Large Utility Sorting on FPGAs

Document Version

Accepted author manuscript

[Link to publication record in Manchester Research Explorer](#)

Citation for published version (APA):

Manev, K., & Koch, D. (Accepted/In press). *Large Utility Sorting on FPGAs*.

Citing this paper

Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights

Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy

If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [<http://man.ac.uk/04Y6Bo>] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Large Utility Sorting on FPGAs

Abstract—Sorting is a core operation in many applications. In particular large problem sorting has received increased attention due to many big data problems that require sorting. Most large problem sorters use variants of merge sorting. Traditional merge sorters are implemented of trees having a linear cost in resources with respect to the number of sequences merged. In this paper, we present an FPGA implementation that scales logarithmic and allows therefore merging several thousand sorted sequences in one run through the chip. For achieving high throughput, we use speculative execution techniques, pipelining of critical paths, simplified inter-level tree communications and parameterizing stage sizes for allowing trade-offs between utilization/throughput and stalling rates. Our implementation can merge sort up to 4096 sequences of 64-bit keys at 253MHz (16K sequences at 188MHz). In contrast with previous works, we research and present an implementation that sorts end-to-end on real hardware showing that the resource utilization of the needed buffering infrastructure is much higher than the sorter itself. A case study utilizing a Xilinx VC709 board merges 1024 sequences of the Graysort benchmark from and to DRAM at 80% peak DDR-3 memory throughput.

I. INTRODUCTION

Sorting is essential in many applications thus it is receiving increasingly more attention in particular for large problem sizes (as common in the big data era). For instance, sorting is regularly used for join implementations for relational databases combining two tables after firstly sorting their keys [1]. In large-scale datacentre services, scoring documents returned from a query is followed by sorting the returned records by their calculated scores [2]. In a distributed model using MapReduce for processing big data sets, sorting is a key step [3]. There are further very special applications with very specific requirements on sorting. For example, the data gathered by the Large Hydron Collider needs to be sorted at extremely low latency [4].

Sorting is one of the best studied computer science problems and probably every computer scientist came across Donald Knut’s famous compendium on sorting [5]. Also dedicated hardware approaches have a long history (e.g., [6]) and the need of high performance sorting may be continuously growing due to the trend to deal with larger and larger data sets.

Over recent years many high-performance merge sorter implementations on FPGAs have been proposed. Most of them aim high performance - in [7] the authors achieve 197 Gb/s for a 32-way merge sorter and later [8] achieve 618 Gb/s. On the other hand, in [9] the authors focus on the number of merged sequences, while achieving up to 4096 merged sequences at once.

Fig. 1 - Example of a traditional 16-way merge sorter.

This paper proposes a parallel merge sorter, which can merge a large amount of data sequences in a single run. The three most significant contributions of this paper are:

- We show that in practice the utility (i.e., the number of merged sequences) of a hardware merge sorter is often more important than just pure throughput for large scale problems.
- We propose a highly optimized design that can merge up to 16,384 sequences of data at once and which runs at 188 MHz and utilizes only 5.1K slices and 132 BlockRAMs.
- We demonstrate a case study targeting Graysort [10] on a Xilinx VC709 board. Our system is capable of merging 1024 sequences of data between two DDR-3 memory channels at 80% peak DDR-3 memory throughput.

A. Traditional FPGA Merge Sorting

Most software algorithms for sorting are not easily applicable in hardware as they often require much conditional execution and random memory accesses. Merge sort, on the other hand, is a very promising approach for sorting in hardware. This holds in particular for *External Sorting* where the problem is too large to fit into RAM and where the problem and even temporary data is commonly stored in mass storage devices (e.g. SSDs or HDDs). Merge sort is organized in a very strict data flow and in the naive implementation, there is almost no need for any control logic. The naive implementation of a hardware sequence merger utilizes a balanced binary tree structure as shown in Fig. 1. The comparing cells are implemented with processing elements that can sort when meeting two conditions: the output FIFO is not full and both input FIFOs hold data to sort. If these conditions are met, sorting takes place from the input FIFOs and (lets say) the smaller value will be pushed

into the output FIFO. The FIFO buffer sizes do not have to be large because they basically decouple operation from control such that no global control is needed and each sorting cell is self-timed and controlled by the output and input FIFO fill levels. Each stage N of such a tree consists of 2^N FIFOs and 2^{N-1} compare units. An implementation of a e -way merge sorting tree consists of $2e - 2$ FIFOs and $e - 1$ comparators.

1) Traditional Merge Sorter Advantages:

- This basic approach uses only little control logic and is easy to implement.
- Only fast local communication between processing elements is needed because every sorting cell is connected to at most three other cells in the binary tree.
- The critical path includes essentially only the key comparison and a 2:1 MUX.

2) Traditional Merge Sorter Disadvantages:

- The linear complexity growth of the proposed binary tree makes it unfeasible for implementing a design that merges a large number of sequences.
- For FPGA implementations, this design intends to under-utilize the FIFOs between the stages that would either be implemented using BRAM or distributed memory primitives (e.g., SRL16 primitives in the case of Xilinx FPGAs) that could hold much more entries than actually needed.
- When starting the sort process, all buffers at the input side will request data at the same time which requires large amount of resources for the input buffers.

3) *Observations:* When analyzing a traditional e -way merge sorter, we can observe the following characteristics which we will exploit for an improved implementation.

- In steady state where all FIFOs are full, then whenever one output record is read, only one empty token will be replaced from the entire higher level of the tree (as shown by dotted arrows on Fig. 1).
- Consequently in each level of the tree, only one compare unit is active as well as one (input) FIFO pull request and one (output) FIFO push request. This means that it must be possible to implement the sorter cell and the FIFOs using shared resources.
- Additionally, a load unit that would be in charge to fill the input FIFO buffers (that would sit above Stage 4 in Fig. 1) will at most receive one refill request per cycle which implies that we don't need any complex multi-channel arbitration scheme.

It should be mentioned that the observations apply to decision trees in general and that much of the work presented in this paper can serve as a design patterns for implementing such trees efficiently on FPGAs.

II. LARGE UTILITY MERGE SORTER

Because external sorting has to work on data from mass storage devices where access (throughput and latency) is relatively expensive (as compared to memory or even on-FPGA data access), it is of paramount importance to minimize the number of major runs. We use the term *major run* to express a major sorting step where the entire problem is read

from mass storage, then processed and finally written again (to the same or another) mass storage device. A major run may include one or more *minor runs* through local memory (e.g., DDR memory) that work on smaller problems. For example, unsorted data may be read from disk, then passed through a linear sorter that works entirely with on-FPGA memory, followed by an intermediate small merge sorter step through DDR memory which produces a temporary merged sequence that is merged again by another run through the chip before writing the generated sequences finally to disk.

If we assume for the previous example a system that is providing an aggregated disk throughput that allows reading or writing the entire problem in time T , then we need for sorting at least the time $2 \times T \times \text{number_of_run}$. Similarly, the energy needed for sorting is mostly depending on the problem size and the number of major runs. Because sorting cannot deliver a result before all input records had been seen by the sorter at least once, external large problem sorting cannot be accomplished in a single major run. Consequently, most external sorting approaches aim for two major runs where in a first run large sorted sequences are generated that are merged (ideally) in just one final major run. Therefore, external sorting through mass storage devices takes at least the time $4 \times T$ (for two major runs requiring disk read and disk write).

In order to perform large problem sorting in just two runs (or a few runs), the goal is not that much to deliver high throughput per run (as long as we can saturate the mass storage or memory throughput), but on the *utility* per run. With utility of a sorter, we refer to the amount of work this sorter would perform per run which in the case of merge sorter is the number of channels merged (i.e. e).

Because after each major and minor run the produced sorted sequence gets longer, in particular the final merge step needs attention as this sort step is normally to be accomplished with orders of magnitude less memory than needed to store the entire problem. As a practical example, let us assume that we want to sort a problem that is 1 TB in size and that the first major run generated 1024 sequences that are each 1 GB long (on disk). When using a high utility merge sorter able of merging those 1024 streams in a single run, RAM is only needed to buffer mass storage access. However, when considering a minor temporary run using 32-way sorter cascaded with another 32-way sorter (for effectively sorting $32 \times 32 = 1024$ streams), this run would require memory to store at least $1 \text{ TB} / 32 = 32 \text{ GB}$ in addition to the memory needed for buffering mass storage access, which would be more DDR memory capacity than available on most FPGA boards.

A. Analyzing Merge Sorters

In this paragraph we are focusing on a e -way merge sorter only. An e -way merge sorter merging in multiple runs n presorted sequences of a problem with the total size d that can output r records per clock cycle and that operates at frequency f , takes a total sorting time of:

$$t_{\text{sort}} \sim \frac{\lceil \log_e n \rceil \cdot d}{r \cdot f}$$

Here $r \cdot f$ is the throughput of the sorter, which may be bound by the available I/O throughput. Therefore the actual

throughput is the min of $r \cdot f$ and the effective DDR/SSD I/O. If an e -way merge sorter saturates the available effective I/O, $r \cdot f$ effectively becomes a constant value that is bound by the maximal effective throughput of the particular run configuration. While decreasing n and d obviously results in less time to sort, these values are user/problem defined and we treat them as unchangeable values (constants - c). Therefore, considering saturated I/O by a design, the effective time to sort relates to the sorter's properties is:

$$t_{\text{sort}} \sim \frac{[\log_e c_n] \cdot c_d}{c_r \cdot c_f} \sim \log_e c \sim (\log e)^{-1}$$

This means that in practice improving the utility of a sorter ends with pure performance increase once the design becomes I/O bound. After external throughput is saturated the only available utility improvement is increasing the number of sorted sequences per run.

In practice, we can use any sorter that takes unsorted data and produces small sequences [11] and then merge them within one or more minor runs. Any subsequent major run can turn the SDRAM into a buffer for hiding latencies and allowing big burst sizes with mass storage. This sorting pattern scales and the possible amount of data to be sorted in m major runs is e^{m-1} . In order to sort big data problems efficiently in FPGAs, merge sorters with large number of merged sequences are highly beneficial for the subsequent runs. Even Intel's new FPGA devices with embedded fast HMB memory [12] would not help here because for external sorting we are I/O bound by the mass storage devices or because of the limited (HMB) memory sizes available. This would even hold for temporal storage in DRAM which is significantly slower than HMB. Finally, improving sort utility minimizes the number major and minor runs and consequently is an important strategy for great energy savings.

B. Top Design

The proposed design of an e -way merge sorter consists of $\log_e$ sorter cells. Each of these sorter modules are in charge for all the sorting in one of the stages of a traditional hardware merge sorter implementation (see Fig. 1). The cells are arranged in a linear fashion, meaning that cell k (implementing stage k) communicates only with the adjacent cells $k + 1$ (referred to as the previous cell) and $k - 1$ (referred to as the next cell).

C. Sorting Cell

Assuming steady state, every stage k of the sorter consists of a single sorting cell that represents 2^{k-1} sorter cells and 2^k FIFOs as shown in Fig. 1. While a single combined compare unit can be used for the sorter cells, we need at least two FIFO elements, because it requires the content of two different FiFOs for the compare. This imposes an implementation requiring at least two random access memories for mapping the buffered data as shown in Fig. 2.

1) *Logical FIFOs*: The logical FIFOs provide an abstraction layer that implements a first-in first-out memory with parameterized depth, width, and number of channels. The module maps multiple channels to a single shared (dual ported) memory. This uses the observation that for all left and right

Fig. 2 - Naive proposed structure of sorting cells

FIFOs in the traditional merge tree there is only a single aggregated read per input branch and a single aggregated write operation needed per clock cycle. To implement the actual logical FIFOs, we use two additional memories (i.e. essentially register files) for holding the read and write pointers per channel. The dual-ported memory buffer uses then a concatenation of the channel ID and the corresponding pointer for addressing (as we use powers of two for logical FIFO sizes). With this, a push writes data into the corresponding buffer followed by an increment and update of the corresponding write pointer register file entry. Respectively, a pop command results in an increment of a read pointer (given by the channel ID).

2) *Synchronization*: In the traditional design (Fig. 1), each compare cell checks for space in its output buffer. However, continuous polling on all channels is not easily possible when sharing one buffer for multiple mapped FIFOs. We overcome this as each stage is generating refill requests to the previous stage. This means that in each stage of the tree (Fig. 1) only one input FIFO channel will be pushed and that results in a corresponding refill request to the stage above. For a refill request, the corresponding FIFO channel identifier is passed to the previous cell.

D. Control Flow

1) *Initial and Steady States*: The previously described scheme demands that the data buffers are already full. We therefore incorporated an initializing phase that fills all channels bottom up and left to right before changing to full streaming mode.

2) *Flags*: The implementation of the proposed design adds one flag bit to each record (which can most of the time fit into the carry bit of the used BRAM buffer). The flags indicate if a record is finished and not valid. With this the sorter terminates if all sequences signal being finished. This also allows the sorter to have channels that stay empty and allows us to disable individual channels if not all can be used simultaneously, hence making the sorter more generic.

E. Communication and Stalling

The proposed design does not use request buffers between sorting stages. The sorter cells themselves consume only relatively few resources (less than 2K LUTs), which allows them to be placed closely together and consequently with

Fig. 3 - Pipeline inside Logical FIFOs

Fig. 4 - Pipeline of CAS

small wiring delays. Not considering request buffers inside the sorting cells results in constant request serve time. However, there are three stalling sources that are considered.

Firstly, the input side of the design may be stalled by the load unit (e.g., due to any disk or external memory congestion). The second stall source is respectively at the output side of the sorter. The design could also stall internally, if any of the stages does not have data to serve a request from the next cell. The forward stalling is implemented using a propagating stall signal. Any cell can also backward stall the design by simply not issuing requests to the previous cells. The first two stalling sources cannot be bypassed, and are 'desired' in a sense that they mean that the design fully utilizes the available memory throughput of the used hardware. The third stalling source can be eliminated if buffer sizes in each stage are big enough to handle fully skewed data without any buffer becoming empty. If this is not ensured, the design needs additional logic to check whether a request can be served and stall the sorting otherwise.

F. Critical Path Optimization

So far, the proposed design is suffering from a long critical path. The requests to an intermediate channel in a stage require reading of the read pointers, reading of the two records to compare, comparison, and a pop operation to one of the logical FIFOs elements (increment the corresponding read pointer). In related work [11] a methodology for pipelining the compare-and-select element has been proposed where future records are speculatively pre-compared. Our proposed design cannot directly benefit that approach because different intermediate channels may be read in consecutive clock cycles which does not easily allow our sorter to look into future records. To overcome this, we implement a pipeline stage with a multiplexer and a channel comparison as shown on Fig. 4. The figure is simplified and in our implementation we are actually using three speculative comparisons that run in parallel with the selector logic (which also includes checking the flag bits) and we pipeline the channel compare. Any consecutive requests accessing the same intermediate channel result in

Fig. 5 - Top Design implementing Variable Buffer Sizes

logical FIFOs speculatively peeking into the second elements of the corresponding FIFOs as shown on Fig. 3. It is also important that the read pointers are updated correctly upon pop operation. The sorter unit will then compare if two consecutive sorting steps request data from the same logical FIFO channel and in this case multiplex the precomputed compare result. As we implement read pointers using distributed memory, we can access them combinatorial without an extra cycle which simplifies pointer updates. All these modifications allow us to pipeline the accesses to the read pointer and FIFO memories with relatively relaxed timing. This is important as the read pointers have to be routed to eventually a larger number of BRAMs as we implement eventually thousands of logical FIFO channels in a single RAM.

G. Variable Buffer Sizes

The proposed design was set to eight records per FIFO inside each stage (corresponding to FIFOs of depth 8 in Fig. 1) as this is the smallest power of 2 buffer size that prevents the sorter from internally stalling due to empty buffers, as described in subsection II-E. Having smaller FIFO depths results in additional logic to prevent hazardous reads.

Moreover, considering randomly distributed data, we can calculate that the probability of consecutively requesting the same channel in stage k is 2^{-2^k} and if the number of records inside each FIFO is m , then the total number of records in that stage is $m \cdot 2^k$. It is therefore clear that the stall probability is higher in the stages near the root and lower in the leaf stages, while the memory utilization follows the opposite trend. Because of this observation, we propose to divide the design into stages (sorting cells) using variable buffer sizes. We propose that higher numbered stages (leaf stages that are less likely to stall) have a minimal FIFO sizes elements per channel, while the last stages (root stages) having sufficient amount of elements such that they can execute any number of consecutive requests without having ever the risk to stall the design. As already presented, we evaluate a minimal FIFO depth of 8 elements to be sufficient in order to not stall. We speculatively assign a depth of 2 for the FIFOs inside stages where we can take the potential risk of stalling (closer to leaf stages) as presented in Fig. 5.

In order to omit hazardous prevention logic, we implement special behaviour in the stage that separates the stages with different FIFO depths (stage K in Fig. 5). We infer sorting cells with sufficient depth size as *large buffer stages* opposed to the rest being with FIFO depths of only 2. Stage K implements a small sliding window of the recently requests. The constant serve times of the cells enable stage K to make predictions about data availability and prevent hazardous requests to

TABLE I
EVALUATION RESULTS OF OUR E-WAY MERGE SORTER COMPARED TO THE PROPOSED SORTER IN [9]¹.

E	Freq (MHz)	Proposed Sorter				A Cost-Effective and Scalable Merge Sorter Tree on FPGAs [9]				
		Slices	BRAMs	Active Rate	Throughput (Gb/s)	Freq (MHz)	Slices	BRAMs	Active Rate	Throughput (Gb/s)
512	258	2.8K	4	0.998	16462	185	0.9K	28	1	11840
1024	255	3.0K	8	0.998	16271	175	1.0K	38	1	11200
2048	253	3.3K	16	0.998	16143	155	1.1K	42	1	9920
4096	250	3.7K	31	0.998	15952	150	1.3K	76	1	9600
8192	208	4.0K	66	0.992	13192	-	-	-	-	-
16384	188	5.1K	132	0.992	11924	-	-	-	-	-

Fig. 6 - Active rates in steady state compared to amount of large buffer stages (K in Fig. 5) for Graysort

Fig. 7 - Utilization and frequency trends while increasing the utility

the previous stages (N to $K+1$). The value of K has been implemented as a parameter, which allows us to easily evaluate and make design decisions (see Fig. 6, Table II).

H. Skewed Data

Previous works [7] propose an optimization that targets stalling issues with skewed data but that cannot guarantee stable sorting. Stable sorting is however essential for most practical applications like database acceleration.

In the event of skewed data, the last proposed optimization can be omitted by parametrizing the design with all stages set to be last buffer stages (see subsection II-G). This ensures the design will not have any empty internal buffer and it will automatically remove the stall detection logic from synthesizing.

III. EVALUATION

We synthesized our sorters using Xilinx Vivado 18.1 with selected options `AreaOptimized_High` and `Performance_Explore` for a Xilinx VC709 board featuring a XC7VX690T-2 FPGA. In this section we implement multiple configurations for the stage sizes and number of merged channels in out-of-context place and route for the proposed design. In Section IV, we present and evaluate a case study running on a VC709 board.

A. Stall Rates

We use detailed simulation including a real memory latency generator that utilizes obtained memory profile from experiments we have ran on the hardware. This ensures accurate latency and throughput numbers as well as providing us with further means for evaluating and improving the design.

Fig. 6 represents the observed active rate in steady state. It also includes sorting with low latency and over provisioning of throughput (BRAM case) as well as DDR3 latency with over-provisioned throughput (DDR3/BRAM case). As already identified, we observe for the latter two cases that having more than 6 large buffer sorting cells results in very high active rates.

To evaluate stall rates we use randomly generated data with uniform distribution. As expected, we observe that the number of merged channels does not affect the stall rates from within the design, while the amount of *large buffer stages* does. Table II lists measured stall rates for a 2048-way merge sorter with 10 different parameter options for the number of large buffer stages (see K in Fig. 5). The listed number of clock cycles denote the time to initialize a design and is constant (i.e. independent to the size of the input data), thus we present the original numbers in clock cycles rather than relating them to execution time. We observe that a minimum of six *large buffer stages* results in active rate higher than 99%. Note that even 33k cycles for filling the pipeline is a relatively small time span when merging eventually terabytes of data per run.

B. Throughput

As the proposed design implements a single rate merge sorter, the achieved throughput is much lower than in previously proposed multirate sorters [7], [8], but as explained in section II, this is not necessary a drawback as those sorters can only sort a rather limited number of sequences (up to 32 sequences in [7], [8]), which in turn means multiple more runs to sort the same problem size. If we compare our solution with the previous best high utility merge sorter [9] as summarized in Table 1, we see that our sorter achieves a significantly higher throughput. Both sorters are single rate,

¹Values derived from results reported.

Amount of Large Buffer Stages (K in Fig. 5)	Initialize State (Clock cycles)	Active Rate in Steady State
2	16,414	0.5791
3	16,442	0.7687
4	16,506	0.9078
5	16,634	0.9709
6	16,890	0.9918
7	17,402	0.9979
8	18,426	0.9995
9	20,474	0.9999
10	24,571	0.9999
11	32,762	1.0000

Table II - Results showing the relation between amount of large buffer stages and initialize times/active rates

but our proposed sorter achieves higher operating frequencies and also scales much better in this direction, which enables it to be faster when $E = 16K$ compared to the previous best at only $E = 512$. Again for external sorting, utility is commonly the key and not the throughput. Even the slowest sorter configuration with 16k channels will likely saturate a fast NVMe SSD (please note that for external sorting we need read and write simultaneously). A solution to further improve throughput can be implemented by running multiple variants of the high utility merge sorter in parallel and combining their resulting streams with a high-throughput design like in [8]. Instantiating K times the proposed e -way design, buffering their output in deep FIFOs to overcome skewed data and putting that into the input side of an K -way multirate merge sorter will result into a $(K \times e)$ -way merge sorter that produces up to K output elements per clock cycle. Moreover, large problem sorting is typically performed on large keys which means high throughput due to wide data paths.

C. Scalability

Fig. 7 shows the trends for slice and BRAM utilization as well as throughput, when the number of sorted sequences is increased of our e -way merge sorter. The slice utilization and achieved frequency scales in logarithmic fashion with e , which proves high scalability for the proposed design. On the other hand, the amount of stored records can physically scale only in linear fashion with respect to the BRAM utilization, which is the major limitation of most merge sorter architectures. However, the organization of eventually thousands of logical FIFOs into one shared buffer reduces memory fragmentation issues that would arise if each FIFO would demand its own BRAM (or BRAMs). Table I shows that the proposed design has greater logic utilization than the previously proposed wide merge sorter, but all the evaluated sizes keep the design around or under 5K Slices, which can be a negligible amount in modern FPGAs, while the design proves high operational frequencies even for larger utility values ($E > 4K$). Additionally, the Slice-to-BRAM ratios of the Virtex 7 device family varies between 68 and 236. Considering $e = 4096$ our design has 119 Slice-to-BRAM ratio, while the other wide sorter has a ratio of 17 and both ratios trend to decrease with the increase of e . This shows our solution fits much better the resource

Fig. 8 - VC709 Graysort Merger floorplan: Green - Sorter; Yellow - Memory interfaces; Cyan - Additional BRAMs utilized; Red - Other infrastructure

distribution of Virtex 7 devices. It is also important that the sorter utilizes less BRAMs, as they are highly needed for an infrastructure to buffer memory operations, as described in the next section.

IV. CASE STUDY: GRAYSORT MERGE SORTER

We have implemented multiple variants of our proposed merge sorter for challenging the Graysort benchmark [10] which is sorting a 100TB problem with 100-byte records that use 10-byte keys (80 bits). In our implementation we detach the data (90 bytes per record) from the key-data pair in order to minimize the width of the datapath of the sorter. We anticipate that this result in energy savings, because the wide data remains in a static position while the key and a small portion of data are propagating through the sorter. Although utilization of the proposed sorter is rather small, incorporating it into a full system requires substantial amounts of on-chip memory to buffer large SDRAM burst sizes. Current

Fig. 9 - Sorter stall sources as a ratio of the total runtime (Note: multiple sources can stall simultaneously!)

technology DDR memory actually performs poor on random accesses (which is the anticipated memory access pattern when merging thousands of streams). For achieving high effective speeds, the burst sizes have to be of significant size. This means that we have to pay a high latency penalty for the first access to a random memory location, but that further consecutive data can be accessed relatively cheaply.

A. Data Records

The aim of the researched case study was to provide high utility merge sorter that can execute sorting from external DRAM and store the resulting sequence to another external memory. Our target hardware (Xilinx VC709) consists of two DDR3 modules that can operate at 64 bit data width whilst running at 1600MHz (800MHz DDR), thus providing a theoretical maximal throughput of 12.8GB/s per module. Xilinx provides a memory interface that has a 512 bit data width and runs at 200MHz which does not directly map the 800 bit problem. We decided to pack 7x800 bit records (=5600 bits) into 11x512 bits (=5632 bits) memory bursts. We call this unit a record block. Before we start sorting, in our tests we generate the input data by using a 64-bit time-dependant self-seeded LFSR and write data directly to one of the DDR modules. As the data payload can be arbitrary, we reserve 80 bits of it for a hash value of the rest of the record, thus we can verify on the fly at the output of the sorter that record are intact and in correct order (by comparing consecutive records and by recomputing the hash).

B. Infrastructure

Fig. 8 shows the utilization of the memory interfaces and the needed infrastructure compared to the utilization of the proposed sorter alone. The infrastructure (marked in red) is needed for record generation, conversion of data widths and verifying the generated output. It is also noticeable that the sorter utilizes most of the available BlockRAMs (marked in cyan) as just the buffering of two blocks of Graysort records per sequence in a configuration of 2K sequences requires already more than 2.8MB of on-chip memory (> 600 BRAMs). The infrastructure also incorporates numerous performance counters both FSM dependant and trigger armed that are being read using Xilinx Integrated Logic Analyzers. Some of the obtained data is presented in Table III.

Fig. 10 - Performance counts for the DDR job queues showing trends according to burst sizes

C. Experimental Setup

We were interested in measuring the overall achievable sorting speed of such an end-to-end system on real hardware as well as the impact of memories, while targeting different access patterns. Stalling the sorter externally is a result of full DDR job queues, which in turn means that the sorter fully utilizes the available DDR bandwidth resulting from the corresponding access pattern. We have evaluated three scenarios for Graysort high utility merge sorters representing three different read burst sizes - 1, 2 and 4 blocks of records, while the write of the output is a utilizing fully sequential writing. We aim towards 2048 merged sequences, but we cannot physically store 4 blocks of records for 2048 sequences and that is why the first two setups merge 2048 and the last one merges only 1024 sequences at once. We merge 20,846 Graysort records per sequence for 2048-sequence setups and 41,692 per sequence for 1024-sequence setup (which corresponds to a problem size of 4.269 GB).

D. Evaluation

We have ran every setup a minimum of five times to obtain sufficient data. The mean and bounds of some of the obtained data is presented in Table III, including total runtime, sorter initialize time, number of records passing through the sorter, stalling sources as well as DDR job queues statistics. The datapath width of the sorter is 800 bits, while DDR datapath is only 512 bits and both operate at 200MHz, this explains the differences in stall/enable rates measured.

Fig. 9 and Fig. 10 show the relation between burst sizes and stalling sources of parts of the sorting system. It can be clearly noticed that internal stalling of the sorter is insignificant compared to external stalling caused by full DDR job queues. We can notice that as we increase the burst size, the input DDR memory starts to starve for reading jobs, while the output DDR memory starts to throttle the sorter. The input DDR memory is only read in a burst fashion with random addresses, while the output DDR memory is written fully sequential. We are observing that the latter still becomes the bottleneck once the burst sizes are sufficient.

The setup with largest burst sizes achieves 10.365 GB/s sorting throughput from DDR to DDR memory, which is more than 80% of the maximal theoretical throughput to the available SODIMM memories (12.8 GB/s).

	Read burst = 1 block of records		Read burst = 2 blocks of records		Read burst = 4 blocks of records	
BlockRAM	1,083	(74% of VC709)	1,083	(74% of VC709)	1,072	(73% of VC709)
LUTs	68,842	(16% of VC709)	68,775	(16% of VC709)	67,064	(16% of VC709)
Registers	91,883	(11% of VC709)	91,881	(11% of VC709)	91,351	(11% of VC709)
Throughput	8.266 GB/s	± 0.0002 GB/s	9.584 GB/s	± 0.0001 GB/s	10.365 GB/s	± 0.003 GB/s
Sort runtime	103,293,073	$\pm 1,066$	89,089,393	$\pm 1,869$	82,381,968	$\pm 26,748$
Sorter initialize	116,261	± 231	84,556	± 55	65,003	± 7
Sorter input	42,721,280	± 0	42,721,280	± 0	42,721,280	± 0
Sorter output	42,692,609	± 0	42,692,609	± 0	42,692,609	± 0
Sorter stall internally	83,502	± 515	78,924	± 711	69,773	± 844
Sorter externally stalled on input side	58,558,266	$\pm 733,995$	25,553,380	$\pm 1,184,768$	3,298,525	$\pm 155,302$
Sorter externally stalled on output side	3,092,374	$\pm 1,213,130$	21,569,996	$\pm 1,441,229$	36,392,604	$\pm 146,870$
Read DDR Enable	67,133,440	± 0	67,133,440	± 0	67,133,440	± 0
Read DDR Queue Empty	15,618	± 492	54,241	$\pm 1,711$	474,066	$\pm 25,575$
Read DDR Queue Full	67,126,247	$\pm 2,234$	40,048,814	$\pm 6,529$	31,241,090	$\pm 17,822$
Read DDR Ready	67,148,938	± 492	67,187,303	$\pm 1,683$	67,602,379	$\pm 25,476$
Write DDR Enable	67,088,388	± 0	67,088,388	± 0	67,088,388	± 0
Write DDR Queue Empty	18,868,978	$\pm 1,388,839$	7,203,468	$\pm 558,476$	1,211,970	$\pm 30,918$
Write DDR Queue Full	5,369,752	$\pm 2,043,306$	10,999,177	$\pm 971,318$	14,392,515	$\pm 86,419$
Write DDR Ready	84,890,287	$\pm 1,273,566$	74,006,842	$\pm 513,823$	68,268,683	$\pm 28,582$

TABLE III - Performance counters results measured in clock cycles(@ 200MHz) and resource utilization

This case study show that in practice the total sorting time estimation from section II-A and how important utility is as long as the problem is bandwidth dependant. Latter setup can sort the selected amount of data split into 1024 presorted sequences in 411.909ms. Lets compare to a multirate merge sorter as presented already in any of [7], [8]. They can both achieve throughput that satisfies the available bandwidth of 12.8 GB/s, but can merge only 32 sequences at a time. That would mean they have to read all 1024 sequences from one DDR, 32 at a time and with reprogramming/reinitializing inbetween, and store the resulting data into 32 sequences onto the other DDR memory (what we call a first major run). Then the design would have to be reconfigured/reprogrammed for a second major run that will produce the final resulting sorted sequence. Let's assume that they can magically operate the DDR3 memories at maximal theoretical performance (omitting the need of refresh cycles etc.) of 12.8GB/s and that they can reconfigure/reprogram and reinitialize at zero cost, this would still result in a theoretical minimal time of $DATA_SIZE/12.8 = 333.536ms$ for a single major run and 667.072ms for the whole problem, which is 62% more than what we have achieved for the 1024-sequence setup.

E. Future Work

Graysort benchmark needs to sort 100 TB in total. Assuming a single node, a final major run would have to merge sequences of approximate size $100TB/E$ each to produce the final output. If we sacrifice burst sizes (throughput) we anticipate to be able to fit 8/16K Graysort sequences merged at once. Which could let us run the whole Graysort benchmark on a single FPGA node in only two major runs (which is the theoretical minimum). This would mean we have to produce 6.25GB sequences in the first run and merge them with a single second run.

V. CONCLUSION

In this paper, we propose a high utility parallel hardware merge sorter. We implement highly optimized and scalable

design, including novel approaches for pipelining and allowing variable buffer sizes. We evaluated different designs for merging up to 16384 sequences with up to 66% higher throughput compared to previous best 4096-way merger implementation [9]. A case study targeting a Xilinx VC709 board merges 1024 (2048) sequences of the Graysort benchmark from and to DDR memory, utilizing the DDR-3 memory throughput at 81% (75%) of peak throughput. This is to the best of our knowledge the first implementation of a high utility sorter on an FPGA (end-to-end). With this we enabled FPGAs to be an ideal platform for large problem sorting.

REFERENCES

- [1] P Mishra et al. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.
- [2] A Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.
- [3] J Dean et al. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [4] A Farmahini-Farahani et al. Modular high-throughput and low-latency sorting units for FPGAs in the Large Hadron Collider. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 38–45. IEEE, 2011.
- [5] D Knuth. Sorting and Searching, 2nd edn. The Art of Computer Programming, vol. 3, 1998.
- [6] KE Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.
- [7] W Song et al. Parallel hardware merge sorter. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 95–102. IEEE, 2016.
- [8] S Mashimo et al. High-Performance Hardware Merge Sorter. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 1–8. IEEE, 2017.
- [9] T Usui et al. A Cost-Effective and Scalable Merge Sorter Tree on FPGAs. In *Computing and Networking (CANDAR), 2016 Fourth International Symposium on*, pages 47–56. IEEE, 2016.
- [10] J Gray. Graysort benchmark. *Sort Benchmark Home Page*—<http://sortbenchmark.org>, 2013.
- [11] D Koch et al. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 45–54. ACM, 2011.
- [12] M Deo et al. Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge. *Intel White Paper*, 2016.