Efficiently Removing Sparsity for High-Throughput Stream Processing

Philippos Papaphilippou[§*], Zhiqiang Que[*], and Wayne Luk[*]

[*]Dept. of Computing, Imperial College London, UK {pp616, z.que, w.luk}@imperial.ac.uk

[§]School of Computer Science and Statistics, Trinity College Dublin, Ireland papaphip@tcd.ie

***Abstract*—Big data analytics and machine learning are increasingly targeted by FPGAs due to their significant amount of computing capabilities and internal parallelism. Different programming models are used to distribute the workload to the internals of the FPGAs at different granularities. While the memory bandwidth has been steadily increasing, there are some challenges in the way system-on-chips use this bandwidth. One way system-on-chip architects exploit the increasing memory bandwidth is by widening the datapath width. This is reflected at various points in the system including the widening of vector instructions. On FPGAs, many analytics accelerators are memory-bound, and would benefit from making the most of the available bandwidth. In this paper we present a scalable and highly-efficient building block for building high-throughput streaming accelerators, which removes sparsity on-the-fly without backpressure.**

***Index Terms*—FPGA, stream compaction, aggregation, high-throughput computation, analytics, interconnects, prefix scan**

I. Introduction

There is steady progress with the improvement of memory speed. This has led to the incorporation of high-bandwidth memory (HBM) in modern FPGAs [1] and in general-purpose processors that achieve a memory throughput of over 1 TB/s [2]. However, as the number of processing elements and their performance increases, it is becoming increasingly challenging to efficiently distribute and utilise this bandwidth [3].

From the programmer's perspective, this challenge is reflected on the parallel programming models that make algorithm design decisions less trivial. To maximise efficiency, parallel algorithms need to match specific platforms. For example, merge trees are still considered in state-of-the-art sorters on FPGAs [4], [5], while on CPUs prominent algorithms include radix sort [6], quicksort [7] and others [8]. Memory use is not the only reason for this discrepancy, but still, the main outcome is that there are different parallel memory access patterns, which can benefit from different memory topologies.

One way system-on-chip architects exploit the increasing memory bandwidth is by widening the datapath width. This is reflected at various points in the system including the widening of vector instructions. For instance, this led to the rise of Intel's AVX-512 single-instruction multiple-data (SIMD) instructions [9]. ARM's scalable vector extension (SVE) is a promising alternative for incorporating micro-architecture-independent vector instructions to general purpose processors. Between a processor's L1 and L2 caches, there are already instances where the communication is 1024-bit wide, such as the ARM-based A64FX supercomputer processor [10].

The widening of the datapath is observed in FPGAs as well. An example is with the move from Xilinx' Zynq UltraScale to UltraScale+, which increased the width of high-performance (HP) AXI ports from 64 to 128 bits [11], or up to 256-bits for special memories in Versal [12]. Given that these are designed to run with high operating frequencies [13], moderately complex FPGA designs can benefit from multiplying their widths. For instance, the 128-bit wide HP AXI ports of Zynq UltraScale+ can act as 256-bit buses at half the frequency [4]. Due to the increased flexibility and the absence of instructions, FPGAs can be the best candidate to benefit from wide datapaths.

This paper presents an efficient FPGA-based building block for removing sparsity on-the-fly from wide data streams. The goal is to be able to exploit the proposed approach for building high-throughput streaming accelerators, and without blocking the data stream while doing so. The proposed design is an asymptotically more efficient algorithm for implementing the parallel round-robin arbiter (PRRA) than related work [14].

Our key contributions are as follows:

- An optimal-complexity switch topology for implementing the parallel round-robin arbiter (PRRA).
- Formal proofs for correctness and optimality.
- An open-source FPGA-based evaluation.

Following is the background (section II) on the functionality. Section III introduces the novel architecture. Section IV elaborates on its correctness and optimality. The evaluation (section V) studies its scalability and behaviour. The paper ends with related work and conclusions (sections VI and VII).

II. Background

A. Parallelism in data analytics

Most data-intensive parallel algorithms can be classified into vertical, horizontal [15] and hybrid. Horizontal parallel computation involves multiple workers/cores operating concurrently after splitting data into large independent chunks, such as the map phase of MapReduce [16]. Vertical computation involves fine-grained parallelisation, such as adopting SIMD on a single stream. Sometimes both could be used to efficiently utilise the available resources, such as when multi-cores run vector instructions, which is what the hybrid class stands for.

On FPGA accelerators, a vertical computation is desirable, as it promotes fewer data movements and more-linear access patterns. This is important since such accelerators are usually distant from main memory, and their memory accesses are optimised for high-bandwidth, including with HBM. The length of the accesses from the programmable logic is shown to have a considerable impact on performance [17]. While making thoughtful algorithm selections can enable a design to use multiple workers and faster random accesses [18], this can come at the cost of software and hardware complexity [14].

Finally, in modern systems with FPGAs, there is already an order of magnitude of difference in the operating frequency of the FPGA and the other components. Moving the FPGA nearer to the processor or the data can magnify this gap, which can be detrimental to the adoption of FPGAs as accelerators in the future. Our proposed building block can alleviate this by simplifying high-throughput computation for certain workloads through a physical or emulated wide datapath.

B. Parallel round-robin arbiter

This work considers the parallel round-robin arbiter (PRRA) [14]. The PRRA is a recursive function that reads batches/chunks of sparse data, and performs a permutation so that the valid or desired entries are produced in round-robin fashion. This output can then be stored in memory banks or read by other stream processors, for example. Figure 1 and algorithm 1 illustrate the high-level functionality of the PRRA, where P is the size of the batches received per cycle.

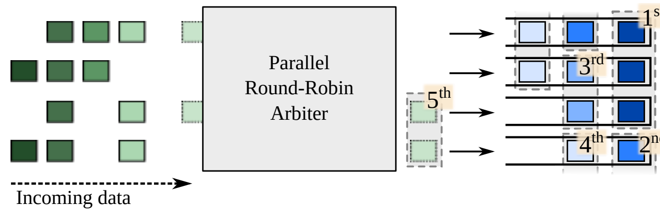

Fig. 1. Example visualisation of the PRRA ($P = 4$)

```
1  int offset;
2  input in[P]; output out[P];
3  while forever do
4      receive (positive clock edge);
5      int j ← 0;
6      for int i ← 0, 1, ..., P-1 do
7          out[(offset + j)%P] ← Null;
8          if in[i].valid then
9              out[(offset + j)%P] ← in[i];
10             j++;
11         end
12     end
13     offset ← (offset+j)%P;
14 end
```

Algorithm 1: PRRA functionality pseudocode

Figure 2 shows the building blocks of the implementation in related work [14]. The idea of this design is that a sorting network [19] sorts its inputs based on their valid bit, whilst an adder tree calculates the popcount of those valid bits. The sorting network is used as a concentrator of the valid entries, while the popcount is the number of valid elements in the batch, which is then used to update the rotation counter. The rotation counter is used by the barrel shifter to make the appropriate rotation to the batch to achieve the PRRA effect.

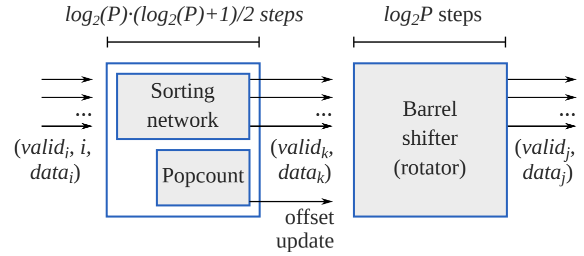

Fig. 2. PRRA implementation using a sorting network

Since most sorting networks are not stable, i.e. they do not necessarily keep the order between equal values, the valid bit is not enough to keep the original order of valid values. Thus, there are two versions of the PRRA. The non-stable version uses the "binary" [20] sorter, while the stable version also uses the port number in the sorter's comparators. This paper only focuses on the stable version as a baseline, since it achieves greater applicability.

III. Proposed Approach

This section provides an alternative implementation for the parallel round-robin arbiter with lower hardware complexity. We notice that the switching logic of parallel round-robin arbiters can be implemented with fewer 2x2 switches by using a reverse butterfly network as a permutation network (III-A). A rolling prefix scan is used to calculate the permutation indices (III-B). The novel architecture is presented in III-C.

A. Reverse butterfly network

The butterfly network, also referred to as a banyan or omega network [21], is a permutation network. Butterfly networks have $log_2(P)$ parallel stages, where P is the number of inputs. There is a control bit for every 2x2 switch, with a high value denoting a swap, and a low value denoting a forwarding.

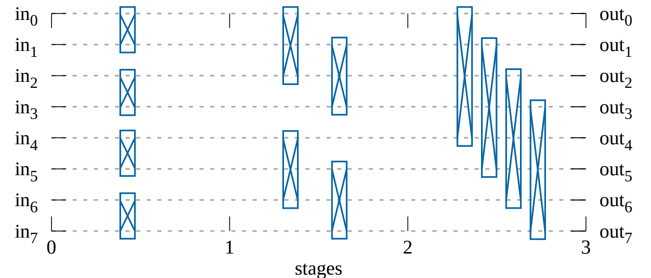

Fig. 3. Reverse butterfly network ($P = 8$)

The reverse butterfly network (also called inverse butterfly) is also a permutation network. It is based on a butterfly network with its inputs and outputs reversed, as presented in figure 3. In the (non-reverse) butterfly network case, a widely used algorithm [22] for the input of the control bits is based on reading the destination index bit-by-bit. More specifically, the most significant (MSB) of the values are used for the switches

of the first stage, MSB-1 for the switches of the second stage, and so on. If the inputs are a permutation of indices (0 to P-1), this becomes a permutation network, but not for all possible permutations. The main difference from the butterfly network is that the control bits are assigned starting from the LSB for the first stage, rather than the MSB.

They can perform an (interesting) strict subset of all possible permutations of their input. Thus, they are known as blocking networks, and this is easy to show. The number of possible switch configurations is smaller than the number of possible permutations (e.g. $2^{3*8/2} = 4096 < 8! = 40320$ for $P = 8$). This subset is key to our proposed approach.

B. Rolling prefix scan

The prefix scan (or prefix sum) is a parallel algorithm that provides the cumulative sum up to each number in a list, and has many applications including in radix sort [23]. The resulting number of additions are more than what would be required by a simple serial implementation with a cumulative sum. However, as a parallel algorithm it is advantageous, as it is able to provide the result in $log_2 N$ number of steps for a list of N numerical elements. SIMD-based implementations have also emerged [15], including on GPUs [24].

The rolling prefix scan is a hardware adoption of prefix sum that allows it to be used for arbitrarily long lists [25]. It is analogous to vertically-parallelised prefix scan for SIMD [15] and similar arrangements. The flexibility for an arbitrary input size makes it desirable for FPGA use. This is because, as a hardware design has size limits, streaming or thrashing applications (not fitting on the FPGA) can benefit from reusing the same logic. A linear access pattern can be achieved, while making full use of the available datapath width.

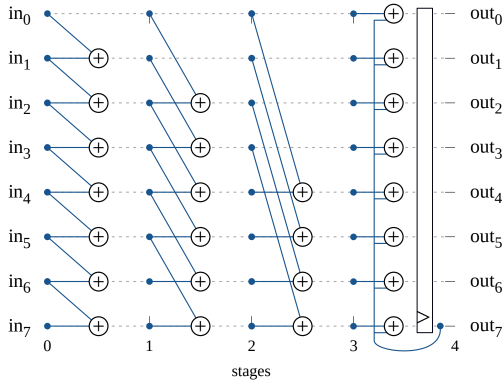

Fig. 4. Rolling prefix scan (P=8)

This modification for hardware implementation is shown in figure 4. The idea is that the first $log_2 P$ steps are an implementation of the original prefix sum in a pipeline, while the last stage keeps the sum of all previous elements. As the last stage is stateful, it is the only stage where it is compulsory to have a register, i.e. in a pipeline. The rolling prefix scan has a depth of $log_2(P) + 1$ stages.

C. Proposed architecture

The proposed architecture consists of two easily pipelinable components connected together, as shown in figure 5.

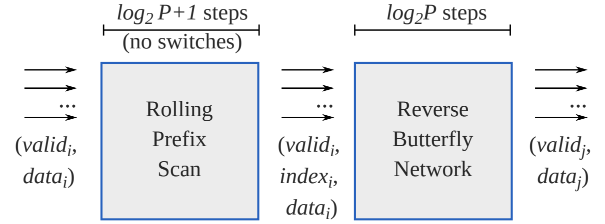

Fig. 5. Novel parallel round-robin arbiter architecture

First, the input goes through a rolling prefix scan (see section III-B) to provide the control bits for the reverse butterfly network. The PRRA's input consists of batches of P elements alongside their valid bits, i.e. $in_i \equiv (valid_i, data_i)$ for $i \in \{0, 1, ..., P\}$. Note that it is not permuted by the rolling prefix scan, hence the reuse of i in the middle of figure 5. The rolling prefix scan only reads the valid bits of the elements in the input, and provides the final positions of each element after its last stage, which holds the last offset. The last offset counter is essentially the last $log_2 P$ bits of the count of all inputs encountered in previous cycles, minus 1. It is initialised with the value $P-1$ on reset, as the positions would start from position 1 instead of 0, since valid bits indicate presence.

Then, the reverse butterfly (see section III-A) acts as a permutation network on the elements of the input batches. Each of the 2x2 switches in the network (as in figure 3) swaps the inputs that are connected to it, when its control bit is high. The swap condition C acts as the control bit and is based on the fields of the carried information alongside the data.

$$C = (in_A.valid \wedge in_A.index[l]) \vee (in_B.valid \wedge \neg in_B.index[l])$$

where in_A and in_B are the first and second inputs to the 2x2 swap switch (sorted spatially), and the $valid$ field is the valid bit. The $index$ field is the desired destination position calculated by the prefix scan. $l \in \{0, 1, ..., log_2(P)\}$ is the stage number, which is here used to perform bit selection on the $index$ field, starting from its LSB position for the first stage of the reverse butterfly network.

IV. Correctness and Optimality

A. Network passability

In order to prove that the reverse butterfly network routes all desired input combinations (passable permutations), it is enough to show that its output is a bitonic sequence with respect to the order of its input (up to one local minimum and up to one local maximum). This is enough because the regular (non-reverse) butterfly is shown to sort all bitonic sequences in the bitonic sorter [19], and any scheduling that would allow routing of bitonic sequences would also enable non-blocking routing for all bitonic sequences for sorted input, using the inverted circuit (with its inputs and outputs swapped) [4].

Figure 6 shows a butterfly network, as found in the bitonic sorter. The shaded box on the bitonic sorter indicates one

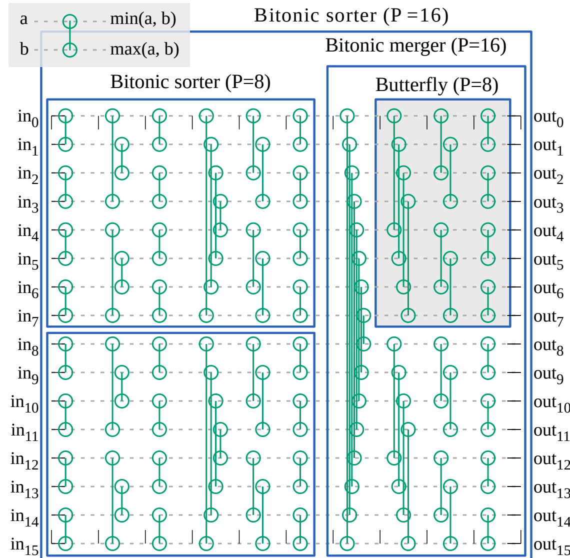

Fig. 6. Building blocks of a bitonic sorter, including a butterfly network

of its two equally-sized butterfly networks near its output. Though, instead of compare-and-swap (CAS) units of the sorting networks that operate on the magnitude of the input, the butterfly as a permutation network uses 2x2 switches.

The order/magnitude of the inputs of the reverse butterfly as a PRRA can be considered to be the port number. The possible order of the outputs can be seen as all rotations (according to the rotation offset) of the concatenation of the valid bit and the local order (there are two different orders: among the subsets for valid and invalid elements). For the stable version of the parallel round-robin arbiter, the local order of the valid subset must be the same as in the input (following port numbers). With respect to the local order of the non-valid set, the order does not matter for the requirements of the parallel round-robin arbiter. Therefore, in order for this to be a bitonic sequence, the local order of elements in the non-valid set is considered to be the same (constant function).

For no rotation, the expected output is a bitonic sequence. This is because its form is $e_0, ..., e_{n-1}, e_n, ..., e_{P-1}$, where e_i is the local order of the ith element, and n is the number of valid inputs. The sublist $e_0, ..., e_{n-1}$ is already sorted according to the port number, and the sublist $e_n, ..., e_{P-1}$ consists of equal values and is below all elements in the first sublist. This only yields up to one local maximum e_{n-1} (or e_0 when all inputs are invalid), and up to one local minimum e_{P-1} (or e_0 when all inputs are valid), and thus is a bitonic sequence. For all other rotations, the output will still be a bitonic sequence, as a rotated bitonic sequence is also bitonic [26]. This finalises the proof that the required permutations to implement PRRA are passable inside the reverse butterfly.

B. Network configuration

The correctness of the swap condition $C = (in_A.valid \wedge in_A.index[l]) \vee (in_B.valid \wedge \neg in_B.index[l])$ can be shown by breaking down the possible cases. When both $in_A.valid$ and $in_B.valid$ are false, then their order does not invalidate the result, as we do not care about the invalid entries in the output. When one of them is valid, then the LSB algorithm (see section III-A) is followed for that stage, which overrides the decisions of the invalid counterpart, which is invalidated. Since the LSB algorithm is followed, the valid entry will follow the correct path to reach its destination as shown by the prefix sum, at least for the studied stage l (a longer format of this proof could include an inductive step).

The remaining case is when both $in_A.valid$ and $in_B.valid$ are true. Assuming that there is a configuration where there is a conflict, both inputs want to follow the same output port of a 2x2 switch (i.e. $in_A.index[l] \neq \neg in_B.index[l]$). Since there is a unique control bit configuration for single entries to be permuted to a designed output index, as given by LSB, the only way for this to happen is with a non-passable permutation. Given the proof in section IV-A, all index sequences given by the rolling prefix sum only produce passable permutations on the reverse butterfly network. This leads to a contradiction and completes the proof for the correctness of C.

C. Optimality

The reverse butterfly network is optimal with respect to the number of stages and 2x2 swap switches. This can be derived as follows. Every input port needs to be able to provide paths to all positions, as it can be rotated by an arbitrary amount according to the output of the previous rotation. Thus, there needs to be at least $log_2(P)$ 2x2 swap switches for each path for this to happen, one for each bit of the binary representation of the destination port. Every switch can have at most 2 inputs, therefore it can be shared to achieve switch reuse. The reverse butterfly network is proven to pass all required permutations of the inputs while always using the maximum number of inputs per switch (two), and thus being optimal in this aspect.

Note that this does not prove optimality with respect to implementation efficiency, as place-and-route relates to wire length, target FPGA architecture etc. It also does not cover the control bit calculation (prefix sum). Its few-bit widths make it of secondary importance for optimisation, but future work includes the exploration of circuits with fewer assumptions.

V. Evaluation

In this section we study the behaviour of the proposed PRRA architecture when implemented on an FPGA[1]. First, the implementation characteristics are observed for an increasing amount of throughput (section V-A). Second, a less technology-dependent comparison is provided (section V-B).

A. Scalability

This part of the evaluation studies the FPGA implementation efficiency of the proposed approach, and compares it to previous work [14]. The studied designs are out-of-context with respect to the use cases, but are tested on a real FPGA as AXI peripherals. Their functionality is for debugging, but it is enough to study the logic's behaviour. A memory-mapped input vector is released into the PRRA upon request, and the result is also stored in memory-mapped registers, which are read with AXI. For this part, the data width

[1] Source available: https://philippos.info/prra

of each element is indicatively set to 64-bits. The resulting implementation dataset is produced with Vivado 2020.1, and the target platform is Xilinx/AMD Alveo U280.

There are two main designs in our design space. The baseline is the sorter-based approach [14], which uses "odd-even sorting" [19]. The proposed approach is referenced as "reverse-butterfly based". The design space for each of the two focuses on its scalability with the number of ports P. Another variable in the design space is the placement of registers in the design's stages, as they are fully pipelineable. This is a manual equivalent to register retiming, and is done to study the behaviour of the pipelines. A Python script generates Verilog HDL code for each of the $(type, P, S)$ combinations, where $type$ is a boolean that selects between the two approaches, $P \in \{2, 4, 8, ..., 256\}$ is the number of ports, and $S \in \{1, 2, 4, 8, 16\}$ denotes the register placement (e.g. $S = 2$ skips every other pipeline register stage).

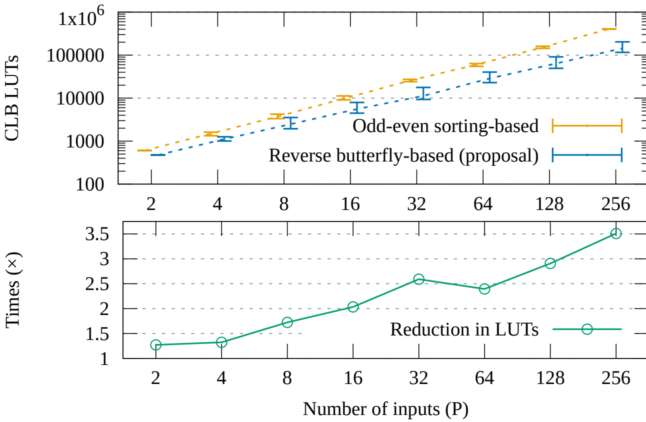

Fig. 7. Look-up table (LUT) utilisation

The first set of results are shown in figure 7 that compares the Look-up table (LUT) utilisation of the designs for different numbers of inputs (P). On the top plot, there are 3 series: the min., the average and the max. LUTs. The reason for multiple values is the existence of the S parameter that studies the register placement, which is summarised here. An observation is that there is little variation to the main logic according to S, as S only impacts register placement. The bottom plot shows the LUT reduction of the best variations per PRRA design (the S values with the fewest LUTs), which demonstrates that the proposal is more efficient in terms of pure logic.

In our design space, the proposed PRRA was always efficient enough to be able to be placed-and-routed successfully. However, the odd-even-based PRRA was not always able to be generated for $P = 256$, as only the $S = 1$ variation made it past place-and-route, hence the absence of any variation for the corresponding data points.

Figure 8 shows how the register utilisation varies according to the obtained maximal operating frequency (f_{max}). The way to read this plot is as follows. Same colour indicates the same PRRA architecture, same point type indicates same P-value, and the points connected by the same line are small variations

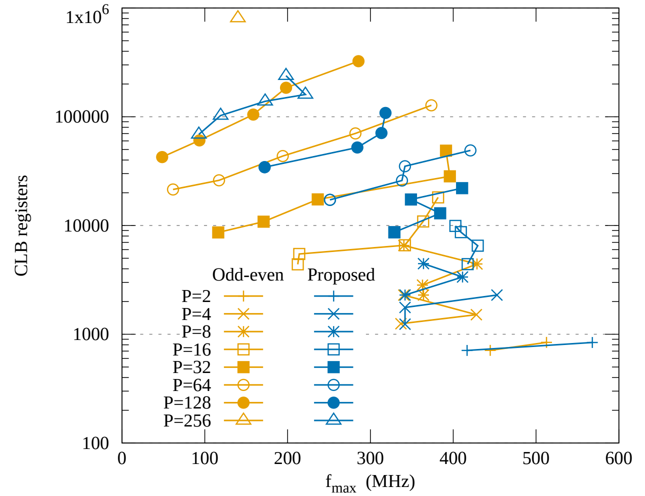

Fig. 8. PRRA register utilisation and maximal operating frequency

occurring from the exploration of the register placement (S, not explicit in the figure) for the same PRRA architecture.

We notice that for the same P value, for $P \geq 16$, the new architecture is always superior to the corresponding line for the prior work "odd-even"-based. That is, every point of the blue lines are in the Pareto front (lower right is better) when combined with the corresponding yellow line. Additionally, as the y-axis is logarithmic, it is easy to see that points from the higher values of P can differ an order of magnitude in their register utilisation between the different architectures.

With respect to f_{max}, for $P \leq 8$, a clear win is not apparent due to the stages being similar (e.g. 5 vs 5 for $P = 4$, but also note the different functions of these). Still, the obtained operating frequencies are already rather high due to the design of both candidates focusing on scalability.

B. Gate count

It is also important to see how the proposed design scales independently of the FPGA architecture technology. The last part of the evaluation provided empirical results with Vivado place-and-route for a specific FPGA. In this part, we isolate the Verilog designs from the rest of the system, such as the AXI interconnect-related logic, and measure the gate count. The comparison is done with Yosys 0.25, and the only allowed gates in our library model are NAND and NOT gates.

In order to widen the applicability of this evaluation, this technology-independent comparison also includes a dataset from Vivado HLS. The way this dataset is obtained is by performing C-synthesis using a behaviourally-described PRRA in C code. The source code for these designs are simple and they follow the pseudocode in algorithm 1. After Vivado HLS produces the Verilog files, the PRRA module is then read by Yosys, also ignoring any additional interfacing beyond the main logic. The idea is to show that this functionality and scalability is not a given with a modern HLS tool.

Figure 9 introduces this comparison, where the proposed architecture is asymptotically more efficient in terms of logic complexity. The difference is magnified for large P values,

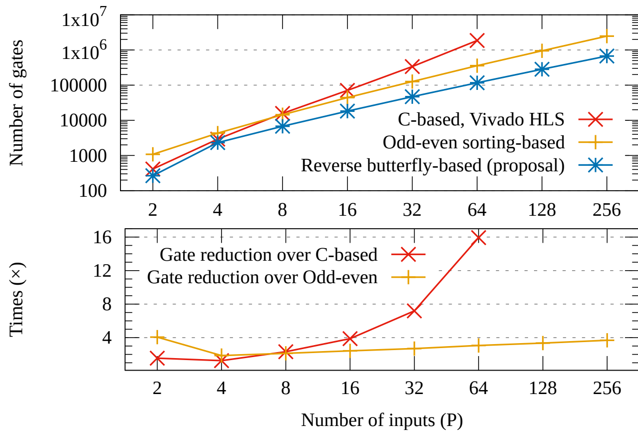

Fig. 9. Gate utilisation comparison

with up to around 4 and 16 times over the odd-even sorter-based and the HLS version. The dataset of Vivado HLS stops at $P = 64$, as C-synthesis starts to take impractically long times surpassing a day. In contrast to the other designs, Vivado HLS is not able to produce a fully-pipelined code. Thus, the latency and initiation interval are manually restricted to 1. For a fairer comparison, the other designs here use a high S-value to only leave one register level and match this latency. Still, irrespective of the register placement and the HLS latency parameter, the logic utilisation (analogous to gate count) varies little between variations of the same design (see figure 7).

VI. Related work

Stream compaction is a common operation having applications in sparse matrix compression, encoding, collision detection and more [24]. On FPGAs the directly competing approach [14] is used as a baseline. There are specialised designs that do not provide this high-throughput streaming functionality. One example [27], while operating on chunks of data, it provides significant speedup on LSM-trees. The PRRA can be considered a vertical parallelisation [15] of stream compaction. The proposed design is reminiscent of modern software equivalents due to the inclusion of the prefix scan [24]. The combination of prefix sum with a permutation network existed before [20] for different applications; they have high complexity due to using sorters for permutation.

VII. Conclusions

High-throughput computation is becoming increasingly important on FPGAs, due to the increased memory bandwidth and lower FPGA operating frequencies when compared to the rest of the system. In this paper, an optimal switch structure is introduced for the implementation of the parallel round-robin functionality. The parallel round-robin arbiter performs stream compaction with high-throughput, which has a variety of applications including aggregation analytics for relational databases and sparse matrix encoding. The proposed architecture has low overhead and a high scalability potential for wide datapaths, which would benefit high-bandwidth systems, near data processing, and future system architectures.

Acknowledgement

The support of EPSRC (EP/L016796/1, EP/P010040/1, EP/V028251/1 and EP/S030069/1), United Kingdom, and the SCSS and the HCI at Trinity College Dublin, Ireland is gratefully acknowledged.

References

[1] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High bandwidth memory on FPGAs: A data analytics perspective," in *30th International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2020, pp. 1–8.

[2] J. Dongarra, "Report on the Fujitsu Fugaku system," *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06*, 2020.

[3] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim, "An analysis of system balance and architectural trends based on top500 supercomputers," in *The International Conference on High Performance Computing in Asia-Pacific Region*, 2021, pp. 11–22.

[4] P. Papaphilippou, "Reconfigurable acceleration of big data analytics," Ph.D. dissertation, Imperial College London, 2021.

[5] P. Papaphilippou, W. Luk, and C. Brooks, "FLiMS: A Fast Lightweight 2-Way Merger for Sorting," *IEEE Transactions on Computers*, vol. 71, no. 12, pp. 3215–3226, 2022.

[6] Intel (R), *Integrated performance primitives: Developer reference, vol. 1, signal processing. [accessed july-2023].* [Online]. Available: https://intel.com/content/www/us/en/docs/ipp/developer-reference/2021-7/.

[7] 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.

[8] *Boost.sort 3.- parallel algorithms*, [accessed July-2023]. [Online]. Available: https://www.boost.org/doc/libs/develop/libs/sort/doc/html/sort/parallel.html.

[9] Intel (R), *Intel intrinsics guide*, [Online; accessed July-2023]. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[10] J. Domke *et al.*, "At the locus of performance: A case study in enhancing cpus with copious 3d-stacked cache," *arXiv preprint arXiv:2204.02235*, 2022.

[11] Xilinx Inc., *Zynq UltraScale+ FPGA Product Tables and Product Selection Guide*, 2016-2021.

[12] Xilinx Inc., "Versal architecture and product data sheet: Overview," *DS950 (v1.15) Advance Product Specification,*, 2022.

[13] X. inc., "LogiCORE IP Product Guide v2.1, AXI Interconnect," *PG059, Xilinx, December*, vol. 20, 2017.

[14] 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.

[15] W. Zhang, Y. Wang, and K. A. Ross, "Parallel Prefix Sum with SIMD," *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB*, vol. 5, p. 31, 2020.

[16] K. Tangwongsan, M. Hirzel, and S. Schneider, "Sliding-window aggregation algorithms," in *Encyclopedia of Big Data Technologies*. Springer International Publishing, 2018, pp. 1–6.

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

[18] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "FPGA-based Multithreading for In-Memory Hash Joins," in *CIDR*, 2015.

[19] K. E. Batcher, "Sorting networks and their applications," in *Proc. of the April 30–May 2, spring joint computer conference*, 1968, pp. 307–314.

[20] T. Jain, "Nonblocking on-chip interconnection networks," Ph.D. dissertation, Technische Universität Kaiserslautern, 2020.

[21] M. J. Narasimha, "The batcher-banyan self-routing network: Universality and simplification," *IEEE Transactions on Communications*, vol. 36, no. 10, pp. 1175–1178, 1988.

[22] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. on Computers*, vol. 100, no. 12, pp. 1145–1155, 1975.

[23] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[24] M. Safari and M. Huisman, "Formal verification of parallel prefix sum and stream compaction algorithms in CUDA," *Theoretical Computer Science*, vol. 912, pp. 81–98, 2022, ISSN: 0304-3975.

[25] P. Papaphilippou, P. H. J. Kelly, and W. Luk, "Simodense: a RISC-V softcore optimised for exploring custom SIMD instructions," in *31st Int. Conf. on Field Programmable Logic and Appl. (FPL)*, IEEE, 2021.

[26] G. Zachmann, "Adaptive bitonic sorting," *Encyclopedia of Parallel Computing, David Padua, Ed*, pp. 146–157, 2013.

[27] X. Sun, J. Yu, Z. Zhou, and C. J. Xue, "FPGA-based compaction engine for accelerating LSM-tree key-value stores," in *36th Int. Conf. on Data Engineering (ICDE)*, IEEE, 2020, pp. 1261–1272.