Time Warp on the GPU: Design and Assessment

Xinhu Liu and Philipp Andelfinger
Institute of Telematics
Karlsruhe Institute of Technology
76131 Karlsruhe
xinhu.liu90@gmail.com,
philipp.andelfinger@kit.edu

ABSTRACT

The parallel execution of discrete-event simulations on commodity GPUs has been shown to achieve high event rates. Most previous proposals have focused on conservative synchronization, which typically extracts only limited parallelism in cases of low event density in simulated time. We present the design and implementation of an optimistic fully GPU-based parallel discrete-event simulator based on the Time Warp synchronization algorithm. The optimistic simulator implementation is compared with an otherwise identical implementation using conservative synchronization. Our evaluation shows that in most cases, the increase in parallelism when using optimistic synchronization significantly outweighs the increased overhead for state keeping and rollbacks. To reduce the cost of state keeping, we show how XORWOW, the default pseudo-random number generator in CUDA, can be reversed based solely on its current state. Since the optimal configuration of multiple performance-critical simulator parameters depends on the behavior of the simulation model, these parameters are adapted dynamically based on performance measurements and heuristic optimization at runtime. We evaluate the simulator using the PHOLD benchmark model and a simplified model of peer-to-peer networks using the Kademlia protocol. On a commodity GPU, the optimistic simulator achieves event rates of up to 81.4 million events per second and a speedup of up to 3.6 compared with conservative synchronization.

1. INTRODUCTION

In the past years, heterogeneous GPU-accelerated simulation approaches have begun to complement traditional CPU-based parallel and distributed simulations to satisfy the computational demands created by the increasing scale and complexity of discrete-event models. Even inexpensive commodity many-core GPUs can provide substantial speedup compared with a purely CPU-based execution.

Since large-scale simulations using CPU-based clusters frequently occupy substantial amounts of computing resources, such approaches must be justified by the gains in productivity. Recent work has also begun to take the energy consumption of distributed simulations into consideration [5]. Further, since researchers typically rely on shared computational resources, the delays between submission of a simulation task and its execution can be substantial. In such cases, a commodity GPU in a researcher's workstation can enable swift feedback at comparatively low resource and energy consumption.

GPU-based discrete-event simulation approaches fall into two main categories: *Hybrid* CPU-GPU-based simulation [2, 4, 11, 23] maintains a CPU-based simulator core, while executing some or all parts of the simulation model on the GPU. Hybrid simulation performs well in case individual simulation events are associated with significant computation. In *fully* GPU-based simulation [1, 13, 21, 22, 28, 32], the simulator core is executed on the GPU as well. Since significant data transfers between the CPU and GPU context are required only during initialization and termination, fully GPU-based simulation can efficiently execute models with fine-grained events as well.

In parallel simulations, the choice of synchronization algorithm has a strong impact on performance. Most previous works on fully GPU-based simulation focused on conservative synchronization, where the simulation correctness is maintained at all times. These works already demonstrated the large event rates achievable on commodity GPUs.

Optimistic synchronization allows for temporary violations of correctness, but restores a previous simulation state in case such a violation occurs. Hence, larger parallelism may be extracted at the cost of occasional rollbacks. On the one hand, optimistic synchronization seems to be an attractive approach to fully GPU-based simulation, since the massively parallel computational resources of a GPU may be utilized more fully. On the other hand, the rollback mechanism tends to increase the frequency of memory accesses, which are associated with particularly large costs on the GPU.

In this paper, we aim to clarify whether the benefits of optimistic synchronization outweigh its drawbacks in the context of fully GPU-based discrete-event simulation and make the following main contributions:

1. Time Warp on the GPU: We propose a design and implementation of optimistic synchronization for parallel discrete-event simulation on a GPU. We identify key simulator parameters and apply heuristic optimization to select suitable parameter combinations at runtime. The simulator and model code is made available to the community[1].

[1] http://github.com/GPUTW

SIGSIM-PADS'17, May 24 - 26, 2017, Singapore, Singapore

ISBN 978-1-4503-4489-0/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3064911.3064912

2. Reverse Computation for CUDA RNG: We show how XORWOW, the default random number generator in CUDA, can be reversed to reduce the cost of rollbacks.

3. Comparison with Conservative Synchronization: The overall performance is evaluated in comparison with a synchronous conservative scheme on the example of the PHOLD benchmark model and a simplified model of peer-to-peer networks based on the Kademlia protocol.

The remainder of the paper is organized as follows: Section 2 discusses required background and related work in parallel discrete-event simulation and general-purpose computation on GPUs. Section 3 describes our simulator design and implementation. Section 4 analyzes performance-critical parameters and describes the autotuning approach. Section 5 presents a reversal of the XORWOW random number generator. Section 6 evaluates the performance of the simulator. Section 7 discusses limitations and open research issues. Section 8 summarizes and concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 Parallel Discrete-Event Simulation

Parallel discrete-event simulation is a method to decrease the wall-clock runtime of a discrete-event simulation by distributing the system state to a number of *logical processes* (LPs), each of which maintains its own simulation time and executes events in non-decreasing timestamp order. Newly created events may be sent to other LPs. To maintain the causal correctness of the simulation, two main classes of synchronization algorithms have been proposed: in conservative synchronization, LPs execute events only in case future causality violations can be ruled out, i.e., no event with smaller timestamp may be received from a remote LP. Optimistic synchronization allows for temporary causality violations and subsequently restores a previous correct simulation state using a rollback mechanism. An introduction to parallel and distributed simulation is given in [8].

YAWNS [19] is a well-known conservative synchronization algorithm that executes the simulation by considering a sequence of intervals in simulated time spanning the current minimum timestamp of any event in the simulation up to a limit determined using the lookahead, i.e., the lowest possible delta between a new event's timestamp and its creation. Valid lookahead values are determined according to simulation model characteristics and may vary over time and between different pairs of simulated entities. In simulations of computer networks, a lookahead value that is valid over the course of the entire simulation is given by the lowest possible link latency between any two nodes in the network. Events are processed in multiple *executions*, wherein each LP either executes one event or is idle. Assuming identical per-event processing times, if events within a window are not evenly distributed to LPs, an increasing number of LPs becomes idle before a new window is determined.

Time Warp [10] is an optimistic synchronization algorithm that can scale to millions of cores [3]. Rollbacks in Time Warp are bounded by global virtual time (GVT), which is a point in simulated time at which the correctness of the simulation state is guaranteed. If none of the LPs has executed any events out of timestamp order yet, the GVT can be calculated as the lowest timestamp of any LP's next event, if any. If a causality violation occurs, the simulation must be rolled back at most to the GVT.

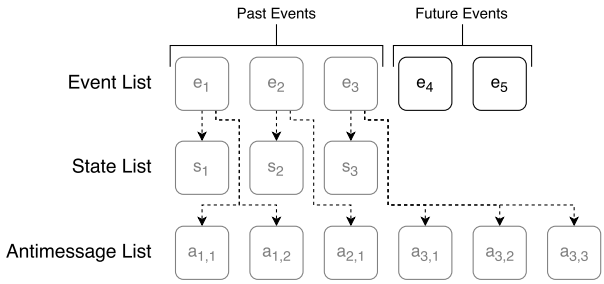


Figure 1: The three lists used in the Time Warp protocol.

Contrary to the single future event list per LP in conservative algorithms, Time Warp requires three lists per LP: The **event list** is similar to the future event list in conservative synchronization, but may temporarily hold past events as well. During a rollback, past events may be undone and subsequently re-executed. The **state list** stores snapshots of an LP's state that are used during rollbacks to restore a correct simulation state. The **antimessage list** contains a copy ("antimessage") of each event sent to a remote LP. Antimessages are sent to remote LPs to cancel previously sent events. The remote LP removes the event from its event list. If the event in question has already been handled, the LP rolls back to an earlier point in simulated time.

Figure 1 illustrates the three lists. The event list contains past and future events. For each past event, there is one state in the state list and as many entries in the antimessage list as there are newly scheduled events.

Depending on the operations performed in event handlers, using *reverse computation* [6] can be applied to arrive at a previous state. While some operations such as incrementation of an integer are non-destructive and can be reversed without any state saving, typically a combination of reverse combination and state saving is applied. The benefits of reverse computation are a decreased memory consumption and – depending on the cost of memory accesses in relation to computation – a potential decrease in simulation runtime.

2.2 GPGPU Basics

While GPUs were originally designed to handle the data-parallel workloads involved in rendering three-dimensional scenes, general-purpose computation on GPUs (GPGPU) enables the use of the hundreds of arithmetic units of modern graphics cards to solve computational problems in a variety of domains such as signal processing or machine learning. Here, we briefly discuss key GPGPU concepts, focusing on the GPGPU platform NVIDIA CUDA [20].

CUDA code is formulated in so-called kernels, which can be called from CPU code. GPU threads execute in groups of 32 called warps. Threads in a warp execute a shared sequence of instructions in lock-step. In case of divergent branching (e.g., through *if-else*), the branches are serialized. Threads are further organized into blocks of configurable size, which are assigned a small region of low-latency memory shared within the block. An important restriction is that the visibility of writes to memory among threads *within* a block can be guaranteed during execution of a kernel without significant overhead, whereas guaranteeing the visibility across blocks is more expensive and typically achieved by terminating the currently running kernel. CUDA provides a

number of atomic operations for memory accesses. Since the graphics memory is designed for high throughput instead of low latency, cache misses can incur up to 600 clock cycles of latency. Given a sufficient number of concurrent warps, a hardware scheduler can hide memory access latencies by dynamically exchanging control between warps.

2.3 GPU-Based Simulation

We differentiate between two classes of approaches that make use of GPUs for acceleration of discrete-event simulations: in *hybrid* GPU-based simulation, event management – i.e., the selection of events for execution and the scheduling of new events – remains on the CPU, while the execution of events is performed in parallel on the GPU (e.g., [2, 11, 23, 4]). A benefit of the hybrid approach is that multiple GPUs can be utilized easily. On the other hand, events must be sufficiently computationally expensive so that the parallel execution on the GPU amortizes the cost of data transfers between host and graphics memory.

In *fully* GPU-based simulation, event management is performed on the GPU as well. Hence, fully GPU-based simulation largely avoids the costs of data transfers between host and graphics memory and can efficiently support models where individual events are associated with only small amounts of computation. Here, we focus on fully GPU-based discrete-event simulation, omitting time-stepped approaches and works that focus on specific model domains (e.g., [17]).

In 2006, Perumalla explored various fully GPU-based simulator designs [22]. Although a traditional discrete-event execution process was not feasible due to hardware limitations at the time, substantial speedup was achieved by exploiting the concurrency of events with identical timestamps.

In 2010, Park et al. proposed a fully GPU-based simulator that uses a tolerance interval in simulated time to increase the number of events that can be executed in parallel [21].

In 2013, Tang et al. presented a fully GPU-based simulator that processes events similarly to the YAWNS algorithm [32]. Instead of strictly limiting execution to events within the current window, parallelism is increased by identifying candidate events that may become safe to execute before a new window is calculated. When limiting the set of receivers of each simulated entity, the authors report an event rate of up to about 13 million events per second.

In 2014, Andelfinger et al. proposed a fully GPU-based conservative simulator implementation that adapts the LP size at runtime to balance parallelism and event management overheads [1]. Similarly, our proposed optimistic simulator adapts the LP size as one of the autotuned parameters.

In 2015, Swenson proposed a number of fully GPU-based simulator designs [28]. An event rate of up to 120 million events per second is achieved under the assumption that each event creates exactly one new event.

The previously discussed approaches are all based on conservative synchronization. A fully GPU-based simulator design using an optimistic synchronization approach was presented in 2013 by Li et al. [13]. The approach requires all events to be created before any events are executed. During the execution phase, all of the events are executed in parallel. Now, events are iteratively canceled and re-executed until a correct final simulation state is reached.

Contrary to some of the discussed works, our proposed simulator poses no restrictions on event creation beyond limits given by the available graphics memory. The core logic of our design is quite similar to CPU-based Time Warp systems in shared memory settings [7]. Differences arise from the fine-grained parallelism of the GPU, which enables us to maintain low list management overheads by forming small LPs of only a few simulated entities each.

2.4 Considered Simulation Models

2.4.1 PHOLD

PHOLD (parallel hold) [9] is a generalization of the sequential hold benchmark model [31] that, in addition to simulated time, also considers a configurable number of simulated entities to which events are assigned. We apply the PHOLD model as follows: a constant number of events (the *population*) is initially created and assigned to simulated entities in a round-robin fashion. The execution of an event creates a new event targeting a random entity after a random delta in simulated time. Target entities are drawn from a uniform distribution and deltas in simulated time are drawn from an exponential distribution. We configure an arbitrary lookahead of 10 by adding this value to each drawn time delta. The PHOLD model accepts three parameters: the number of simulated entities, the population size and the parameter λ of the exponential distribution. Since the performance of the simulation is strongly affected by the relation between λ and the chosen lookahead, we vary λ over multiple orders of magnitude. We refer to the simulated entities as nodes. However, the proposed simulator is not confined to simulations of networks.

Since executing an event involves only a low amount of computation, hold and PHOLD emphasize the event management performance of the simulator (e.g., [25]). As all events are handled in the same manner, there are no divergent branches across parallel event executions. Thus, event execution on the GPU must be expected to be more efficient than with models of real-world systems.

2.4.2 Peer-to-Peer Network

To show the effects of larger event complexity and more significant branching across events, we additionally evaluate the simulator using a model of networks based on Kademlia [16]. Kademlia is a protocol used to form distributed hash tables (DHTs) for key-value storage and retrieval in a peer-to-peer network and is the basis of the BitTorrent Mainline DHT, currently comprised of around 10 million nodes[2]. The value associated with a key is retrieved in a *lookup* by iteratively querying nodes on the path to the key.

After initialization, events of three types occur: 1. Creation of initial requests to immediate neighbors of the initiating node. 2. Reception of a request: on reception, the current node looks up the nearest known nodes to the desired identifier and transmits a response message to the node that initiated the lookup. 3. Reception of a response: a reception event updates a list of the closest nodes to the desired key known to the initiating node and transmits further requests to maintain a configured number of in-flight requests. Link latencies between nodes are drawn from a uniform distribution on $\{10, 11, \ldots, 200\}$. Once a response contains no closer nodes and all nodes in the list have responded, the lookup terminates and a new lookup is scheduled after a random delay drawn from a uniform distribution on $\{10, 11, \ldots, D_{\max}\}$, $D_{\max}$ being a model parameter. Smaller $D_{\max}$ increases the

[2] http://dsn.tm.kit.edu/misc_2917.php

event density in simulated time. Due to the fixed lower bound on the link latency and lookup delay distributions, we use a fixed lookahead value of 10.

Execution of an event involves memory accesses to look up routing table entries, the creation of up to 8 new events, and atomic operations to access global statistics such as the number and duration of key-value lookups.

3. SIMULATOR DESIGN

In the following, we describe the simulation loop executed by the GPU-based simulator, as well as the chosen memory layout. A number of performance-critical simulator parameters arise – the number of simulated nodes that form an LP, the degree of optimism when executing events, and the threshold of inactive LPs before synchronization is performed. These parameters are further analyzed in Section 4.

The user implements a new model by providing the following core functions: `set_model_params()` and `init_node()` initialize the model, `get_lookahead()` returns the lookahead of the model, `handle_event()` and `roll_back_event()` perform forward and backward execution of an event described by a pointer to an event structure (cf. Section 5.2). State saving is performed during event handling by calling `append_state_to_queue()` and `append_antimsg_to_queue()`.

3.1 Simulation Loop

Our goal is to map the steps of a discrete-event simulation loop – selection of events for execution, handling of events, and creation of new events – to the GPU. Since the GPU's threads have shared access to graphics memory, the simulation can act on global knowledge of the simulation state. The single-program, multiple-data paradigm of the GPU suggests a synchronous simulation process, i.e., all threads perform the same simulation step at the same time.

The simulation proceeds as a sequence of *iterations* consisting of multiple steps (cf. Figure 2), the core step being the parallel *execution* of up to one event per LP by each GPU thread. The remaining steps are required for synchronization between LPs and for event management. As a consequence of separating the execution of events from the insertion into event lists, newly created events are not visible to LPs before a new iteration begins. Thus, a challenge lies in balancing synchronization and event management overheads with parallelism: multiple executions can be performed within a single iteration, while still maintaining strict correctness and deterministic results. However, as illustrated in Figure 3, when increasing the number of executions per iteration, more and more LPs become idle, either due to a lack of further events, or due to event lists being at capacity. Due to an increase of the deviation of simulation times between LPs, in the optimistic case, the probability of rollbacks may increase as well. Once a new iteration begins, newly created events are considered for execution, increasing the opportunities to execute events in parallel.

In the following, we contrast the simulation steps per iteration for the conservative and optimistic case:

Conservative synchronization first determines the earliest timestamp $t_{\min}$ in the simulation. Events within $[t_{\min}, t_{\min} +$ lookahead) can be executed safely. This is done in one or more executions, each LP processing its earliest event, if any. During each execution, new events may be created, which are appended in an unsorted fashion to the target LP's future event lists (FEL) using atomic operations. Since events

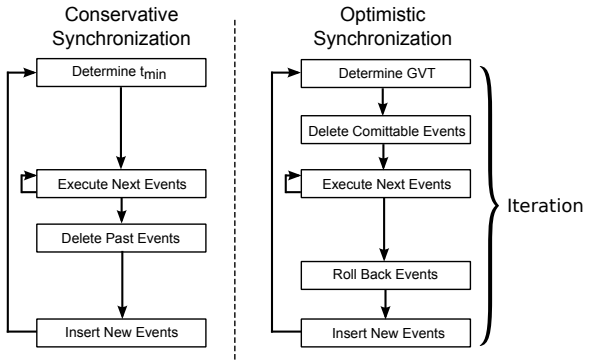


Figure 2: Comparison of the simulation steps during an iteration with conservative and optimistic synchronization.

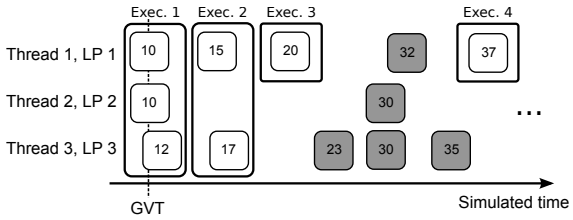


Figure 3: An example of performing multiple executions in one iteration to reduce synchronization overhead. Events grouped in a rectangle are processed in one execution. Since newly created events (gray) are not considered before the next iteration begins, the number of executions per iteration affects both the parallelism and the probability of rollbacks.

are extracted by one thread per queue, atomic operations are not required during extraction. After a number of executions, processed events are deleted from the LPs' FELs. Finally, in preparation for the next iteration, each FEL is sorted, including newly created events.

Optimistic synchronization first calculates the global virtual time (GVT) as a lower bound for future rollbacks. Entries in the event list, state list, and antimessage list with earlier timestamp than the GVT will not be required in any future rollback and are deleted. Now, as in conservative synchronization, LPs process their respective earliest event in parallel. Contrary to conservative synchronization, it is not necessary to limit the consideration of events to a certain range in simulated time. However, we will see in Section 4.1.2 that artificially limiting the optimism can increase performance. As with conservative synchronization, new events are initially appended in an unsorted fashion to the target LPs' event lists. Due to the memory requirements of storing three lists per LP and of storing uncommitted events, there is a need to handle cases where newly created events exceed the capacity of the receiving LP. In such cases, the LP that created the event rolls back its current event and is inactive for the remainder of the iteration.

During the executions within an iteration, the minimum timestamp of any newly received event of each LP is stored. After a number of executions, previously executed events with timestamps larger than or equal to this minimum are rolled back to restore the correctness of the simulation. Finally, the event lists are sorted before a new iteration begins.

To enable a fair comparison between conservative and optimistic synchronization on the GPU, we implemented both mechanisms as part of the same simulator engine, using a

shared implementation of event lists, event insertion, earliest timestamp calculation, and so forth. Both $t_{\min}$ and GVT are determined by a parallel reduction in logarithmic time. Event handling and list operations are performed by a single thread per LP. Thus, in case threads within a warp perform the same operation, e.g., executing an event of the same type or moving an event, the single-instruction, multiple-data (SIMD) architecture of the GPU is exploited.

3.2 Memory Layout

Due to the substantial cost of dynamic memory allocation on the GPU, all memory is allocated statically during initialization. For each list type (event list, antimessage list, state list), a single array in graphic memory is allocated that holds all LPs' lists. We implemented the LPs' lists as circular buffers (cf. Figure 4). For each LP, we store the offset o_{head} of the first entry, the offset o_{future} of the first entry in the simulated future, the offset o_{unsorted} of the first entry that has not yet been inserted into the sorted part of the list, and the offset o_{next} of the next new entry. The offsets separate the list into past, future, and unsorted entries.

List operations are performed by one GPU thread per list. Removal of the earliest entry from a list can be performed in constant time. Inserting a new entry requires linear time, since all entries with larger timestamps are moved one position towards the end of the list. The large cost for insertion can be tolerated in case each list contains only a relatively small number of entries, which we can strive for by using the massive core counts of the GPU to form extremely small LPs, e.g., of only 1-32 simulated nodes in the case of a network simulation. We investigate the impact of different overall numbers of list entries on the simulation performance in Section 6. An assessment of alternative list implementations is part of our future work.

The simulator design poses no limitations of the memory layout used by models. However, to achieve high performance, dynamic memory allocation should be minimized.

The number of nodes that are aggregated to form an LP (*LP size*) can be adapted by adjusting the LP boundaries in memory and merging or separating events according to the new boundaries. In Figure 5, events are represented by their timestamps and underlined in case they have already been executed. First, to create space for the merged list, each LP's events are aligned to the beginning of the LP's segment. The events in the merged list must be sorted according to whether they have already been executed, and in timestamp order. To ensure these properties, some LPs may need to perform rollbacks (cf. Appendix A).

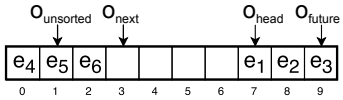


Figure 4: Example of an event list for a single LP. Stored offsets separate past, future and unsorted entries.

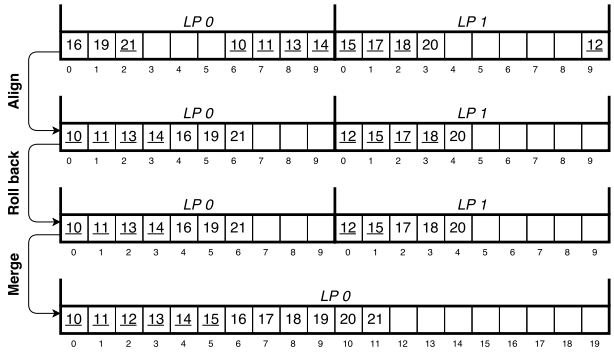


Figure 5: Steps during merging to double the LP size.

The LP size incurs a tradeoff between event insertion overhead and parallelism: a small LP size, e.g., 1 node per LP, leads to small event lists and low overhead for event insertion. At the same time, the probability increases that an LP has no safe event and thus remains idle during an execution.

While with conservative synchronization, the only required list operations are removal of the earliest event and insertion of new events, optimistic synchronization requires event deletion as well: if an LP initiates a rollback, events up to a previous point in simulated time are removed. In case events have been sent since then, antimessages are sent to the receivers of the events. Each antimessage may trigger the deletion of an event. In our implementation, such events are identified using linear search in reverse order during the rollback step and deleted in the insertion step.

4. PARAMETER ANALYSIS AND AUTOTUNING

The GPU-based simulator provides a number of tuning parameters that enable a tradeoff between parallelism and overhead for synchronization and event management. In this section, we study the impact of these parameters on the simulation performance and describe the autotuning approach used to determine a suitable configuration at runtime.

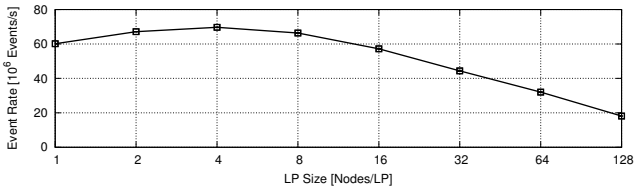
Our measurements system is equipped with 4 12-core Intel Xeon E5-2660 processors (base clock rate: 2.1 GHz, turbo clock rate: 3.3 GHz), 64 GiB of RAM and an NVIDIA GeForce GTX 980 Ti with 6 GiB of RAM and 2816 CUDA cores clocked at up to 1392 MHz. Plotted event rates are averages of 3 runs. 95% confidence intervals are plotted but frequently small enough to be nearly invisible.

4.1 Impact of Parameters

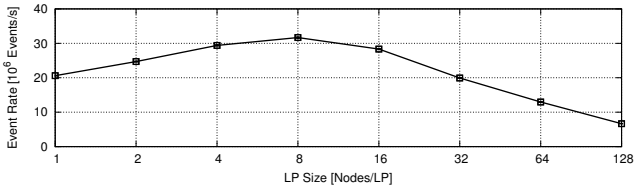
4.1.1 LP Size

On a GPU, typically a much larger number of threads is scheduled than there are physical cores, enabling the GPU's hardware scheduler to perform memory access latency hiding. This has two consequences: first, the large number of threads enables the use of small LPs, e.g., of 1-32 simulated nodes in a network simulation. Second, since the thread count is decoupled from the core count, a suitable LP size cannot be deduced directly from hardware properties.

Figure 6 shows the event rate achieved for the PHOLD model with a network size of 1 048 576 nodes and a population equal to the network size with different LP sizes and conservative synchronization. Initially, events are evenly distributed to the nodes, with timestamps according to the index in each node's queue, i.e., all zero in the cases where the population equals the network size. We can see that with small λ, the LP size has a substantial impact on performance. As discussed in Section 3.1, with lower event density in simulated time, the average number of LPs that have a safe event during an execution decreases. An increase in LP size tends to increase the number of LPs with a safe event at the cost of the increased overheads of larger FELs.



(a) $\lambda = 10^{-2}$



(b) $\lambda = 10^{-4}$

Figure 6: Event rate with different LP sizes for PHOLD using conservative synchronization.

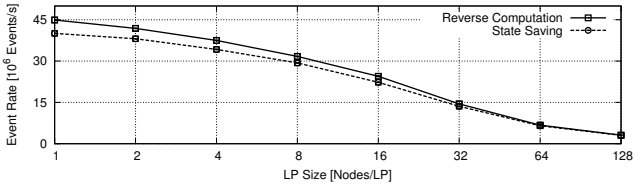


Figure 7: Event rate with different LP sizes for PHOLD with $\lambda = 10^{-2}$ using optimistic synchronization.

With optimistic synchronization, we observed that with these parameters, a one-to-one mapping of nodes to LPs achieves highest performance (cf. Figure 7 for $\lambda = 10^{-2}$). However, when varying the remaining simulator parameters, the optimal LP size varies with the model parameters as well.

4.1.2 Optimism Bound

In its purest form, optimistic synchronization allows for execution of events that lie arbitrarily far in the simulated future. However, with an increasing deviation of current simulation times between LPs, the probability of causality violations increases as well. Hence, as already proposed in early works in the field [27, 24, 30], it can be beneficial to limit the execution of events to timestamps below a certain delta from the GVT. We refer to this delta as the *optimism bound*. Since executing events within [GVT, GVT + lookahead) cannot cause causality violations, suitable optimism bounds are equal to or larger than the lookahead. Figure 8 illustrates the use of an optimism bound.

Figure 9 shows the event rates achieved with different optimism bounds for the PHOLD model with a network size of 1 048 576 nodes and a population of 1 048 576. The optimal optimism bound depends strongly on λ, being around 2 200 units of simulated time with $\lambda = 10^{-2}$ and around 12 000 units with $\lambda = 10^{-4}$. Event rates decline substantially if an exceedingly small or large optimism bound is selected.

4.1.3 Inactivity Threshold

As described in Section 3.1, the simulation proceeds in a sequence of iterations, each of which includes one or more executions of events. If only one execution is performed per iteration, the simulator may perform calculation of GVT or

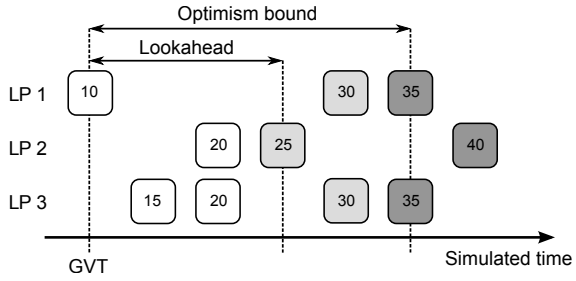
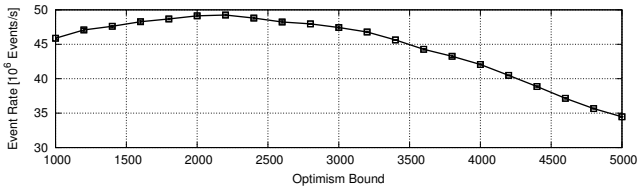
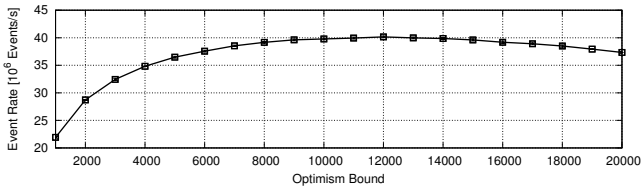


Figure 8: Bounding the optimism in Time Warp. White events cannot cause causality violations and are safe to be processed; light gray events may cause violations; dark gray events are not considered for execution yet.



(a) $\lambda = 10^{-2}$



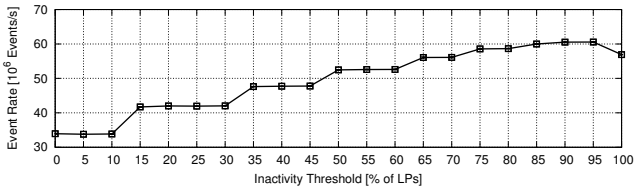
(b) $\lambda = 10^{-4}$

Figure 9: Event rate with different optimism bounds for PHOLD using optimistic synchronization with state saving.

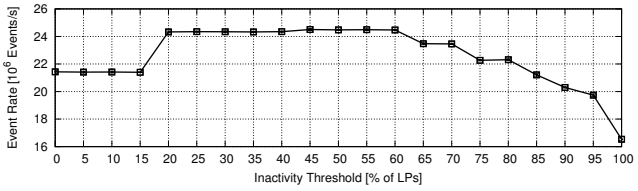
$t_{\min}$, deletion of committable events, roll backs and insertion of new events into lists more frequently than necessary, potentially incurring substantial overhead. However, with increasing numbers of executions in a single iteration, more and more LPs will become inactive due to a lack of events in the current window or within the optimism bound, or due to lists being at capacity. Figure 10 shows the event rates achieved for the PHOLD model with a network size of 1 048 576 nodes and a population of 1 048 576, varying the percentage of inactive LPs (*inactivity threshold*) at which executions for the current iteration are terminated. The LP size was set to 4 nodes per LP. Again, there is a strong dependence on the PHOLD parameter λ. With the relatively large event density in simulated time given by $\lambda = 10^{-2}$, the optimal threshold is around 95%, while with $\lambda = 10^{-4}$, values around $45 - 60\%$ are optimal. With $\lambda = 10^{-2}$, the average number of executions per iteration increases monotonously from approximately 1 to 9 for thresholds of 0% to 95% and peaks at about 18 for a threshold of 100%.

4.1.4 Block Size

The configured number of CUDA threads per block influences the allocation of the GPUs resources to the scheduled threads. The selection of a suitable block size without prior performance measurements can be nontrivial [29]. Figure 11 shows measurement results for the PHOLD model with a network size of 1 048 576, a population of 1 048 576,



(a) $\lambda = 10^{-2}$



(b) $\lambda = 10^{-4}$

Figure 10: Event rate with different inactivity thresholds for PHOLD using optimistic synchronization with state saving.

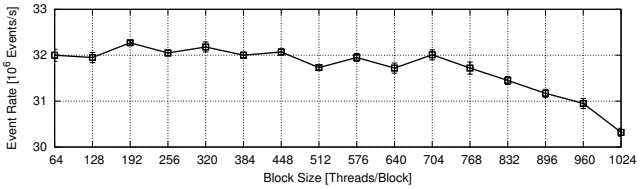


Figure 11: Event rate with different block sizes for PHOLD with $\lambda = 10^{-2}$, optimistic synchronization with state saving.

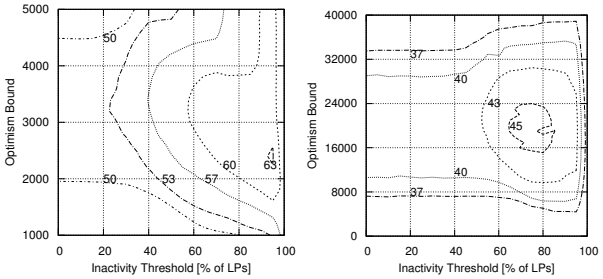
and $\lambda = 10^{-2}$. Since the event rates for block sizes between 64 and 704 did not vary substantially and since the results for other model parametrizations were similar in shape, we selected a block size of 256 threads for all our experiments.

4.2 Autotuning Procedure

We have identified the LP size, the optimism bound and the number of executions per iteration as three simulator parameters with a strong impact on performance. Optimal values for these parameters depend on the simulation model behavior and thus cannot be determined easily prior to a simulation run. The selection of suitable values is further complicated by the dependence across parameters. Figure 12 shows examples of the effect of two of the considered parameters on the event rate when executing the PHOLD model with a network size and population of 1 048 576.

To determine suitable parameter combinations for different models and model parametrizations, we perform measurements and adaptation at runtime. We apply the Nelder-Mead optimization algorithm [18], which searches heuristically for an extremum of a non-linear function without the need for calculating derivatives. The algorithm considers $(n + 1)$ points of an n-dimensional objective function and iteratively replaces the point with the worst function value with a new point with a better function value.

For each new point, the simulator parameters are adapted and the resulting simulator performance is measured. The duration of each measurement is a model-dependent tradeoff between the measurement accuracy and the cost of executing at suboptimal parameter combinations. At each parameter combination, after a warm up time of 200ms of wall-clock time, we measured event rates averaging over 300ms.



(a) $\lambda = 10^{-2}$ (b) $\lambda = 10^{-4}$

Figure 12: Contour graph of event rates for PHOLD, optimistic synchronization with state saving. The dependency between optimal parameter settings and the variation across model configurations illustrates the need for autotuning.

5. REVERSE COMPUTATION

A major concern in optimistic synchronization is the cost of rollbacks. In this section, we describe how XORWOW, the default random number generator in NVIDIA CUDA, can be reversed, i.e., how a previous generator state can be derived solely from the current state.

Although XORWOW has been shown to fail several of the statistical tests of the TESTU01 suite [12, 26], it is used as the default random number generator in CUDA. Some previous works applied reverse computation to combined linear congruential generators [6]. Reversal of the Mersenne Twister RNG [15] has also been described previously[3].

5.1 Reversal of CUDA RNG

As proposed by Marsaglia [14], a pseudo-random number generated by XORWOW is the sum of a number generated by an Xorshift generator and a number generated by a Weyl sequence. Xorshift generates the next random number by calculating the exclusive OR of bit-shifted versions of previously generated numbers according to:

$$X_n = X_{n-1} \oplus (X_{n-1} << 4) \oplus X_{n-4} \oplus (X_{n-4} >> 2) \oplus (X_{n-4} << 1) \oplus ((X_{n-4} >> 2) << 1).$$

The Weyl sequence used in XORWOW is defined by the equation

$$Y_n = Y_{n-1} + 362437 \bmod 2^{32}.$$

In CUDA, the mutable part of the RNG state for XORWOW is defined by an unsigned 32 bit integer d and an array v of 5 unsigned 32 bit integers, i.e., 24 bytes in total.

CUDA provides a function `curand_init()` to skip an arbitrary offset in the generated sequence. The number of clock cycles required to execute this function for XORWOW depends logarithmically on the offset. On an NVIDIA GeForce GTX 980 Ti, setting the offset to 2^{18} and 2^{62} required 1.12×10^7 and 1.33×10^8 clock cycles, respectively. At the graphics card's maximum clock rate of 1392 MHz, this translates to 8.1ms and 96ms. Since the target offset in the random number sequence required by rollbacks increases over the course of a simulation, `curand_init()` is too expensive to be a viable alternative to reverse computation.

Figure 13 illustrates the logical operations and state variables of XORWOW. Here, we sketch the basic idea of the reversal process on the basis of the values t and r marked in

[3]https://jazzy.id.au/2010/09/22/cracking_random_number_generators_part_3.html

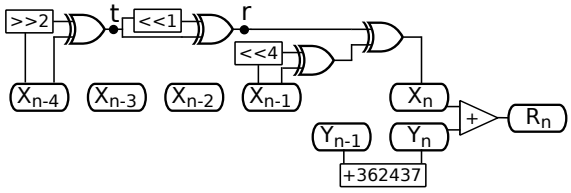


Figure 13: Logical operations and state variables of the XORWOW random number generator.

the figure. The description of the overall reversal procedure is included in the appendix. The goal is to calculate all bits t_i of t using only the bits r_i of r. To simplify the example, we treat t and r as 4 bit values. During forward generation, the relationship between r and t is $r = t \oplus (t << 1)$. On the bit level, we have:

$$\begin{array}{cccccc} & t_1 & t_2 & t_3 & t_4 \\ \oplus & t_2 & t_3 & t_4 & 0 \\ \hline & r_1 & r_2 & r_3 & r_4 \end{array}$$

During reversal, the goal is to determine t from r. First, obviously, $t_4 = r_4$. The remaining bits can be determined in an iterative fashion: $t_i = r_i \oplus t_{i+1}, i \in \{1, 2, 3\}$. The same idea is applied to reverse all other Xorshift operations of XORWOW. Since the Weyl sequence used in XORWOW involves only a simple addition, its reversal is trivial. The resulting full reversal procedure is described in Appendix B.

Figure 14 compares the time required by forward generation and reversal of 1 048 576 random numbers per thread, varying the number of parallel threads (at least 10 repetitions, 95% confidence intervals). On our test system, reversal of a random number generation is around a factor of 12 to 16 slower than forward generation, independently of the number of parallel threads. Using 1 thread, our test system performed about 5.95×10^7 generations or about 3.70×10^6 reversals per second, while with 1 048 576 threads, 3.84×10^{11} generations or 2.61×10^{10} reversals per second were achieved.

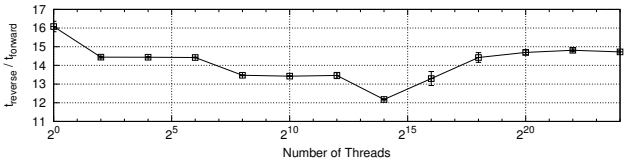


Figure 14: Comparison of time required for 1 048 576 forward random number generations ($t_{forward}$) or reversals ($t_{reverse}$) per thread using the XORWOW generator.

5.2 Memory Savings

The state of the XORWOW generator in CUDA is comprised of 24 bytes of mutable state and 24 bytes holding parameters for generating numbers following the normal or Poisson distribution. In the PHOLD model, an event requires 16 bytes of memory. Since the system state of the PHOLD model is defined solely by the RNG states of all nodes, reverse computation enables us to eliminate the state list and the antimessage list entirely. However, past events must still be stored until the GVT is updated.

In the peer-to-peer network model, a node's state is defined by its RNG state, the state of any current lookup, i.e., the number of in-flight messages and the list of closest nodes to the desired key, as well as statistics such as the number of executed lookups. Overall, each state comprises 144 bytes. Events hold data associated with requests and responses and comprise 52 bytes. With reverse computation, each state can be reduced to 76 bytes. The remaining state variables can be determined computationally, e.g., the number of executed lookups is decremented in case an event representing a final response message is rolled back.

6. PERFORMANCE EVALUATION

Our performance measurements of the simulator implementation focus on the following questions:

- Does the autotuning mechanism determine parameter combinations close to the optimum?
- To what degree does the simulation performance benefit from reverse computation?
- How does the overall performance of the optimistic simulator compare to its conservative counterpart?

The measurements were performed on an NVIDIA GeForce GTX 980 Ti of the system specified in Section 4. The host memory usage was below 100 MiB. One CPU core was fully utilized for performing CUDA API calls. The CPU usage can be reduced at some cost in performance by using interrupts instead of polling to interact with the GPU.

6.1 Parameter Autotuning

To study the effectiveness of the parameter autotuning, we compare event rates achieved using parameter autotuning to the results of optimal parameter settings. We first performed a parameter sweep to determine the combination of LP size, optimism bound, and number of executions per iteration that achieved the highest event rate. The results from these runs are compared to runs using autotuning. This process was repeated for different values of λ and for the three considered synchronization schemes during brief simulation runs spanning 10^9 events, or between approximately 10s and 30s of wall-clock time. Table 1 compares the results of the runs with the highest average event rate ("Optimal Param. Rate") with the results using autotuning when considering the highest event rate during any measurement interval of 300ms ("Adaptive Peak Rate") and the average over a simulation run of 10^9 events ("Adaptive Avg. Rate").

We can see that the autotuning process succeeds in selecting suitable parameter combinations. Due to fluctuations in event rates, the adaptive peak rate occasionally even exceeds the average rate at the optimal parameter combination. Since during the autotuning process, some time is spent in non-optimal parameter combinations, the average rate is significantly lower than the peak rate. This effect diminishes when increasing the simulation duration. Averaged over 10^9 events, autotuning attained 78.29% or more of the event rate of the optimal parameter combination.

6.2 Synchronization Approaches

The overall event rate using the different simulator variants was evaluated for the PHOLD and peer-to-peer network models in simulations spanning 5×10^9 and 1×10^9 events, respectively. Runs with very low event density in simulated time ($\lambda = 10^{-4}$ with 16 384 nodes for PHOLD and $D_{\max} = 10$min for the peer-to-peer network model) were limited to 2×10^9 and 2×10^8 events. For PHOLD, the population was chosen equal to the network size. The measurement results are shown in Figure 15. With $\lambda = 1$

	$\lambda = 1$	$\lambda = 10^{-2}$	$\lambda = 10^{-4}$
Optimal Param. Rate	93.00	85.96	35.95
Adaptive Peak Rate	90.84	82.78	34.74
Adaptive Avg. Rate	75.16	74.73	31.98
Percentage of Opt.	80.82%	86.94%	88.96%

(a) Conservative synchronization.

	$\lambda = 1$	$\lambda = 10^{-2}$	$\lambda = 10^{-4}$
Optimal Param. Rate	63.47	63.16	45.45
Adaptive Peak Rate	65.07	63.86	46.60
Adaptive Avg. Rate	55.66	56.87	37.11
Percentage of Opt.	87.69%	90.04%	81.65%

(b) Optimistic synchronization with state saving.

	$\lambda = 1$	$\lambda = 10^{-2}$	$\lambda = 10^{-4}$
Optimal Param. Rate	84.24	85.35	58.46
Adaptive Peak Rate	84.17	84.47	57.33
Adaptive Avg. Rate	70.92	68.91	45.77
Percentage of Opt.	84.19%	80.74%	78.29%

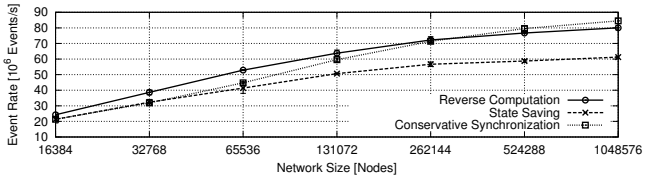
(c) Optimistic synchronization with reverse computation.

Table 1: Comparison of event rates [10^6events/s] using the optimal parameter configuration or autotuning for PHOLD with a network size and population of 1 048 576.

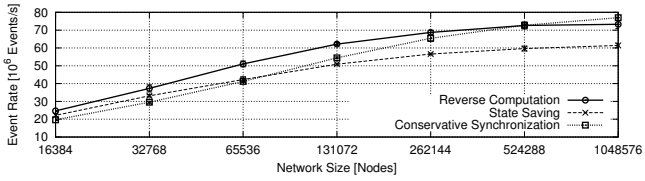
and $\lambda = 10^{-2}$, for most parameter combinations, a modest increase in event rate up to a factor of 1.3 is achieved using reverse computation compared with conservative synchronization. With network sizes of 524 288 nodes and above, conservative synchronization outperforms reverse computation. In these cases of high event density, the overhead for maintaining additional lists and performing rollbacks in optimistic synchronization outweighs any gain in parallelism. In contrast, with $\lambda = 10^{-4}$, the event density is low enough so that optimistic synchronization with reverse computation substantially outperforms conservative synchronization for all considered network sizes, by a factor of up to 3.0. Reverse computation consistently achieved higher event rates than state saving, by a factor of up to 1.3. State saving still significantly outperformed conservative synchronization for low event densities or small networks, by a factor of up to 2.9. Figure 16 shows the comparison for the peer-to-peer network model, varying the maximum delay between key-value lookups by the individual nodes. LPs were formed by merging neighboring queues in memory without consideration of the network topology. Optimistic synchronization with reverse computation consistently outperformed conservative synchronization, by a factor of up to 3.6. For nearly all parameter combinations, reverse computation achieved higher event rates than state saving, by a factor of up to 1.2. Again, state saving still consistently outperformed conservative synchronization, by a factor of up to 3.4.

As expected, due to the larger complexity of the model compared to PHOLD, the overall event rates with the peer-to-peer network model are lower. In fact, for all three synchronization approaches, the event rates achieved for $D_{\max} =$ 10min and small network sizes seem to be within the range achievable by sequential CPU-based simulators. However, optimistic synchronization reduces the event density required to efficiently execute models on the GPU. The size of the confidence intervals depends chiefly on the selection of simulator parameters through autotuning, which varies to different degrees depending on the model parametrization.

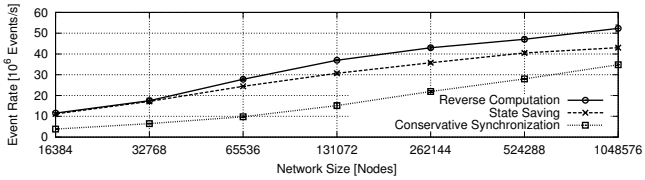
Table 2 lists an example of the runtime spent on the five core simulation steps with the different synchronization approaches for runs spanning 10^{10} events each. Conservative



(a) $\lambda = 1$



(b) $\lambda = 10^{-2}$



(c) $\lambda = 10^{-4}$

Figure 15: Overall performance comparison for PHOLD.

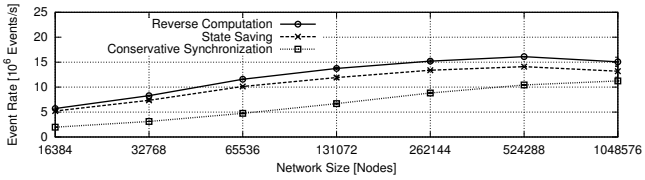
	Conserv.	State Saving	Rev.Comp.
$t_{\min}$ or GVT	2.99 ± 0.03s	2.38 ± 0.19s	2.45 ± 0.26s
Delete Events	7.18 ± 0.02s	10.08 ± 0.62s	9.96 ± 0.92s
Handle Events	84.69 ± 0.25s	116.12 ± 0.86s	76.77 ± 0.76s
Rollbacks	N/A	2.17 ± 0.37s	2.73 ± 0.64s
Insert Events	15.25 ± 0.03s	15.98 ± 0.40s	16.38 ± 0.61s
Other	2.03 ± 0.04s	1.66 ± 0.13s	1.53 ± 0.13s
Total	112.14 ± 0.33s	148.39 ± 1.56s	109.82 ± 1.53s

Table 2: Composition of runtime for PHOLD with a network size and population of 1 048 576 and $\lambda = 10^{-2}$.

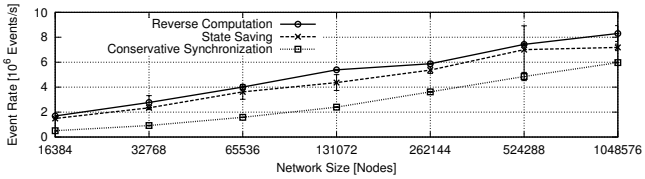
synchronization and reverse computation spend less time on deleting events than state saving, since only one list must be managed and past events can be deleted immediately. Further, handling events is substantially less expensive than with state saving, since no state information must be stored. As in most above results, reverse computation achieves the lowest runtime overall, since the cost of rollbacks is amortized through higher parallelism during event handling. The average number of events processed per execution for conservative synchronization, state saving and reverse computation was about 1.17×10^5, 1.28×10^5 and 1.69×10^5.

Figure 17 shows the effect of varying the population at a fixed network size of 131 072 with $\lambda = 10^{-2}$. The results show the tradeoff between parallelism and costs of list operations: the event rate increases with larger event density, since more events tend to be processed per execution. However, around a population of 1 048 576, the costs for list operations begin to outweigh further increases in parallelism. With the given combination of model parameters, conservative synchronization achieves higher event rates than reverse computation at populations of 1 048 576 and above.

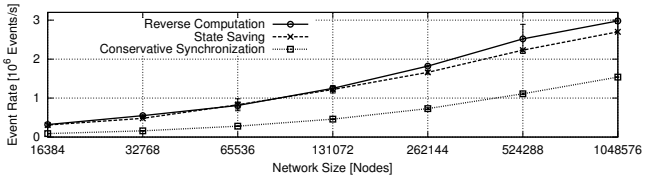
To summarize our measurements, we make two main observations: first, reverse computation outperformed state saving in nearly all cases. Second, optimistic synchronization outperformed conservative synchronization in nearly all cases. Particular benefits are seen at low event densities.



(a) $D_{\max} = 10\text{s}$



(b) $D_{\max} = 1\text{min}$



(c) $D_{\max} = 10\text{min}$

Figure 16: Overall performance comparison of the simulator variants for the peer-to-peer network model.

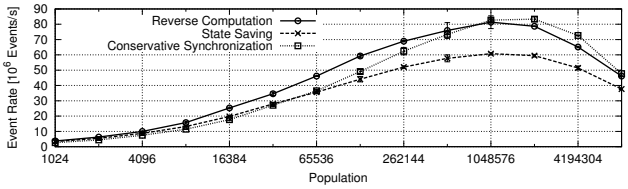


Figure 17: Event rate when varying the population for the PHOLD model with a network size of 131 072 and $\lambda = 10^{-2}$.

7. DISCUSSION

A number of avenues in the design of fully GPU-based simulators remain to be explored:

Asynchronous execution: Similarly to most previous GPU-based approaches, the LPs in our simulator execute synchronously. Zhu et al. have previously shown that an asynchronous approach is feasible in a conservative context when exploiting properties of logic simulation models [33]. Since synchronous simulations may frequently force LPs to wait for other LPs to progress, an asynchronous approach may unlock further parallelism in the optimistic case as well.

Memory distribution: We assumed that each LP is assigned the same overall amount of memory for its lists. Further, the memory allocated for each of the three lists was the same across all LP. Since the list size affects blocking of LPs and thus the required frequency of GVT calculation, a dynamic adaptation of list sizes may increase performance.

List implementation: One of the focal points of existing works on fully GPU-based simulation has been the FEL or event list implementation. Here, we use simple per-LP circular buffers with constant-time removal and linear-time insertion of events. In our evaluation, we have seen the resulting significant dependence of the simulation performance on the lists' loads. Insertion overhead and parallelism is balanced by dynamically resizing LPs. We observed a substantial decrease in performance when increasing the combined size for the FELs or event lists beyond around 2 GiB, e.g., from an event rate of about 3.2×10^7 events per second at 2 GiB to fewer than 2.0×10^7 events per second at 3.2 GiB (cf. Figure 18). The substantial drop-off in performance beyond specific overall allocation sizes given highly scattered memory accesses seems to be due to details of the translation lookaside buffer implementation of current NVIDIA cards[4]. Potentially, an alternative list implementation may reduce this effect. Our future work includes a systematic comparison of FEL or event list implementations on the GPU.

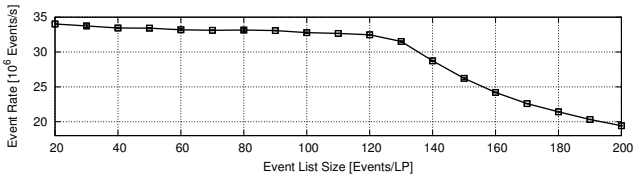


Figure 18: PHOLD event rate with different per-LP event list allocation sizes with a network size and population of 1 048 576, $\lambda = 10^{-2}$, 1 node per LP using state saving. Event rates decline strongly when allocating space for more than 128 events per LP, or 2 GiB of memory for all event lists.

Inter-GPU communication: Since our focus is on models associated with fine-grained events, we minimize data transfers between host and graphics memory by executing the simulation on a single GPU, achieving PHOLD event rates of up to 8.14×10^7 events per second. The highest PHOLD event rates events rates we are aware of have been reported by Barnes et al. [3]. For similar PHOLD parametrizations to our experiments, event rates were between one and two magnitudes larger than those reported by us. However, these results were achieved on a system with 32 768 CPU cores, in contrast to the single GPU used in our experiments. For larger PHOLD instances and when parametrizing PHOLD so that communication is reduced, up to 5.04×10^{11} events per second were reported by Barnes et al. on a system using about 2 million CPU cores. To enable GPU-based simulations at larger scale, future work could explore whether some of the benefits of fully GPU-based simulation can be maintained while applying multiple GPUs using direct inter-GPU communication and synchronization.

Real-world models: Finally, a general open issue in fully GPU-based simulation is the viability of the approach to execute models of the complexity found in widely used CPU-based simulators. Although the considered peer-to-peer network model goes beyond the simplistic assumptions in the PHOLD model, many realistic models may be comprised of events whose costs due to branching and memory accesses dwarf the gains through parallelization in the GPU context. Hence, feasibility studies are required before more comprehensive porting efforts are undertaken.

8. CONCLUSION

The proposed fully GPU-based implementation of Time Warp achieves average event rates of up to 81.4 million events per second on a commodity GPU. Our evaluation shows that the optimistic synchronization substantially outperforms conservative synchronization in nearly all consid-

[4]https://devtalk.nvidia.com/default/topic/878455/cuda-programming-and-performance/gtx750ti-and-buffers-gt-1gb-on-win7/1

ered scenarios, by a maximum factor of 3.6. In particular, optimistic synchronization improves the viability of GPU-based simulation for models with comparatively low event density. Due to a reduction in costly accesses to graphics memory, reverse computation outperformed rollbacks based on state saving by a factor of up to 1.3. Further, autotuning at runtime successfully approximates the performance achieved with optimal simulator parameter combinations.

We consider evaluations of the suitability of different realistic model types for execution using a fully GPU-based simulator the most pressing issue for future work. We hope that the publicly available simulator code will be used by the community to further explore the possibilities and limitations of fully GPU-based discrete-event simulation.

9. REFERENCES

[1] 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.*, pages 3471–3482. IEEE, 2014.

[2] P. Andelfinger, J. Mittag, and H. Hartenstein. GPU-Based Architectures and Their Benefit for Accurate and Efficient Wireless Network Simulations. In *Proc. of the Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 421–424. IEEE, 2011.

[3] P. D. Barnes, Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp Speed: Executing Time Warp on 1,966,080 Cores. In *Conf. on Principles of Advanced Discrete Simulation*, pages 327–336. ACM, 2013.

[4] B. Ben Romdhanne. *Large-Scale Network Simulation over Heterogeneous Computing Architecture*. Dissertation, EURECOM, 2013.

[5] A. Biswas and R. Fujimoto. Profiling Energy Consumption in Distributed Simulations. In *Proceedings of the Conf. on Principles of Advanced Discrete Simulation*, pages 201–209. ACM, 2016.

[6] C. Carothers, K. Perumalla, and R. Fujimoto. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation*, pages 224–253, 1999.

[7] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a Time Warp System for Shared Memory Multiprocessors. In *Proc. of the Winter Simulation Conference*, pages 1332–1339. Society for Computer Simulation International, 1994.

[8] R. Fujimoto. Parallel and Distributed Simulation. In *Proceedings of the Winter Simulation Conference*, pages 45–59. IEEE Press, 2015.

[9] R. M. Fujimoto. Performance Measurements of Distributed Simulation Strategies. Technical report, DTIC Document, 1987.

[10] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Lang. and Systems*, 7(3):404–425, 1985.

[11] 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*, pages 23–32. IEEE Computer Society, 2012.

[12] P. L'Ecuyer and R. Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Trans. on Math. Software*, 33(4):22, 2007.

[13] X. Li, W. Cai, and S. J. Turner. GPU Accelerated Three-Stage Execution Model for Event-Parallel Simulation. In *Conf. on Principles of Advanced Discrete Simulation*, pages 57–66. ACM, 2013.

[14] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[15] M. Matsumoto and T. Nishimura. Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[16] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. *Peer-to-Peer Systems*, pages 53–65, 2002.

[17] M. Nanjundappa, A. Kaushik, H. D. Patel, and S. K. Shukla. Accelerating SystemC Simulations Using GPUs. In *International High Level Design Validation and Test Workshop*, pages 132–139. IEEE, 2012.

[18] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 1965.

[19] D. M. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *Journal of the ACM*, 40(2):304–333, 1993.

[20] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. Version 7.0, NVIDIA Corporation, 2015.

[21] H. Park and P. A. Fishwick. A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation. *Simulation*, 86(10):613–628, 2010.

[22] K. S. Perumalla. Discrete-Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, pages 74–81. IEEE Computer Society, 2006.

[23] S. Raghav, M. Ruggiero, A. Marongiu, C. Pinto, D. Atienza, and L. Benini. GPU Acceleration for Simulating Massively Parallel Many-Core Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1336–1349, 2015.

[24] P. L. Reiher, F. Wieland, and D. Jefferson. Limitation of Optimism in the Time Warp Operating System. In *Proceedings of the Winter Simulation Conference*, pages 765–770. ACM, 1989.

[25] R. Rönngren and R. Ayani. A Comparative Study of Parallel and Sequential Priority Queue Algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[26] M. Saito and M. Matsumoto. A Deviation of CURAND: Standard Pseudorandom Number Generator in CUDA for GPGPU. In *Proceedings of 10th Int'l Conf. on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, 2012.

[27] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. In *SCS Multiconference on Distributed Simulation*, pages 34–44, 1988.

[28] B. P. Swenson. *Techniques to Improve the Performance of Large-Scale Discrete-Event*

Simulation. Dissertation, Georgia Institute of Technology, 2015.

[29] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. uBench: Exposing the Impact of CUDA Block Geometry in Terms of Performance. *Journal of Supercomputing*, 65(3):1150–1163, 2013.

[30] S. J. Turner and M. Q. Xu. *Performance Evaluation of the Bounded Time Warp Algorithm*. University of Exeter, Department of Computer Science, 1990.

[31] J. G. Vaucher and P. Duval. A Comparison of Simulation Event List Algorithms. *Communications of the ACM*, 18(4):223–230, 1975.

[32] 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*, pages 367–372. ACM, 2013.

[33] Y. Zhu, B. Wang, and Y. Deng. Massively Parallel Logic Simulation with GPUs. *ACM Trans. on Design Automation of Electronic Systems*, 16(3):29, 2011.

APPENDIX

A. ROLLBACKS WHEN MERGING LPS

Figure 19 illustrates a situation where merging without a prior rollback leads to an incorrect result: since the earliest future event of LP 0 has a timestamp of 16 and LP 1 has already executed an event at timestamp 18, two past events with timestamps 17 and 18 are placed before the future event at timestamp 16. Such violations can be avoided by a rollback that ensures that no LP has past events with timestamps greater than or equal to the next future event of the respective other LP. The above describes the minimum requirement for rollbacks before merging. In our implementation, since the LP size is only changed infrequently, all LPs are simply rolled back to the GVT before merging. Finally, two lists are merged by iteratively selecting the event with the earliest timestamp and lowest originating LP from the two lists. Since all past events in the event lists are rolled back, the state lists and antimessage lists are empty and can be merged simply by adjusting their offsets in memory.

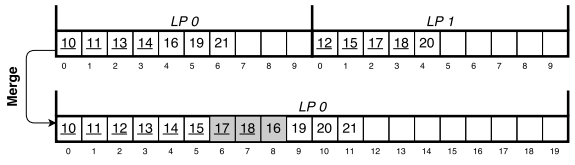


Figure 19: Invalid ordering of list entries when omitting rollback before merging.

B. REVERSAL OF XORWOW IN CUDA

Listing 1 shows the forward random number generator code for XORWOW in CUDA 7.5.

Reversing d is straightforward: d is simply decremented by the constant 362437. Reversal of $v[1]$ through $v[4]$ is achieved by assigning the current value with the next smaller index, i.e., $v'[i] = v[i-1]$ with $i \in \{1,2,3,4\}$. To reverse $v[0]$, we start with the last occurrence of t in Listing 1:

$$v[4] = v'[4] \oplus (v'[4] << 4) \oplus t \oplus (t << 1)$$

where $v'[4]$ denotes the old value of $v[4]$ and $\oplus$ denotes the XOR operation. Since the old value of $v[4]$ is the new value

Listing 1: Implementation of XORWOW in the NVIDIA CUDA Toolkit 7.5 (`curand_kernel.h`).

```
unsigned int
curand(curandStateXORWOW_t *state) {
    unsigned int t = (state->v[0] ^
                      (state->v[0] >> 2));
    state->v[0] = state->v[1];
    state->v[1] = state->v[2];
    state->v[2] = state->v[3];
    state->v[3] = state->v[4];
    state->v[4] = (state->v[4] ^
                   (state->v[4] << 4)) ^
                   (t ^ (t << 1));
    state->d += 362437;
    return state->v[4] + state->d; }
```

of $v[3]$, we replace $v'[4]$ with $v[3]$:

$$v[4] = v[3] \oplus (v[3] << 4) \oplus t \oplus (t << 1)$$

Thus, we have:

$$t \oplus (t << 1) = v[4] \oplus v[3] \oplus (v[3] << 4)$$

We define $r := t \oplus (t << 1)$ and let r_i denote the i-th bit of r and let t_i denote the i-th bit of t. Then, we have:

$$r_i = t_i \oplus t_{i+1}, i \in \{1, 2, \ldots, 31\}$$

$$r_{32} = t_{32} \oplus 0 = t_{32}$$

The bits of t can be calculated as follows:

$$\begin{aligned} t_{32} &= r_{32} \\ t_{31} &= r_{31} \oplus t_{32} = r_{31} \oplus r_{32} \\ t_{30} &= r_{30} \oplus t_{31} = r_{30} \oplus r_{31} \oplus r_{32} \\ &\ldots \\ t_1 &= r_1 \oplus t_2 = r_1 \oplus r_2 \oplus r_3 \oplus \ldots \oplus r_{32} \end{aligned}$$

Let v_i denote the i-th bit of v'[0], the old value of v[0]. We have:

$$\begin{aligned} t_1 &= v_1 \oplus 0 = v_1; t_2 = v_2 \oplus 0 = v_2 \\ t_i &= v_i \oplus v_{i-2}; i \in \{3, 4, \ldots, 32\} \end{aligned}$$

Finally, the bits of $v'[0]$ can be calculated as follows:

$$\begin{aligned} v_1 &= t_1; v_2 = t_2 \\ v_3 &= v_1 \oplus t_3 = t_1 \oplus t_3 \\ v_4 &= v_2 \oplus t_4 = t_2 \oplus t_4 \\ v_5 &= v_3 \oplus t_5 = t_1 \oplus t_3 \oplus t_5 \\ v_6 &= v_4 \oplus t_6 = t_2 \oplus t_4 \oplus t_6 \\ &\ldots \\ v_{31} &= v_{29} \oplus t_{31} = t_1 \oplus t_3 \oplus t_5 \oplus \ldots \oplus t_{31} \\ v_{32} &= v_{30} \oplus t_{32} = t_2 \oplus t_4 \oplus t_6 \oplus \ldots \oplus t_{32} \end{aligned}$$

Listing 2 shows our CUDA implementation of the reversal.

Listing 2: CUDA implementation of the reversal function for XORWOW.

```
__device__ void
curand_reverse(curandStateXORWOW_t *state) {
    unsigned int r = state->v[4] ^
      state->v[3] ^ (state->v[3] << 4);
    unsigned int t = 0;
    for (int i = 0; i < 32; i++) {
            t = t ^ r; r = r << 1;
    }
    unsigned int v0 = 0;
    for (int i = 0; i < 32; i += 2) {
            v0 = v0 ^ t; t = t >> 2;
    }
    state->v[4] = state->v[3];
    state->v[3] = state->v[2];
    state->v[2] = state->v[1];
    state->v[1] = state->v[0];
    state->v[0] = v0;
    state->d -= 362437; }
```