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

Chi Zhang, Ren Chen, Viktor Prasanna

Computer Engineering Technical Report Number CENG-2015-10

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

01 2016

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*—As a fundamental database operation primitive, sorting requires efficient implementation and high performance in terms of latency, throughput, and energy consumption. 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 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 parallel on FPGA. The partial results from the FPGA are then merged on the CPU. We target Intel HARP as our experimental platform incorporating Intel Xeon E5-2600 v2 processors and Altera Stratix V FPGA. For a range of data set size, we improve 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 of 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 [4], [5], [6], [3]. 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], [8], [6]. 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.

Nowadays, 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 HARP [9], [10], [11], [12] promise massive parallelism by offering continuing advances in hardware acceleration through FPGA technology. Advances in interconnection bandwidth among heterogeneous devices also makes data communication much more efficient and cooperative computation between FPGA and CPU feasible. As CPU and FPGA are able to communicate through coherent memory in these platforms, cache hit rate is increased and overall data communication latency is reduced.

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 particular, the key idea is to make CPU the master computation and dispatching unit while FPGA as an Accelerator Function Unit (AFU). We develop a merge sort based accelerator (MSA) which supports processing streaming data. Several MSAs are employed on the AFU and work in parallel to exploit the massive data parallelism on FPGA. We fix the supported 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 keeps on merging the sorted sub data sets from FPGA. As a result, computation threads of CPU and FPGA are able to work in parallel by overlapping the CPU and FPGA computation. A large shared memory workspace is allocated for storing large intermediate data and reducing the memory usage burden on the FPGA. Our experimental results show that high throughput can be sustained using our proposed hybrid design to sort large data sets. Besides, our design demonstrates significant performance improvement compared with FPGA-only and CPU-only baselines. Specific contributions include:

- A high throughput merge sort based hybrid design on a CPU-FPGA heterogeneous platform.
- A divide and conquer based strategy to exploit the task parallelism on FPGA and the thread parallelism through overlapping CPU and FPGA computation.
- Demonstrate detailed system design approach to map a large scale merge sort tree onto the heterogeneous CPU-FPGA platform.
- Analysis of the performance improvement and resource utilization of the hybrid design with FPGA-only and CPU-only baselines.
- Analysis of the design tradeoffs between system performance and resources utilization of the proposed hybrid design for sorting on Intel HARP.

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

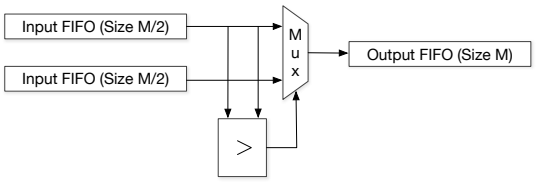


Figure 1. FIFO-based 2-to-1 merge unit

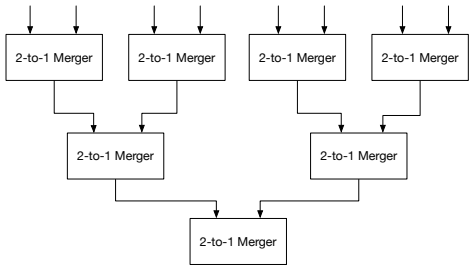


Figure 2. Logical Sort Tree Structure (depth=3)

II. BACKGROUND AND RELATED WORK

The role of sorting is to arrange an array of data in an ordered sequence for future search or merge. Efficient sorting algorithms in software have been developed including mergesort, heapsort and quicksort. However, software approaches are often limited by throughput and memory bandwidth utilization. Recently, several customized hardware designs have been developed to accelerate sorting operation for high throughput [6], [4], [8]. In this section, we evaluate several merge sort implementations on hardware and give a brief introduction on 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 propose 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 whole data sequence, where N is the input data 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 memory for several times. Another disadvantage is the supported problem size for this architecture is small due to high FPGA on-chip memory usage.

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

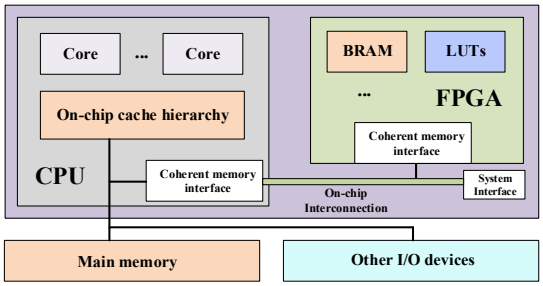


Figure 3. Target architecture integrating general purpose processors and FPGA

B. Merge Sort Tree

Another widely used sort design is to employ multiple levels of merge unit which look like a binary logical sort tree structure as described by [14] in Figure 2. For depth N merge sort tree, it requires $O(2^N)$ FIFO entries which consume a large amount of memory resource. Besides, the area consumption grows exponentially with the problem size. For large size N, the data buffering process in the nodes at the bottom of the tree has to be performed using external memory. This results in throughput performance decline as external memory bandwidth is relatively much smaller than the on-chip memory bandwidth.

C. Intel HARP Heterogeneous Platform

1) System overview: The target heterogeneous system platform with integrated CPU and FPGA is shown in Figure 3. Specifically, we employ the Intel Heterogeneous Architecture Research Platform (HARP) as our primary experimental platform where Intel Xeon E5-2600 v2 processor and Altera Stratix V[15] FPGA are integrated. The CPU and FPGA communicate through Intel QuickPath Interconnection (QPI) [16] with 6.4 GT/s full bandwidth. Studies in [16][17] show that QPI offers much higher bandwidth with low latency, packetized, point-to-point interconnect compared to FSB, which is suitable 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.

2) Shared Memory Approach: Figure 4 shows the data communication scheme through a shared memory workspace between CPU and FPGA. The DRAM access granularity for FPGA is cache line width, which is 512 bits, providing a total of 10 GB/s memory bandwidth. However, the FPGA cannot access the DRAM directly. Instead, it has to send read/write request to the coherent cache system so as to access DRAM data. The shared memory scheme enables efficient data communication between CPU and FPGA, as well as simplifying and facilitating the design of FPGA.

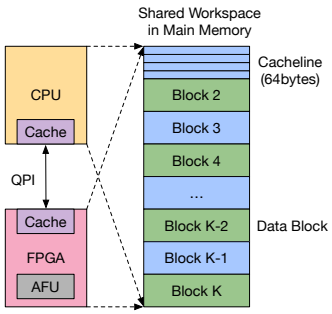


Figure 4. Shared Memory Scheme between CPU and FPGA

III. DESIGN METHODOLOGY

A. Divide and Conquer Strategy

The divide-and-conquer strategy is based on the shared memory scheme on HAPR such that it allows CPU and FPGA to manipulate data concurrently by continuous data transfer through QPI. The detailed strategy goes as followed:

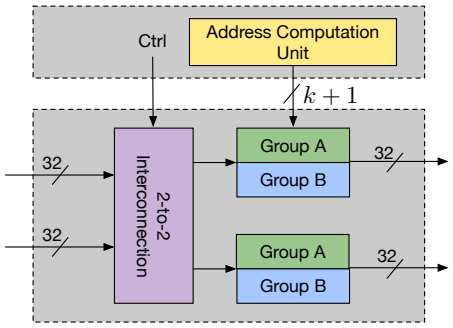
- Divide: Break the whole data sequence into K blocks as denoted in Figure 4. Each block contains M cache lines as shown in the first block.
- Acceleration: CPU continuously sends blocks of data to FPGA while FPGA keeps sorting unit block data and send back to shared memory through QPI.
- Conquer: As long as CPU detects sorted data blocks in shared memory, it will start merging data blocks.

B. Data Streaming Merge Sort Acceleration Unit

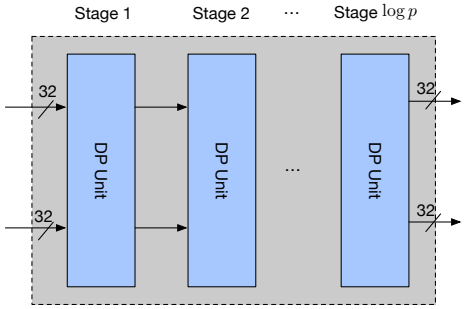
In this section, we propose our high throughput merge sort based design, which supports streaming data processing. To process streaming data, we divide the sorting design into stages and partition the memory of each stage so that they can store serial sorted data from previous stage and simultaneously output data for the next stage. This fully pipelined design also takes the advantage of FPGA block RAMs to achieve high performance. In order to clarify our design, we first define several key parameters as followed

1) p: data Parallelism. We define data parallelism p to be the number of sorted keys output from the merge sort unit. In order to make our design more efficient, we always choose p to be a power of 2.
2) k: stage index of data permutation unit in merge sort unit. The total number of stages should be $o(\log p)$.

1) Staging and Memory Partitioning: Like the merge tree structure, it is necessary to break the sorting operation into stages in order to reduce the system complexity and prevent feedback loop. A typical data permutation unit (DP Unit) is shown in Figure 5(a). To support streaming data permutation, we use two memory blocks, each containing two groups called Group A and Group B. Each permutation cycle, either Group A is performing storing sorted data stream from previous stage and Group B is performing



(a) Data Permutation (DP) Unit



(b) Streaming Data Processing Kernel

Figure 5. High Throughput Merge Sort Accelerator Design

sending out data for next stage to sort or Group B is performing storing sorted data stream from previous stage and Group A is performing sending out data for next stage to sort. In this way, all the I/Os are utilized at every clock cycles without doing extra and duplicate work. Hence, the proposed design achieves the theoretical maximum throughput for serial merging. For every consecutive p keys coming into the merge sort accelerator, after certain delay, they would be popped out in sorted order.

The memory consumption for the DP Unit is determined by its stage index k as $o(2^k)$. Thus, every 2^k cycles, a sorted data chunk of size 2^k has been sent out or stored in the next stage. Unfortunately, the area consuming of this approach also goes exponentially with the number of stage, which is logarithmic to the data parallelism as the Merge Tree approach. However, with the shared memory approach, we will see that p is actual determined by shared memory data block size, which is flexibly adjustable according to available hardware resources.

2) Interconnection: The 2-to-2 interconnection contains a comparator and a demultiplexer. The 'Ctrl' signal determines the data flow direction and whether the output is in ascending order or in descending order.

3) Overall MSA architecture: The overall merge sort accelerator design is a serial concatenation of DP Units with $\log p$ stages as shown in Figure 5(b). The data would be fed into this merge sort accelerator serially and be popped out at

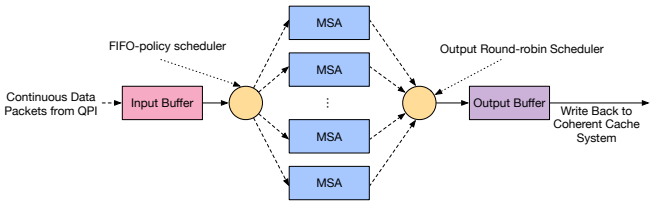


Figure 6. AFU design on FPGA

the same rate after certain delay. The latency of the proposed design is $2^2 + 2^3 + \cdots + 2^{\log p+1} \sim o(p)$. The total memory consumption would be $(2^3 + 2^4 + \cdots + 2^{\log p+2}) \cdot 32\text{Bytes} \sim o(p)$.

C. Accelerator Function Unit Design

1) Overall Architecture Design: The top level design of Accelerator Function Unit is illustrated in Figure 6. The data would be fed into AFU in the form of packets from QPI, along with the information of virtual address in main memory. In order to maximumly utilize the CPU-FPGA platform computational capacity, we follow the "divide-and-conquer" strategy by dividing the workspace into blocks as shown in Figure 4, each containing several cachelines. Thus, we define

1) N: Size of the workspace, the total number of cachelines in shared workspace. Accordingly, the total number of keys to be sorted is $16N$. Without losing generality, we assume N to be a power of 2.
2) M: Block size, number of cachelines in one block. In order to take the advantage of CPU binary merge speed, we design M to be a power of 2.
3) K: The total number of unsorted blocks, which is equal to $\frac{N}{M}$.

First, we could notice that $p = 16M$ because each block contains $16M$ keys and we design AFU to be able to sort a block of data without requesting the same block again.

Then, we would like to justify that by using 8 parallel merge sort accelerators, the data consuming rate matches the QPI bandwdith. In [18], it is measured that the FPGA read bandwidth through QPI is approximately 6 GBytes/s. For FPGA operating at 200 MHz, each merge sort accelerator consumes 4 Bytes data per cycle. Thus, for 8 parallel accelerators, the total data consuming rate is $200 \times 4 \times 8 = 6400$ MBytes/s $\approx$ 6 GBytes/s.

However, in real systems, packets would not arrive at constant speed. Instead, data burst and interval occurs often times, which pulls down the QPI bandwidth utilization.

2) Scheduler Design: For input scheduler, the FIFO policy would maximize the utilization of QPI bandwidth by dispatching newly arrived cacheline to any idle merge sort accelerator.

The output scheduler would be performing round-robin checks of the 8 parallel merge sort accelerators. There should be a register at the output of each merge sort accelerators to prevent sorted data being flushed. We would argue that under such scheme, for every merge sort accelerators, the output scheduler would fetch the sorted data block into output buffer before the next data block being sorted from the same sorter.

Algorithm 1 Streaming Merge Algorithm

```
function MERGEAREA(a, b)
        ▷ merge block area [a, b − 1] and [b, 2b − a − 1]
end function

procedure SMA
input: serial sorted blocks from FPGA
output: entire sorted workspace
initialization: i ← 1, total block number K
    while i ≤ K do
        if block i + 1 not sorted by FPGA then
            continue
        end if
        MERGEAREA(i, i + 1)
        Let p ← 4, k ← i
        while (i + 1)%p == 0 do
            k ← k − p/2
            MERGEAREA(k, k + p/2)
            p ← p × 2
        end while
        i ← i + 2
    end while
end procedure
```

Proof: For each block of data, it takes $\frac{p}{16} = M$ cycles to store into the output buffer because of cacheline granularity. For each merge sort accelerators, it takes $p = 16M$ cycles for the next data block to be sorted. For a particular merge sort accelerator i, it takes at most $7M$ cycles before the scheduler starts serving it when the other merge sort accelerators all have valid output. Then it takes M cycles for i to store its output into the buffer. Since $7M + M = 8M < 16M$, we could conclude that for every merge sort accelerator, the output scheduler would fetch the sorted data block into the output buffer before the next data block being sorted for the same merge sort accelerator. ■

Actually, we can see from the proof that the maximum possible parallel merge sort accelerators should be 15. However, increasing merge sorters would only increase the hardware complexity and consume more resources without increasing the QPI bandwidth utilization according to the bandwidth match analysis before.

D. Software Engine Design

To exploit CPU computational ability in this architecture, we develop a slightly different merge algorithm, which supports merging streaming sorted blocks coming from FPGA

concurrently. It is based on eager evaluation, indicating that before block j has been received from FPGA, all the possible work of merging should be completed already. Our streaming merge algorithm is defined as Algorithm 1. Note that the MERGEAREA function takes two arguments a and b, which is the starting block index from a to $b-1$ and from b to $2b-a-1$. Before this merge operation, we can make sure that all data between these two areas 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 present data as possible. The space complexity and time complexity is the same as normal merge sort as $O(N)$ and $O(16N \log K)$, but this approach will keep CPU doing useful work without just waiting for FPGA.

E. Overlapping Computation with CPU and FPGA

In practice, we typically would like to increase the block size to put more stress on the FPGA while saving the latency of CPU to achieve a balance. Ideally, the maximum computation overlapping occurs when FPGA is sorting the last block and CPU finishes merging all the rest blocks. Then, the system latency would be $o(16N \log K)$. However, since the maximum feasible block size for FPGA is fixed, if the problem size keeps increasing, CPU would be the master sorting unit, which slows down the whole system.

IV. Performance Analysis

In this section, we propose several performance metrics, make performance analysis and examine resource consumption based on the architecture proposed in section III. A brief summary of performance asymptotic analysis on various architecture is shown in Table I, where p is the data parallelism and n is the input size of various sorting architecture.

A. Performance Metrics

1) FPGA resource consumption: It is measured after place & route including on-chip memory consumption and logic utilization. The on-chip memory consumption is a bottleneck for FPGA sorting stages described in section III, which determines the maximum overall throughput improvement we can achieve using hybrid design.

2) QPI bandwidth utilization: We measure QPI bandwidth using approximation of average block sorting rate in HARP coherent cache system. Let T_i be the timestamp (in seconds) of block i to be sorted. The problem size is $64N$ bytes, where N is the total number of unsorted cachelines. Thus we can estimate the QPI bandwidth as

$$\text{QPI bandwidth} \approx \frac{64N}{T_N - T_1}\text{Bytes/s} \tag{1}$$

Note that it is different from overall latency since we only start the timer when we detect the first block sorted.

Design	Latency	Logic	Memory	Throughput
Merge Sort Based[6]	$o((n \log p)/p)$	$o(p \log n)$	$o(np)$	$o(p)$ or $o(p \log p)$
Merge Sort Based[8]	$o(n)$	$o(\log n)$	$2n + o(n)$	$o(1)$
Parallel sorting network[19]	$o(\frac{n \log n}{p \log p})$	$o(p \log^2 p)$	$p \log^2 p$	$o(\frac{p \log p}{\log n})$
Hybrid MSA	$o(n)$	$o(\log n)$	$o(n)$	$o(p)$

Table I
Asymptotic analysis of performance of various sorting architectures

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

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

B. Performance Analysis

1) QPI bandwidth utilization: The maximum measured QPI read/write bandwidth is around 6 Gbytes/s [18]. However, it is achieved by carefully overlapping read/write request on FPGA, which is not supported in our design. The QPI bandwidth measurement approach mentioned in part A also assumes a constant cache hit rate on both CPU and FPGA side. The FPGA on-chip cache is 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. For large problem size, CPU cache miss rate will go high and causes decrease on the QPI write bandwidth utilization.

2) Overall latency: It is determined by FPGA data parallelism, overlapping computation time and problem size. The actual latency varies with different set of input data. In order to make fair comparison, we conduct the experiments using the same pseudo-random data. In equation 2, the CPU latency is bounded by $16N \log \frac{16N}{p}$, where N is the problem size and p is the data parallelism. With the growth of problem size, the CPU latency grows dramatically and becomes the dominant factor for overall throughput. Even though, the shared memory approach achieves significant throughput improvement compared to FPGA-only and CPU-only baseline approach within a range of data set size.

C. Resources Consumption

We summarize logic utilization and memory consumption of the various architecture in Table I. Apart from that, the FPGA peripheral and Intel QPI IP consumes fixed amount of resources on chip including input and output buffer, scheduler logics, a 64 KBytes on-chip cache, QPI interconnect protocol module, address translation and reorder buffer.

Modules	BRAM Size	BRAM utilization	Registers utilization	Logic (in ALMs)
Merge Sort Accelerator	3.084 MB	51.4%	18693	65%
FPGA peripheral	1.66MB	27.7%	155125	28%
Intel QPI IP	80.25KB	1.3%	71269	5%

Table II
RESOURCES CONSUMPTION FOR BLOCK SIZE 1024 ON ALTERA STRATIX V FPGA

With the growth of data parallelism, MSA will consume more memory, which becomes resources bottleneck on FPGA. The logic consuming basically comes from 2×2 interconnection mentioned in section III, which is linear to the number of serial merge stages and logarithmic to the input size.

V. EXPERIMENT AND RESULTS

In this section, we provide a detailed experimental setup and evaluation of the proposed merge sort approach on heterogeneous architecture, along with a performance and resources utilization comparison to the state-of-art designs and CPU-only and FPGA-only baseline approach.

A. Experimental Setup

Our design is target at Intel HARP, which integrates 10 Intel Core Xeon E5-2600 v2 processor running at 2.8 GHZ with 128 GB DDR3 memory and Altera Stratix V FPGA with 2 channel of external DDR3 memory up to 64GB.[11] The Intel provided FPGA on-chip cache is 64 KBytes, along with QPI interface and address translation unit. In our design, the FPGA clock frequency for 6.4GT/s QPI is 200MHZ. In order to exploit the relations between performance and resource utilization and demonstrate the advantage of using FPGA and CPU, we develop a highly parameterized AFU on FPGA and do real testing with $M = 1, 8, 32, 128, 1024$, where M is the number of cachelines per block. The input key sequence is 16KBytes $\sim$ 512MBytes. Since we use CPU generated random keys as input, the testing performance varies when we switch input data but will keep stable under probabilistic meaning.

B. Maximum Resources Consumption

The resources consumption for block size equal to 1024 is shown in Table 2, which is the maximum data parallelism this FPGA could support. Apart from FPGA on-chip cache, additional block memory is consumed for reordering buffer, which is offered by Intel as System Protocol Layer. The AFU peripherals convert parallel cachelines into serial input and do the opposite at the output. Therefore, a huge amount of logic and registers are consumed, which can be optimized in the future. The merge sort accelerators consumes 3.084 MB block rams, which is almost half of Altera Stratix V FPGA on-chip memory.

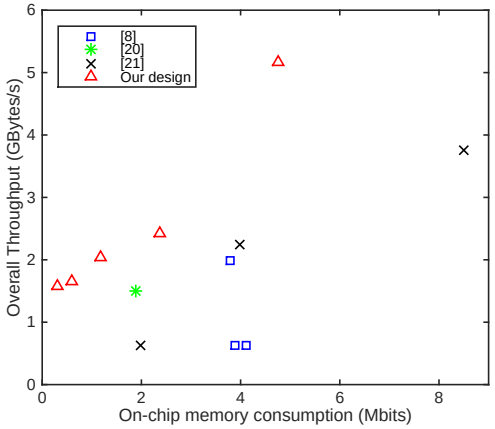


Figure 7. Performance comparison of various sorting designs

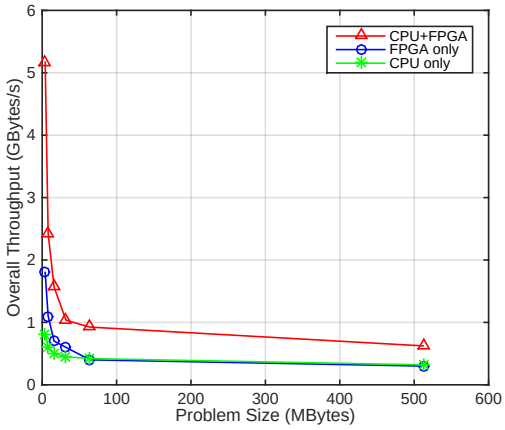


Figure 8. Throughput comparison for various input size

C. Performance Evaluation

1) State-of-art comparison: To compare with state-of-art designs, we generate pseudo-random $16K$ key sequence as input. We compare our design with state-of-art approaches[8], [20], [21] in terms of 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 as a factor of $\log \frac{1}{p}$ as shown in Figure 7. By utilizing both CPU and FPGA, we achieve a peak throughput of 5.1 GBytes/s when the problem size is 16 KByte, which is 85% of the QPI bandwidth. On average, we achieve 2.3x throughput improvement compared to state-of-art designs. In this example, we also take advantage of FPGA on-chip cache due to small problem size. QPI only fetches data from main memory for limited amount of times and the FPGA on-chip cache hit rate is very high, which contributes to the low overall system latency.

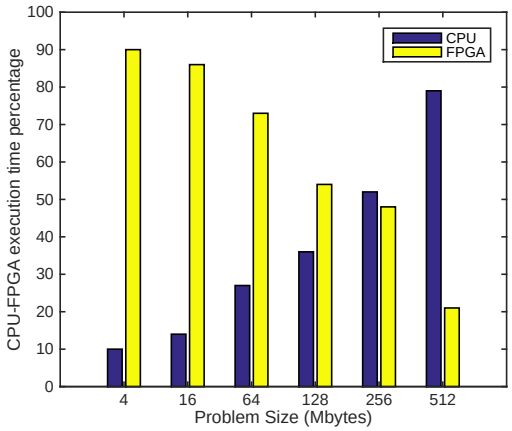


Figure 9. CPU and FPGA execution time break down

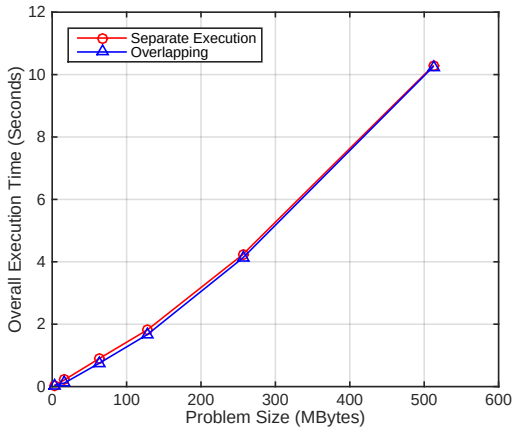


Figure 10. Overall execution time for various problem sizes

2) FPGA and CPU baseline comparison: To show the performance improvement compared to FPGA-only and CPU-only approach, we use the maximum available FPGA resources by implementing 8 merge sort accelerators and 1024 cachelines as its input size. The problem size ranges from 16 KBytes to 512MB and the overall throughput is shown in Figure 8. Compared with CPU-only approach, CPU+FPGA approach achieves 2.9x throughput improvement on average. For FPGA-only approach, the peak throughput cannot be maintained for large problem size. 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 only data buffering between the merge stages. This on-chip memory requirement exceeds the capacity of the most state-of-art FPGA-only approach[22]. Therefore, to process large set of data, external memory is needed and multiple times of data transfer between FPGA and external memory largely impair the overall FPGA throughput. Compared with FPGA-only approach, we achieve 1.9x more throughput on average.

3) Execution time: To illustrate the execution time of the system and how the execution time is broken down using both CPU and FPGA, we experiment problem sizes from 4MBytes to 512MBytes and track the execution time of CPU and FPGA separately. The input size of FPGA is 1024 cachelines, which utilizes maximum available hardware resources on FPGA. The CPU-FPGA execution time break down is shown in Figure 9 and actual overall execution time is shown in Figure 10. For small problem size, the number of CPU merge operation is limited such that the overall execution time is low. For large problem size, the CPU latency starts to dominate and the overall latency increases dramatically as shown in Figure 10. For CPU-FPGA hybrid approach, we can further take the advantage of shared memory approach by overlapping CPU merging and FPGA acceleration. Figure 10 shows that we can achieve approximately 10% acceleration on average for problem size below 256 MBytes by overlapping CPU and FPGA execution procedure. However, when CPU becomes the dominant factor for large problem size, the overlapping becomes negligible.

VI. Conclusion and Further Work

In this paper, we presented a "divide-and-conquer" strategy for streaming data processing merge sort on heterogeneous platform. We exploited the shared memory approach by comparing the performance and resources utilization with the state-of-art approach and FPGA-only and CPU-only baseline approach.

Compared to FPGA-only approach, the shared memory and FPGA on-chip cache facilitates the data access of FPGA. The high bandwidth QPI makes streaming data transfer between main memory and FPGA efficient. It also breaks the on-chip memory and external memory bandwidth limitation, which is the primary bottleneck of FPGA-only approach. The drawback of this approach lies in that the CPU latency becomes the bottleneck when problem size grows larger. We summarize the benefits of design as

- Support streaming data processing up to maximum 512MB problem size with 5.1 GBytes/s peak throughput.
- Utilize approximately 80% hardware resources and exploit heterogeneous architecture computational capacity by overlapping CPU and FPGA to achieve high performance
- On average, we improve throughput by 2.9x and 1.9x compared with CPU-only and FPGA-only approach, respectively.

In actual implementation, we did not make the best effort to achieve the peak QPI bandwidth (6 GBytes/s) by carefully overlapping read/write request on FPGA [18]. Further work

involving efficient utilization of QPI bandwidth is worth considering.

In our design, the FPGA peripheral consumes too much memory for registering cacheline data in order to work coherently with unpredictable QPI instant packet pattern. However, such an approach limits the maximum data parallelism and FPGA computational capacity, which can be improved in the further.

Other interesting area is how to effectively utilize the FPGA-CPU coherent cache system in order to improve performance.

ACKNOWLEDGMENT

We would like to thank Andrew Schmidt and the Information Sciences Institute, University of Southern California for their his assistance in conducting the experiments. Also, we would like to thank Intel and Altera for their donation of HARP system to USC.

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] Altera Corporation, "Stratix v device overview." [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_51001.pdf

[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," accepted by 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.

[19] 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.

[20] 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

[21] 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

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