 1 shows a FIFO-based balanced 2-to-1 merge unit commonly used in a merge sort based hardware design. It assumes the input to be sorted data sequences. In [13], the author proposes to divide data into chunks and use a primary general purpose processor to pre-sort each chunk through software-based quicksort. Then odd-even based merge network is employed to perform $O(\frac{N}{2M}\log^2\frac{N}{M})$[1] operations in order to sort the complete data sequence, where N is the problem size and M is the output FIFO size as shown in Figure 1. There are two main drawbacks in this approach. Although it uses CPU and FPGA to perform sorting, it fails to fully utilize the computation resources as the data parallelism on FPGA is not exploited. Instead, the sorting process is performed in serial and data chunks have to be transferred between FPGA and external memory several times. Another disadvantage is that the supported problem size for this architecture is small due to extensive FPGA on-chip memory usage.

B. Merge Sort Tree

Another widely used hardware sort design is to employ multiple levels of merge units which have a similar structure of the binary logical sort tree in Figure 2 [14] . For merge sort tree of depth d, the input size is 2^d. It requires $O(2^d)$ FIFO entries which consume a large amount of memory resource. Besides, the area consumption grows exponentially with the problem size. For large d, the data buffering process in the nodes at the bottom of the tree has to be performed using external memory. Since the external memory bandwidth is much smaller than the on-chip memory bandwidth, the throughput will decrease dramatically when the problem size grows larger.

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

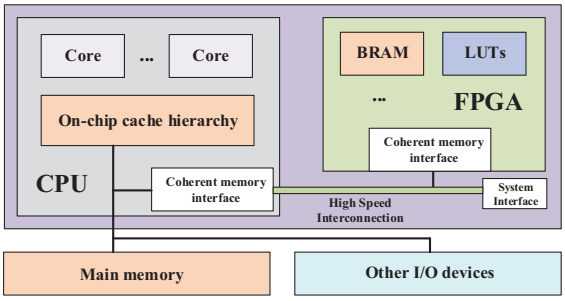


Figure 3. Target architecture integrating general purpose processor and FPGA

C. CPU-FPGA Heterogeneous Platform

1) System Overview: The target heterogeneous system platform with integrated CPU and FPGA is shown in Figure 3. The FPGA logic consists of LUT logic and on-chip memory resource such as block RAMs for implementing customized hardware accelerator. The CPU is a regular multi-core general purpose processor with the conventional on-chip cache hierarchy. Coherent memory interfaces are integrated in both FPGA and CPU for fast data communication. High speed interconnection is employed for the physical connection between FPGA and CPU.

2) Shared Memory Model: Figure 4 shows the shared memory model between CPU and FPGA. The DRAM access granularity for FPGA is cacheline[2]. However, the FPGA cannot access the DRAM directly. Instead, it has to send read/write request to the coherent cache system to access the DRAM data. The CPU and FPGA share the last level cache on CPU as shown in Figure 4. The shared memory enables efficient data communication between CPU and FPGA.

III. DESIGN METHODOLOGY

A. Divide and Conquer Strategy

The divide-and-conquer strategy is based on the shared memory model discussed in Section II. It allows the CPU and the FPGA to process data concurrently with low data communication overhead. The detailed strategy is described as following:

- Divide: Partition the complete data sequence into K sub-blocks as denoted in Figure 4. Each sub-block contains M cachelines as shown in the first sub-block.
- Accelerate: The FPGA keeps on requesting sub-blocks of data from shared memory, sorts the received data and sends the sorted sub-blocks of data back to the CPU through CPU-FPGA interconnection.

[2]In this paper, we assume that the size of each cacheline is 512 bits and keys to be sorted are 32-bit unsigned integers.

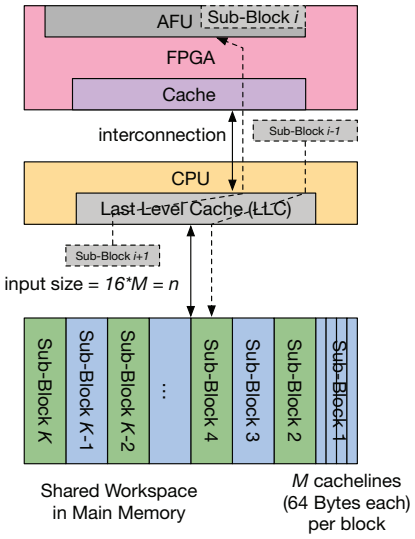


Figure 4. Shared Memory Model

- Conquer: As long as the CPU detects sorted sub-blocks of data sent by FPGA, it starts to merge them until the complete data set in the shared workspace is in sorted order.

B. Merge Sort Accelerator Design

To process streaming data, we fully pipeline the Merge Sort Accelerator into several stages. We partition the memory of each stage so that they can store the sorted data from the previous stage and simultaneously output data for the next stage. This fully pipelined design also takes advantage of FPGA Block RAMs (BRAMs) to achieve high performance. The key parameters of MSA are:

1) n: Input size n is the maximum number of consecutive keys that can be sorted by one Merge Sort Accelerator (MSA). Without loss of generality, we choose n to be a power of 2.
2) k: Stage index of the data permutation unit in MSA. The total number of stages in MSA is $\log n$.

To clearly illustrate our design, we first propose the idea of fully pipelined design. Then, we introduce the detailed interconnection and datapath. Finally, we demonstrate the overall MSA architecture.

1) Fully pipelined design: Similar to the merge sort tree structure introduced in Figure 2, the sorting design is composed of several pipeline stages without feedback loop. A data permutation unit (DP Unit) is developed as shown in Figure 6a and employed at every stage of the pipeline. To support streaming data permutation, we use two memory blocks, each containing two groups. More details on the datapath will be illustrated later.

2) Interconnection: The 2-to-2 interconnection contains a comparator and a demultiplexer as shown in Figure 5. The 'Ctrl' signal determines whether the output sequence is in ascending order or in descending order.

3) Datapath: The datapath is shown in Figure 5. We define a permutation cycle at stage k to be the period from the time when stage $k-1$ starts to store keys in Group A of the upper memory block to the time when all the keys at stage k are sorted. The highlighted arrows illustrate the active datapath during the first half of a permutation cycle. The memory groups in dark gray are active for write while the memory groups in light gray are active for read. The smaller of x and y, which is the input of stage k, is stored in the Group A of the upper memory block. In the meantime, the value stored in Group B of stage k serves as input to stage $k+1$. During the second half of permutation cycle, the demultiplexer routes the input to the group A of the lower memory block. The size of one group at stage k is 2^k. Thus, it takes $2 \cdot 2^k = 2^{k+1}$ cycles to complete one permutation cycle in stage k. At the end of every permutation cycle, the keys in each group are in sorted order.

4) Overall Merge Sort Accelerator Architecture: In order to sort n consecutive keys, MSA needs to have $\log n$ stages of data permutation units shown in Figure 6b. The data are fed into MSA serially and popped out at the same rate after certain latency. The latency of MSA is $2^2 + 2^3 + \cdots + 2^{\log n+1} = o(n)$. The total memory consumption is $(2^3 + 2^4 + \cdots + 2^{\log n+2}) \cdot 32 = o(n)$. Note that the memory consumption of this approach is exponential in the number of stages. However, with the divide and conquer strategy, we will see that the input size n is actually determined by the sub-block size of the shared memory, which is adjustable according to available hardware resources.

C. Accelerator Function Unit Design

1) Overall Architecture Design: The high level design of the Accelerator Function Unit is illustrated in Figure 7. The input data are fed into AFU in the form of packets through CPU-FPGA interconnection, along with the information of virtual address in the main memory. In order to fully utilize the computation capability of the CPU-FPGA platform, we employ the "divide-and-conquer" strategy introduced in Section III-A by dividing the workspace into sub-blocks as shown in Figure 4, each containing M cachelines. To illustrate our idea, we use the following notations:

1) N: Problem size, which is the total number of cachelines in the shared workspace. Without loss of generality, we assume N to be a power of 2.
2) M: Sub-block size, which is the number of cachelines in one sub-block. For the sake of simplicity, we choose M to be a power of 2.
3) L: The number of MSAs in parallel in the Accelerator Function Unit.

Let K denote the total number of unsorted sub-blocks in the shared workspace. According to the definition above, $K = \frac{N}{M}$. In order to reduce the data communication

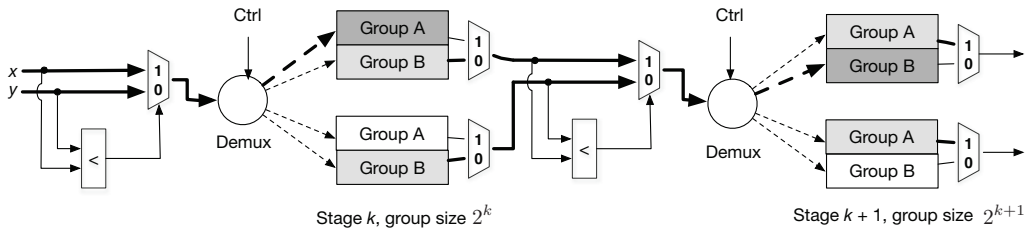
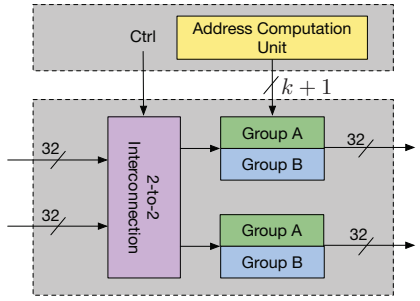
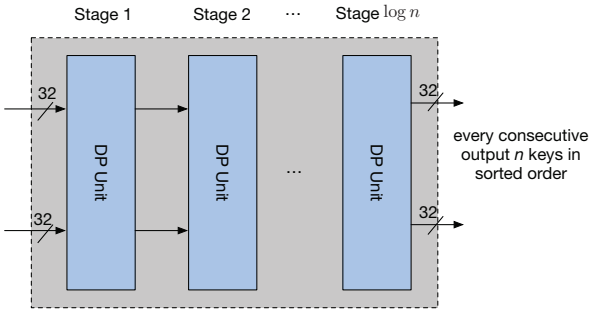


Figure 5. MSA Datapath between stage k and stage $k+1$



(a) Data Permutation (DP) Unit



(b) Streaming Data Processing Kernel

Figure 6. Merge Sort Accelerator Design

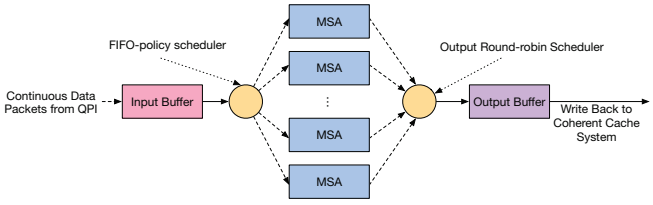


Figure 7. Overview of AFU design on FPGA

overhead, we design AFU to be able to sort one complete sub-block data without requesting the same sub-block again. Thus, the input size n of each MSA is equal to the number of keys in one sub-block, which is $16M$ as shown in Figure 4.

Assume the CPU-FPGA interconnection bandwidth is B and the FPGA frequency is F. The MSA data consumption rate is 4 Bytes per cycle. In order to make the AFU data consumption rate match the CPU-FPGA interconnection bandwidth,

$$F \times L \times 4 = B \tag{1}$$

Thus, the number of MSAs working in parallel in the AFU is $\lceil \frac{B}{4F} \rceil$.

2) Scheduler Design: For the input scheduler, the FIFO policy maximizes the utilization of CPU-FPGA interconnection bandwidth by dispatching newly arrived cacheline to any idle MSA.

The output scheduler performs round-robin[3] check of the L parallel MSAs. There is a register at the output of every MSA to prevent the sorted data being flushed before written in the output buffer.

Assume the number of keys in one cacheline is t. We claim that if $L < t$, then for every MSA, the output scheduler will fetch the sorted sub-block into the output buffer before the next sub-block being sorted by the same MSA.

For each sub-block, it takes $\frac{n}{t} = M$ cycles to be written into the output buffer because of the cacheline granularity. Each MSA takes $n = t \cdot M$ cycles for the next sub-block to be sorted. Using the round-robin scheduling, serving a MSA can be delayed by at most $(L-1) \cdot M$ cycles due to the other $L-1$ MSAs. Then it takes M cycles to store the result into the output buffer. Since $(L-1) \cdot M + M = L \cdot M < t \cdot M = n$, we can conclude that for any MSA, the output scheduler will fetch the sorted sub-block into the output buffer before the next sub-block being sorted by the same MSA.

D. Software Engine Design

To exploit CPU computational capability in this architecture, we develop a slightly different merge algorithm, which supports merging streaming sorted sub-blocks coming from the FPGA concurrently. Our streaming merge algorithm is defined as Algorithm 1. Note that the MERGEAREA function takes two arguments a and b. This function merges two groups of sub-blocks keys starting from a to $b-1$ and from b to $2b-a-1$. Before this merge operation, all the keys inside these two groups of sub-blocks are in sorted order. Compared with the traditional "bottom-up" approach, it can be viewed as "left-to-right" approach, where we try to merge as many available data as possible. The space complexity

[3]Actually, a static scheduler can be used

Algorithm 1 Streaming Merge Algorithm

```
1: function MERGEAREA(a, b)
2:         ▷ merge block area [a, b − 1] and [b, 2b − a − 1]
3: end function
4:
5: procedure SMA
6: input: serial sorted blocks from FPGA
7: output: entire sorted workspace
8: initialization: i ← 1, total block number K
9:     while i ≤ K do
10:        if block i + 1 not sorted by FPGA then
11:            continue
12:        end if
13:        MERGEAREA(i, i + 1)
14:        p ← 4, k ← i
15:        while (i + 1)%p == 0 do
16:            k ← k − p/2
17:            MERGEAREA(k, k + p/2)
18:            p ← p × 2
19:        end while
20:        i ← i + 2
21:    end while
22: end procedure
```

and time complexity are the same as "the normal" merge sort, which are $O(N)$ and $16N \log N/M + O(N \log N/M)$. However, our approach enables the overlapping of the CPU and the FPGA computations.

E. Overlapping CPU and FPGA computations

Increasing the sub-block size puts more computation workload on the FPGA while reducing the CPU computation time. To achieve a balance, when FPGA completes sorting sub-block j, where $1 \leq j \leq N/M$, the CPU should complete merging sub-blocks 1 to $j-1$. Then, the overall sorting latency is $16N \log N/M + O(N \log N/M)$. When the maximum input size for a single MSA is fixed, if the problem size keeps increasing, CPU merging time becomes the dominating factor in the total sorting time, which determines the overall system performance.

IV. Performance Analysis

A. Performance Metrics

1) FPGA resource consumption: The resource consumption including on-chip memory consumption and logic utilization are measured after place & route. The amount of on-chip memory limits the depth of the FPGA based sorting design as discussed in Section III. This determines the maximum overall throughput improvement we can achieve using our hybrid design.

Design	Latency	Logic	Memory	Throughput
Merge Sort Based[6]	$O((n \log p)/p)$	$O(p \log n)$	$O(n)$	$O(p/\log p)$
Merge Sort Based[8]	$O(n)$	$O(\log n)$	$2n + O(n)$	$O(1)$
Parallel Sorting Network[15]	$O(\frac{n \log n}{p \log p})$	$O(p \log^2 p)$	$p \log^2 p$	$O(\frac{p \log p}{\log n})$
Merge Sort Accelerator	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$

Table I
ASYMPTOTIC PERFORMANCE OF VARIOUS SORTING ARCHITECTURES

2) CPU-FPGA interconnection bandwidth utilization: CPU-FPGA interconnection bandwidth utilization is measured by the volume of data transferred between the FPGA cache and the last level cache of CPU within unit amount of time. It measures the throughput of FPGA merge sorter design.

3) Overall latency: The overall latency is the time from the first cacheline leaves the memory to the time all the keys are sorted. Let t_{FPGA} and t_{CPU} be the execution time of FPGA and CPU respectively, $t_{overlapped}$ be the time CPU and FPGA operate concurrently. We can calculate overall latency as

$$t_{overall} = t_{FPGA} + t_{CPU} - t_{overlapped} \tag{2}$$

B. Throughput and Overall Latency

The throughput of the proposed MSA is bounded by the CPU-FPGA interconnection bandwidth. However, the overall sorting latency is reduced through concurrent processing on CPU and FPGA.

The overall latency is determined by FPGA data parallelism, overlapped computation time and problem size. In equation 2, the CPU latency grows as $16N \log \frac{16N}{n}$, where N is the problem size and n is the MSA input size. As the problem size increases, the CPU latency becomes the dominant factor affecting the overall performance. Still, the CPU-FPGA platform achieves significant latency improvement compared to FPGA-only and CPU-only baselines for a range of data set size.

C. Resource Consumption

We summarize logic utilization and memory consumption of the various architectures in Table I, where p is the data parallelism and n is the input size. We can see from Table I that the logic and memory consumption is less than state-of-art designs. As the input size increases, MSA consumes more memory, which becomes the resource bottleneck on the FPGA. The logic resource is mainly consumed by the 2-to-2 interconnection (see Section III), as the number of 2-to-2 interconnection increases linearly with the number of serial merge stages and logarithmically with the MSA input size.

Modules	BRAM size	BRAM utilization	Register utilization	Logic (in ALMs)
Merge Sort Accelerator	3.96 MB	66%	12193	31%
MSA peripherals	73 KB	1.2%	6096	5%
Intel QPI IP	80.25 KB	1.3%	71269	5%

Table II
RESOURCE CONSUMPTION FOR SUB-BLOCK SIZE 1024 ON ALTERA STRATIX V FPGA

V. EXPERIMENTS AND RESULTS

A. Intel QuickAssist QPI FPGA Platform

We conducted our experiments on the Intel Heterogeneous Architecture Research Platform (HARP), a pre-production of Intel QuickAssist QPI FPGA Platform for academic use. This platform integrates 10 Intel Core Xeon E5-2600 v2 processors running at 2.8 GHz with 128 GB DDR3 memory and Altera Stratix V FPGA with 2 channel of external DDR3 memory up to 64 GB [11]. The FPGA on-chip RAM is approximately 6 Mbits. The CPU-FPGA interconnection is Intel QuickPath Interconnection (QPI) [16] with 6.4 GT/s theoretical bandwidth[4]. Studies in [16], [17] show that QPI offers much higher bandwidth with low latency, packetized, point-to-point interconnect compared to FSB, thus it is well suited for continuous, large volume data transfer between heterogeneous devices. The FPGA on chip cache also makes it much more efficient for hardware to access data. Results in this work were generated using pre-production hardware and software, and may not reflect the performance of production or future systems.

B. Experimental Setup

In our experiments, the FPGA clock frequency for 6.4 GT/s QPI is 200 MHz. The maximum QPI read/write bandwidth is around 6 GBytes/s [18]. The cacheline size is 64 Bytes and each cacheline contains 16 keys. According to Equation 1, the number of MSAs in AFU is 8. In order to evaluate the impact of resource utilization on the performance and demonstrate the advantages of using FPGA and CPU, we develop a highly parameterized AFU on FPGA and test our design on the hardware platform with $M = 1, 8, 32, 128, 1024$, where M is the number of cachelines per block. The size of the input key sequence was 16 KBytes to 512 MBytes. In order to make a fair comparison, we use CPU generated pseudo-random data as input keys.

C. QPI bandwidth utilization

The maximum QPI read/write bandwidth is around 6 GBytes/s [18]. The FPGA on-chip cache is a direct-mapping cache of size 64 KBytes. For problem size smaller than 64 KBytes, the FPGA will benefit much from high on-chip cache hit rate and the throughput will approach 6 GBytes/s.

[4]Giga Transactions per second

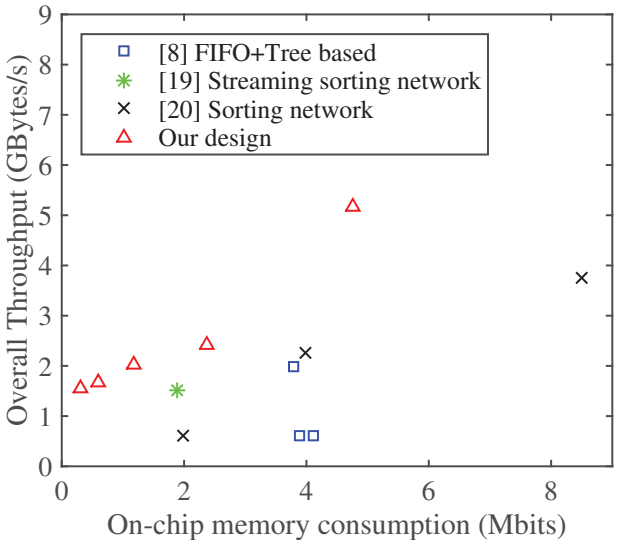


Figure 8. Performance comparison of various sorting designs

For large problem sizes, the CPU cache miss rate increases and leads to reduced QPI write bandwidth utilization.

D. Resource Utilization

Besides the MSAs in the FPGA, the MSA peripherals and Intel QPI IP consumes a fixed amount of resources on the chip including input and output buffers, scheduler logic, a 64-KByte on-chip cache, QPI interconnect protocol module, address translation and reorder buffer.

The resource consumption for sub-block size M=1024 is shown in Table II. Besides the FPGA on-chip cache, additional block memory is consumed by the reordering buffer provided by the Intel QPI IP. The MSA peripherals convert the parallel cacheline keys into serial input and perform the reverse operation at the output. The merge sort accelerators consume 3.96 MB block rams, which is almost two-thirds of Altera Stratix V FPGA on-chip memory.

E. Experimental Evaluation

1) Comparison with state-of-the-art: To compare with the state-of-art designs, we generate pseudo-random $16K$ key sequences as input. We compare our design with state-of-art approaches [8], [19], [20] with respect to on-chip memory consumption and system throughput. It is obvious that with the growth of on-chip memory consumption, the throughput increases in our design because FPGA sorting blocks become larger and CPU latency decreases with a factor of $\log \frac{1}{n}$, where n is the input size of each MSA on FPGA. By utilizing both CPU and FPGA, we achieve a peak throughput of 5.1 GBytes/s when the problem size is 16 KBytes, which is 85% of the QPI bandwidth reported in [18]. On the average, we achieve 2.3x throughput improvement compared with state-of-the-art designs. In this

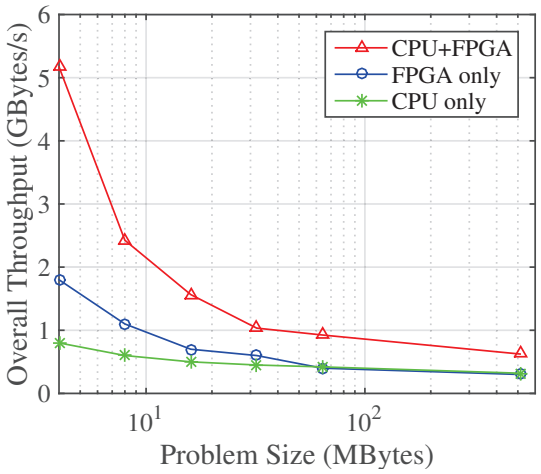


Figure 9. Throughput comparison for various problem sizes

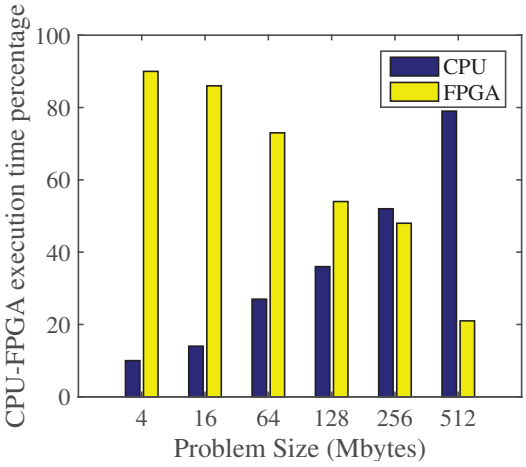


Figure 10. CPU and FPGA execution time break down

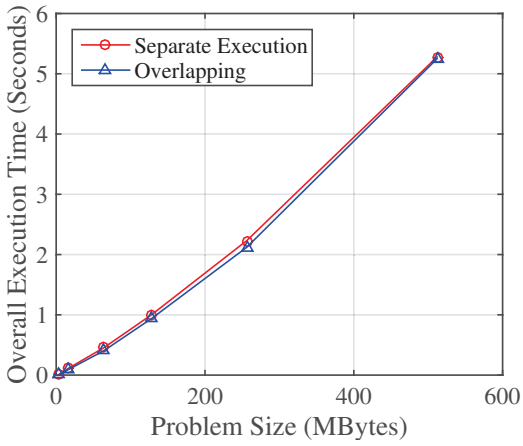


Figure 11. Overall execution time for various problem sizes

implementation, we also take advantage of FPGA on-chip cache for small problem size [21]. QPI only fetches data from the main memory for limited amount of times and the FPGA on-chip cache hit rate is very high, which contributes to the low overall latency.

2) Comparison with FPGA-only and CPU-only baselines: To show the performance improvement compared with FPGA-only and CPU-only baselines, we use the maximum available FPGA resources by implementing 8 MSAs on FPGA with sub-block size $M = 1024$ cachelines. The problem size ranges from 16 KBytes to 512 MBytes and the overall throughput is shown in Figure 9. Compared with CPU-only approach, CPU+FPGA approach achieves 2.9x throughput improvement on the average. For the FPGA-only approach, the peak throughput cannot be maintained for large problem sizes. For example, to sort 1 MByte 64-bit keys, 20 cascaded merge sort stages are needed to achieve peak throughput, and at least 16 MBytes on-chip memory is required for data buffering only between the merge stages. This on-chip memory requirement exceeds the capacity of the most of the state-of-art FPGA-only designs [22], [23], [24]. Therefore, to process large size input, external memory is needed with multiple rounds of data transfer between the FPGA and external memory. This impairs the overall FPGA throughput. Compared with the FPGA-only approach, we achieve 1.9x improved throughput on the average.

3) Execution time: According to Equation 2, the total execution time is determined by the FPGA execution time, the CPU execution time and the overlapped execution time. To illustrate the breaking down of the execution time of the system between the CPU and FPGA, we vary the problem size from 4 MBytes to 512 MBytes and measure the execution time of CPU and FPGA separately. The size of the input to each MSA on FPGA is 1024 cachelines, which utilizes the maximum available hardware resources on the FPGA. The CPU and FPGA execution time break down is shown in Figure 10. The overall execution time is shown in Figure 11. For small problem sizes, the number of CPU merge operations is small and the overall execution time is low. For large problem sizes, the CPU latency starts to dominate and the overall latency increases as shown in Figure 11. For CPU-FPGA hybrid approach, we can further exploit the shared memory by concurrent processing on CPU and FPGA. We can achieve approximately 8.22% acceleration on the average for problem size below 256 MBytes by overlapping CPU and FPGA execution time. For large problem sizes and CPU execution time becomes the dominant factor, the overlapped execution time becomes negligible.

VI. Conclusion and Future Work

In this paper, we presented a divide-and-conquer strategy for merge sort on a heterogeneous platform. We optimized sorting on a CPU-FPGA platform and compared the results

with FPGA-only and CPU-only baselines and the state-of-art with respect to throughput, execution time and resource consumption.

Future work includes optimizing the design by further exploring the design space of the sorting architecture discussed in Section III. We will also consider power consumption as a performance metric. We will also consider improving the performance by pre-caching the data for FPGA in the shared last level cache using an idle thread on the CPU.

ACKNOWLEDGMENT

We would like to thank Andrew Schmidt and the staff at the Information Sciences Institute, University of Southern California for their assistance in conducting the experiments.

REFERENCES

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

[2] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification. http://hybridmemorycube.org/files/SiteDownloads/HMC_Specification%201_0.pdf.

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

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

[5] A. Becher and et.al., "Energy-aware sql query acceleration through FPGA-based dynamic partial reconfiguration," in *Proc. of IEEE FPL*, Sept 2014, pp. 1–8.

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

[7] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv*, vol. 38, pp. 1–37, 2006.

[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] Microsoft Corporation, "An FPGA-based reconfigurable fabric for large-scale datacenters." [Online]. Available: http://research.microsoft.com/en-us/projects/catapult/

[10] Xilinx Inc, "Zynq-7000 all programmable soc." [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[11] Intel Inc., "Xeon+FPGA platform for the data center." [Online]. Available: http://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[12] Micron Technology, Inc., "The Convey HC-2 computer." [Online]. Available: http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC-2_Architectual_Overview.pdf.

[13] R. Marcelino, H. Neto, and J. Cardoso, "A comparison of three representative hardware sorting units," in *Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE*, Nov 2009, pp. 2805–2810.

[14] K. Fleming, M. King, M. C. Ng, A. Khan, and M. Vijayaraghavan, "High-throughput pipelined mergesort." in *MEMOCODE*, 2008, pp. 155–158.

[15] S. Olariu, M. Pinotti, and S. Zheng, "An optimal hardware-algorithm for sorting using a fixed-size parallel sorting device," *IEEE Transactions on Computers*, vol. 49, no. 12, pp. 1310–1324, 12 2000.

[16] D. Ziakas, A. Baum, R. Maddox, and R. Safranek, "Intel® quickpath interconnect architectural features supporting scalable system architectures," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, Aug 2010, pp. 1–6.

[17] B. Mutnury, F. Paglia, J. Mobley, G. Singh, and R. Bellomio, "Quickpath interconnect (QPI) design and analysis in high speed servers," in *Electrical Performance of Electronic Packaging and Systems (EPEPS), 2010 IEEE 19th Conference on*, Oct 2010, pp. 265–268.

[18] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-fpga systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 264–273. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847269

[19] M. Zuluaga, P. Milder, and M. Püschel, "Computer generation of streaming sorting networks," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1245–1253. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228588

[20] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1007/s00778-011-0232-z

[21] R. Chen and V. K. Prasanna, "Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations," in *Proc. of IEEE Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–8.

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

[23] R. Chen and V. K. Prasanna, "Energy-efficient architecture for stride permutation on streaming data," in *Proc. of IEEE Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 2013, pp. 1–7.

[24] R. Chen, H. Le, and V. K. Prasanna, "Energy efficient parameterized fft architecture," in *Proc. of IEEE Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2013, pp. 1–7.