2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-6460-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPSW63119.2024.00038

A Fast Scalable Hardware Priority Queue and Optimizations for Multi-Pushes

Samuel Collinson, Allan Bai, Oliver Sinnen
Parallel and Reconfigurable Computing Lab, Department of Electrical, Computer, and Software Engineering
University of Auckland, New Zealand

***Abstract*—Priority queues (PQ) are an essential data structure for many important algorithms. Hardware implementations of priority queues can accelerate such algorithms significantly. Usually a choice has to be made between very fast (fixed-size) hardware PQs or scalable PQs. Fast hardware queues can provide single cycle push and pop operations, but are limited in size. Scalable hardware queues use embedded memory to achieve scalability, but thereby compromise on speed. This paper proposes a fast and scalable hardware priority queue that can be used for general purpose. It is based on a modular and flexible hybrid design, using a shift register queue combined with a heap-based queue. In the best case, i.e. when only the first queue is needed, push and pop operations only take a single cycle. In an experimental evaluation on a Xilinx FPGA we demonstrate that performance of the hybrid queue maintains a good balance between performance and scalability even for applications where the size of the working set of data can have large variance. We further propose an optimisation of the heap-based queue to support workloads that repeat several consecutive push operations followed by a single pop operation, which is the workload, for instance, in state space search or ray-tracing.**

I. Introduction and Related Work

Priority queues (PQs) are an abstract data structure similar to a stack or queue, but with an associated priority value for each data element. This priority value is used to determine the order in which elements are extracted from the queue, where higher priority elements are extracted first and elements with the same priority are extracted based on their order in the queue. A PQ typically implements two primary instructions, insertion of an element into the queue with a set priority, called `push` (also called enqueue or insert), and extraction of the highest priority element, called `pop` (also called max-delete or dequeue), both of which are typically followed by a sorting algorithm to maintain the order of the queue.

Whilst PQs are traditionally implemented in software, it can be more efficient to use hardware to perform direct comparisons with multiple data items at once, which suits the parallel nature of FPGAs. Such hardware implementations of PQs can be very attractive in many application areas.A major challenge in the design of hardware PQs is the trade-off between performance and scalability. Small PQs can be extremely fast in hardware, with single cycle operations, see Section II, but are usually of (small) fixed-size. However, for many applications PQs need to be scalable to large sizes, which might not be precisely known a-priori.

PQs are commonly found within the context of communication and computer networking hardware appliances. The length issue of the queue has often been solved by simply dropping data elements when the queue is full [1].The area of prioritised IP packet queues for network routing has seen research into scalable hardware PQs [2], [3]. Graph search algorithms, such as those that solve the shortest path problem, often require PQs.Due to the difficulty of implementing fast and scalable PQs in hardware, some work on hardware implementations of the Dijkstra shortest path algorithm have avoided them and resorted to basic forms of sorting [4], [5]. Achieving speedups in hardware implementations of graph algorithms proves to be non-trivial [6]. Having fast and scalable PQs might help to implement graph algorithms in FPGAs more efficiently and easily. In data base systems, PQs are also employed for different aspects, in particular as part of sorting procedures [7], [8]. PQs are also heavily used in ray tracing, when traversing the acceleration hierarchy tree [9].

The importance of PQs and the issue of scalability motivated research into different designs of hardware PQs in the past. A very fast design is based on a large shift register and parallel comparators [10] (Section II). The downside is the $O(n)$ comparators needed, which strictly limits the scalability. Other work uses systolic arrays [11] or sorting tree networks [5], which, like the shift register based design, incur high hardware costs limiting their scalability. The recursive design in [12] reduces the hardware costs, trading off performance. More scalable approaches are pipelined heaps, where for many designs $O(1)$ operation complexity can be achieved through the pipelining. Examples are [13], [14], [2]. An approach that uses different memory types (SRAM and DRAM) and an amortized constant operation time was proposed by [15]. As similar approach was taken by [16]. Hybrid approaches with hardware-software co-design were proposed in [10], [17].

A hybrid hardware-based solution was proposed in [8], where a fast, but not scalable register-array solution is combined with a scalable memory based solution. While this solution provides high performance and scalability, it is specialised on replacement operations. Consecutive `pop`'s or `push`'s are not supported, which naturally occur in many application areas, including state space search or ray tracing.

The authors of [18], [19] recently proposed a hybrid priority-queue for networking, which is implemented in C++ and high-level synthesis (HLS) is used to generate a queue/set of queues for a given network application. The focus is on flexibility, and fast, pipelined operation of the queue using HlS for fast design space exploration.

979-8-3503-6460-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPSW63119.2024.00038

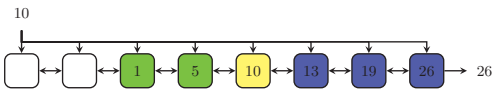


Fig. 1: Shift-register based single-cycle PQ with 8 elements. Item with priority 10 (yellow) pushed onto the queue

This paper proposes a fast and scalable PQ that can be used for general purpose. It is based on a modular and flexible hybrid design, using a shift register queue combined with a heap-based queue. In the best case, i.e. when only the first queue is needed, `push` and `pop` operations only take a single cycle. In an experimental evaluation on a Xilinx FPGA we demonstrate that performance of the hybrid queue maintains a good balance between performance and scalability even for applications where the size of the working set of data can have high variance. The implementation is optimised for dual-port RAM typically found in FPGAs. We further propose an optimisation of the heap-based queue to support workloads that repeat several consecutive `push` operations followed by a single `pop` operation, which is the workload required when ray tracing [20], [21] or for best-first graph search algorithms like A*.

The rest of this paper is organised as follows. Our design for the fast single-cycle queue is detailed in Section II, and the design of the scalable heap queue is presented in Section III. The design of the hybrid queue is proposed in Section IV followed by the experimental evaluation in Section V. In Section VI we propose the design optimisation of the heap queue to efficiently support workloads with bursts of `push` operations. Finally the paper is concluded in Section VII.

II. Single-cycle Priority Queue

Priority queues can be efficiently implemented in hardware using a series of comparators and a shift register, as shown in [10]. Figure 1 illustrates a single-cycle queue in the moment a new element (10) is pushed. Input data is stored in the first register of lower priority and all registers of lower priority are shifted across, effectively sorting the data in a single-cycle. This means that the storage element must decide to retain the same value (if it is of higher priority, blue in figure), accept the input value (if it is the first element of lower priority, yellow) or accept the value of the element above it (if it is of lower priority, greeen). Data can be extracted by outputting the highest priority register and shifting all other registers along, also in a single-cycle, meaning the storage element must also accept the value of the option below it.

A detailed view of the storage element is shown in Figure 2. The comparator in the top left compares the priority of a new data element with the priority of the data element currently stored. This value is passed to the `CMP OUT` port which is connected to the `CMP IN` port of the storage element below.Likewise on the other side. Using these two values, the control logic selects the next value for the storage element.

While highly efficient, the primary drawback of this single-cycle queue is the heavy usage of the available logic elements,

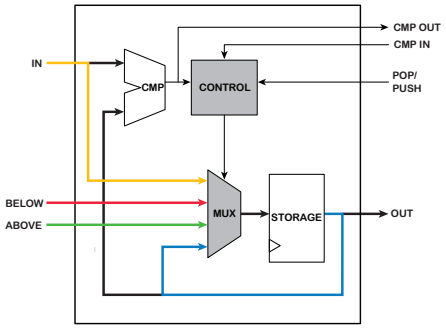


Fig. 2: Storage element used in the construction of the shift-register priority queue.

with each storage element requiring a separate comparator unit and 64 4:1 multiplexers (e.g. for 32 bits of data with a 32 bit priority). If the application deals with a data set of unknown size or a size with high variance, very large priority queues are required to process the data.

Implementation details for the single-cycle priority queue are given in Section IV-A and scalability is evaluated in Section V-5.

III. Heap

A heap is a tree-based data structure that satisfies the *heap property*: Given node A, which is the parent of node B, then the priority of node A is higher than the priority of node B. This relative ordering of parents and children applies across the entire heap and ensures the root node will always have the highest priority [22].

Heaps are typically stored using an array of data with the first index in the array containing the root node. For a heap where the first array index is 1 and given a node at location n, the parent node can be found at $\lfloor n/2 \rfloor$ and the left and right child nodes can be found at $2n$ and $2n+1$ respectively. A visual illustration of a heap as a tree is shown in Figure 4a.

Algorithm 1: `HeapifyUp` – restoring heap property, when given index and parent violate property

```
Data: heap: array of size heap.length.
Input: i: index to start heapify operation from
Function HeapifyUp(i):
    parent ← i/2;
    if i > 1 and heap[parent] < heap[i] then
        swap heap[i] with heap[parent];
        HeapifyUp(parent);
```

`push` is implemented in a heap by storing the new element to be pushed at the bottom of the heap. Then the `HeapifyUp` function, defined in pseudo-code in Algorithm 1 for an array-backed heap with an index starting at 1, is called recursively to swap the current target value with its parent when the heap property is not fulfilled.

For a `pop` operation, the highest value at the head is removed and replaced with the lowest value on the heap and

Algorithm 2: `HeapifyDown` – restoring heap property, when given index and 2 children violate property

Data: heap: array of size heap.length.
Input: i: index to start heapify operation from

Function `HeapifyDown` (i) **:**
  left $\leftarrow 2 \times$ i;
  right $\leftarrow 2 \times$ i $+ 1$;
  largest $\leftarrow$ i;
  if left $\leq$ heap.length **and** heap[left] $>$ heap[largest] **then**
    largest $\leftarrow$ left;
  if right $\leq$ heap.length **and** heap[right] $>$ heap[largest] **then**
    largest $\leftarrow$ right;
  if largest $\neq$ i **then**
    swap heap[i] **with** heap[largest];
    `HeapifyDown` (largest) ;

(a) Example `push` operation, where 13 is to be pushed.

(b) Bottom value 7 of single-cycle PQ spills onto heap PQ.

Fig. 3: `push` operation for a small hybrid PQ (single-cycle PQ in orange, heap PQ in purple).

this is the initial target location. The heap property is then ensured by the `HeapifyDown` function, defined in pseudocode in Algorithm 2, again recursively. In general, `pop` and `push` operations take $O(\log N)$ cycles for heaps of size N.

Implementation of the heap, including optimisations for FPGA dual-port RAM, is detailed in Section IV-A and scalability is evaluated in Section V-5. Optimisation of this heap for multi-push workloads is presented in Section VI.

IV. Hybrid Queue

This section presents the novel hybrid queue, a combination of the single-cycle queue from Section II and the heap from Section III. It combines the performance advantage of a single-cycle queue, and the resource and scalability advantage of a heap queue.

Provided the size of the working data set is distributed such that it remains at a certain mean for the majority of execution time, the hybrid queue has been designed in such a way that most of the operations are performed with the speed of the single-cycle queue. When the data size moves to larger extremes and the single-cycle queue becomes full, extra values are accommodated by "spilling" from the bottom of the single-cycle queue into the heap queue. For these larger extremes, performance degrades to that of the heap queue, but allows the hybrid queue to be both fast during normal operation as well as being able to cope with larger data sets in extremes. Figure 3 shows an example of a `push` operations for a small hybrid queue.

The combination of a single-cycle queue and a heap allows us to maintain a good balance between performance and scalability for applications where the size of the working set of data can have large variance. Implementation of the hybrid queue is described in Section IV-A below and performance is evaluated in Section V.

A. Implementation

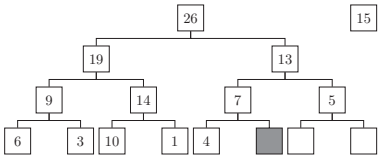
1) Single-Cycle Queue: The single-cycle queue is implemented in VHDL using the same method as in [10]. It is constructed from a number of modular storage elements, implemented in VHDL from the logic shown previously in Figure 2.Each block is linked with the master `push`/`pop` control signals and the new element input. Each block also communicates the result of comparison with a new element to the block below it, and accepts the comparison result of the block above it. Using the above signals, the decisions for `push` and `pop` are constructed in a `case` statement which is synthesized into the multiplexer and control logic that will select the data to be stored in the storage register on the next clock edge. The architecture utilises `generic` parameters and `generate` statements at synthesis time to facilitate trivial scalability of data width, priority width, and queue size. The critical path of the implemented single-cycle queue is the comparator in each storage unit, which has a maximum operating frequency of 450MHz on the used Xilinx FPGA [23]. However, the fanout of the input data can become the critical path when the queue size is greater than approximately 100 elements or when FPGA utilization approaches capacity.

2) Heap: The arithmetic required to calculate parent and child locations, described previously in Section III, is particularly convenient for implementation in hardware as it can be done with logical left and right shifts.

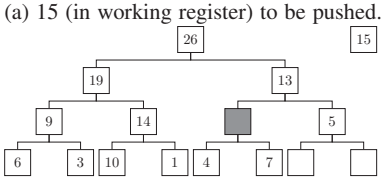
The heap `push` operation was implemented in VHDL by adapting the `HeapifyUp` function in Algorithm 1 to work efficiently with FPGA dual-port memory. This has two ports which memory can be read from independently, but only one fixed port which is able to write to memory. As can be seen in Algorithm 1, swapping of the value in $heap[index]$ with the value in $heap[parent]$ is needed, which requires two write operations and therefore two cycles on the dual-port memory, as only one value can be written at a time.

However, when analysing the memory access patterns of Algorithm 1, we see that the original `push_data` value is moved up through the tree and written at multiple locations. So instead of writing this value multiple times only to have to swap, this value is instead stored in a working register (`working_reg`) and only written to memory just before the `push` operation terminates. Eliminating continuously writing this value means each value swap on FPGA dual-port memory can be completed in a single cycle, plus one cycle at the end to write the working register value. Figure 4 illustrates the principle of the optimized implementation.

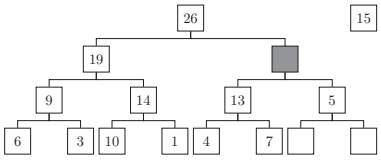
This technique saves a cycle for every swap. Additionally, because the parent value only needs to be read from memory for each level of the heap, the read of the next parent can be



(a) 15 (in working register) to be pushed.



(b) Working register (15) > target location.



(c) Final slot for 15 (< parent value 26).

Fig. 4: Optimiszed heap `push` using working register to reduce memory writes. Current target location in grey.

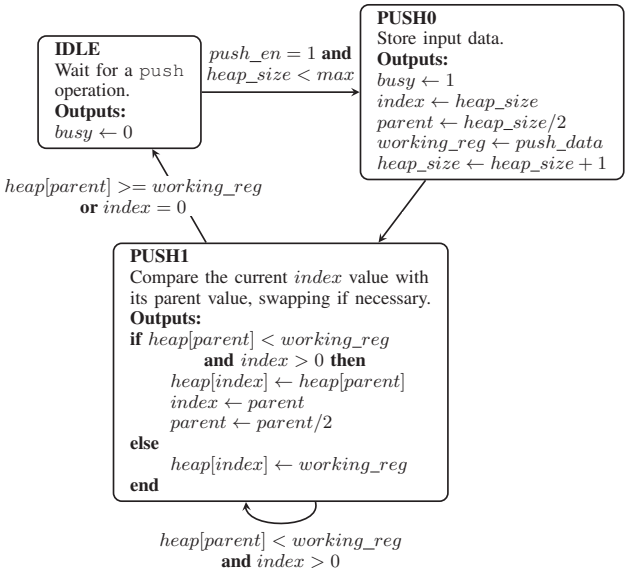


Fig. 5: FSM for implementation of `push` operation.

completed in parallel with the write of the swap value to allow this functionality to be implemented in a single state. Figure 5 shows the resulting FSM.

The heap `pop` operation was implemented in a similar manner to the `push` operation, by adapting the `HeapifyDown` function in Algorithm 2. We use the same working register method as for `push` to save unnecessary writes.

The comparison of the left and right children in Algorithm 2 is done in parallel, which requires additionally comparing the left and right child to each other to ensure the highest priority

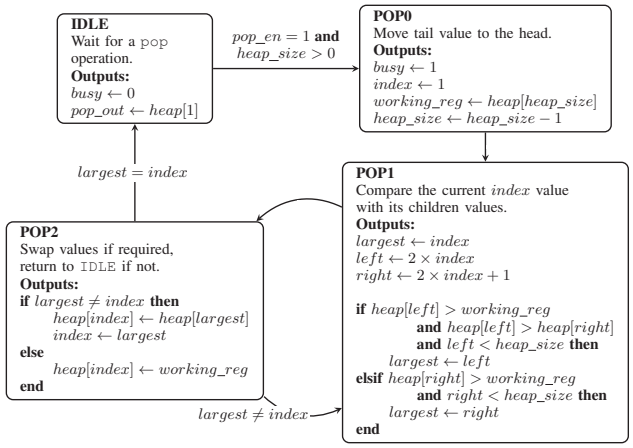


Fig. 6: FSM for implementation of `pop` operation.

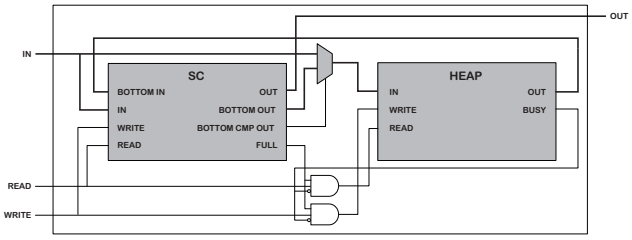


Fig. 7: HybridQ wrapper around the single-cycle queue (labelled SC) and the heap.

of the two is selected. As these steps require two reads for the left and right children and a write for the swap, which cannot be done in a single cycle using dual-port memory, the function must be spread over two states. The resulting FSM to implement the complete operation is shown in Figure 6.

To complete the control logic implementation the FSMs in Figure 5 and Figure 6 are combined into a single FSM by merging their `IDLE` states together and requiring the same conditions to transition to the next state for each operation. From both of these FSMs it can be seen that there are three comparators required. Two are used to compare the output from each read port to the working register, and one to compare the two outputs to each other. Note that the array in Algorithm 1 and Algorithm 2 is indexed starting at 1, not 0 as is done with FPGA memory. To account for this, dedicated logic is added in front of each port input address to decrement the address when it is presented. This allows us to keep using the simple and FPGA friendly arithmetic for finding parent and child nodes in the control logic. The combination of control logic, comparators and dual-port memory make up the complete heap implementation. The critical path of the implemented heap are the three comparators.

3) Hybrid Queue: The hybrid queue implementation, shown in Figure 7, is created in VHDL as a wrapper around the existing single-cycle queue and the heap modules, described previously. The single-cycle queue acts as the primary queue

and is directly connected to the external control signals and data input. Once this queue is filled, overflowing data is directed to the heap. Signals are emitted by the single-cycle queue to control the operation of the heap when the single-cycle queue is considered to be full. The heap emits a busy signal while maintaining the heap, which blocks all input activity until complete. The busy signal is not emitted when the single-cycle queue is not full, allowing the hybrid queue to process data at single-cycle speeds, only degrading in performance when spilling into or out of the heap.

V. Evaluation

In this section the performance and scalability of the hybrid queue is evaluated and compared against the fast single-cycle queue and scalable heap implementations. The performance is evaluated by comparing the execution cycles and time of multiple usage profiles – combinations of `push` and `pop` operations designed to simulate different possible application workloads. The mean and standard deviation of the population size of the data in the queue and the order the data is processed is varied to see the affect different usage profiles have on performance.

1) Method: Testing is performed by creating VHDL test benches that `push` and `pop` data onto the given queue to fit a selected usage profile, executing the test bench in Modelsim, and timing the execution for each type of queue. We use Xilinx ISE 14.4 and target an older Virtex-5 LX330T-2 FPGA to consider an FPGA where resources are somewhat limited. The priority queues evaluated are: a single-cycle queue, a heap, and hybrid queues with single-cycle queue sizes of 100, 300, 500 and 780. The heap is implemented with the Xilinx BlockRAM primitive which has a size of 1,024 elements when using the minimum configuration. We use a single-cycle queue size of 1,024 to match this. When the length of the hybrid queue is mentioned in the following sections, the length of the single-cycle queue within it is what is being discussed. The same heap of size of 1,024 is used within all the hybrid queues, due to the minimum size of the BlockRAM primitive, *i.e.* the hybrid queue of length 780 can potentially store 1,804 elements. We artificially limit this queue size during comparison by ensuring we never exceed 1,024 elements on the queue at any one time.

To create usage profiles, data sets are first generated in ordered, reverse ordered and random ordered sequence. The length of these sequences is referred to as the number of data points in the figures in the following sections. A set of normally distributed integers with a given standard deviation is then generated representing the desired queue population size. `push` or `pop` operations are recorded accordingly until the queue population is equal to the generated integer, whereupon the next generated integer is used. Once all of the elements in the data set have been pushed onto the queue, the queue will `pop` until empty before finishing. For example, for a generated desired queue population of [3, 2, 4], the following sequence of `push` and `pop` operations is generated: [`push`, `push`, `push`, `pop`, `push`, `push`, `pop`, `pop`, `pop`, `pop`].

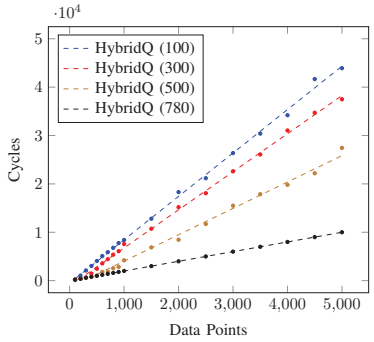


Fig. 8: Execution cycles of hybrid queues for 500 mean queue population, 25% SD (125), random ordered data.

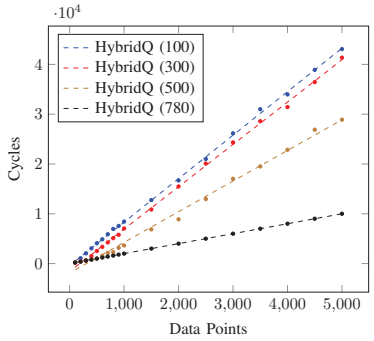


Fig. 9: Execution cycles of hybrid queues for 500 mean queue population, 10% SD (50), random ordered data.

2) Effect of Mean Queue Population: Let us first look at the interplay between the mean queue population sizes in relation to hybrid queue sizes. Exemplary, Figure 8 shows the execution cycles for a usage profile with a mean queue population of 500, standard deviation of 25% (125) and random ordered data.

Two interesting things can be observed. First, as expected, the larger the single-cycle queue inside the hybrid queue, the better the performance. When that size is larger than the mean population (i.e. HybridQ(780)) the performance benefit is very significant. Second, the single-cycle queue part can still be beneficial even if the mean population size is significantly larger than the single-single queue part, as illustrated by the difference between the sizes of 100 and 300.

This is because, for the same queue population, when more elements are stored in the single-cycle queue, less elements spill onto the heap, meaning operations complete faster as they require maintenance of a smaller heap.

3) Effect of Standard Deviation of Mean Queue Population: Figure 9 looks at the same example as before, but with lower standard deviation of the queue population size. Unlike the mean queue population, the standard deviation has very little effect on execution time. It can be seen that the difference between the execution cycles of the 100 and 300 length hybrid queue sizes are less differentiated when the standard deviation

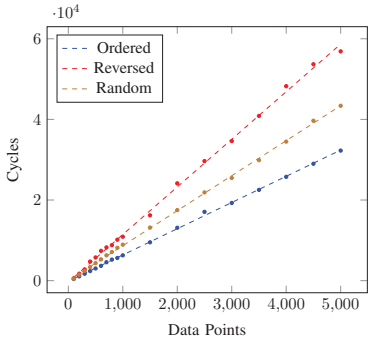


Fig. 10: HybridQ(100) for 500 mean, 25% SD (125).

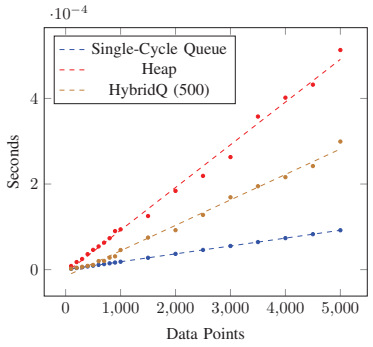


Fig. 11: Execution times (seconds) of different queues for mean population size 500, SD 25% (125).

is lower. This is likely due to less variance in the population of the hybrid queue, and thus the population of the heap is more consistent, resulting in relatively more consistent data.

4) Effect of Data Order: For a heap, ordered data is the best case scenario and reverse ordered data is the worst case scenario, as less traversing of the heap data structure is necessary.

As large single-cycle queue sizes will avoid this cost, let us look at the smallest hybrid-queue of our experiments in Figure 10. Naturally, random ordered data sits in between ordered and reverse ordered data, preforming no better than the best case of ordered data and no worse than the worst case of reverse ordered data. For larger queues, the order of data has qualitatively the same effect, but it is much less pronounced and closer to the ordered performance.

5) Effect of Queue Type on Logic Utilization and Performance: Figure 11 shows the execution times in seconds between single-cycle queue (size 1024), heap and hybrid queue (size 500) for random data with a mean queue population of 500 and 25% (125) standard deviation. Table I shows the required logic utilization and achievable clock frequency for each queue type when synthesized in Xilinx ISE 14.4 targeting a Virtex-5 LX330T-2 FPGA. In contrast to before, the execution time here relects both the number of cycles and the execution frequency.

Queue	LUTs	Registers	FMax (MHz)
Single-Cycle	$1.27 \cdot 10^5$	65,536	108.7
Heap	11,536	2,352	87.8
HybridQ (500)	62,194	32,000	91.7

TABLE I: Logic utilization for single-cycle PQ, heap and hybrid PQ with size 1,024

The performance from Figure 11 and logic utilization from Table I shows the hybrid queue offers a great compromise between resource usage and speed even for this unfavourable workload where the mean population size is the size of its single-cycle queue part. For lower mean population sizes and higher standard deviation its adantage will be more pounced.

VI. Optimisation for Burst-Heavy Workloads

For a heap where bursts of `push` operations are followed by a single `pop` operation, such as when traversing acceleration hierarchy for ray-tracing or in state space search, each incoming `push` operation must be queued, and the `pop` operation must wait until the preceding `push` operations are processed. Also, if a `push` operation is requested during maintenance of the heap for a `pop` operation, the `push` will need to be queued until maintenance for the `pop` is complete, because the order of the operations can not be modified.

The need to queue requests for both of these situations can be removed by accepting and storing `push` operations at the tail of the heap during any stage of heap maintenance, for either `push` or `pop` operations, and processing them after the current maintenance is completed. This is achieved by keeping an additional `push_location` register, writing any incoming values to the location stored in this register and incrementing it afterwards. When the heap reaches its `IDLE` state, it compares the value in the `push_location` register with the value of the `heap_size` register. If the value in the `push_location` register is larger, the heap proceeds to maintain the heap for the stored `push` values, incrementing `heap_size` until it matches the value of the `push_location` register. These modifications allow `push` operations to be accepted at any stage of operation, while `pop` operations must wait for heap maintenance to complete before being processed, desirable for a burst-heavy workload while maintaining the scalability of the heap.

VII. Conclusions

This paper proposed a fast scalable hardware priority queue, which is based on a hybrid design. It combines the performance of a single-cycle priority queue with the scalability of a heap based queue, achieved in a modular fashion. Various FPGA-specific optimizations improve the performance of the heap queue. In the experimental evaluation it was demonstrated that the resulting hybrid queue provides an excellent trade-off and which is especially interesting for workloads with low mean and high deviation of the population size. Moreover, it can be easily tuned with the choice of the single-cycle queue part.

REFERENCES

[1] D. R. Augustyn, A. Domański, and J. Domańska, "A choice of optimal packet dropping function for active queue management," in *Computer Networks* (A. Kwiecień, P. Gaj, and P. Stera, eds.), (Berlin, Heidelberg), pp. 199–206, Springer Berlin Heidelberg, 2010.

[2] S.-W. Moon, K. G. Shin, and J. Rexford, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, vol. 49, pp. 1215–1227, Nov. 2000.

[3] M. Suzuki and K. Minami, "Concurrent heap-based network sort engine - toward enabling massive and high speed per-flow queuing," in *Communications, 2009. ICC '09. IEEE International Conference on*, pp. 1–6, June 2009.

[4] M. Tommiska and J. Skyttä, "Dijkstra's shortest path routing algorithm in reconfigurable hardware," in *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, FPL '01, (London, UK, UK), pp. 653–657, Springer-Verlag, 2001.

[5] K. Sridharan, T. K. Priya, and P. Kumar, "Hardware architecture for finding shortest paths," in *TENCON 2009 - 2009 IEEE Region 10 Conference*, pp. 1–5, Jan 2009.

[6] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: Graphlet counting case study," in *FPT*, pp. 1–8, 2011.

[7] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv.*, vol. 38, no. 3, 2006.

[8] M. Huang, K. Lim, and J. Cong, "A scalable, high-performance customized priority queue," in *Proceeding of 24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany*, pp. 1–4, 2014.

[9] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, (New York, NY, USA), pp. 15–22, ACM, 2005.

[10] R. Chandra and O. Sinnen, "Improving application performance with hardware data structures," in *2010 IEEE Int Symp on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–4, April 2010.

[11] C. E. Leiserson, "Systolic priority queues," in *Proc. of Caltech Conf. VLSI*, pp. 200–214, 1979.

[12] Y. Afek, A. Bremler-Barr, and L. Schiff, "Recursive design of hardware priority queues," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pp. 23–32, ACM, 2013.

[13] A. Ioannou and M. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *IEEE/ACM Transactions on Networking*, vol. 15, no. 2, pp. 450–461, 2007.

[14] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *In Proceedings of Infocom 2000*, March 2000.

[15] H. Wang and B. Lin, "Per-flow queue management with succinct priority indexing structures for high speed packet scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1380–1389, 2013.

[16] X. Zhunag and S. Pande, "A scalable priority queue architecture for high speed network processing," in *Proc. of 25TH IEEE International Conference on Computer Communications INFOCOM 2006*, pp. 1–12, Apr. 2006.

[17] C. Kumar, S. Vyas, R. Cytron, C. Gill, J. Zambreno, and P. Jones, "Hardware-software architecture for priority queue management in real-time and embedded systems," *International Journal of Embedded Systems (IJES)*, vol. 6, no. 4, pp. 319–334, 2014.

[18] I. Benacer, F.-R. Boyer, and Y. Savaria, "A fast, single-instruction–multiple-data, scalable priority queue," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 1939–1952, 2018.

[19] I. Benacer, F.-R. Boyer, and Y. Savaria, "Hpqs: A fast, high-capacity, hybrid priority queuing system for high-speed networking devices," *IEEE Access*, vol. 7, pp. 130672–130684, 2019.

[20] S. Collinson and J. Morris, "Fast digital rendering for special effects," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 631–634, 2012.

[21] S. Collinson and O. Sinnen, "Flexible hierarchy ray tracing on fpgas," in *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 330–333, Dec 2013.

[22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction To Algorithms*. MIT Press, 2001.

[23] Xilinx, "LogiCORE IP floating-point operator v5.0." http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf, 03 2011.