

FLiMS: Fast Lightweight Merge Sorter

Philippos Papaphilippou
Dept. of Computing
Imperial College London
London, United Kingdom
pp616@imperial.ac.uk

Chris Brooks
Science Innovation
dunnhumby
London, United Kingdom
Chris.Brooks@dunnhumby.com

Wayne Luk
Dept. of Computing
Imperial College London
London, United Kingdom
w.luk@imperial.ac.uk

Abstract—We have developed a highly-efficient and simple parallel hardware design for merging two sorted lists residing in banked (or multi-ported) memory. The FPGA implementation uses half the hardware resources required for implementing the current state-of-the-art architecture. This is achieved with better performance and half the latency, for the same amount of parallelism. The challenges for the merge operations in FPGAs have been the low clock frequency due to the feedback datapath of the merger being the critical path for timing, and also the high resource utilisation in recent attempts to eliminate/remove the feedback datapath. Our solution uses a modified version of the bitonic merge block, as found in a bitonic sorter, repurposed for performing parallel merge for streaming data. As with the state-of-the-art, it can be considered feedback-less since it only nests one parallel comparison for any desired level of parallelism. This leads to high operating frequency designs, 1.3 times higher than the previous best on our test platform. Since the new design uses 2 times fewer hardware resources, it allows more parallelism and leaves room for additional logic.

Index Terms—bitonic merge, bitonic sort, merge join, parallel, FPGA, databases, operators, distributed algorithms, Zynq, SIMD

I. INTRODUCTION

The merge operation is widely used for a variety of applications, including in many popular sorting algorithms, such as mergesort and timsort, as well as in hardware for database operations, including sort-merge join. It is thus very desirable to accelerate and recent research has proposed a variety of merge accelerators on FPGAs, each of them having their own shortcomings in comparison with the others.

With the rise of machine learning and big data, the need for processing huge amounts of data leads to high-performance architectures and FPGA-designs for specialised tasks. This has also led to the introduction of reconfigurable hardware in the datacenter, for accelerating big data computation, as well as System-On-Chip (SoC) solutions for bringing FPGAs closer to general purpose computation with tighter integration with the operating system and user software. However, using FPGAs for high-performance computation is still an emerging topic, as the inter-chip communication can be a bottleneck, and thoughtful designs that limit the data movement are highly desired.

This paper introduces a novel merge accelerator. Our focus is on the computational aspect and hardware utilisation of this accelerator when the data to be processed already reside locally, such as in an FPGA's block RAM. While this is useful by itself, it can also be used as a building block for

bandwidth optimised designs, since the proposed block RAM usage strategy allows it to be used for big data streaming situations, as they can be used as FIFO queues efficiently.

The basic block of our parallel merger is a repurposed portion of a bitonic sorter, as seen in figure 1. The bitonic sorter [1] is a parallel algorithm for sorting in $\log_2(N) \cdot (\log_2(N)+1)/2 \in \Theta(\log^2(N))$ steps, where N is the number of input elements. As seen in the example in figure 1, the input is a sequence of 16 numbers, a_0 to a_{15} , and the sorted result will be in output after 10 steps. In each step, the bitonic sorter performs $N/2$ compare-and-swap (CAS) operations in parallel. The CAS block takes two inputs, such as a_0 and a_1 and swaps them if a_0 is less than a_1 (for descending order). The sorting order can be changed from descending to ascending and the reverse by just flipping all the CAS blocks in the structure.

Since the bitonic sorter is based on mergesort, it has a hierarchical block, labeled bitonic merger in figure 1, that merges two sorted lists of P elements to a sorted list of $2P$ elements. Many hardware mergers utilise an optimization of the bitonic merger, the $2P$ -to- P bitonic partial merger which outputs just the top half of the result. This block is combined with additional logic to work as a parallel merger for longer lists as streams. All current designs require the two input sequences of the bitonic partial merger to be sorted. *The new idea is based on relaxing this condition.*

Our contributions are the highly-efficient design and the evaluation, as well as automatic Verilog generator scripts for a user-defined P for FPGAs and equivalent SIMD code for experimental evaluation on high-end general-purpose processors. All code is available online at <http://philippos.info/merge>.

II. CHALLENGES

There are several challenges in developing hardware for efficiently merging sorted lists in parallel. Section IV-B explains how these challenges are met by our design.

C1. Long critical path

The most important challenge for merger designs has been the long feedback datapath length. This is because, while the bitonic merge block is pipelineable, there is the need to decide the new input on each cycle. At any point during a merging operation (on descending elements), the merger input will consist of $2P$ elements: the lower P elements of the last output and the top P of the two input queues. Both of those sublists are data dependent to the output of the merger, because the

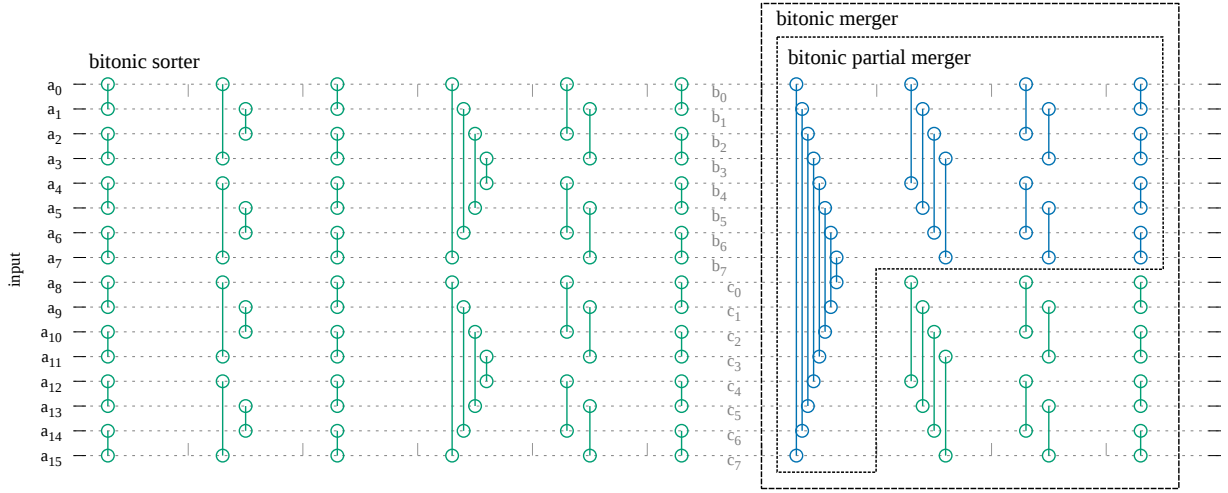


Fig. 1. A 16-to-8 bitonic partial merger block as found in a bitonic sorter for a list of 16 elements

lower P of the output needs to be fed back to the input and the top P next elements will be selected based on what has already been written in output and made way for new elements. On the first cycle, the input is straightforward, as the two sublists will be the top P elements for each of the A and B input queues, but on subsequent cycles, only P new elements will need to be dequeued from the queues and will be compared against the elements that have not yet appeared in the result vector.

The problem with a long feedback datapath length is that it can impact the operating frequency considerably and result in a massively decreased performance, in comparison with how the pipeline would be able to perform without this being a critical path. This has been the main focus of the last research regarding parallel merge operations, to eliminate or remove completely the feedback datapath [2], [3].

C2. Hardware complexity

One other considerable challenge is the hardware complexity. The bitonic merge block is itself relatively compact, as it comes from the bitonic sorter, which is already efficient and pipelinable. What has made the merge hardware expensive in previous works was the overhead of additional or alternative hardware to eliminate the feedback. Too high resource utilisation, such as registers, lookup tables and block memory in FPGAs, can render the design slow, power hungry or not possible to be implemented at all.

C3. Scalability

The hardware utilisation increases with increasing the degree of parallelism P . How the hardware scales according to P is an important aspect of a hardware merger design, because a fast but non-scalable merger will limit P and therefore performance.

C4. Bandwidth bottleneck and skewed datasets

While the memory access throughput is matter of the memory system, it might be allowed for the accelerator to receive

the data from each of the input lists A and B with a fixed bandwidth, less than P . A problem with such a configuration is when the data to be merged is skewed [4]. When the data are skewed, the merger only dequeues from one of the input queues for long periods of time. This results in stalled cycles from underutilising the bandwidth of one of the queues.

C5. Latency

There are cases in which the length of the pipeline can be a bottleneck to performance. Even if a hardware merger is fully pipelineable and massively parallel, it might be required to be used multiple times for small amounts of data apart from big streaming data. For example, the merger may be needed to sort local data as an intermediate step in a parallel algorithm. In such cases the pipeline length is desired to be short.

III. RELATED WORK

In 2016, Song et al. proposed an architecture for parallel merge using FPGAs [4]. This work focuses more on merging multiple sequences and multiple inputs of limited bandwidth, thus using a tree structure that uses a merger structure hierarchically. In figure 2, we can see a high-level view of the merger. This merger works as follows: two input queues, A and B , output 0 to P elements each per cycle, according to how many made it in the last result of P elements. This is known from just the first stage of the $2P$ -to- P bitonic partial merger and is used as a feedback to select the amount of elements to be dequeued from A and B . Since some elements remain from the previous cycle, each input of the bitonic partial merger block needs to be rotated by an offset equal to the dequeued elements. This is done to ensure that the bitonic partial merger gets sorted inputs. The problem with this approach is that the crossbars consisting of barrel shifters create a critical path that increases with P . This leads to lower frequency designs and it does not scale very well [2].

In 2017, Mashimo et al. [2] proposed a lower-latency-feedback architecture, SHMS, to solve the long critical path

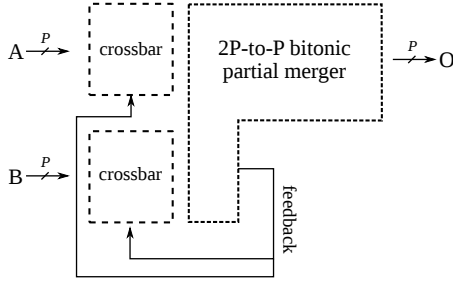


Fig. 2. High-level view of the merger used in PMT [4]

problem with the previous approaches. While achieving much higher frequencies, as high as 3.14 times more than PMT [4] for 32 different input queues (also focusing in multiple inputs), the register utilisation was much worse. For 32 inputs, it uses 7.26 times more registers than PMT. This does not scale well, not from the long feedback datapath length, but due to the high register utilisation.

Recently, Saitoh et al. [3] proposed a feedback-less architecture, MMS, to increase the performance and scalability of the merge operator. MMS uses two 2P-to-P partial merge blocks along with shift registers and an extra comparator and multiplexer. In figure 3, we can see the high-level view of this design. MMS is our baseline to compare the hardware utilisation reduction and performance speedup in our own design for two input queues.

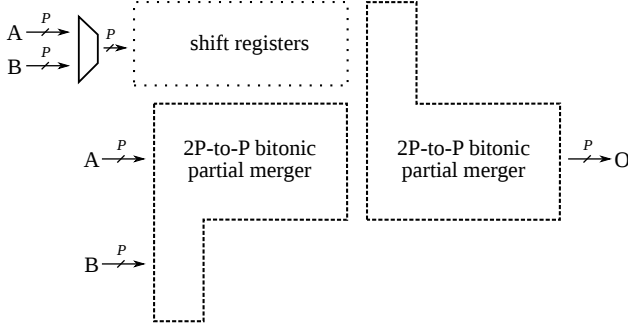


Fig. 3. High-level view of the merger used in MMS [3]

IV. NEW MERGE SORTER DESIGN

The proposed parallel algorithm can be broken down into 3 segments: a *distributed algorithm* for handling the input from multiple banks/queues, the pipelined *bitonic partial merger* (minus the first stage) and finally the *output logic*.

a) *Distributed algorithm*: After the data are written into the BRAM banks in a round robin fashion, a set of independent entities (or nodes) are responsible for controlling the input to the merge sorter. The P entities respond to the same clock and their behaviour can be described by a distributed algorithm.

Let A and B be the input FIFO queues containing the two sorted lists to be merged. Let $A_0, \dots, A_{P-1}$ and $B_0, \dots, B_{P-1}$ be the queues corresponding to the respective banks for A and B . Each of the entities $e_0, e_1, \dots, e_{P-1}$ have as input the

pair of queue heads $(a_0, b_{P-1}), (a_1, b_{P-2}), \dots, (a_{P-1}, b_0)$ respectively, with a_i being the head of A_i and b_i being the head of B_i . Each of the entities outputs one number per cycle when valid.

Each entity e_i has the integer registers cA_i and cB_i to store the last heads that were dequeued from banks A_i and B_{P-1-i} respectively. It also has a register for the sorting network input in_i . On each cycle with valid input, if $cA_i > cB_i$, it means that (see proof) cA_i will make it into the top P in the result of the pipeline and therefore cA_i is copied into in_i . In this case, cA_i is replaced by the head a_i , which is dequeued from A_i , but cB_i will remain unchanged, since it will need to be compared again in the next cycle, being in lower P. The equivalent logic goes for the case when $cA_i \leq cB_i$.

In algorithm A, we can see the pseudocode for the distributed algorithm. This algorithm essentially also includes the first stage of the partial bitonic merge, since it selects the current top P on each cycle and inserts them into registers for the rest of the sorting network to sort and produce the P-sized chunk of output.

```

1 int  $i$ ;                                ▷  $i$  is the entity tag
2 int  $cA_i, cB_i, in_i$ ;                  ▷ 32-bit registers
3 while forever do
4   receive (positive clock edge);
5   if  $cA_i > cB_i$  then
6      $in_i \leftarrow cA_i$ ;
7      $cA_i \leftarrow \text{dequeue}(a_i)$ ;
8   else
9      $in_i \leftarrow cB_i$ ;
10     $cB_i \leftarrow \text{dequeue}(b_{P-i})$ ;
11  end
12 end

```

Algorithm A: Distributed algorithm pseudocode

There is some extra logic regarding the validity of the input which is omitted for the sake of simplicity and portability to different architectures and is relatively trivial to construct. This is useful to handle the end of the input queues correctly.

b) *Sorting network*: The sorting network is responsible for sorting the top-P result of the “distributed algorithm” segment. It is a partial bitonic merger minus the first stage. As it is optimised for this architecture, it is not considered a sorting network on its own and does not sort arbitrary number sequences. For its input though, it behaves correctly (see proof).

c) *Output logic*: The output logic is the most straightforward. On each cycle, if the output of the partial bitonic merger is marked as valid, it is written down as a P-sized chunk of the result, one number in each of the P output banks, which collectively consist the output queue O . The output can then be read in a round robin fashion from banks O_0 to O_{P-1} .

In figure 4, we can see a visualisation of the proposed parallel merger for $P=8$. In table I, an example execution is provided for $P=4$.

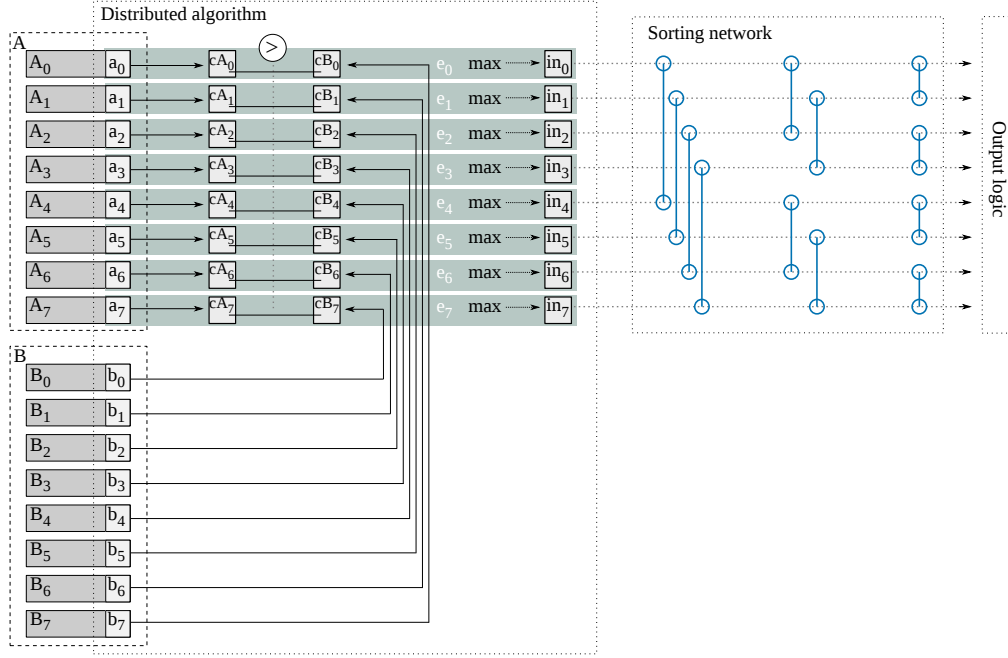


Fig. 4. The proposed parallel merger for P=8

Cycle	Input A	Input B	cA	cB	Output (after the pipeline delay)
0	3 3 4 5 11 16 17 26 26 29	0 7 8 9 12 15 18 19 21 22			
1	3 3 4 5 11 16	0 7 8 9 12 15	17 26 26 29	22 21 19 18	22 26 26 29
2	3 3 4	0 7 8 9 12	17 5 11 16	15 21 19 18	17 18 19 21 22 26 26 29
3	3 3	0 7	4 5 11 16	15 12 9 8	11 12 15 16 17 18 19 21 22 26 26 29
4			4 5 3 3	7 0 9 8	5 7 8 9 11 12 15 16 17 18 19 21 22 26 26 29
5			4 3 3	0	0 3 3 4 5 7 8 9 11 12 15 16 17 18 19 21 22 26 26 29

TABLE I: Merging two descending sequences: Example execution for P=4 and random sorted lists A and B.

A. Proof

In order to prove that the design behaves as expected we will show that it is functionally equivalent to a trivial merger implementation, where on each cycle the input comes sorted to a 2P-to-P bitonic partial merger, as with the merger used in PMT in [4]. We will show that (1) the distributed algorithm block always selects the top P out of the 2P-sized input and that (2) the optimised sorting network always sorts this top P list before writing the result in output.

(1) By induction, if the merger has worked correctly on all previous cycles, it will load $k \in \mathbb{Z} \cap [0, P]$ elements from A and $P - k$ from B, as the top P will be written sorted in output and P new elements will need to be compared with the lower P remaining from the previous cycle. This is because to find the top P out of the two lists we need to have at least the top P of each. At cycle 0, the parallel merger behaves in the same way as the 2P-to-P bitonic partial merge because the input lists are sorted, and therefore we can now focus on the inductive step.

The cA and cB register lists, which hold the last P heads dequeued from each of the lists A and B, receive elements from the respective banks without performing any additional

rotation. Since A and B are written in a round robin fashion, cA and cB are also updated in a round robin fashion and at any stage during the merging, they contain a sorted list each, shifted by an offset $l_A, l_B \in \mathbb{Z} \cap [0, P]$.

On each cycle, the relationship between l_A and l_B is known. In order for the merge operation to output P elements, it will need to dequeue a total of P elements from the lists A and B. Since A and B are sorted, the dequeued elements will be the top k from A and the top $P - k$ from B and therefore each residing in consecutive (or in circle) registers in cA and cB. On each cycle, l_A and l_B will be updated according to the number of dequeued heads, as they will be filled with new elements. That is $l'_A = (l_A + k) \bmod P$ for A and $l'_B = (l_B + (P - k)) \bmod P$ for B, on each next cycle. Since this happens on every cycle, we can conclude that this relation is steady by the following:

$$\begin{aligned}
 l'_B &= (l_B + (P - k)) \bmod P \\
 &\Rightarrow l'_B = (l_B + (P - (l'_A - l_A))) \bmod P \\
 &\Rightarrow l'_B = (l_B - l'_A + l_A) \bmod P \\
 &\Rightarrow (l'_B + l'_A) \bmod P = (l_B + l_A) \bmod P
 \end{aligned}$$

Initially, $l_A = 0$ and $l_B = 0$, and therefore it holds that $(l'_B + l'_A) \bmod P = 0$ on any cycle. Since $l_B = l_B \bmod P$, it

always holds that $l_B = (P - l_A) \bmod P$.

From the distributed algorithm, the first stage is equivalent to the first of the bitonic merge (as seen in fig. 1), which compares each element pair in $\{(Ta_i, Tb_{P-1-i}) | \forall i \in \mathbb{Z} \cap [0, P)\}$, where $Ta_0, \dots, Ta_{P-1}$ are the current top P elements in A and $Tb_0, \dots, Tb_{P-1}$ are the current top P elements in B , sorted in descending order. For the first cycle this is trivial. We notice that whatever the rotation combination from l_A and l_B is, the correct comparisons will be made, as

$$\begin{aligned} (Ta_{(i+l_A) \bmod P}, Tb_{((P-1-i)+l_B) \bmod P}) &\equiv \\ (Ta_{(i+l_A) \bmod P}, Tb_{((P-1-i)+(P-l_A)) \bmod P}) &\equiv \\ (Ta_{(i+l_A) \bmod P}, Tb_{(P-1-i-l_A) \bmod P}) &\equiv \\ (Ta_{(i+l_A) \bmod P}, Tb_{(P-1-(i+l_A)) \bmod P}). \end{aligned}$$

(2) The sorting network receives the correct top- P output from (1) and the task is to sort it correctly. Originally, this structure is supposed to sort bitonic sequences of size P [1]. The output of the distributed algorithm block is a rotated bitonic sequence, as we saw that the comparisons will be rotated by l_A . A rotated bitonic sequence is also a bitonic sequence [5] and therefore the input for the sorting network has the correct property, which completes the proof.

B. Properties

The feedback datapath is just one comparison in parallel. It is the same as MMS [3], with the exception that instead of one, there are multiple parallel comparisons happening at the same time, on which the input is directly dependent for each next cycle. Since the delay of those is less or equal to just one stage of the sorting network, it is not a critical path and the design is considered feedback-less. This does not change with increasing P , helping to solve the challenges C1 and C3.

The length of the pipeline is equal to that of the partial bitonic merger or the bitonic merger, which is $1 + \log_2(P)$. In contrast, MMS [3] has double the stages, as it combines two partial bitonic mergers and it equals to $2(1 + \log_2(P))$. (In this case slight variations might be used in the shift register length but this is according to the implementation [3]). The reduced latency helps to solve the challenges C2 and C5. Having a short latency might also be desirable in alternative parallel architectures where pipelining is not explicit (see VI-A).

The biggest contribution is the reduction of the hardware utilisation. Having one instead of two partial bitonic mergers approximately halves the register and lookup-table utilisation. There are $\log_2(P)$ (for the sorting network stages) times $P/2$ (comparisons per stage) CAS modules plus $P/2$ max modules for the first stage that also select what to be dequeued from the input banks. Each entity which contains a max module contains 3 registers instead of just 2 as found in a CAS module, that would have been used in an unmodified 2P-to-P partial bitonic merger. In contrast with MMS, it also removes the need for the shift registers which also consume resources. As with MMS, the use of a partial bitonic merger as a building block provides much better scalability than SHMS [2]. This helps solve the challenges C2, C3 and C4.

Finally, the fact that it is designed to dequeue elements from different banks enables the usage of longer FIFO queues, such as with using BRAM blocks and can help situations with limited bandwidth and skewed datasets, as described in challenge C4. Additional properties of the distributed algorithm approach are discussed in VI-A and VI-B.

V. EVALUATION

The purposes of this section are to provide a proof of concept, that the design actually works on real hardware, and a simplistic use case and to get actual hardware resources utilisation metrics from a commodity toolchain, such as Vivado 2018.1. Additionally, the experimentation also gives insight about the scalability of the design in terms of the degree of parallelism and its impact on various hardware resource metrics.

A. Implementation for Zynq

We have implemented both the proposed and the state-of-the-art parallel merger designs for a Zynq-based board.

The target system is MYIR's Z-turn board, which has the Xilinx Zynq 7020 FPGA System On Chip (SoC). The device runs GNU/Linux on the two embedded ARM cores and being a SoC, the software can seamlessly interact with accelerator designs. The board is relatively cheap, but the FPGA performance is respectable for our purposes, allowing a 64-way parallel merge. That being said, the data movement speed between the FPGA, CPU and main memory is relatively low in comparison with high-end servers with FPGA co-processors and requires additional steps to take full advantage of the available interfaces [6]. However, the data movement is outside the scope of this project and the relatively low-end device highlights the simplicity of the design and the fast (for BRAM data) and reliable operation. The BRAM functionality can be easily changed to queues for merging streaming data, or even as intermediate queues for merging multiple input queues of throughput one in a hierarchical way [4].

The data movement goes as follows. A simple C program writes two sorted arrays of 8192 random 32-bit integers each into a scratchpad memory. The scratchpad memory is an already reserved main memory region outside the kernel space. In the simple C program, the scratchpad is memory-mapped (mmap) rather than memory-allocated (malloc), which means it is firstly pointed out by a specific physical address. By convention in the Linux operating system, this has the disadvantage of not being cached and thus being slow when accessed intensively by a CPU [7]. Then the C program sends an instruction to a Direct Memory Access (DMA) Intellectual Property (IP) residing on an AXI interconnect, to copy the scratchpad data to the FPGA (for this and the scratchpad to work, the kernel's device tree has to be edited manually [8]). The hardware merger is implemented as an AXI peripheral and accepts one integer per FPGA cycle. The AXI peripheral keeps all data in BRAM blocks which are divided in multiple (2P) banks, and written in a round robin fashion, to provide P-level of merge parallelism. When the data are loaded, the DMA

IP sends an interrupt to indicate completion, and then the C program sends a signal for the merger to start the merging operation. The CPU is busy-waiting until completion, after which a similar approach is followed to retrieve the sorted data back to the scratchpad. Finally, the C program compares the result with the expected and prints out if it is valid.

B. Validation

For validation, the procedure described above is repeated one thousand times to ensure that the peripheral is stable and correctly behaving for different input on real hardware. This is done for all the mergers whose P yielded a low-enough resource utilisation and allowed an implementation on this board.

C. Results

Apart from the validation, the main focus is to get the hardware utilisation as reported by Vivado, for different values of P and for both merger types. The Vivado version is 2018.1 and supports various hardware description language standards. Due to the fact that the Verilog standard supported by the Zynq IP is the 2001 one, which restricts the interface definitions, a python script was developed to write the peripheral code for different Ps. This was useful also for correctly inferring BRAM in a form of a banked memory for different values of P in an automated way.

In figure 5 and figure 6, we can see the Look-up-table (LUTs) and Flip-flop registers (FF) resource utilisation respectively, as reported by Vivado. The top half of each figure represents the actual dataset and the lower half is the improvement measurement based on the previous values. There is also the “fitting” series, which is based on linear regression on each of the datasets before combining them for speedup, to show the trend and identify outliers.

The resource utilisation values are only for the AXI peripheral that contains the respective merge design and excludes the static logic in the rest of the block design, such as the Xilinx DMA IP and the AXI interconnect. Since the utilisation of the excluded logic is the same for all runs, the conclusions would be similar, as each series would be moved upwards by a constant. The rest of the design consumes about 5200 additional LUTs, 6700 FFs and 6 BRAMs and these values remain fairly steady across different implementations and Ps. The AXI peripheral code, still contains some code for the correct use of the protocol so we expect the improvement to be closer to the predicted when P is large.

When the hardware resources of 7z020 were not enough for a design, the device definition was being changed to 7z030 to allow higher resource utilisation and avoid BRAM being translated into other forms of FPGA memory. It is interesting to note that the state-of-the-art design for P=64 is most likely feasible for 7z020, but took Vivado multiple hours to implement, using a relatively high-end computer, due to the resource utilisation reaching the device’s limits.

As we can see in figure 5 and figure 6, the proposal consumes around half the resources in comparison with the state-of-the-art, MMS [3]. For the larger Ps, the LUT improvement

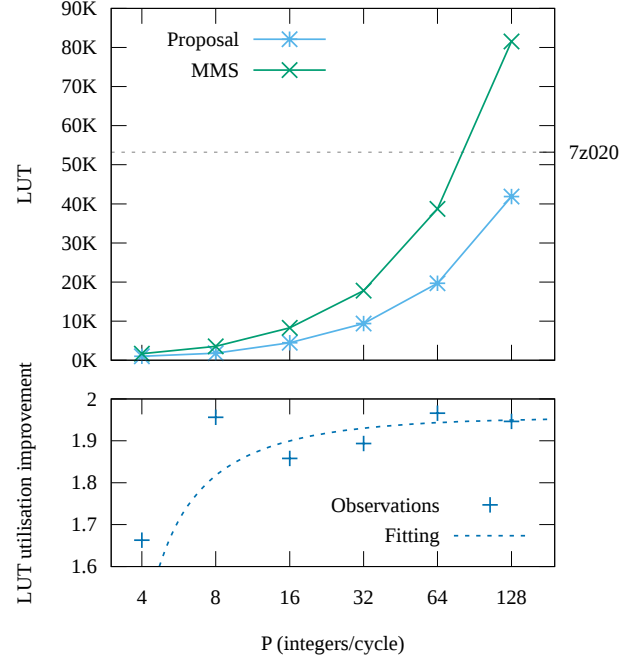


Fig. 5. Vivado-reported LUT utilisation for the respective AXI peripherals

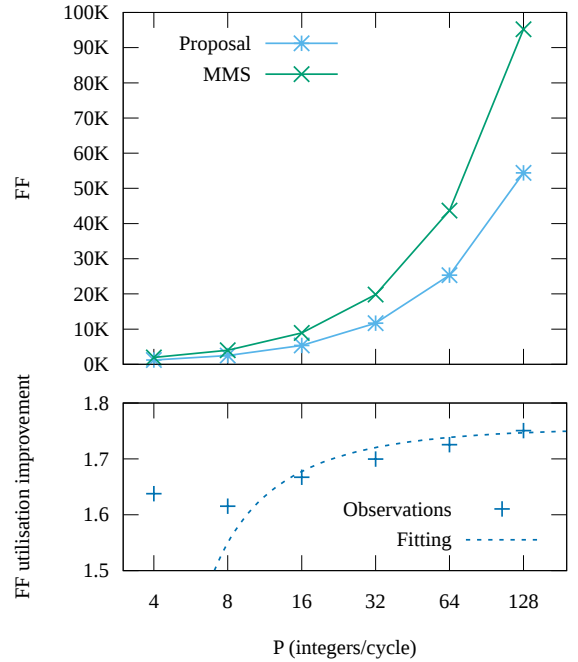


Fig. 6. Vivado-reported FF utilisation for the respective AXI peripherals

is close to 2 (fig. 5 bottom) and the FF improvement is close to 1.8 (fig. 6 bottom).

Finally, we would like to see how the lower resource utilisation improves the operating frequency of the merger. While the critical path in the pipeline has the same width, the new resources allocation resulted in a more efficient routing.

In figure 7, we can see the resulting frequencies. The ranges of the operating frequency might sound underwhelming in comparison with the reported ones in previous works, but this is subject to the lower capabilities of the device and any differences in the rest of the logic. On the Z-turn board, the overall speedup of the proposed merger is about 1.3 times, for the same Ps.

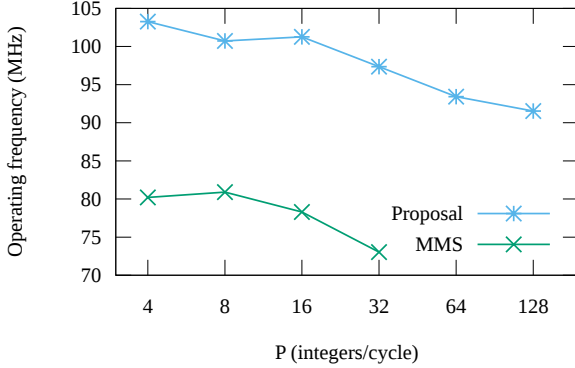


Fig. 7. Peripheral operating frequency based on WNS times

Table II summarises the obtained values. The maximal operating frequencies (“Freq.”) are found using the worst negative slack (WNS) for the same peripheral clock, as described by this method [9]. The LUT and FF values are only for the AXI peripherals, as with the figure 5 and figure 6. The LUT total and FF total are the values corresponding to the entire design, which must be kept under the device limits in order to generate a bitstream. Finally, there is the bitstream column which describes if the design was feasible, in which case it was validated afterwards in the real SoC platform. The larger designs failed due to resource limitations (“limits”) of the device used (for 7z030 as well), or the synthesis was aborted because it was taking too long to complete (“slow”). In the “7z030” case, a bitstream was generated but for the higher-end device.

design, P	Freq.	LUT	LUT (total)	FF	FF (total)	bitstream
MMS, 4	80.2	1648	6996	1967	8659	yes
8	80.9	3545	8764	4014	10713	yes
16	78.3	8269	13349	8922	15621	yes
32	73	17791	22621	19827	26526	yes
64	-	38704	42906	43692	50391	no (slow)
128	-	81518	-	95212	-	no (limits)
FLiMS, 4	103.3	991	6334	1201	7894	yes
8	100.7	1812	7154	2485	9184	yes
16	101.3	4450	9800	5352	12051	yes
32	97.4	9396	14736	11666	18365	yes
64	93.4	19687	25020	25321	32020	yes
128	91.5	41879	47170	54387	61087	no (7z030)

TABLE II: Results summary for FLiMS and MMS [3]

VI. DISCUSSION AND FUTURE WORK

A. SIMD preliminary evaluation

In general, if a design is efficient on FPGAs, it does not necessarily mean that a similar approach would work

competitively in CPUs, since they are two very different architectures. However, it is interesting to see how the challenges change for FLiMS when moving to the Single instruction, multiple data (SIMD) model, as found in today’s general purpose processors. The feedback-less design could translate to higher Instruction-level-parallelism (ILP) due to the weaker data dependencies in the out-of-order execution. The short pipeline length could lead to a lower instruction count and the absence of a centralised control mechanism (since it is a distributed algorithm) could lead to a low branch misprediction rate.

For this reason, we have conducted a simple experiment to see how a manually vectorised merge implementation would benefit a modern processor (Intel i7-8809G) with Advanced Vector Extensions for 256-bit registers (AVX2). The main function produces two sorted random inputs of 2^{24} elements each and runs the merge function to get them merged. The lists reside in main memory without any explicit banking scheme, but they are aligned, for SIMD performance. The baseline is a simple serial merge function in C, where two pointers are accumulated according to what is written to the output after each comparison.

The SIMD implementation uses min, max and shuffle (or permute for AVX2 [10]) instruction combinations to implement the CAS modules in the sorting network in parallel, similar to what is described by Chhugani et al. [11].

There are two approaches to implement the distributed part. The first is to maintain two vectors of 8 seemingly independent pointers that point to consecutive/rotated locations in the input lists and they are updated accordingly to satisfy the algorithm requirements (i.e the vector of pointers for the second list is in reverse order). This is achieved by calling a mask/gather intrinsic, which selectively updates the vectors cA and cB with the respective memory locations as pointed by the pointers.

An issue here is that the gather instruction is relatively expensive in AVX2 [10]. The second approach is to fetch whole blocks of 8 integers and update the cA and cB values with the use of masks. This is more efficient because loading bigger memory chunks requires fewer loads and additional cheaper instructions manipulating local data.

Compiler	gcc 8.1.1	clang 6.0.1	icc 18.0.3
FLiMS: time (s)	0.023	0.023	0.022
write throughput (GB/s)	2.77	2.69	2.81
Serial: time (s)	0.06	0.036	0.061
write throughput (GB/s)	1.03	1.73	1.03
speedup	2.67	1.55	2.73

TABLE III: Preliminary results for manual vectorisation with P=16 on i7-8809G, merging 2 lists of 16777216 32-bit integers each.

In table III, we can see the preliminary results for merging using 3 different compilers. The compiler flags for each case are $-O3$ and $-march = native$ to attempt automatic vectorisation of the serial function and support the respective Intel intrinsics subset for the current architecture. The results are averages of 25 measurements. In this experiment, there were some other optimisations, such as to use P=16 instead

of $P=8$, as emulating a merger with double the output rate proved to perform better.

The successor set AVX-512 supports easier mask manipulation, though equivalent AVX-512 implementations with double the P value yielded similar results on a compatible processor. Input lists of different size to go well beyond the cache size were also tried and there was not any significant performance difference. A multi-thread case has not been explored. With just one thread, the memory bandwidth does not seem fully utilised.

This is not the first bitonic sort-based merger that uses SIMD. Chhugani et al. [11] used a full bitonic merger (not a bitonic partial merger) to feed the lower output vector back into one of the inputs. Therefore, it has more instructions emulating the sorting network. This is adopted in subsequent works [12], [13]. One question is whether the instruction issue width is wide enough to exploit the weakened dependencies with FLiMS. Another question is whether the removal of the rotation-related logic with FLiMS, in comparison with a software-equivalent of the FPGA-related work in figure 2, has further improved CPU performance. We may have improved the cycles by a fraction, but other parts of the routine, such as the CAS units emulation, that consists of multiple intrinsics, may already occupy most of the time.

In conclusion, the best merging algorithm for SIMD units depends on the variety of intrinsics the processor supports and how they are implemented. Thoughtful code optimisations or processor support for faster or more specialised intrinsics, such as CAS or efficient vector reverse could possibly make it (or similar approaches) the best option for general purpose processors.

B. Future work

There are many ways this topic can be expanded, mainly regarding applications. The merger could be applied in FPGA sort designs, such as a full mergesort implementation. It can be used to replace all mergers in sorting structures, such as in parallel merge trees [4], where more than two input queues are merged. The distributed algorithm in our proposal takes place in just one FPGA and was used to explain the functionality, but when combined in a merge tree, it could be used to merge multiple input queues from different nodes (or memory locations). It can also be applied in a full H/W sort-merge join operation for databases to improve current approaches [14]. In order to sort using multiple attributes, we would need a modification to perform stable sort, as the regular mergesort, or solve the tie-record problem for satellite data/key-value pairs on the fly at a hardware cost [3]. It would be useful to also explore implementations for alternative architectures such as OpenCL for GPUs and manual vectorisation with multi-threading for high-end CPUs and co-processors. The SIMD approach for general purpose processors sounds applicable to efficient database sort-merge join implementations as well, with additional vector registers to be occupied by the keys [12]. It would also be interesting to mathematically prove any optimality properties this design exhibits.

VII. CONCLUSIONS

The new parallel merge architecture achieves better FPGA performance for half of the hardware resources than the state-of-the-art architecture [3]. The lower resource utilisation allows higher throughput designs, creates room for additional logic and yields a higher operating frequency for the same degree of parallelism. The new merger design can be seen as an efficient building block for a variety of applications.

ACKNOWLEDGMENT

This research was sponsored by dunnhumby. The support of the United Kingdom EPSRC (grant number EP/I012036/1, EP/L00058X/1, EP/L016796/1, EP/N031768/1 and EP/K034448/1), European Union Horizon 2020 Research and Innovation Programme (grant number 671653) is gratefully acknowledged.

REFERENCES

- [1] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, 1968, pp. 307–314.
- [2] S. Mashimo, T. Van Chu, and K. Kise, "High-performance hardware merge sorter," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, IEEE, 2017, pp. 1–8.
- [3] M. Saitoh, E. A. Elsayed, T. Van Chu, S. Mashimo, and K. Kise, "A high-performance and cost-effective hardware merge sorter without feedback datapath," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2018, pp. 197–204.
- [4] W. Song, D. Koch, M. Luján, and J. Garside, "Parallel hardware merge sorter," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, IEEE, 2016, pp. 95–102.
- [5] G. Zachmann, "Adaptive bitonic sorting," *Encyclopedia of Parallel Computing*, David Padua, Ed, pp. 146–157, 2013.
- [6] S. Erusalagandi, *Leveraging Data-Mover IPs for Data Movement in Zynq-7000 AP SoC Systems*, WP459 (v1.0), 2015.
- [7] L. Nava, *Contiguous memory on arm and cache coherency*, 2015. [Online]. Available: <https://aelseb.wordpress.com/2015/04/11>.
- [8] K. Kouzoubov, *Sample minimal Vivado project for Parallella FPGA - sample DMA*, <https://github.com/Kirill888/parallella-fpga-dummy-io>, 2016.
- [9] E. Billauer, Vivado: Finding the "maximal frequency" after synthesis, 2017. [Online]. Available: <http://billauer.co.il/blog/2017/01/vivado-minimal-period-timing/>.
- [10] Intel (R), *Intel intrinsics guide*. [Online]. Available: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>.
- [11] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.
- [12] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009.
- [13] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ACM, 2010, pp. 351–362.
- [14] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, ACM, 2014, pp. 151–160.