An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics

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

Chris Brooks
Science Innovation
dunnhumby, UK
Chris.Brooks@dunnhumby.com

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

***Abstract*—This work improves on the latest research about sorting acceleration on FPGAs. An efficient design is introduced for sorting data that fit on-chip, with the additional functionality to merge sorted sublists recursively, for an input of arbitrary length. While many-leaf mergers are conventionally single-rate, a novel technique in our approach is to use a parallel merge tree only for the latest stages of the merge tree, to enable bandwidth-adapted multi-rate many-leaf merge. Our open-source RTL generator produces sorting peripherals with customisable parallelism and data format. We evaluate our FPGA design as an 128-bit wide peripheral on an MPSoC platform, with a speedup of up to 49 times over the A53 core for sorting, and up to 27 times speedup for our specialized database analytics application.**

***Index Terms*—FPGA, sorting, generator, database acceleration, mergesort, analytics, high-throughput, stream processing, distinct count, group by**

I. Introduction

FPGAs are increasingly being introduced as means to accelerate big data and database analytics [1], [2]. A considerable fraction of such operations relies on sorting [3], [4], or their performance can be improved when operating on sorted data after making the appropriate algorithmic changes [5]. For this reason, the research for efficient FPGA-based sorters has been fairly active recently, with one of the main considerations being the disproportionate advantage of the throughput with which the CPU accesses the main memory.

A challenge for the hardware designer is to maximise performance while adhering to a number of constraints related to the storage medium and data movement. These include the storage capacity, the data width and number of ports, the latency for different access patterns [6], address segmentation and NUMA-related restrictions. In research, using a representative storage medium is not always the case, such as with the majority of sorters mentioned in the related work, that are BRAM-based even though are destined for large-scale sorting.

When performing mergesort using FPGAs, there currently seems to be a trade-off for supporting either high-throughput/few-leaves or single-rate/many-leaves. For high-throughput mergers, a limitation of the current solutions is their scalability to a high number of inputs, and require a high aggregate bandwidth. The most recent variation has only reported merging for up to 64 inputs [7]. On the other hand, many-leaf mergers can merge a couple of thousand input lists simultaneously, which can have significant benefit due to the reduced number of data passes. A data pass involves reading every data from DRAM and writing them back once and is considered a performance bottleneck. However, an evaluation [6] that demonstrated saturation of the PCI Express bandwidth used 800-bit records, which may not always be the norm.

Our sorting solution combines features from the state-of-the-art hardware designs for mergesort, while taking into account different architectural constraints. Our sorter generator produces a hardware merge sorter in Verilog, with any desired datapath length, data width and payload width. The same logic is also used to perform efficiently the first sorting pass, that is to produce a series of sorted sublists, before merging them together. This functionality, while crucial to applicability, performance and resources utilisation, is absent from most large-scale sorters, which usually assume already-sorted sublists or rely on existing hardware sorter modules.

Finally, we also evaluate this solution for our in-house specialised use case, in order to demonstrate its applicability to database analytics. The task is to calculate the number of distinct values per key (group), from an input consisting of key-value pairs. A fully-pipelined high-throughput stream processor is attached to the output of the sorter as an add-on, in order to produce the results on-the-fly. By achieving task-pipelining, this additional operation becomes transparent, as no immediate data need to be stored temporarily for processing afterwards.

The list of our key contributions is as follows:

- The first merge sorter design applicable to both efficiently sorting small lists and recursively merging them together.
- The first open-source many-leaf merge sorter. A Verilog generator script produces merge sorters with customisable bandwidth, data and payload width.
- Improvements on current techniques, such as by closing the gap between high-throughput and many-leaf sorters.
- A full system evaluation for sorting and for a specialised analytics accelerator (distinct counter with groups).

II. Related work

In this section we present a concise survey on the latest research related to merge sort-based FPGA sort acceleration.

A. High-throughput sorters

High-throughput merge sorters can merge a number of sorted lists simultaneously, while providing an output rate of more than 1 element per cycle. This can be achieved by

building a merge tree (PMT [8]), mainly consisting of high-throughput mergers of 2 lists and FIFO queues.

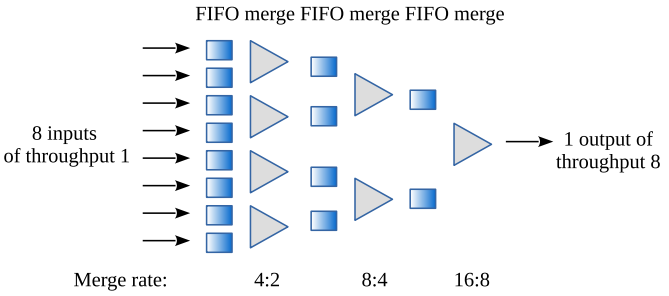

Fig. 1. Parallel merge tree (PMT [8]), for 8 input lists

Figure 1 shows how these building blocks can be arranged to merge 8 sorted inputs of throughput 1, with an output rate 8. The 'merge rate' denotes the throughput capacity of the merger for each level. For instance, a merger of rate 4:2 merges two inputs of width 2 (times the element width) and outputs two elements per cycle. The difference in widths from level to level is managed by rate converters and the appropriate stall signals. A simpler unoptimised tree [3], [9] can also be implemented with mergers of the maximum throughput, which also squares the aggregate bandwidth requirements of the inputs [8].

B. High-throughput sorter building blocks (2-way mergers)

A merger for 2 already-sorted sublists of fixed length can be modified to merge 2 lists of arbitrary length in streaming fashion. Then, it can be used as building block for a parallel merge tree, to merge many lists simultaneously.

At some point, the most attention was drawn on removing the expensive feedback length that existed in traditional merger designs [3], [8], that prevented scalability in terms of operating frequency for an increased degree of parallelism (w) (HMS [10], MMS [11], VMS [12]). Lately, FLiMS [13] and WMS/EHMS [7] offered further improvements by focusing on efficiency, for minimising the required hardware resources, usually with a subsequent improvement in operating frequency.

Most of them are based on two popular sorting networks: Batcher's odd-even mergesort and the bitonic sorter [14]. Those two sorting networks are pipelines of $(log_2(n) \cdot (log_2(n) + 1))/2$ stages, and each stage consists of up to $n/2$ compare-and-swap (CAS) units, where n is input size in powers of 2. The functionality of the CAS units is to sort two values. Collectively, a sorting network in its entirety can sort any list of n inputs. These two sorting networks have the same number of stages and can be built hierarchically using 2 sorters of half the input and an appropriately sized merger to merge two equally-sized sorted sublists. The merger part consumes the last $log_2(n)$ stages in both sorting networks.

Figure 2 summarises the architecture of FLiMS [13]. The CAS network in the bottom half of the figure shows how the merger part of the respective sorting network was adapted for streaming merge (for throughput $w = 4$). The CAS units are coloured in green. FLiMS uses the merger of a bitonic sorter

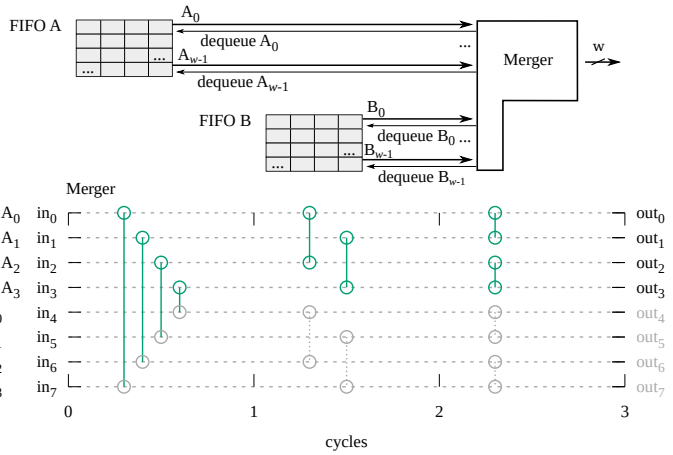

Fig. 2. FLiMS [13] architecture (top) and CAS network (bottom)

of $2w$ elements and prunes all CAS units (greyed out in the figure) related to the unused sorted bottom w in the output (out_4 to out_7 in figure 2). The first stage (half-cleaner) also acts as a selector logic, with its CAS units replaced by MIN units (see algorithm A, ignoring the highlighted code).

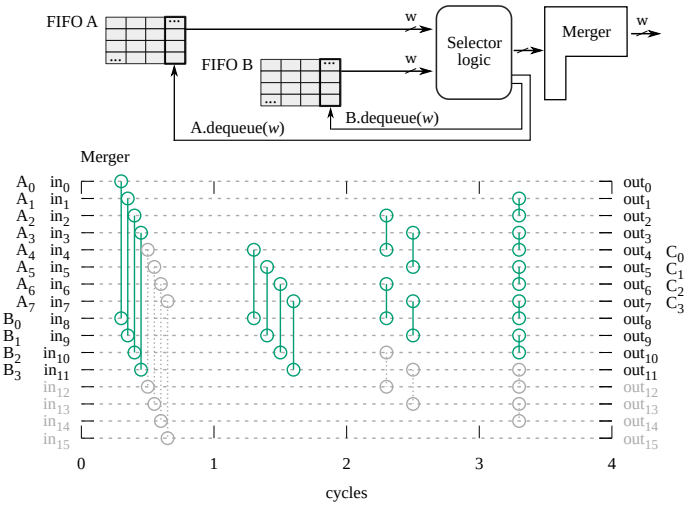

Fig. 3. WMS [7] architecture (top) and CAS network (bottom)

Another state-of-the-art approach is WMS [7], shown in figure 3. It utilises the merger of odd-even mergesort instead, and for $4w$ inputs (hence the higher latency), but with additional pruning, as the first input list has double the length of the second one (and some other optimisations). Still, WMS fundamentally requires more resources and pipeline stages, also adding the complexity of a selector logic. One advantage of WMS is the centralised selector logic, which dequeues whole rows of length w from the input FIFOs. This could be useful in special occasions, such as when dequeuing very narrow data from block RAM.

In our solution we adopt FLiMS for our PMT, due to its decentralised nature and the moderately-wide input widths in our uses, which was more promising for our implementation.

C. Many-leaf sorters

Many-leaf (or large-utility) sorters try to merge as many input queues as possible (currently up to a few thousands [6], [15]), with minimal hardware resources. The comparators are proportionate to the number of merge stages in the merge tree ($O(log_2(k))$, where k is the number of input queues).

Even though their output rate is a single element per cycle, they can still have a considerable advantage over the current high-throughput sorters. This is especially true for large-scale sorting, because it requires less merge phases, and subsequently fewer data passes. This was first demonstrated for data stored in the on-chip block memory (BRAM) [15] and afterwards with data stored in FPGA's DRAM in a bare-metal configuration [6]. The latter was a more realistic use case, as DRAM is more appropriate for large-scale data.

D. Small-scale sorters

This category of sorters refers to efficient sorters of on-chip data, such as the aforementioned sorting networks [14], [16]. They can be used to replace the first stages of mergesort [6] as well as in other FPGA sorting approaches [17].

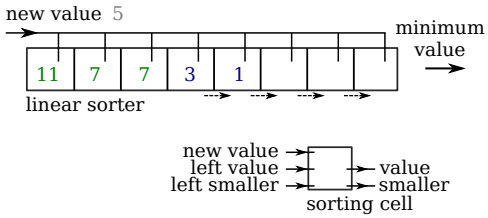

Fig. 4. Linear sorter example

A simple small-scale sorter is the linear sorter [18], [19] and can be seen as a parallel version of insertion sort [20]. As shown in the example of figure 4, a new value is inserted in every cycle. This value is broadcasted across all sorting cells, which compare it against their current value. The result of each comparison is known to the immediately right cell, after a sub-cycle delay. When a cell currently holds a smaller value, it adopts the value on its left. When a cell is the leftmost to have a value smaller than the new value, it adopts the new value instead. This procedure produces a sorted list in $n + 1$ cycles, where n is the input size. There are many variations, such as fast systolic sorters at the expense of latency [21].

III. Solution

Our proposed design works in 3 phases internally, with regards to the data movement. As seen in figure 5, initially, the unsorted data are streamed from DRAM directly to the sorter, and the immediate results are stored to BRAM. These data in BRAM are sorted in chunks of k elements.

In the second phase, the sorter works as a merge sorter of k sorted lists of length k from BRAM. In other words, the sorter is able to provide a sorted list of size $k \times k$ with a single pass.

The third phase is the merge operation from data in DRAM. The same phase applies to input lists of arbitrary length. Thus, the sorter can sort an input of size k^{P+1}, where P is the total number of passes. During this phase, the merger still reads

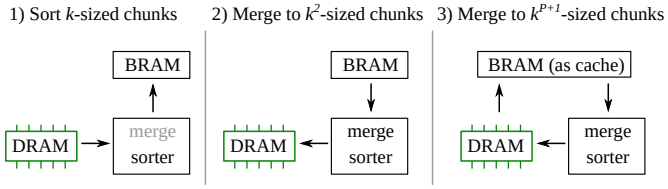

Fig. 5. Data movement in high-level

from BRAM, which works as a cache to DRAM, to hide the latency from accessing with a random access pattern. This memory is already divided in k lists of length k, which now work as buffers and filled upon request, with a throughput of w records per cycle. The building blocks of our solution are described in detail in the following subsections.

A. Merge-capable linear sorters

The linear sorter is modified such that it can also be used for merging a number of sorted lists. The idea is that, a many-leaf merger essentially outputs the minimum head of all queues to be merged, representing the minimum of all unprocessed input. And this can be emulated by a linear sorter. By appending the source list index to the value as a payload, we can keep track of the source of the extracted head. This is useful for fetching the next head to be processed on the immediately next cycle. In this way, we can keep exactly one head for each of the input queues, and appropriately representing the merge operation.

Figure 6 illustrates an example for merging 8 lists (named A through H). The minimum head (1) is extracted as an output, as it represents the minimum of the heads of already sorted lists, which subsequently also represents the minimum of all input. The next head is indicated by the source list (A), and the next head (6 from list A) is dequeued to be processed on the next cycle.

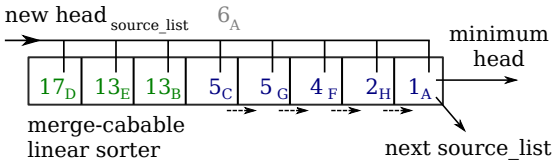

Fig. 6. Merge-capable linear sorter example

In order to make it practical for general use, some additional functionality is added, such as for supporting list endings:

a) Enqueue a 'bubble': This is used to adjust merge capacity, as well as to support the ending of an input queue.

b) Flush current values: Only useful in sorting mode, this is triggered automatically, as soon as the sorter fills up, in order to support fully-streaming of multiple chunks for sorting. The cells with flushed values behave as shift registers to eventually propagate the sorted chunk, while ignoring new requests.

B. Modified parallel merge tree

The modified linear sorters are appropriate for both sorting and merging, but they are limited to single rate input/output. In this subsection, we propose to use an upgraded parallel merge tree (PMT [8]), to merge the results of multiple merge-capable

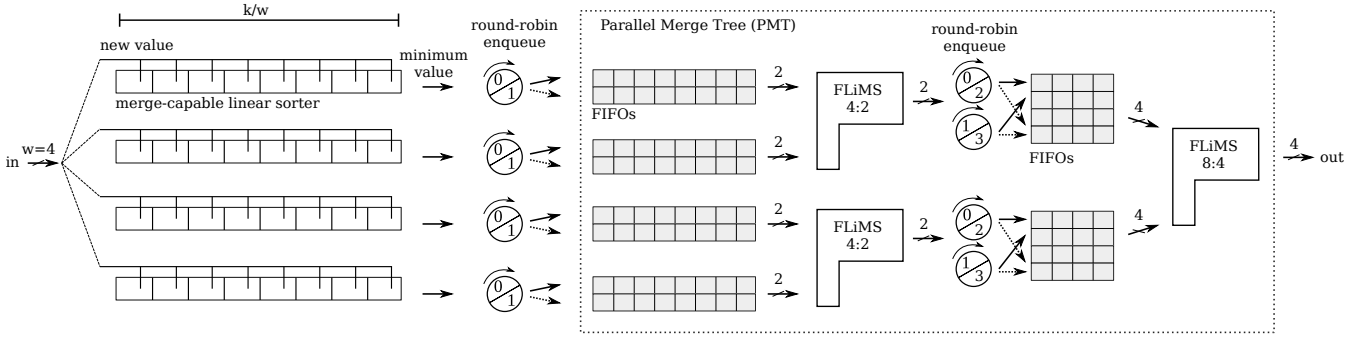

Fig. 7. The merge-capable high-throughput sorter (for a throughput of 4). Wire widths are multiples of the value width.

linear sorters. In this way, we can efficiently support a custom amount of throughput, as well as many-leaf merging.

The parallel merge tree here adopts FLiMS [13] as a building block, an efficient high-throughput merger for 2 lists.

Figure 7 shows how the parallel merge tree increases the throughput of our merge sorter. First, there are w linear sorters, where w is the degree of parallelism. Each linear sorter has a length of k/w, where k is the total merge capacity or the sorted chunk size. This structure is able to both sort an input of k elements, and merge k sorted lists of arbitrary length.

In sort mode, the value of a two bit counter is appended to the most significant bits of all output of the linear sorters. This 2-bit counter is incremented when a new sorted chunk is flushed to the PMT. It is used by FLiMS to be able to give the correct sorting priority to independently sorted chunks. Note that one bit is not enough, as during the half-cleaners stage, between one value with a chunk order of 0 and one with 1, it is not possible to differentiate easily which came first.

C. On-chip buffers

This subsection describes each of the buffers required for efficiently merging or sorting using our approach. Note that there are many such design choices.

a) PMT FIFOs: In order for the first phase to be able to support bursts of length k, the FIFOs after the linear sorters have a depth of k/w elements. This is required because the linear sorters are flushed automatically when full. A ready signal represents the space availability for a forthcoming burst. With respect to the FIFOs of all levels in the merge tree, each FLiMS module of a previous level (as well as the linear sorters) must stall when there is no space left for its output.

b) Burst buffer: The output to DRAM is written in bursts. A burst buffer is near the output and is enabled from the second phase and beyond. When no space is left, the FLiMS of the last level in the PMT is stalled.

c) List buffers (BRAM): The block memory is divided in $w \times w$ banks. The first dimension is for the w linear sorters, which merge an independent range of k/w lists. The second dimension is for being able to fill a buffer with a throughput w. In the first phase (sort), the output of the PMT is written across all w partitions, with round-robin priority per sorted chunk, to distribute the merging workload evenly across the w linear sorters. In the second phase, each linear sorter reads from its own partition with a throughput of 1, and the merged output is redirected to DRAM.

d) Fill request buffers: In the third phase, the linear sorter functionality remains similar, and each list buffer of size k represents data in DRAM. When a list buffer is halfway empty, a request is stored in a fill request queue. When a list buffer is empty or a fill request buffer is full, a stall signal is triggered for the respective merge-capable linear sorter.

e) Next head buffers: For an efficient merge operation, there needs to be a buffer between the linear sorters and BRAM. This is because the BRAM introduces a latency of 1 cycle and this could potentially halve the overall throughput. The next head buffers prefetch the next head for each of the k lists, as soon as a head is read, so that the BRAM latency is hidden.

IV. Properties and optimisation

A. As a small-scale sorter

Our approach provides a good combination of logic, storage and time requirements. Table I, compares the complexities of a series of small-scale sorters. The proposed sorter combines the minimum for storage and time, that are in $O(n)$ and $O(n/w)$, where n is the input size and w the throughput. The only exception in time is that of sorting networks, which are only practical for small n. With respect to the logic, there are two competing streaming sorting networks [22]. The first, although with lower complexity, it requires a rather high value for n to become more logic efficient, that would have more implications on its storage which is in $O(n \log^2 n)$. The second one has the least logic, but as its time is not linear, its overall throughput as a stream processor is reduced considerably.

A sorter is fully-streaming when processes consecutive chunks without the need to wait in-between [22]. Our proposal is fully-streaming for chunks of size $k = \sqrt{n}$ (phase 1). During phase 2, the logic is being reused for merging, unless we double the logic. When processing multiple chunks separately, the overall throughput is getting halved. For data that fit on-chip (k^2), in contrast to [23], there is virtually zero latency between reading all the unsorted data and writing the result.

B. As a large-scale sorter

In terms of logic complexity, it is inferior to many-leaf mergers. The number of comparator cells in many-leaf mergers [6], [15] is $\log_2 k$, whereas our proposal requires a comparator

TABLE I
COMPARISON WITH VARIOUS SMALL-SCALE SORTERS, INSPIRED BY [22], [23]

Approach	Throughput	Logic	Storage	Time	Storage type	Fully-Streaming	Merging
Linear sorter [18]	1	$O(n)$	$O(n)$	$O(n)$	FF	Yes	No
Bitonic and odd-even mergesort [14]	n	$O(n\log^2 n)$	$O(n\log^2 n)$	$O(\log^2 n)$	FF	Yes	No
Interleaved linear sorter [24]	$1 \leq w \leq n$	$O(wn)$	$O(wn)$	$O(n/w)$	FF	Yes	No
Streaming sorting networks [22]	$2 \leq w \leq n$	$O(w\log^2 n)$	$O(n\log^2 n)$	$O(n/w)$	BRAM	Yes	No
Folded streaming sorting networks [22]	$2 \leq w \leq n$	$O(w)$	$O(n)$	$O(n\log^2 n/w)$	BRAM	No	No
Partially ordered set linear sorter [23]	$1 \leq w \leq n/2$	$O(w^2+wn)$	$O(n)$	$O(n/w)$	BRAM	No	No
Proposed design	$1 \leq w \leq \sqrt{n}$	$O(w^2\log w+\sqrt{n})$	$O(n)$	$O(n/w)$	BRAM/DRAM	No	Yes

per list. However, the related work (II-A to II-C) require and assume pre-sorted sequences, making them more appropriate for merging only. In many-leaf sorting a small-scale sorter is required to be present anyway, and for small-scale sorting, our proposal offers improvements over the related work.

One recent large-scale FPGA sorter uses samplesort as its main algorithm [17]. Although not merge-based, the partitions need to be sorted individually, using an open-source small-scale sorter [25]. In principle, samplesort's advantage is the elimination of a merge phase, although many-leaf merge can hide this phase if it happens on-the-fly. Being mostly focused on a specialised full-system implementation, the evaluation for small lists, narrow data and different distributions is left for future work. It is also not appropriate for data with many duplicates. Nevertheless, samplesort could still benefit from our work as an efficient small-scale sorter.

C. Rate mismatch

Parallel merge trees can suffer from rate mismatch, that happens when the input data distribution lead to underutilisation of certain mergers, resulting in reduced throughput [6], [8], [15]. A notable example is when the data are already (or nearly) sorted [8]. In our proposal this can be avoided in small-scale sorting. In the sort (1st) phase, the data arrive in bundles of w, and each value goes to a different linear sorter. This will lead to sets of w sorted chunks of similar distribution and therefore the utilisation of the merge tree will be uniform. In the merge (2nd) phase, each of the w partitions of k/w sorted chunks will have similar distribution overall, as the chunks are stored across the partitions with a round-robin priority. However, as each sorted chunk will end up in its entirety to the output (ignoring cases of duplicates), the PMT will need to have buffers of adequate length. During the 3rd phase, as the sorted regions can exceed the on-chip buffer sizes, the sorter's throughput will eventually slow down to 1 for sorted input.

D. Skewed datasets optimisation

Another contributor to rate mismatch is when there are a lot of duplicates in the input (skewed datasets). PMT [8] proposes a simple solution which causes the merger blocks to 'oscillate' when there are duplicates. This ensures that the input queues are consumed with a similar rate, that collectively balances the utilisation of the merge tree. However, PMT's mergers inherit the long feedback problem, which was addressed in subsequent works [7], [11]–[13]. We propose the equivalent optimisation for FLiMS [13] (figure 2), while keeping the decentralised nature of the selector stage. The code for the new selector units is illustrated in algorithm A. An 1-bit register called $side_{Last}$ represents the queue out of which the head was dequeued during the previous cycle, and is appended to the least significant bit in the comparison, to enforce a sort priority on equal values.

```
1  int i;                                          ▷ i is the entity tag
2  reg cA_i, cB_i, in_i;                  ▷ registers of data width
3  reg side_Last;                                  ▷ 1-bit register
4  while forever do
5      receive (positive clock edge);
6      if {cA_i, !side_Last}<{cB_i, side_Last} then
7          in_i ← cA_i;
8          cA_i ← dequeue(A_i);
9          side_Last ← 0;
10     else
11         in_i ← cB_i;
12         cB_i ← dequeue(B_{w-1-i});
13         side_Last ← 1;
14     end
15 end
```

16 **Algorithm:** A - Modified MIN_i unit pseudocode

In order to prove that FLiMS continues to sort correctly, the selector stage must be shown to still produce a bitonic sequence [13] (up to one local maximum and up to one local minimum). On each cycle, each of the w MIN_i units compares a_j to b_{w-1-j}, where i is a rotation of j by a common offset $o \in [0, 1, ..., w-1]$ and a and b represent the input lists in ascending order. This emulates a half-cleaner, that selects (and dequeues) a total of the smallest w elements from a total of $2w$ elements. The resulting sequence has up to one local maximum (if we consider the rotation). Our modification only takes effect when $\exists h \in [0, 1, ..., w-1] : a_h = b_{w-1-h}$. We notice that this happens only consecutively and for the maximum value, as $(a_0, a_1, ..., a_{w-1})$ is monotonically increasing and $(b_{w-1}, b_{w-2}, ..., b_0)$ is monotonically decreasing. As a consequence, the position of the maximum (split) in the bitonic sequence can be at the start, end or between this region of duplicates. The $side_{Last}$ registers correspond to the last half-cleaner decisions, which is of the form $\{1\}^m\{0\}^n, m+n = w$, after considering the offset. The region of duplicates will be a sublist of this expression, with its 1s and 0s replaced by consecutive duplicates from a and b respectively. As a result, there will be up to one local maximum (split) in this region, and therefore up to one local maximum in the entire half-

cleaner result, which consists a bitonic sequence. Dequeuing consecutive entries ensures the integrity of the input data.

E. Throughput optimisation

One complication from using a form of linear sorters in our solution could be a reduction to the operating frequency. This is because linear sorters require broadcasting to multiple comparator cells, that becomes expensive for many cells [20]. Our proposal solves this problem partially, due to the arrangements to achieve high-throughput. It essentially splits a sorter of length k to w linear sorters of length k/w. Therefore, each broadcast is more local, as there are w independent linear sorters that are w times shorter, for the same chunk size k.

An optimisation is proposed to further increase the operating frequency of the generated sorters and saturate the bandwidth of the port. Since our approach allows a custom degree of parallelism (w), a sorter with a width multiple of the datapath width can be generated instead. For example, by doubling the value of w and appropriately interleaving the input and output, the sorter logic can operate at half the operating frequency of the base clock. This emulates a sorter of half the value of w and double the frequency (as in our evaluation). This modification is relatively inexpensive, as only w changes in the logic complexity (see table I) and w is relatively a small number. This optimisation can also help in situations where the value width is high and the comparator logic latency contributes to the critical path of the design.

F. Input buffer rate

A design choice for balancing the resources utilisation for a specific device is to use additional BRAM for the 3rd phase (merging from DRAM). Instead of only reusing the BRAM of the 2nd phase (merging from BRAM, totalling $k \times k$ values), our generator script accepts one more parameter, the buffer rate (r). It represents the ratio of the BRAM to be used as input buffers in the 3rd phase over the BRAM required for the 2nd phase. By increasing r, the input buffers are lengthened accordingly, improving the performance when merging from DRAM. As the input buffers can be filled with longer DMA transfers, a higher throughput can be achieved [26] and the memory access pattern from DRAM becomes more linear.

V. Evaluation

In this section, we evaluate example applications of the sorter design. As our proposal has a plethora of applications, all source code is provided for further exploration[1].

The evaluation platform was Avnet's Ultra96 board, which combines a Xilinx Zynq UltraScale+ ZU3EG device with 2GB RAM shared between the programmable logic and the four A53 (mid-range) ARM cores, running Linux at a frequency of 1.2 GHz. The generated peripherals have an 128-bit-wide datapath and most data transfers are done by a DMA engine.

The software part of the sorter behaves as a callable C function. The baseline for comparing to CPU sorting is GCC C++ standard library's std::sort() implementation (single-core), that is considered a widely-available well-performing baseline for evaluating high performance sorting in software [27].

[1]**Source code available:** http://philippos.info/sorter

A. Sorter generator script

The generator script produces a Verilog sorter for user-specified values for w, buffer rate r and a flag to enable the skewness optimisation. The remaining parameters can also be modified after generation: total merge capacity or inner chunk size k, value width d, comparison width (as means to facilitate payloads) and various queue lengths. The combination of w and the data width specifies the throughput of the sorter (e.g. $w = 4$ and 32-bit values for reading and writing at 128 bits per cycle). All integer parameters are in powers of 2, except the comparison width.

B. Use case 1: Sorting

The first application for evaluation is sorting a list of 32-bit integers. The first selected configuration was a sorter with $k = 128$, $w = 4$, 128-bit datapath and an operating frequency of 250MHz. This frequency can saturate the achievable bandwidth of that port at 128 bits per cycle [28] and was achieved with the throughput optimisation (internally with $w = 8$).

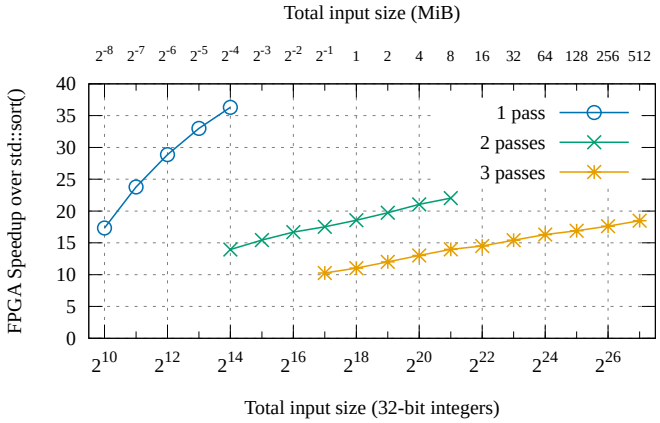

Fig. 8. Sorting performance

As shown in figure 8, the sorter achieves a speedup of up to around 36 times for data that fit in a single pass ($k^2 = 16384$) and above around 15 times for all other input sizes. A big contributor to performance is the number of passes, especially for the 2nd phase and above, as the memory access pattern becomes less linear. Different combinations of chunk sizes for the same number of passes seems to have a negligible variation in performance (showing the maximum in the figure).

Also evaluated is a sorter configuration optimised for single pass sorting. With k set to 256, it is able to sort 65536 32-bit values in a single pass and with a speedup of 49x over the A53 core and 19x over an Intel i7-8809G core running at 4.2 GHz (in turbo boost). Note that the i7 is on a different system and with a more advanced memory. Table II illustrates the results of the evaluation, as well as the hardware resource utilisation.

C. Use case 2: Distinct count with groups

The second application is for accelerating database analytics. Our emerging task is to find the distinct count of keys per group. The input is a list of $< group\_id, key >$ pairs, with each field being 32-bit wide, totalling 64-bits per

pair. The resulting data are 64-bit-wide pairs as well, with the last 32-bit value representing the distinct count of the respective group. For instance, the distinct count of the list $(A1, A2, B2, B2, C3, G4, G5, G7)$ is $(A2, B1, C1, G3)$.

The proposed hardware accelerator, illustrated in figure 9, adapts some building blocks from [26] to perform this task as a non-blocking high-throughput stream processor. The output of the sorter bypasses the distinct counter, until the last pass takes place, when the sorted output gets its final form to be processed on-the-fly.

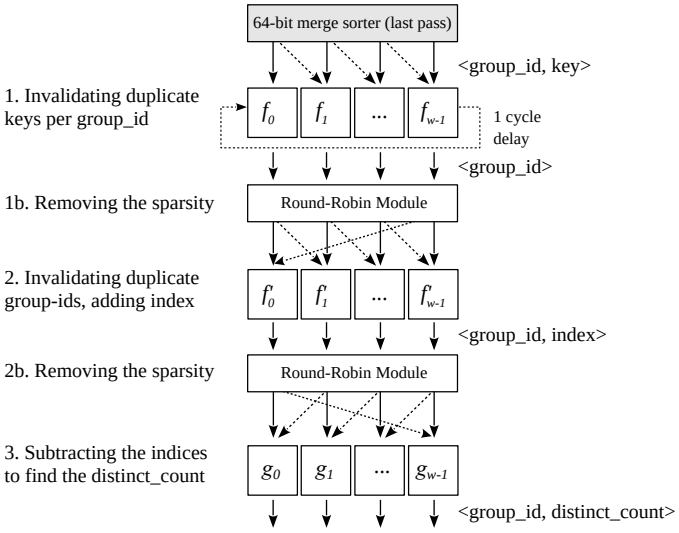

Fig. 9. Distinct counter engine

The functionality of this pipeline goes as follows. First, the f_i modules invalidate duplicate occurrences of the same $< group\_id, key >$ combination. This is achieved by comparing each entry with its immediate predecessor, as the stream is already sorted. Similarly, the f'_i modules only allow to pass unique $group_id$s, but with an index appended, representing their order just before removing duplicate $group_id$s. Finally, the g_i modules subtract the indices of consecutive $< group\_id, index >$ entries to provide the final distinct count per $group\_id$. Any occurring sparsity is removed using round-robin modules. Also known as parallel round-robin arbiters [29] (not to be confused with [30]), they rearrange arriving entries to achieve a round-robin arbitration effect to the output, but with full throughput.

In order to estimate the performance under different inputs, we average multiple runs of two opposite cases. One with 100% unique groups, that consumes the most time, as the number of lines in output is the same as sorting. The second type of input is with near 0% unique groups (all duplicates), which outputs a couple of lines in the last pass, maximising the FPGA workload ratio to data movement. The performance variation between the two types is relatively small, as sorting still takes the majority of time. This is especially true for inputs that require more than one pass, because the data movement savings actually happen only during the last pass, and also the CPU performance improves for many duplicate values as well.

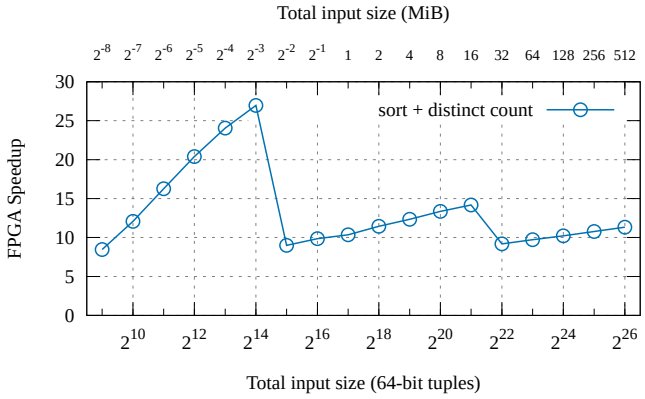

Fig. 10. Distinct count performance

Figure 10 and table II summarise the results of this exploration. The average speedup of the selected configuration over std::sort() plus distinct count on the A53 core reaches up to 27 times, for 16384 tuples, and over 9 times for the other input lengths. Most variation between the input types happens at 16384 (single pass), where the 27 times speedup is actually the average of around 24 and 30 times for the inputs with 100% unique groups and near 0% unique groups respectively.

TABLE II
RESULTS SUMMARY (FOR BOTH USE CASES)

Design configuration[a]				Vivado 2019.2 report				FPGA Speedup over[c]			
							$f_{max'}$[b]	A53		i7	
k	w	d	r	LUT	FF	BRAM	(MHz)	Min	Max	Min	Max
1: sort											
128	4(×2)	32-bit	8×	26K	39K	128	250	15	36	6	17
256	4(×2)	32-bit	1×	44K	61K	64	214	6	49	2	19
2: sort & distinct											
128	2(×2)	64-bit	4×	35K	50K	120	250	9	27	3	12

[a] All designs feature the skewness optimisation and no payload.
[b] Nearest lower frequency to f_{max} evenly dividing 1500MHz.
[c] Max. is at input size $k \times k$ (1 pass). Min. only refers to > 1 passes.

VI. Future work and Conclusions

Future work includes bringing our evaluation closer to production level, starting from a higher-end FPGA to a distributed FPGA system, to overcome the bandwidth limitations [31]. An operation that could be desirable in database applications is stable sorting [6], that is to be able to output records with the same key in the same order as they appear in the input. It sounds possible to modify our solution to support stable sort, although the skewed dataset optimisation at IV-D would not be applicable. Other future work could include comparison with SIMD [27], [32] and GPU-based [33] solutions. Ideally, this design will be integrated in an analytics platform and enable hot-plugging of a series of high-throughput stream processors [34], such as our proposed distinct counter, by exploiting partial reconfiguration in the datacenter [35].

This paper introduces a novel hardware sorter adaptable to different input-sizes, bandwidth and use cases. As a small-scale sorter, it has a competitive combination of logic, storage and time requirements, but also works as a merger of multiple

input lists. The merging capability was embedded to the linear sorter, to enable scalability to arbitrary problem sizes. The approach to use a parallel merge tree to combine single-rate mergers provides the means to saturate the bandwidth while still performing many-leaf merge. Additional enhancements allow efficient sorting of skewed and sorted datasets, as well as design choices for increasing the transfer speeds to saturate the available bandwidth. Our open-source script generates Verilog code for a sorter of a custom degree of parallelism, merge capacity, data and payload width. Finally, we provide example use cases implemented indicatively on a resource-constrained platform, including our in-house analytics application.

ACKNOWLEDGEMENT

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

REFERENCES

[1] P. Papaphilippou and W. Luk, "Accelerating database systems using FPGAs: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 125–130.

[2] J. Fang, Y. T. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory database acceleration on FPGAs: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 33–59, 2020.

[3] 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.

[4] S.-W. Jun, S. Xu *et al.*, "Terabyte sort on FPGA-accelerated flash storage," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 17–24.

[5] M. Roozmeh and L. Lavagno, "Implementation of a performance optimized database join operation on FPGA-GPU platforms using OpenCL," in *2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. IEEE, 2017, pp. 1–6.

[6] K. Manev and D. Koch, "Large Utility Sorting on FPGAs," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018.

[7] E. A. Elsayed and K. Kise, "Towards an efficient hardware architecture for odd-even based merge sorter," in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2019, pp. 249–256.

[8] W. Song, D. Koch, M. Luján, and J. Garside, "Parallel hardware merge sorter," in *24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 95–102.

[9] R. Kobayashi and K. Kise, "Face: Fast and customizable sorting accelerator for heterogeneous many-core systems," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 2015, pp. 49–56.

[10] S. Mashimo, T. Van Chu, and K. Kise, "High-performance hardware merge sorter," in *25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 1–8.

[11] 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.

[12] M. Saitoh and K. Kise, "Very massive hardware merge sorter," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 86–93.

[13] P. Papaphilippou, C. Brooks, and W. Luk, "FLiMS: Fast Lightweight Merge Sorter," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 78–85.

[14] 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.

[15] T. Usui, T. Van Chu, and K. Kise, "A cost-effective and scalable merge sorter tree on FPGAs," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2016, pp. 47–56.

[16] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, 2012.

[17] H. Chen, S. Madaminov, M. Ferdman, and P. Milder, "FPGA-Accelerated Samplesort for Large Data Sets," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 222–232.

[18] C.-Y. Lee and J.-M. Tsai, "A shift register architecture for high-speed data sorting," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 11, no. 3, pp. 273–280, 1995.

[19] R. Perez-Andrade, R. Cumplido, C. Feregrino-Uribe, and F. M. Del Campo, "A versatile linear insertion sorter based on an FIFO scheme," *Microelectronics Journal*, vol. 40, no. 12, pp. 1705–1713, 2009.

[20] D. Koch and J. Torresen, "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*, 2011, pp. 45–54.

[21] W. Luk, V. Lok, and I. Page, "Hardware acceleration of divide-and-conquer paradigms: a case study," in *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp. 192–201.

[22] M. Zuluaga, P. Milder, and M. Püschel, "Streaming sorting networks," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 4, pp. 1–30, 2016.

[23] D. Li, L. Huang, T. Gao, Y. Feng, A. Tavares, and K. Wang, "An Extended Non-Strict Partially Ordered Set Based Configurable Linear Sorter on FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[24] J. Ortiz and D. Andrews, "A streaming high-throughput linear sorter system with contention buffering," *International Journal of Reconfigurable Computing*, 2011.

[25] M. Zuluaga, P. Milder, and M. Püschel, "Computer generation of streaming sorting networks," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1241–1249.

[26] P. Papaphilippou, H. Pirk, and W. Luk, "Accelerating the merge phase of sort-merge join," in *29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 100–105.

[27] B. Bramas, "A novel hybrid quicksort algorithm vectorized using AVX-512 on Intel Skylake," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 8, no. 10, pp. 337–344, 2017.

[28] K. Manev, A. Vaishnav, and D. Koch, "Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems," in *International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 179–187.

[29] P. Papaphilippou, J. Meng, and W. Luk, "High-Performance FPGA Network Switch Architecture," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Association for Computing Machinery, 2020, pp. 76–85.

[30] S. Q. Zheng, M. Yang, J. Blanton, P. Golla, and D. Verchere, "A simple and fast parallel round-robin arbiter for high-speed switch control and scheduling," in *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, vol. 2. IEEE, 2002, pp. II–II.

[31] A. Dollas, "Big data processing with FPGA supercomputers: Opportunities and challenges," in *2014 IEEE computer society annual symposium on VLSI*. IEEE, 2014, pp. 474–479.

[32] J. A. Watkins, "A Fast and Simple Approach to Merge Sorting using AVX-512," 2019.

[33] D. P. Singh, I. Joshi, and J. Choudhary, "Survey of GPU based sorting algorithms," *International Journal of Parallel Programming*, vol. 46, no. 6, pp. 1017–1034, 2018.

[34] G. Alonso, Z. Istvan, K. Kara, M. Owaida, and D. Sidler, "doppioDB 1.0: Machine learning inside a relational engine," *IEEE DE Bull*, vol. 42, no. 2, 2019.

[35] C. Dennl, D. Ziener, and J. Teich, "Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2013, pp. 25–28.