Performance Evaluation of Priority Queues for Fine-Grained Parallel Tasks on GPUs

Nikolai Baudis, Florian Jacob, Philipp Andelfinger
Institute of Telematics, Karlsruhe Institute of Technology
Email: nikolai.baudis@gmail.com, florian.jacob@student.kit.edu, philipp.andelfinger@kit.edu

***Abstract*—Graphics processing units (GPUs) are increasingly applied to accelerate tasks such as graph problems and discrete-event simulation that are characterized by irregularity, i.e., a strong dependence of the control flow and memory accesses on the input. The core data structure in many of these irregular tasks are priority queues that guide the progress of the computations and which can easily become the bottleneck of an application. To our knowledge, currently no systematic comparison of priority queue implementations on GPUs exists in the literature. We close this gap by a performance evaluation of GPU-based priority queue implementations for two applications: discrete-event simulation and parallel A* path searches on grids. We focus on scenarios requiring large numbers of priority queues holding up to a few thousand items each. We present performance measurements covering linear queue designs, implicit binary heaps, splay trees, and a GPU-specific proposal from the literature. The measurement results show that up to about 500 items per queue, circular buffers frequently outperform tree-based queues for the considered applications, particularly under a simple parallelization of individual item enqueue operations. We analyze profiling metrics to explore classical queue designs in light of the importance of high hardware utilization as well as homogeneous computations and memory accesses across GPU threads.**

I. Introduction

Considering the increasing scale of networked systems in Internet of Things applications and smart cities, the evaluation of the complex behavior emerging from fine-grained interactions among large numbers of entities is becoming increasingly challenging and computationally demanding.

For many years now, the fine-grained parallelism of graphics processing units (GPUs) has been applied to accelerate computations where the operations and memory access patterns are known a priori as in, e.g., linear algebra codes or signal processing. In contrast, applications in the context of large-scale networked systems – such as discrete-event simulation, job scheduling, minimum spanning tree calculation, and path search – frequently comprise computational problems of irregular and hard-to-predict structure. Priority queues as the core data structure of many such applications have been studied for many decades. However, due to the substantial differences in the execution model between CPUs and GPUs, the relative merit of classical data structures requires a reconsideration to achieve high performance on GPUs.

In this paper, we evaluate the performance of GPU-based priority queues, focusing on two applications: 1. Discrete-event simulation of interacting entities, e.g., in a networked system. 2. Finding the shortest path among multiple pairs of cells on a grid, e.g., in an agent-based simulation. The defining property of these applications is a substantial or full independence across task instances, whereas each instance is comprised of steps that provide only limited independence.

We explore priority queue implementations on different levels of granularity: using a single GPU thread per queue, groups of threads per queue, and applying the full GPU resources to a single queue. The considered queue designs are circular buffers, implicit heaps, splay trees, and the GPU-specific parallel heap [1].

A challenge lies in the lockstep execution of groups of threads on the GPU: highest performance depends on a minimization of divergent branches and scattered memory accesses. To study the priority queues with respect to these characteristics, in addition to overall performance measurements of the priority queue designs, we present profiling results to explore the causes for the differences in performance.

Our measurement results can help guide future implementations of applications relying on GPU-based priority queues. To allow the community to reproduce our results or to build on the implementation, our code is made publicly available[1].

II. Background and Related Work

A. General Purpose Computation on GPUs

General Purpose Computation on Graphics Processing Units (GPGPU) enables the use of the thousands of arithmetic units of modern GPUs for parallel computations by scheduling large numbers of threads that execute GPU functions called kernels. In NVIDIA CUDA [2], which we use for our implementation, threads are organized in warps, which are groups of 32 threads that execute in lockstep. Blocks of up to 1 024 threads share low-latency memory and can be synchronized efficiently. A hardware scheduler hides memory access latencies by exchanging control among warps dynamically. Accesses to adjacent locations in memory by threads within a warp are coalesced, i.e., translated to a single memory transaction. Due to the lockstep execution within a warp and memory access coalescing, performance is reduced if threads within a warp frequently execute divergent branches or scattered memory accesses, which poses a challenge when considering the dynamic nature of irregular computational tasks.

[1] https://github.com/gpupq/gpupq

©2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. doi:10.1109/MASCOTS.2017.15

B. Irregular Computations on GPUs

In the past years, many previous works have focused on supporting the efficient execution of irregular computations on GPUs. Generally, most proposed approaches aim to reduce synchronization overheads and workload imbalances among threads. This can be achieved by fetching new tasks during execution of a kernel [3], [4], [5] and by minimizing the use of atomics and centralized data structures in the distribution of tasks to the GPU threads [6], [7], [8]. Recently, microarchitectural modifications to the GPU hardware have been proposed to support higher efficiency in the implementation of task lists [9] and the scheduling of new work during execution of a kernel [10].

In 2011, Barrientos et al. proposed a method for k nearest-neighbors search for database query processing on GPUs based on per-thread heaps [11]. In 2012, Merrill et al. presented a parallelization of the breadth-first search problem achieving asymptotically optimal work complexity [12]. Since in breadth-first search, a frontier of newly discovered vertices can be processed without regard for a certain ordering, the use of priority queues is not required. In the past years, multiple framework have been proposed that provide generic operations for implementing graph algorithms on GPUs [13], [14], [15].

C. Considered Applications

1) Parallel Discrete-Event Simulation: In discrete-event simulations, a modeled system is represented by state variables that are modified by events at discrete points in simulated time. A *sequential* simulation performs a simple loop: among the currently existing set of events (pending event set, PES), the event with the lowest timestamp is selected and executed. During execution, further events may be added to the PES. This process is repeated until the PES is empty or a termination condition is met. Since individual events may be associated with only marginal computational costs, an efficient priority queue design is crucial to achieve reasonable performance.

In *parallel* discrete-event simulations [16], the model is partitioned into logical processes (LPs) that are executed on separate processor cores and maintain a separate PES each. Events are exchanged among LPs to reflect the interactions in the modeled system. To maintain the causal ordering among events, the model time is synchronized across LPs.

Frequently, parallel discrete-event simulators are benchmarked using the PHOLD model [17], in which a fixed set of simulated entities exchanges messages. When an entity receives a message, the message is sent to a random entity after being delayed by a random amount of simulated time. This behavior is implemented as follows: an initial population of events is enqueued into the PES. During execution of an event, a new event is created with a time delta and a receiving simulated entity drawn from random distributions. Since each event requires only minor computations and interactions among entities are frequent, the PHOLD model exercises the core components of the simulator, i.e., the priority queue implementation and the synchronization mechanism.

2) A Path Search:* The A* algorithm [18] is an extension to Dijkstra's algorithm for finding shortest paths in a graph. To reduce the number of considered vertices, A* applies a user-supplied heuristic to select the candidate vertex that minimizes the estimated distance to the destination vertex. The state of a search is kept in two main data structures: the open and closed list. The open list holds all discovered vertices that have not been examined yet. Vertices are assigned a cost comprised of the distance from the source vertex and the estimated distance to the destination according to the heuristic. In each iteration, the vertex v with the lowest cost is examined: neighbors of v that are not in the closed list are stored in the open list, and v is stored in the closed list. Pointers link vertices to their neighbors with the lowest cost. Once the destination vertex has been examined, the shortest path is traced backwards using the neighbor pointers. Typically, the open list is a priority queue.

D. Priority Queue Design

A priority queue is a data structure that supports the operations *enqueue*, i.e., inserting an item with a priority, and *dequeue*, i.e., extracting the highest-priority item.

1) Priority Queues on CPUs: A variety of previous works [19], [20], [21] evaluated priority queues using synthetic benchmarks or concrete applications [22]. Generally, linear data structures such as linked lists were found to perform well only for miniscule queue lengths up to around 50 items. At larger item counts, while the conclusions vary, heaps, splay trees [23] or more complex proposals such as the calendar queue [24] or ladder queue [25] achieved highest performance.

To handle increasingly large item counts, a number of parallel priority queues have been proposed. Due to our focus on single-GPU execution, we omit approaches targeting distributed memory. The general ideas are: taking into account the cache hierarchy of CPUs [26], a separation of items into fully sorted high-priority items and lower-priority items with partial sorting [26], [27], decoupling of enqueue and dequeue operations [28], and splitting of a global queue into smaller segments considered individually [29], [30].

2) Parallel Priority Queues on GPUs: In 2012, He et al. [1] proposed a generic parallel priority queue design for GPUs. The queue allows for bulk enqueue and dequeue operations at the same time. The queue is structured similarly to a binary heap, but holds multiple items per node. Heap operations

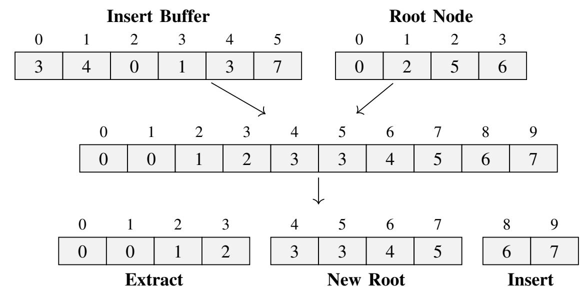


Fig. 1. Joint enqueue/dequeue operation in the parallel heap.

can be performed in parallel to an application's operations on the dequeued items. Enqueue and dequeue operations are performed at the same time. This is accomplished by merging the root node and the newly enqueued items into a sorted buffer, dequeuing the smaller half, and retaining the larger half as the new root node (cf. Figure 1). Now, the heap property is restored over multiple enqueue-dequeue cycles by merging newly enqueued items with existing items downwards along the heap levels. The required merging steps can be parallelized across multiple heap nodes.

Some previous works described GPU-based priority queues when proposing methods for **parallel discrete-event simulation**: In 2010, Park et al. presented a GPU-based simulator that stores all LPs' events in a global array. In each iteration, events that are ready to be processed are marked. After a thread identifies the earliest event of an LP, the event is executed. Event management is simplified by the assumption that each event creates exactly one new event. Thus, a newly created event can take the place of its predecessor without the need to determine a storage position and without mutual exclusion among threads. We omit some later works that make the same assumption [31], [32]. Wenjie et al. organize events in a two-dimensional structure [33]. In their approach, each new event is placed in a randomly selected column, linked lists connecting each LP's events in timestamp order. Both Zhen et al. [34] and Andelfinger et al. [35] store each LP's events in a separate array. Parallel access to an LP's queue is performed using atomic operations. In [35], the number of simulated entities assigned to each LP is adapted to balance idle threads and the cost of queue operations.

Works applying priority queues for **shortest path problems** on GPUs frequently focus on the single-source shortest path problem [36], [37], [38], in contrast to the independent source-destination pairs considered in our work. A focus of these previous works is on item deduplication. A number of previous works considered the execution of an individual search on a GPU [39], [40]. Zhou et al. [41] use multiple thousand binary heaps to hold intermediate results. The following previous works consider the parallel execution of multiple searches on a GPU. We omit previous works that do not describe the queue implementation. Bleiweiss performed A* search on up to 340 vertices, processing up to 115 600 searches in parallel [42]. Priority queues were implemented as circular buffers and implicit heaps, with heaps achieving best performance. Demeulemeester et al. [43] followed their A* implementation. Silva et al. performed up to 300 000 A* searches in parallel on graphs of 400 vertices [44], implementing priority queues as implicit heaps.

III. Considered Priority Queue Designs

In the following, we describe the considered queue variants. For each queue, we state the thread assignment, i.e. the number of GPU threads operating on each queue. An overview of the memory layout of all queue variants is given in Figure 2.

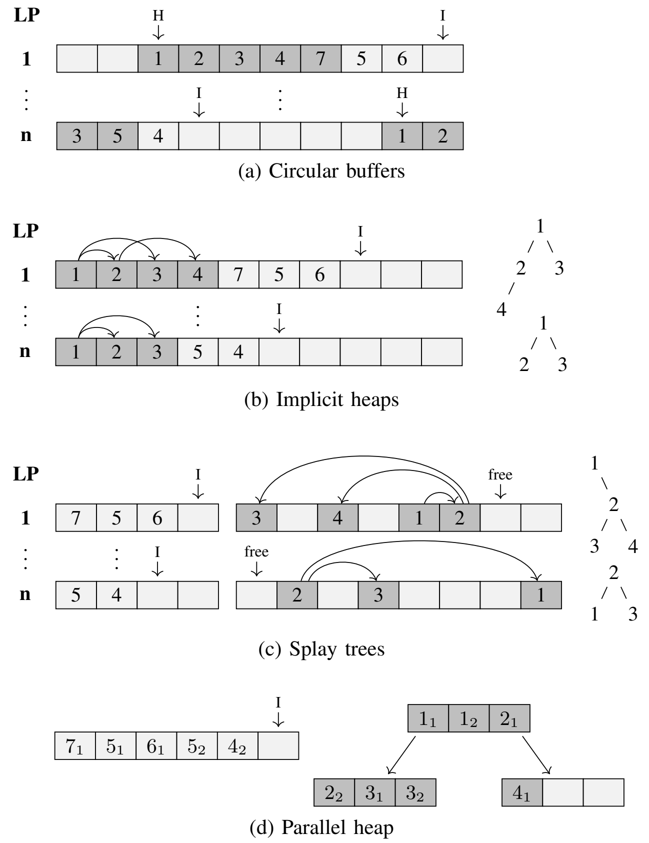


Fig. 2. Considered queue implementations. Dark gray items have already been enqueued.

A. Circular Buffers

In this queue design, a circular buffer (CB) is used to store the items pertaining to each single LP linearly in a sorted fashion. Each buffer maintains pointers to the first item in the buffer and the position of the next new item. At first, new items are appended without sorting to the end of the buffer. In the following, we consider the enqueue operation for an item to be finished once the item resides in the correct position in the data structure. The required sorting step is performed before further items are dequeued from the buffer. We consider the following approaches:

Insertion Sort (CB-SEQ): Each new item is enqueued by linearly traversing the existing items back-to-front or front-to-back, moving items of lower priority by one position until the correct position of the new item is found. Thread assignment: 1 thread per buffer.

Insertion Sort, Single Pass (CB-SEQ-SP): Several – in our implementation eight – new items are sorted and stored in a temporary cache located in low-latency memory. Now, the existing items are linearly traversed front-to-back, each thread handling one buffer. Contrary to CB-SEQ, all items to be enqueued are processed in a single pass over the circular buffer. However, while CB-SEQ terminates once the insertion position is reached, CB-SEQ-SP traverses the entire circular buffer. At each position, the new item in the buffer is the

```
INPUT: Q; e; s; k; t
Q_0 ← −∞; Q_s ← ∞; i ← s − k + 1
insert ← False; __shared__ finished ← False
while i > −k+1 and finished == False do
  for j ∈ {i, .., i+k−1} in parallel do
    c ← dummy; synchronizeThreads()
    if j > 0 then
      if c > e then copy ← True; break
      if j == s or Q_j > e then
        insert ← True; insertpos ← j
      if t == k − 1 then finished ← True
      if copy == True then Q_j ← c
    synchronizeThreads()
    if finished == True then break
  i ← i − k
if t == 0 and (s == 0 or Q_0 > e) then
  insert ← True; insertpos ← 0
synchronizeThreads()
if insert == True then Q_insertpos ← e
```

Symb.	Description
Q	Queue as sorted list
Q_i	Item at index i
e	Item to be enqueued
s	First empty Q index
k	#Threads in the block
t	Thread idx $\in\{0,k-1\}$

Fig. 3. Parallel insertion algorithm (CB-PAR).

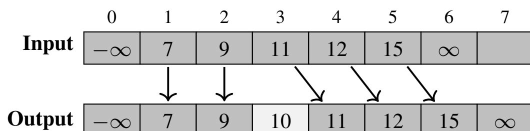


Fig. 4. Enqueuing ’10’ using CB-SEQ or CB-PAR as in Alg. III-A. Each arrow shows where an input item considered by a thread is stored in the output array. CB-SEQ performs these steps sequentially using a single thread, with CB-PAR, thread block size k steps are executed at once.

higher-priority item of the existing item and the highest-priority item in the cache. The other item is inserted into the cache. Thread assignment: 1 thread per buffer.

Insertion Sort, Parallel (CB-PAR): We propose a simple parallelization of the enqueue operation. Each buffer is handled by an entire thread block of configurable size k. For each new item, the buffer is linearly traversed back-to-front or front-to-back with a step size of k. We describe the enqueue process for back-to-front: at step i, thread t compares the item at positions $k \times i + t$ (*current*) and $k \times i + t - 1$ (*left*) with the new item (*new*). In case *new* is of higher priority than *current* and *left*, *left* is moved to the position with the next higher index. In case *new* is of lower priority than *current* and of higher priority than *left*, *new* is stored at the position of *current*. In case both *current* and *left* have lower priority than *new*, no action is taken. Read and write accesses to items are separated by barrier synchronizations, i.e., it is enforced that all threads must have completed the read operation at the current position before the subsequent write accesses are performed. In effect, CB-PAR performs the copying operations on block granularity, in contrast to the thread granularity of CPU-SEQ. Pseudo-code is provided in Algorithm III-A. An enqueue operation example is displayed in Figure 4. Thread assignment: k threads per buffer.

B. Traditional Tree-Based Queue Variants

Implicit Heap: Each LP's items are stored in an implicit binary heap. After an iteration, new items are appended at the bottom of the heap and the heap property is restored. After the highest-priority item is dequeued, the last item of the heap becomes the new root. Now, items are swapped downwards until the heap property is restored. As in the above approaches, the fact that items of an implicit heap are packed tightly in memory allows us to efficiently store each heap in a small array per LP. Thread assignment: 1 thread per heap.

Splay Tree: Splay trees, proposed by Sleator and Tarjan [23], are binary search trees that heuristically adjust to the access patterns to the tree's items. Contrary to implicit heaps, splay trees are typically implemented as pointer-based data structures. We ported the C implementation by Sleator[2] to CUDA. Since dynamic memory allocation on the GPU is extremely expensive, we implemented a simple *next fit* allocation scheme ([45], page 453): a call to `malloc_()` performs a linear search in a circular buffer and returns the first unused item. A pointer to the new item is stored to begin subsequent searches after the position of the new item. A call to `free_()` simply marks the corresponding item as unused. Thread assignment: 1 thread per heap.

C. Parallel Heap

As a priority queue targeting GPUs explicitly, we implemented the parallel heap according to He et al. [1]. We implemented the required merging steps using the moderngpu library[3], which performs sorting based on parallel mergesort. Contrary to the previous queue designs, the parallel heap is designed to handle large item counts using bulk operations. Thus, we handle items from all task instances, i.e., all simulated entities or parallel A* searches, using a single queue. To still determine the highest-priority item for each task instance, we adapted the algorithm for dequeuing items from the root node: after sorting the merged buffer, the smallest item per task instance is marked. We parallelized this step using atomic operations for the interaction among threads. Subsequently, all marked items are dequeued from the queue. During the merging step, which dominates the heap's runtime, the thread assignment is managed by the moderngpu library.

IV. Measurements

A. Experimental Setup

We performed our experiments on a system equipped with an 8-core Intel Core i7-6700 processor, 16 GiB of RAM using an NVIDIA GeForce GTX 1060 with 6 GiB of GDDR5 RAM and 1280 CUDA cores clocked at up to 1809 MHz. Preliminary experiments were performed on an NVIDIA GTX 980 Ti with similar results. We report averages of three runs with 95% confidence intervals. Experiments with CB-SEQ and CB-PAR were executed both with back-to-front and front-to-back traversal of the circular buffers. Since applications can easily select the preferable direction using runtime measurements, we report the results achieved with the higher-performing direction for each parametrization.

Parallel discrete-event simulation: To synchronize the model time across simulated entities, we implemented the YAWNS algorithm [46], which executes two alternating steps:

[2] www.link.cs.cmu.edu/link/ftp-site/splaying/top-down-splay.c
[3] https://github.com/moderngpu/moderngpu

first, the minimum timestamp t_m among the remaining events is determined. On the GPU, this step is performed in logarithmic time using parallel reduction. Event execution requires a configurable lookahead value l, which is a lower bound on the delta between an event's timestamp and the model time at its creation. For each simulated entity, the earliest event is executed in case its timestamp is below $t_m + l$.

The execution of an event is comprised of drawing two uniformly distributed random numbers u_1, u_2 using CUDA's default random number generator XORWOW. u_1 is transformed to an exponential variate x with rate parameter λ using the inverse transform: $x := -\ln(u_1)\lambda^{-1}$. Given the current model time t and the number N of entities, a new event is created at timestamp $t_m + l + x$, targeting entity $\lfloor u_2 N \rfloor$. When decreasing λ, events become more sparse in model time and the number of executable events per iteration decreases.

Events pertaining to a simulated entity reside in a separate priority queue. During execution of events, each GPU thread acts on a single simulated entity, i.e., if a simulated entity has no events with timestamps below $t_m + l$, the corresponding thread remains idle. Since no conditional statements are required during the event's execution, threads do not diverge. Previous works have proposed sorting events to efficiently support divergent event handlers (e.g., [47]).

A* path search on grids: We focus on quadratic grids of 64×64 up to 1 024×1 024 cells, performing as many parallel searches as possible given the graphics memory limits. Such scenarios can be found in agent-based simulation and multi-agent systems. We limited the grid size so that at the largest considered size, a reasonable number of searches can still be performed in parallel. On a grid of side length l, we place $l^2 \times d$ quadratic obstacles uniformly at random, d being a configurable parameter. The obstacle side length is an exponential variate with parameter η, while enforcing a minimum size of 1. Source and destination cells are chosen uniformly at random on non-obstructed cells on the grid.

The open list is a priority queue of cells to be visited. Contrary to PHOLD, we dequeue a configurable number n of items at once. Since searches are performed separately, items are not exchanged among priority queues.

As the heuristic guiding the selection of the next candidate cells, we use the minimum distance between two cells on a grid: $h = \sqrt{2}d_{\text{min}} + d_{\text{max}} - d_{\text{min}}$, with d_{min} and d_{max} being the minimum and maximum of the x and y distance between the current cell and the destination. Figure 5 shows two finished searches, visualizing the found paths and the visited cells.

Since we focus on relatively small grids, one thread block handles each search. Using multiple blocks and large numbers of threads for each search could increase the performance for large grids, but would shift the focus of our evaluation away from enqueue operations towards item deduplication across threads. Since synchronization of memory accesses within a thread block can be performed during a kernel execution, in our implementation, threads can interact without temporarily yielding control to the CPU. In each iteration, we dequeue the n cells with the lowest f value from the open list, f being the sum of the distance from the source to the current cell and the estimated distance from the current cell to the destination. Groups of 8 threads examine each cell's neighborhood. The grid is held in texture memory to benefit from caching based on spatial locality. To limit the degrees of freedom in our measurements and to focus on the performance of the priority queue implementations, we perform our experiments with a fixed n of 8 (i.e., 64 threads per block). The thread block size during priority queue operations is configured separately.

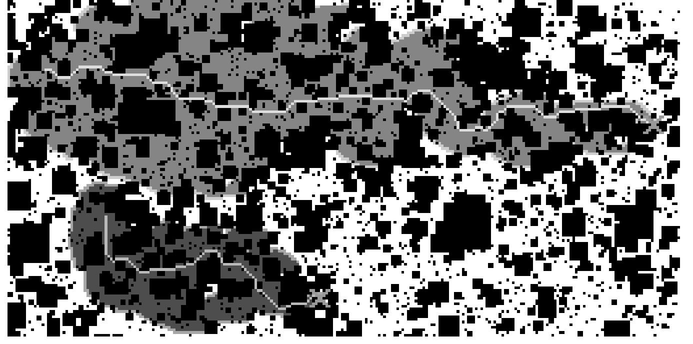


Fig. 5. Two A* searches on a 256×128 grid (cropped to conserve space; all measurements were performed on quadratic grids). White lines: shortest paths, dark areas around paths: visited cells. While the closed list contains all visited cells, each open list contains all unvisited neighbors around the already visited cells around the open list's contour. The figure illustrates the potentially large memory demand of the closed list.

In each iteration, each thread stores up to one new unobstructed cell in a buffer. Newly found cells are enqueued in the open list. Lookup tables are maintained to efficiently discard cells already in the open or closed list.

The memory demands of the A* search limit the number of parallel searches. By using 2-byte half-floats to store distances, we reduced the memory requirements to 16 byte per queue item. The data structures with the largest memory requirements are the closed lists, which can grow close to the overall grid dimensions in cases where no path exists between source and destination, and the lookup tables for avoiding duplicate work.

B. Application Performance

In the following, we present measurement results comparing the priority queue implementations for discrete-event simulations and A* path searches. The presentation of the measurement results is structured with reference to three key performance-determining aspects:

1. Occupancy of the GPU's hardware resources. Large numbers of warps are required to exploit the GPU's cores and to enable hiding of memory access latencies.

2. Thread Divergence within a warp. Conditional branches taken only by a subset of threads are executed by all threads, discarding the results of inactive threads.

3. Memory Accesses not served from caches. If there are insufficient warps to hide memory access latencies, warps may stall for multiple hundred clock cycles on a memory access.

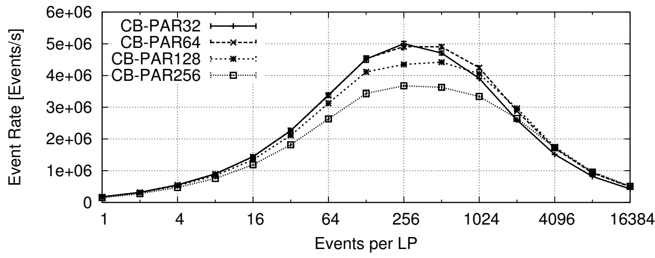


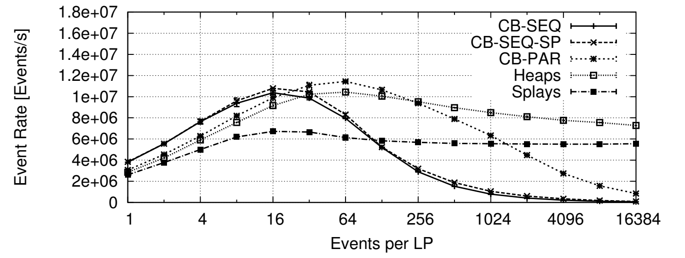
Fig. 6. PHOLD performance of the circular buffer parallel insertion sort with 1 024 LPs and $\lambda = 10^{-3}$, varying the number of threads per block acting on a queue. Due to the small difference between configurations, 32 threads per block were used in all further experiments.

The differences in observed performance among the queues result from the tradeoffs among the three aspects in the queue designs, in light of a given scenario.

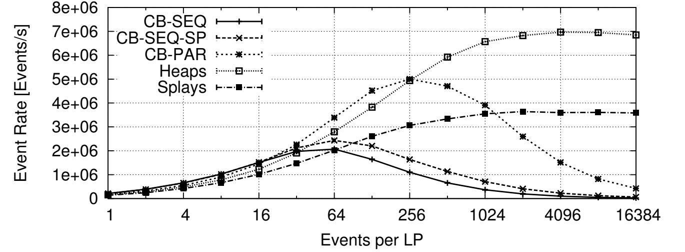
1) Parallel Discrete-Event Simulation: We measured the performance of discrete-event simulation of the PHOLD model with 1 024 and 32 768 simulated entities, varying the average number of events per LP. The largest considered number of events across all LPs was 6.7×10^7. The number of events that can be processed in parallel depends strongly on the ratio between the lookahead and the rate parameter λ of the exponential distribution. We configured a fixed lookahead of 10 units of simulated time and varied λ. The considered parameter combinations were chosen according to our previous works in GPU-based simulation [35], [48] to cover cases of low utilization where the GPU could be outperformed by a single CPU core, up to configurations approaching full GPU utilization. Initially, events with timestamps according to the configured exponential distribution are assigned uniformly at random to LPs. Each configuration was terminated when 60 seconds wall-clock time or $2^{20} \times 1000$ executed events were reached. The key performance metric in discrete-event simulation is the number of events executed per second wall-clock time (*event rate*).

Block Size: We first consider the number of threads per block (block size) used when performing queue operations. For CB-SEQ, CB-SEQ-SP, heaps, and splays, we observed no substantial dependence of the performance on the block size. Thus, a block size of 256 threads was used in all experiments. With CB-PAR, the block size determines the number of threads operating on each queue. With larger block size, more items of each queue are read and written in parallel, while occupying more hardware resources per queue and increasing the synchronization overhead among threads. Figure 6 shows our measurement results with 1 024 LPs and $\lambda = 10^{-3}$, varying Events per LP. We see that for item counts of 256 and below, 32 threads per block achieved highest performance, while for larger numbers of events per LP, using 64 threads per block was beneficial. However, the benefit is slight and in our experiments with larger LP counts, 32 threads per block achieved the highest performance in most cases. Thus, all remaining runs were performed at 32 threads per block.

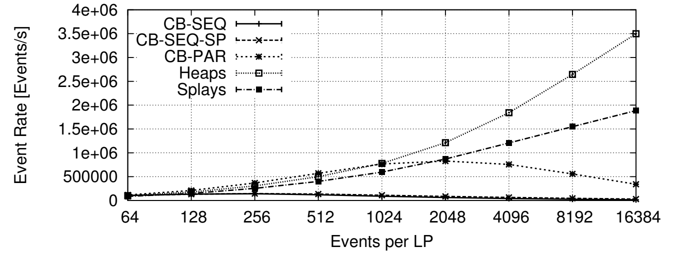
Overall Performance: Figure 7 compares the discrete-event



(a) $\lambda = 10^{-1}$



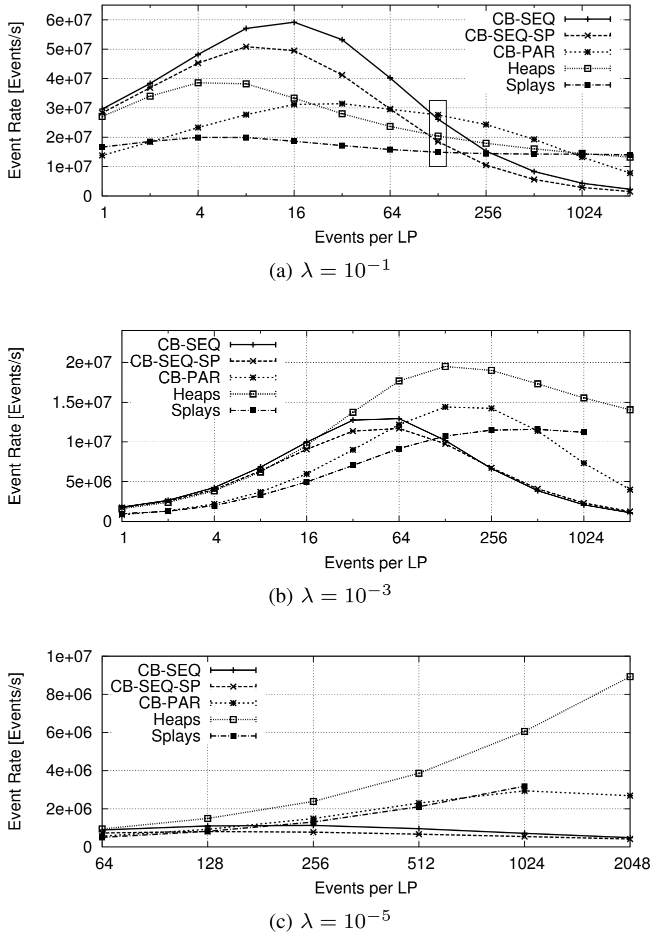
(b) $\lambda = 10^{-3}$



(c) $\lambda = 10^{-5}$

Fig. 7. Event rates achieved for PHOLD with 1 024 LPs. In general, the CB variants perform best for low queue lengths, but for larger queue lengths, they are outperformed by the heap variants. CB-PAR shows good performance because there are only 1 024 queues. With a small λ, more Events per LP are needed to achieve the same event density and simulation performance.

simulation performance across the queue variants for 1 024 LPs. The parallel heap will be evaluated separately. Generally, since the number of concurrently executable events depends on the event density in model time, higher events per LP increase the performance. However, the increase in the cost of queue operations at some point outweighs the increase in parallel event executions. Above a certain event count threshold, the measurement results adhere to the expected asymptotic behavior of the queue variants: the performance with circular buffers diminishes roughly linearly with the events per LP, while the tree-based variants show only minor decreases in performance, indicating the logarithmic cost of their queue operations. With 1 024 LPs and all configured values of λ, the circular buffer with parallel insertion achieved high performance up to about 256 events per LP. Above these event counts, the tree-based variants achieved highest performance. The sequential approaches based on circular buffers performed best at up to 16 events per LP. With the very low event density in simulated time with $\lambda = 10^{-5}$, all absolute event rates are quite low. We have previously shown



(a) $\lambda = 10^{-1}$

(b) $\lambda = 10^{-3}$

(c) $\lambda = 10^{-5}$

Fig. 8. Event rates achieved for PHOLD with 32 768 LPs. While the results are similar to Figure 7, here CB-SEQ outperforms CB-PAR across a wide parameter range, since CB-SEQ achieves higher utilization at large numbers of queues. The results of the parametrization from (a) marked with a rectangle are discussed in more detail in Table I.

that optimistic synchronization can improve performance in such situations [48].

With $32\,768$ LPs (cf. Figure 8), due to the increase in parallelism, the absolute event rates increase substantially. The curves maintain the general shape of the previous results. However, at $\lambda = 0.1$, CB-SEQ increasingly outperform the other priority queue variants up to about 128 items per LP. CB-PAR performed best only in rare cases. Again, when approaching the largest feasible event count given the memory capacity, implicit heaps achieved highest performance. The additional memory requirements for memory allocation with splay trees limited the events per LP to $1\,024$.

In Table I, we explore the causes for the observed results using metrics from profiling runs for that parametrization using the CUDA profiler `nvprof`. We focus on the kernel that performs the enqueue operation, which dominates the runtime with most queue variants. For the comparison, we selected the results marked by a rectangle in Figure 8 (a), since the queue implementations achieved similar performance for this parametrization, highlighting the differences in the causes for the observed performance. To cope with the overhead of profiling, the profiling runs were terminated after 2^{20} events for DRAM Transactions, and after 10×2^{20} events for the other profiling metrics.

Due to the additional costs of splay operations when dequeuing items, the runtime with splay trees is not as strongly dominated by the enqueue step.

The metrics' definitions are provided in Table II, taken from the CUDA programming guide [2]. The Relative Cost of Enqueuing is the fraction of overall execution time spent in the kernel that performs the enqueue operation. Theoretical Occupancy shows how many warps could fit in the limited number of GPU registers, while Achieved Occupancy shows the average number reached at execution time. Both values are relative to the maximum number supported by the GPU. Frequent divergent branches reduce the Warp Execution Efficiency, as threads that do not take a branch are passive. We instrumented the code to derive the number of DRAM Transactions per Executed Event and the number of Comparisons per Executed Event. These metrics are determined across an entire run, not only for the enqueue step. DRAM Transactions per Executed Event shows how frequently the queue has to perform memory accesses, which in addition to indicating algorithmic properties is subject to caching and coalescing.

Occupancy: Since circular buffers with sequential insertion sort (CB-SEQ) schedule only one thread per queue, whereas circular buffers with parallel insertion sort (CB-PAR) schedule 32 threads per queue, the theoretical and achieved occupancy with CB-SEQ is much higher than with CB-PAR.

Thread Divergence: With CB-PAR, the threads within a warp perform nearly the same steps. Hence, CB-PAR achieves much higher warp efficiency than CB-SEQ. Heaps achieve the highest warp execution efficiency. Since the heap depth increases only logarithmically with the number of items, there is only little variance across threads in the traversed number of items. Since at each depth, all threads compare a parent node with its children and potentially swap two nodes, thread divergence is low. In contrast, since in the worst case, CB-SEQ, CB-SEQ-SP, and CB-PAR consider all items in the queue, the potential for thread divergence is high. Due to the divergent splay operations depending on the access patterns to the stored items, splays achieve much lower warp efficiency than heaps.

Memory Accesses: Due to the smaller number of item comparisons required, heaps and splays require much fewer overall instructions than the other queue variants. Nevertheless, CB-SEQ outperformed heaps, demonstrating the tradeoff between caching of memory accesses and asymptotic behavior. Figure 9 shows DRAM Transactions per Executed Event when varying the number of events per LP. Since in CB-SEQ, each thread iterates linearly over adjacent items, most item accesses can be served from the cache. In an implicit heap, the offset between parent and child nodes increases exponentially with the heap depth, resulting in frequent cache misses once parent and child nodes reside in separate cache lines. Accordingly, DRAM Transactions per Executed Event is lower with CB-

TABLE I
PROFILING METRICS FOR ONE ITEM ENQUEUE OPERATION IN PHOLD WITH 32 768 ENTITIES, 128 EVENTS PER LP AND $\lambda = 0.1$. ALL CIS ARE <1%

Metric	CB-SEQ	CB-SEQ-SP	CB-PAR	Heaps	Splays
Relative Cost of Enqueuing	0.78	0.84	0.80	0.69	0.43
Instructions Executed	1.23×10^7	1.85×10^7	1.37×10^7	$\mathbf{6.05\times10^5}$	2.89×10^6
Instructions per Cycle	0.81	**0.93**	0.90	0.04	0.20
Theoretical / Achieved Occupancy	1.0 / 0.43	1.0 / 0.24	0.5 / 0.24	**1.0 / 0.97**	0.75 / 0.65
Warp Execution Efficiency	0.17	0.76	0.78	**0.83**	0.18
Global Hit Rate	0.70	0.48	**0.77**	0.28	0.35
DRAM Transactions per Exec. Event	**48.83**	89.72	55.92	66.16	86.27
Comparisons per Exec. Event	63.94	97.36	129.30	**12.77**	26.35
Event Rate	2.61×10^7	1.85×10^7	$\mathbf{2.76\times10^7}$	2.04×10^7	1.49×10^7

TABLE II
DEFINITIONS OF CUDA PROFILING METRICS [2]

Metric	Description
Theoretical Occupancy	Max. num. of warps supported per multiprocessor
Achieved Occupancy	Ratio of avg. active warps per active cycle to max. num. of warps supported on a multiprocessor
Warp Exec. Efficiency	Ratio of avg. active threads per warp to max. num. of threads per warp supported on a multiprocessor
DRAM Transactions	Device memory read/write transactions
Global Hit Rate	(Cache) hit rate for global loads

SEQ for small numbers of events per LP. At larger event counts, the benefit of CB-SEQ's more frequent cache hits is dwarfed by the linear dependence of the overall item accesses on the number of events per LP. The tradeoff between caching and the number of comparisons is also visible in the Global Hit Rate and Comparisons per Event metrics for CB-SEQ and heaps in Table I. We assume that the low Executed IPC of heaps is a result of the frequent cache misses.

Splays perform larger numbers of comparisons than heaps, but require fewer DRAM transactions. We suspect that the cause is that in contrast to the rapidly increasing offset between parent and child nodes in implicit heaps, in our pointer-based splay implementation, parent and child nodes may still reside in the same cache line independently of the tree depth.

Comparing the DRAM transactions between CB-SEQ and CB-PAR shows the effects of the coarser granularity of CB-PAR: CB-SEQ terminates once the insertion position has been found, whereas CB-PAR operates on a granularity of 32 threads, i.e., typically, a number of items will unnecessarily be considered.

Parallel Heap: Contrary to the previous queue variants, the parallel heap is intended for bulk enqueue and dequeue operations on large item counts and thus maintains the heap property across *all LP's* items. As described in Section III-C, if all events are stored in a single queue, an additional step is required to select the earliest event per LP. In our experiments, this step required around 10% of the total runtime. As shown on an example in Figure 10, the parallel heap performance was substantially lower than the other queue variants.

We additionally explored the raw performance of the parallel heap under idealized conditions by dequeuing all items from the root node in bulk. The same number of new items is then created according to PHOLD. Figure 11 shows measurement results under these idealized conditions for $\lambda = 0.1$. The parallel heap achieves rates up to 1.52×10^8 items per second, demonstrating its potential for applications allowing bulk operations. Still, with larger numbers of heap nodes, the performance decreases below the level achieved using the best-performing of the other queue variants. We cannot conclude whether the limited performance of the parallel heap could be improved by optimizations to our implementation or whether the parallel heap is inherently unsuited for the considered small item counts per entity. Since the computational cost of event handling in PHOLD is low, we did not implement parallel execution of enqueue and dequeue operations with the application code. Applications with large item processing costs would benefit from executing these steps in parallel. Future work could consider a finer-grained parallel heap implementation to support multiple parallel heaps concurrently, e.g., using one thread block per parallel heap.

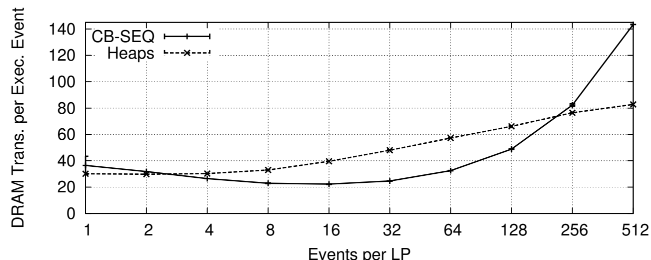


Fig. 9. DRAM transactions per executed event, PHOLD with 32 768 LPs, $\lambda = 10^{-1}$. While the linear item accesses of CB-SEQ frequently request cached items, the increasing spread between parent and child nodes in the implicit heaps results in frequent cache misses. Still, the heaps' logarithmic dependence of the item accesses on the item count leads to fewer DRAM transactions above 256 events per LP.

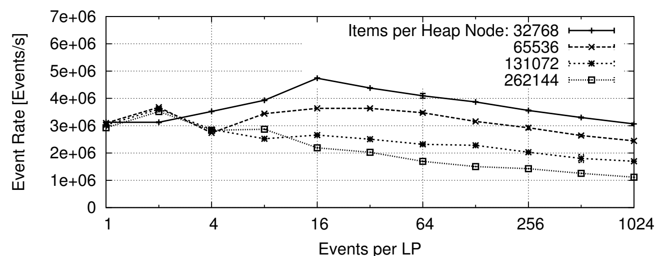


Fig. 10. PHOLD performance of the parallel heap with 32 768 LPs, $\lambda = 0.1$. Because this heap is working at a node granularity of a large number of items and puts all items in a single heap, the changes in effort are more discrete than in other queues, which results in more jagged curves. The overall performance is substantially lower than with the other considered queues.

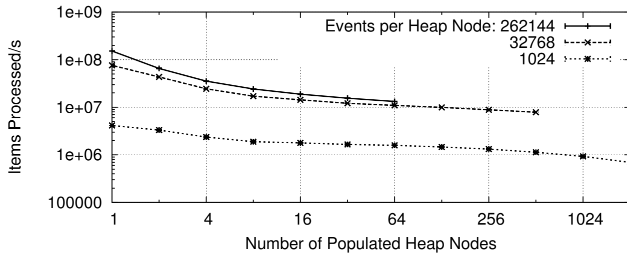


Fig. 11. Performance of the parallel heap for bulk operations adapted to the granularity of the heap. Under these idealized conditions, the parallel heap outperforms the previous approaches. However, with our implementation, this is only the case for small numbers of populated heap nodes.

TABLE III
A* SEARCHES FINISHED PER SECOND WITH THE LARGEST FEASIBLE NUMBER OF PARALLEL SEARCHES (PAR.) AT OBSTACLE DENSITY 0.2 AND $\eta = 1$. ∅LEN.: AVG. QUEUE LENGTH, ∅INS.: AVG. INSERTION POSITION.

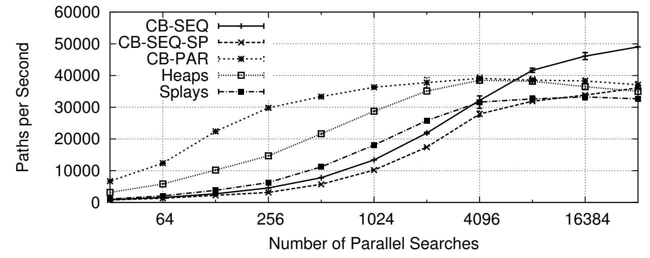
Scenario				Searches per second				
Size	Par.	∅Len.	∅Enq.	CB-SEQ	CB-SEQ-SP	CB-PAR	Heaps	Splays
64^2	32768	57.6	0.30	**49016.2**	36104.9	37088.5	35009.4	32656.0
128^2	8192	114.3	0.37	10676.8	8177.7	**12412.0**	**12436.6**	11428.9
256^2	2048	235.7	0.43	960.5	906.6	**2620.9**	**2956.3**	2601.8
512^2	512	322.6	0.45	48.9	43.0	**327.9**	**314.7**	249.9
1024^2	128	532.8	0.35	3.8	3.5	**41.7**	**40.6**	24.4

2) A Path Search:* We varied the grid side length in powers of two between 64×64 and 1 024×1 024 and configured obstacle densities of 0.1, 0.2, 0.4, with exponentially distributed sizes with $\eta = 1$ and a minimum size of 1. Since the results were similar for the different obstacle densities, we focus on a density of 0.2. The number of parallel path searches was increased up to the graphics memory capacity.

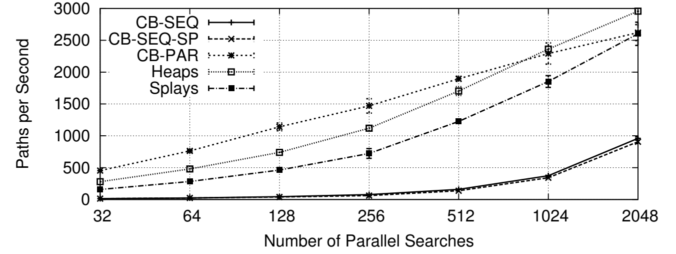
Table III lists the number of path searches per second wall-clock time achieved using the different priority queue implementations. The average open list length emerges from the grid size, the obstacle placement and the A* implementation (e.g., the number of open list items considered in parallel, deduplication, and the chosen heuristic). ∅Enq. denotes the average relative position within the priorities of the existing items at which a new item resides. We observe that circular buffers achieved best performance for 64×64 grids. CB-SEQ performed best as it benefits the most from low item counts and large numbers of parallel queues. With larger grids, queue lengths increase and the number of parallel agents decreases. Hence, CB-PAR and implicit heaps outperformed the other approaches, both to a similar degree. This is consistent with the PHOLD results, where for item counts up to about 530, CB-PAR and Heaps achieved similar performance. At a grid size of 1 024×1 024, 128 parallel searches sufficed to exhaust the graphics memory. Although higher absolute A* performance might be achievable when employing more than 64 threads per search, we maintained this configuration for comparability across grid sizes. At 1 024×1 024 cells, CB-SEQ and CB-SEQ-SP were outperformed by more than one order of magnitude.

The results suggest that the overall performance depends strongly on the number of parallel searches, i.e., further speedup would be possible given larger graphics memory capacity. To investigate this observation further, Figure 12 shows the A* performance when varying the number of parallel searches. The rightmost data points correspond to the results of Table III. At 64×64 grids, CB-SEQ benefits strongly from larger numbers of parallel searches and outperforms the other approaches at 8 192 and 16 384 parallel searches. We suspect that this is caused by the highly variable queue lengths across parallel A* searches: since threads within a warp act on separate queues and may diverge to a large degree, large numbers of warps are required to compensate.

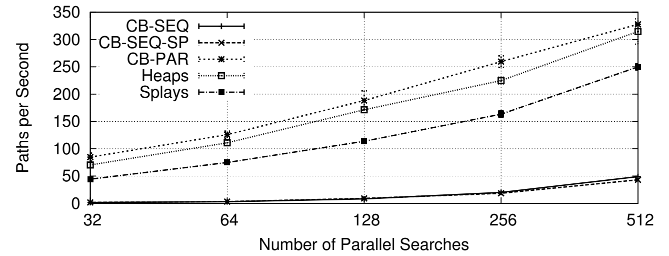
With 256×256 grids, the results indicate that a larger graphics memory capacity would enable substantial performance increases. The same trend was observed with larger grids. With CB-SEQ as our best-performing queue for a grid size of 64×64, we achieved an average time to completion for a path search of 20.4µs. Bleiweiss [42] and Silva [44] achieved similar results of about 10-20µs in 2008 and 2011, but considering much smaller vertex counts.



(a) 64×64 grid.



(b) 256×256 grid.



(c) 512×512 grid.

Fig. 12. A* searches finished per second with $0.2 \times \text{size}^2$ obstacles. The rightmost points are determined by memory limits and correspond to Table III. At 64×64, CB-SEQ benefits the strongest from larger agent counts. With grids larger than 64×64, the incline indicates further that performance gains could be achieved with larger memory capacity.

C. Discussion

The measurement results showed that priority queues based on circular buffers frequently outperformed binary heaps and splay trees for item counts up to around 500. Whether operations on individual priority queues should be parallelized depends on the number of parallel queues. While parallel per-queue operations more frequently achieve efficient coalesced memory accesses, occupancy may be reduced substantially.

In discrete-event simulations, the number of items per queue depends on the simulation model. Here, we experimented with arbitrarily chosen item counts of up to 16 384 per queue, and up to 6.7×10^7 items in total. In A* path searches, the effective queue size emerges from the grid size, the chosen heuristic, and the placement of obstacles. Since the open list contains only discovered cells for which neighbors have not been examined, with two-dimensional grids, we observed average queue sizes of only up to about 500 items, suggesting queues based on circular buffers. While the performance measurements results were quite similar for the two considered applications, future work could consider further distributions of item priorities and different GPU architectures. During our implementation and experimental design, some measurements were performed on an NVIDIA GTX 980 Ti based on the Maxwell architecture. We observed the same trends as with the NVIDIA GTX 1060 based on the Pascal architecture used in our final measurements.

Ideally, the implementer of a priority queue-based algorithm should not be required to select an optimal queue variant for each problem and instance. Autotuning approaches, which have previously been shown to be highly beneficial in the GPU context [49], [35], might help in selecting a suitable queue. Switching among the insertion methods for circular buffers comes at no runtime cost. Switching to a tree-based design, however, would require a reorganization of the items in graphics memory.

In discrete-event simulations, many threads may remain idle in cases where the event density in simulated time is low. The literature proposes two solutions: first, merging queues of multiple simulated entities increases the probability of having events that can safely be executed [35]. Second, applying optimistic synchronization instead of the conservative approach used in this paper has been shown to increase the performance at low event densities [48].

Our A* results indicate that the graphics memory capacity limits the performance for all but the smallest considered problem sizes, since larger numbers of parallel agents could utilize the GPU's cores more fully. If, as we assumed in our experiments, searches consider separate source-destination pairs and thus cannot be considered in aggregate, efforts to decrease the size of the largest data structures such as the closed list could unlock substantial performance gains.

Future work could consider alternative memory layouts to further increase the opportunities for memory access coalescing across problem instances. For instance, in preliminary experiments, we achieved promising results by interleaving the nodes of multiple separate heaps in graphics memory.

V. Conclusions

We presented a performance study of priority queues for fine-grained parallel tasks on GPUs. Contrary to previous works, we focused on large numbers of small-scale problem instances as represented by the per-entity computations in parallel-discrete-event simulations and large number of A* path searches in the context of agent-based simulations and multi-agent systems. The considered queue variants were simple linear circular buffers, tree-based queues with sequential access and a GPU-specific proposal from the literature.

The overall queue performance is dominated by three aspects: occupancy of the GPU hardware, divergence across threads, and uncached memory accesses. Our main observations are as follows: with circular buffers, the hardware utilization at *low* numbers of parallel queues can be increased by parallelizing individual enqueue operations using a separate thread group per queue, at the cost of limiting the utilization at *higher* numbers of queues. Due to their cache-friendly memory access patterns, circular buffers frequently outperformed tree-based queue variants up to about 500 items per queue for the two considered applications. At larger item counts, due to the logarithmic asymptotic behavior of heaps and a highly homogeneous behavior across threads, heaps outperform the other approaches.

Since the relative performance of the queue variants was shown to vary substantially depending on the application parametrization, an attractive direction for future work is the automated selection of suitable queue variants at runtime through autotuning. We hope that future studies and applications can benefit from our publicly available implementation.

References

[1] X. He, D. Agarwal, and S. K. Prasad, "Design and Implementation of a Parallel Priority Queue on Many-Core Architectures," in *International Conference on High Performance Computing*. IEEE, 2012, pp. 1–10.

[2] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*. Version 8.0, NVIDIA Corporation, 2017.

[3] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics*. ACM, 2009, pp. 145–149.

[4] S. Solomon and P. Thulasiraman, "Performance Study of Mapping Irregular Computations on GPUs," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[5] S. Tzeng, B. Lloyd, and J. D. Owens, "A GPU Task-Parallel Model with Dependency Resolution," *Computer*, vol. 45, no. 8, pp. 34–41, 2012.

[6] S. Tzeng, A. Patney, and J. D. Owens, "Task Management for Irregular-Parallel Workloads on the GPU," in *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010, pp. 29–37.

[7] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-Free Irregular Computations on GPUs," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013, pp. 96–107.

[8] C.-C. Kao and W.-C. Hsu, "An Adaptive Heterogeneous Runtime Framework for Irregular Applications," *Journal of Signal Processing Systems*, vol. 80, no. 3, pp. 245–259, 2015.

[9] J. Y. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs using Fine-Grain Hardware Worklists," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 75–87.

[10] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, " Dynamic Thread Block Launch: a Lightweight Execution Mechanism to Support Irregular Applications on GPUs," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 528–540.

[11] R. J. Barrientos, J. I. Gómez, C. Tenllado, M. P. Matias, and M. Marin, "kNN Query Processing in Metric Spaces Using GPUs," in *European Conference on Parallel Processing*. Springer, 2011, pp. 380–392.

[12] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[13] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES'14. New York, NY, USA: ACM, 2014, pp. 2:1–2:6.

[14] J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, June 2014.

[15] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "GBTL-CUDA: Graph Algorithms and Primitives for GPUs," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 912–920.

[16] R. M. Fujimoto, "Parallel Simulation: Parallel and Distributed Simulation Systems," in *Proceedings of the 33nd Winter Simulation Conference*. IEEE Computer Society, 2001, pp. 147–157.

[17] ——, "Performance Measurements of Distributed Simulation Strategies," DTIC Document, Tech. Rep., 1987.

[18] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[19] W. M. McCormack and R. G. Sargent, "Analysis of Future Event Set Algorithms for Discrete Event Simulation," *Communications of the ACM*, vol. 24, no. 12, pp. 801–812, 1981.

[20] D. W. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM*, vol. 29, no. 4, pp. 300–311, 1986.

[21] R. Rönngren and R. Ayani, "A Comparative Study of Parallel and Sequential Priority Queue Algorithms," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 2, pp. 157–209, 1997.

[22] C. L. L. Hendriks, "Revisiting Priority Queues for Image Analysis," *Pattern Recognition*, vol. 43, no. 9, pp. 3003–3012, 2010.

[23] D. D. Sleator and R. E. Tarjan, "Self-Adjusting Binary Search Trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[24] R. Brown, "Calendar Queues: a fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[25] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng, "Ladder queue: An O(1) Priority Queue Structure for Large-Scale Discrete Event Simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 15, no. 3, pp. 175–204, 2005.

[26] P. Sanders, "Fast Priority Queues for Cached Memory," *Journal of Experimental Algorithmics (JEA)*, vol. 5, p. 7, 2000.

[27] S. Gupta and P. A. Wilsey, "Lock-Free Pending Event Set Management in Time Warp," in *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2014, pp. 15–26.

[28] T. Bingmann, T. Keh, and P. Sanders, *A Bulk-Parallel Priority Queue in External Memory with STXXL*. Cham: Springer International Publishing, 2015, pp. 28–40.

[29] T. Dickman, S. Gupta, and P. A. Wilsey, "Event Pool Structures for PDES on Many-Core Beowulf Clusters," in *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. ACM, 2013, pp. 103–114.

[30] H. Rihani, P. Sanders, and R. Dementiev, "Brief Announcement: Multiqueues: Simple Relaxed Concurrent Priority Queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 80–82.

[31] J. Sang, C.-R. Lee, V. Rego, and C.-T. King, "A Fast Implementation of Parallel Discrete-Event Simulation on GPGPU," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2013, p. 501.

[32] B. P. Swenson, "Techniques to Improve the Performance of Large-Scale Discrete-Event Simulation," Dissertation, Georgia Institute of Technology, 2015.

[33] T. Wenjie, Y. Yiping, and Z. Feng, "An Expansion-Aided Synchronous Conservative Time Management Algorithm on GPU," in *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. ACM, 2013, pp. 367–372.

[34] L. Zhen, Q. Gang, G. Gang, and C. Bin, "A GPU-Based Simulation Kernel within Heterogeneous Collaborative Computation on Large-Scale Artificial Society," *International Journal of Modeling and Optimization*, vol. 4, no. 3, p. 205, 2014.

[35] P. Andelfinger and H. Hartenstein, "Exploiting the Parallelism of Large-Scale Application-Layer Networks by Adaptive GPU-Based Simulation," in *Proc. of the Winter Simul. Conf.* IEEE, 2014, pp. 3471–3482.

[36] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "A New GPU-Based Approach to the Shortest Path Problem," in *Int'l Conf. on High Perf. Computing and Simul.* IEEE, 2013, pp. 505–511.

[37] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *Int'l Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 349–359.

[38] P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine, "Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms," in *Proceedings of the Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 11.

[39] J. T. Kider, M. Henderson, M. Likhachev, and A. Safonova, "High-Dimensional Planning on the GPU," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2010, pp. 2515–2522.

[40] C. McMillan, E. Hart, and K. Chalmers, "Collaborative Diffusion on the GPU for Path-Finding in Games," in *European Conf. on the Applications of Evolutionary Computation*. Springer, 2015, pp. 418–429.

[41] Y. Zhou and J. Zeng, "Massively Parallel A* Search on a GPU," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15. AAAI Press, 2015, pp. 1248–1254.

[42] A. Bleiweiss, "GPU Accelerated Pathfinding," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Eurographics Association, 2008, pp. 65–74.

[43] A. Demeulemeester, C.-F. Hollemeersch, P. Mees, B. Pieters, P. Lambert, and R. Van de Walle, "Hybrid Path Planning for Massive Crowd Simulation on the GPU," in *International Conference on Motion in Games*. Springer, 2011, pp. 304–315.

[44] A. Silva, F. Rocha, A. Santos, G. Ramalho, and V. Teichrieb, "GPU Pathfinding Optimization," in *2011 Brazilian Symposium on Games and Digital Entertainment (SBGAMES)*. IEEE, 2011, pp. 158–163.

[45] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[46] D. M. Nicol, "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations," *Journal of the ACM*, vol. 40, no. 2, pp. 304–333, 1993.

[47] G. Kunz, D. Schemmel, J. Gross, and K. Wehrle, "Multi-Level Parallelism for Time- and Cost-Efficient Parallel Discrete Event Simulation on GPUs," in *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 2012, pp. 23–32.

[48] X. Liu and P. Andelfinger, "Time Warp on the GPU: Design and Assessment," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2017, pp. 109–120.

[49] M. Tillmann, T. Karcher, C. Dachsbacher, and W. F. Tichy, "Application-Independent Autotuning for GPUs," in *International Conference on Parallel Computing*, 2013, pp. 626–635.