Proceedings of the 2014 Winter Simulation Conference
A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, eds.

EXPLOITING THE PARALLELISM OF LARGE-SCALE APPLICATION-LAYER NETWORKS BY ADAPTIVE GPU-BASED SIMULATION

Philipp Andelfinger
Hannes Hartenstein

Steinbuch Centre for Computing and Institute of Telematics
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany

ABSTRACT

We present a GPU-based simulator engine that performs all steps of large-scale network simulations on a commodity many-core GPU. Overhead is reduced by avoiding unnecessary data transfers between graphics memory and main memory. On the example of a widely deployed peer-to-peer network, we analyze the parallelism in large-scale application-layer networks, which suggests the use of thousands of concurrent processor cores for simulation. The proposed simulator employs the vast number of parallel cores in modern GPUs to exploit the identified parallelism and enables substantial simulation speedup. The simulator adapts its configuration at runtime in order to balance parallelism and overheads to achieve high performance for a given network model and scenario. A performance evaluation for simulations of networks comprising up to one million peers demonstrates a speedup of up to 19.5 compared with an efficient sequential implementation and shows the effectiveness of the runtime adaptation to different network conditions.

1 INTRODUCTION

When designing and tuning protocols to be deployed in large-scale networks, simulations can be applied to predict the runtime behavior of the network. However, the runtime for large-scale simulations can be prohibitively high. To counter this problem, parallel and distributed simulation divides the simulated network into a number of logical processes (LPs), each simulated by an individual processor or core. The runtime reductions achieved using parallel and distributed simulation vary depending on properties of the modeled network as well as on the simulation hardware. If the overhead incurred by communication and synchronization between LPs is low, large performance gains can be expected.

Large real-world networks are highly parallel systems that work in real-time. Thus, it should be possible to simulate these systems efficiently by imitating their inherent parallelism. However, the reported success in parallel and distributed simulation of large-scale networks varies immensely. In some cases, a large speedup compared with a sequential simulation was achieved (Park, Fujimoto, and Perumalla 2004), while in other cases there were modest or no performance gains (Dinh, Lees, Theodoropoulos, and Minson 2008, Quinson, Rosa, and Thiery 2012). The low-latency interconnects of modern symmetric multiprocessing (SMP) machines make it possible to achieve performance gains even for models considered as benefitting little from parallelization if a sufficiently large number of processor cores can be allocated. However, in addition to the costs incurred by the energy consumption of large processor counts, SMP systems are typically in shared use within institutions and are accessed through a queuing system. Hence, the use of SMP systems can be impractical if low turnaround times for individual simulation results are required.

In this paper, we present a GPU-based simulator that enables low turnaround times ("high-productivity computing") for simulations of large-scale application-layer networks by exploiting the large number of parallel cores in modern commodity GPUs. By executing all steps of the simulation on the GPU and hence avoiding data transfers between graphics card memory and main memory, the costs for communication and synchronization are reduced. To achieve high performance under various network scenarios, the simulator adapts the number of nodes simulated in each LP based on performance measurements executed at runtime. Hence, it is possible to investigate the behavior of an envisioned large-scale network protocol deployment on commodity hardware efficiently with respect to both simulation runtime and energy consumption.

Our main contributions are the following.

1. Analysis of the parallelism in a large-scale peer-to-peer network: we investigate the immense parallelism of large-scale application-layer networks on the example of the Kademlia protocol, which forms the basis of one of the largest real-world peer-to-peer networks comprising millions of peers. Models of peer-to-peer networks typically exhibit characteristics impeding straightforward parallelization: first, there is a lack of clear subnetwork boundaries to be exploited for minimization of communication between LPs. Second, the high level of abstraction commonly applied in the modeling of peer-to-peer networks leads to low computational costs per simulated message, increasing the impact of simulation overheads. Still, our analysis shows that on average, more than 7000 independent messages are processed concurrently in a network of one million peers, suggesting the use of large numbers of processor cores for simulation.

2. Adaptive GPU-based network simulator: we present a fully GPU-based simulator that is able to adapt to the conditions in a simulated network at runtime. Only small adjustments are required to port a CPU-based sequential network model to the GPU-based simulator. Detailed measurements expose the performance properties of the simulator and show significant runtime reductions up to a factor of 19.5.

The remainder of the paper is structured as follows. In Section 2, we introduce fundamental concepts and discuss related work in the field of GPU-based simulation. In Section 3, we analyze the parallelism in the evaluation network model. In Section 4, we describe the design and implementation of the proposed GPU-based simulator. In Section 5, we evaluate the performance of the simulator and discuss remaining challenges. Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

In this section, we introduce parallel and distributed network simulation and the issue of synchronization between processors. We introduce the use of GPUs for general-purpose computations. Finally, we discuss existing work in the field of GPU-based network simulation.

2.1 Parallel Discrete-Event Network Simulations

In parallel discrete-event network simulations, the simulated network is partitioned into a number of *logical processes* (LPs) so that a number of interacting processors share the computational load. For each LP, a *future event list* (FEL) holds the events to be executed in timestamp order. Synchronization is required to maintain the temporal relationships between events occurring in separate LPs. Commonly, synchronization is achieved by executing only *safe* events that at the given point in simulated time can be guaranteed not to trigger a future violation of timestamp order. The *lookahead* denotes the delta in simulated time up to which events are safe according to properties of the simulated network.

A well-known round-based synchronization algorithm is YAWNS (Nicol 1993): each round begins by determining the minimum timestamp t_{min} among all events in the simulation. Given a lookahead value determined according to properties of the simulation model, the window of safe events is given by $[t_{min}, t_{min} + lookahead]$, in the following referred to as the *lookahead window*. The proposed GPU-based simulator applies the YAWNS algorithm to the many-core domain.

2.2 General Purpose Computation on Graphics Processing Units

General purpose computation on graphics processing units (GPGPU) is a programming paradigm enabling the utilization of the massively parallel hardware of modern graphics cards originally optimized for data-parallel problems in the graphics domain for general computational tasks. The simulator presented in this paper was implemented based on the GPGPU framework NVIDIA CUDA. Hence, our brief introduction to GPU architecture will follow NVIDIA's terminology (NVIDIA Corporation 2013).

A CUDA hardware device contains a number of streaming multiprocessors (SMs), each comprising a number of CUDA cores. Computational tasks are organized in thread blocks that are assigned to SMs by a hardware scheduler. Threads are executed in groups of 32 called warps that execute in lockstep. If a sufficient number of warps are to be executed, the hardware scheduler hides memory access latencies by exchanging active warps in case of memory accesses. GPU functions, so-called kernels, are executed using API calls from the CPU context. Kernel input and output data is transferred over the PCI-E bus. As the data transfer bandwidth of the PCI-E bus is significantly lower than the bandwidth between the GPU and graphics memory, frequent data transfers can limit the performance of CUDA programs. Additional overhead is incurred for the exchange of the execution control between the GPU and the CPU.

CUDA hardware is classified by its compute capability (CC), a version number indicating a device's feature set. To allow for interaction between computations of different threads, barrier primitives synchronize memory accesses between threads of *the same block*. Devices starting with CC 3.5 additionally support memory access synchronization between threads of *multiple blocks* through so-called dynamic parallelism. Devices prior to CC 3.5 support only **API-based** inter-block synchronization: when returning the control flow from the GPU to the CPU, all previous writes to graphics memory are guaranteed to be visible to all threads during future kernel executions. Hence, a need for frequent synchronization of memory accesses is reflected by repeated control flow exchanges between the GPU and the CPU. Xiao et al. presented a method enabling **software-based** inter-block synchronization from GPU code independently of dynamic parallelism (Xiao and Feng 2010). When calling a new barrier function, a global variable is incremented atomically by each block until all blocks wait at the barrier. Then, the barrier function terminates and the threads of all blocks can access any data written to memory prior to the barrier call. To avoid deadlocks, only as many thread blocks as there are SMs can be scheduled with this method, allowing for up to $\#SMs * 1024$ threads with CC 2.0 and above. Without software-based synchronization, it is possible to schedule up to 65535^3 blocks for CC 2.0, and up to $(2^{31}-1)^3$ blocks for devices with CC 3.0. We evaluate the proposed simulator for both the API-based and the software-based synchronization method.

2.3 GPU-Based Network Simulation

Previous work differs from our proposed simulator in at least one of the following ways: first, all of the previous works assume a fixed LP size for each simulation run. As we will show, adaptation of LP size at runtime is critical to achieve high simulation performance when varying simulation scenarios. Second, in most works, only some of the tasks involved in the simulation are performed on the GPU, requiring repeated data transfers between the GPU and CPU contexts. Finally, some of the works exhibit significant limitations in their generic applicability to arbitrary system models.

In 2006, Perumalla (Perumalla 2006) proposed alternatives for GPU-based discrete-event simulations improving on a time-stepped execution method. While the proposed approach is shown to achieve high speedup for a diffusion model, limitations in the GPU hardware available at the time restrict the parallelism to events with identical timestamps, resulting in limited expected performance gains for arbitrary models.

In 2010, Park et al. (Park and Fishwick 2010) proposed a framework for purely GPU-based discrete-event simulations, achieving a speedup close to 10. All events within a fixed amount of simulated time are considered to occur simultaneously and are considered safe to be executed in parallel, thus introducing a numerical error. Although the authors give analytical upper bounds for the introduced error, not all simulation studies can tolerate the effects of the numerical error resulting from their method.

In 2011, Andelfinger et al. studied architectures enabling the use of GPUs to accelerate simulations of wireless networks (Andelfinger, Mittag, and Hartenstein 2011). They proposed a hybrid CPU-GPU architecture that executes events containing large amounts of data parallelism on a GPU, while performing event scheduling tasks on a CPU. The proposed architecture exploits both data parallelism within single events as well as parallelism exposed by handling multiple events concurrently. However, the architecture is not suited to exploit the fine-grained parallelism between large numbers of inherently sequential events.

In 2012, Kunz et al. (Kunz, Schemmel, Gross, and Wehrle 2012) presented a hybrid CPU-GPU simulator to accelerate parameter studies. The authors propose aggregating events both within and across simulation runs at the same time. A CPU-based event scheduler handles the choice of independent events to be processed by the GPU in each upcoming simulation round. In our work, by performing all simulation steps on the GPU, data transfers are reduced to the transfer of initial scenario parameters to the GPU memory at the start of the simulation and the transfer of simulation results at the end of the simulation.

In 2013, Li et al. (Li, Cai, and Turner 2013) proposed a three-stage execution model for GPU-based simulation. Events are executed optimistically and cancelled if causality violations are detected. The approach is applicable only to models where event generation can be performed prior to event execution and where no additional events need to be created during the simulation itself.

In the same year, Wenjie et al. (Wenjie, Yiping, and Feng 2013) proposed a synchronous conservative time management algorithm for GPU-based simulation. After determining the lookahead window, safe events are executed concurrently in multiple batches. Between batches, an attempt is made to exploit the changed system state to mark some of the events outside the current lookahead window as safe, improving simulation runtime by up to 30%. In our measurements, calculating a new lookahead window has proven to be inexpensive compared with the other simulation steps. Therefore, in contrast to Wenjie et al., we calculate a new lookahead window as soon as the remaining number of events in the lookahead window is insufficient to efficiently utilize the GPU's resources.

3 CASE STUDY: PARALLELISM IN KADEMLIA-BASED NETWORKS

In this section, we analyze the inherent parallelism of large-scale networks on the example of the Kademlia protocol (Maymounkov and Mazières 2002) used to form a distributed hash table (DHT) for efficient lookup of key-value pairs in an application-layer overlay network. Lookups of key-value pairs are performed by iteratively querying peers with decreasing distance to the peers storing the value associated with the desired key. A protocol parameter determines the number of queries to be performed concurrently during the lookup. Kademlia forms the basis of the BitTorrent Mainline DHT, one of the largest peer-to-peer networks in existence, currently comprising 6 to 10 million peers (Jünemann, Andelfinger, and Hartenstein 2011). The DHT is used by the main BitTorrent network (Carothers, LaFortune, Smith, and Gilder 2006) to locate nodes downloading specific files. Intuitively, parallelism in the DHT results from lookups performed by different peers at the same time and from individual queries performed concurrently during each lookup.

We quantify the parallelism contained in the network model using critical path analysis (Berry and Jefferson 1985). Based on event traces created during sequential simulation runs, event dependency graphs as shown in Figure 1 are created. Dependencies model precedence requirements to enforce the temporal relationships between events. For our purposes, there are two conditions under which an event depends on another: either, one event is created by the other, or one event pertains to the same LP as the other and is its immediate successor in simulated time. Events are ready to be processed once all their dependencies have already been executed. In the figure, dashed ellipses indicate groups of events that become ready to be executed at the same time and can hence be executed in parallel. Figure 1(a) shows the dependency structure on the peer level, corresponding to the full parallelism in the network model. In Figure 1(b), peers A and B are combined in one LP, which is reflected by reduced parallelism. We determine an upper bound for the parallelism of the network model in a round-based fashion: in each round, from the set of all events in the trace, we select those events that have no dependencies on any other event. In contrast to traditional critical path analysis, in each round, we only consider events within the lookahead window. The selected events can be executed safely: first, the creating event has already been processed and second, there is no event with lower timestamp to be executed in the same LP. Dependencies of other events on the selected events are removed and the selected events are eliminated. Under the assumption of equal processing times per event, the average number of events eliminated per round gives an upper bound for

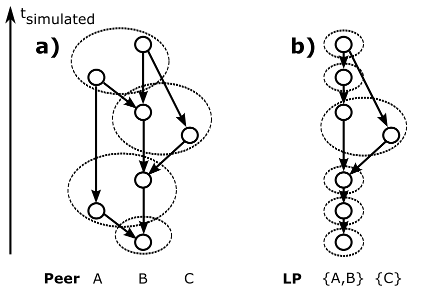
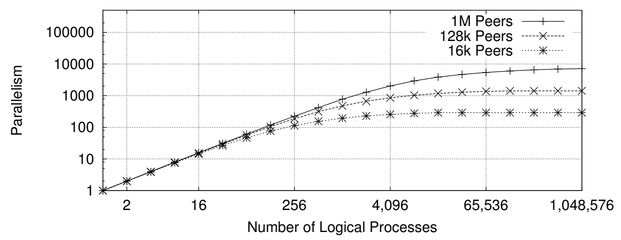
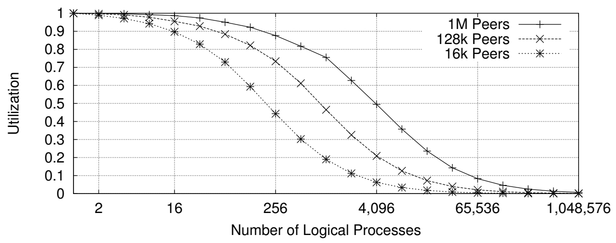


Figure 1: Example of an event dependency graph for a simulation of a network of three peers on the peer level (a), and when combining peers to form LPs (b).



(a) Critical Path Parallelism



(b) Utilization of LPs

Figure 2: Critical path parallelism and utilization of LPs in a simulation of a Kademlia-based network for different network sizes and varying the number of LPs.

the speedup achievable by parallel simulation. Of course, real-world simulation performance will typically be significantly lower due to the overheads for communication and synchronization between processors.

In Figure 2(a), we plot the parallelism p_{max} in the Kademlia network model for a fixed lookahead of $10ms$ and a random assignment of peers to LPs. We vary the number of LPs l for three network sizes: 16,384, 131,072 and 1,048,576 peers, each peer performing a lookup every 30s on average, for 10^7 events total. For up to 32 LPs, p_{max} scales almost linearly for all network sizes, p_{max} being larger than 80% of l. The largest values for p_{max} are 290.0 with 16,384 peers, 1416.8 with 131,072 peers and 7117.4 with 1,048,576 peers. Figure 2(b) shows the utilization of the available LPs given by p_{max}/l. If a utilization of more than 50% is desired, the largest achievable parallelism is 75.9 with 16,384 peers using 128 LPs, 313.1 with 131,072 peers using 512 LPs, and 1285.0 with 1,048,576 peers using 2048 LPs.

In summary, the Kademlia model contains substantial parallelism that could make use of hundreds or thousands of processor cores. In the simulator evaluation in Section 5, we will investigate how much of the identified parallelism can be translated to simulation speedup compared with a sequential implementation.

4 GPU-BASED NETWORK SIMULATOR

The main challenge in parallel simulation is the synchronization between LPs. As in the YAWNS algorithm (Nicol 1993), the proposed simulator enforces timestamp order by alternating between two tasks:

1. *Selection*: from all events remaining to be executed, select the set of *safe* events that can be executed without the possibility of causing a future violation of timestamp order.

2. *Execution*: execute the selected events, potentially creating new events.

The steps are repeated until a termination criterion, e.g, the execution of a configured number of events, is satisfied. Executing these steps on a many-core GPU is associated with a number of challenges (C1-C4):

C1. Inter-block synchronization of memory accesses is required frequently during simulation runtime. However, on the GPU, synchronization of memory accesses between thread blocks is a costly operation.

C2. Dynamic allocation of memory from the GPU context is expensive, suggesting the use of statically allocated memory regions. However, if transfers between graphics and main memory are to be avoided, the limited amount of memory available must be managed so that it can hold the shifting simulation state.

C3. Graphics memory is optimized for high throughput instead of low access latency.

C4. The number of active threads required for efficient utilization of the GPU depends both on the GPU device itself and on the code to be executed and cannot be easily determined prior to runtime.

We address C1 by comparing the performance of two different approaches to memory access synchronization in our simulator implementation. In the *purely GPU-based* variant, this is reflected by a call to the software-based synchronization method. In the *API-based* variant, a return of the control flow to the CPU and a separate kernel launch are required for synchronization. Challenge C2 is addressed by using a statically allocated memory region to hold FELs, and by adapting FEL sizes at runtime if size limits are exceeded. Challenge C3 is addressed by representing FELs using a simple data structure that does not require scattered memory accesses. To address C4, we employ performance measurements that allow the simulator to balance the number of active threads with simulation overheads at runtime. In addition, the *Selection* step is repeated if an insufficient number of safe events remain in the lookahead window.

4.1 Execution Procedure

Initially, a fixed number of simulated nodes is assigned to each LP. Initial events pertaining to the simulated nodes are created and inserted into their respective LP's FEL. Now, the simulation proceeds in a round-based fashion as shown in Algorithm 1. Simulation steps that require subsequent inter-block synchronization in every loop iteration are marked with **[S]**.

Algorithm 1 Execution procedure of the GPU-based simulator engine.

```
repeat
    determineLookaheadWindow() [S]
    repeat
        numEventsCurrentIteration ← selectSafeEvents()
        handleSafeEvents() [S]
        checkQueueOverflow()
        numEventsTotal ← numEventsTotal + numEventsCurrentIteration [S]
    until numEventsCurrentIteration < minEventsPerIteration
    insertNewEvents() [S]
until numEventsTotal ≥ finalNumEvents
```

In the following, we describe each of the steps of the execution procedure in detail.

determineLookaheadWindow(): We determine the events that are safe to be executed by finding the minimum timestamp t_{min} in any of the LPs' FELs. All events with timestamps smaller than or equal to $t_{max} = t_{min} + lookahead$ are safe, as any new event created by a safe event will have a timestamp larger than or equal to t_{max}. In the following, the interval $[t_{min}, t_{max}]$ will be referred to as the *lookahead window*.

For each LP, the event at the LP's FEL head is selected and a *parallel reduction* is performed to find the lowest timestamp of all selected events: in each iteration, a number of concurrent threads calculate the minimum of two remaining elements of input data each. This way, given a sufficient number of threads, the global minimum is determined in $log(n)$ iterations. If there are fewer threads t than there are LPs l in the simulation, the parallel reduction is repeated $\lceil l/t \rceil$ times to cover all LPs' earliest events. Our implementation of the parallel reduction is based on (Sengupta, Harris, and Garland 2008).

The following three steps address the execution of safe events and repeated until fewer than a configured number of safe events remain. Each step is repeated $\lceil l/t \rceil$ times, handling t LPs during each repetition.

selectSafeEvents(): Each thread selects an LP's earliest safe event, if any. If there is no event in an LP's FEL or the earliest event is not safe, the thread remains idle during the current repetition.

handleSafeEvents(): All threads that have selected a safe event call the event handler defined by the network model, passing the selected event as an argument. Each event has a type field and a memory

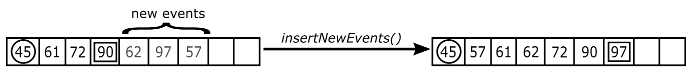


Figure 3: During event execution, newly created events are appended to the target LP' FEL. In a subsequent step, the new events are inserted into the FEL in timestamp order.

region for event data. The model behavior is specified in the event handler function, which can in turn delegate event handling of different event types to specified functions.

If new events are to be created, the event handler calls the simulator function enqueueEvent(). Any new event is appended to the target LP's FEL. In graphics card memory, FELs are represented as ring buffers located in memory regions of equal size. Figure 3 shows the insertion of new events into a single LP's FEL. The FEL head is denoted by a circle, while the tail is denoted by a square. In enqueueEvent(), new events are appended in an unsorted fashion. As multiple threads may create new events for the same LP concurrently, the target LP's FEL tail is advanced atomically before storing the new event at the new tail position, eliminating the possibility of race conditions.

checkQueueOverflow(): When simulating only small numbers of peers in each LP, the limited amount of memory available on the graphics card restricts the number of events that can be contained in a single LP's FEL. If load imbalances in the simulated network lead to an overflow of any LP's FEL, excess events are stored in a temporary buffer of fixed size shared by all LPs. The FEL overflow is resolved by doubling the number of simulated network nodes, e.g., peers, per LP and thus combining the capacities of neighboring FELs until all events fit into their respective LP's FEL (cf. Section 4.2).

insertNewEvents(): As a last step before a new lookahead window is determined, the events enqueued during the handleSafeEvents() step of the current round are inserted into FELs in timestamp order (cf. Figure 3). In each iteration, each thread handles the insertion of all new events for a single LP.

4.2 Adaptation of Logical Process Size

In the simulator configuration, there is a tradeoff regarding the number of simulated network nodes assigned to each LP. Low numbers allow the simulator to expose the parallelism in the network model (cf. Section 3), but may lead to i) many idle threads if LPs' FELs do not contain safe events in most rounds, ii) large costs for aggregation of all FELs' minimum timestamps for advancing the lookahead window. On the other hand, large numbers of nodes per LP limit the exploitable parallelism and increase the overhead for insertion of events into FELs, as the number of events in each FEL increases with larger LPs.

An optimal LP size depends on a number of factors: the dependencies between events as given by the network model, the event density in simulated time, as well as hardware characteristics such as the number of hardware threads available, the number of active threads required to exhaust the graphics card's memory bandwidth, and the costs for FEL management. Network model properties can vary during runtime and in particular cannot be easily predicted prior to a simulation run, as determining the network model's runtime behavior is typically the main goal of the simulation study itself. Hence, for high performance, the simulator should be able to adapt to the conditions of the network scenario at runtime.

LPs are resized as illustrated in Figure 4. First, each GPU thread aligns the FEL of one LP to the first element of the reserved memory area. Then, if the number of nodes per LP is to be increased, events of all LPs with index $2k+1$ are appended at the tail of LPs with index $2k$. Now, the LP count is halved and insertNewEvents() is called to insert the new events into the sorted FELs. This way, both the number of nodes assigned to each LP and the maximum number of events in each LP's FEL is doubled. If the number of nodes per LP is to be halved instead, each thread iterates over the events of one FEL, separating events into two FELs, one for a new LP with index $2k$, and one for a new LP with index $2k+1$. As timestamp order has already been established by previous simulation rounds, events can be copied to their new position in the existing order. Halving the number of nodes per LP halves the maximum size of each FEL (f_{max}) as well. If an existing FEL holds more than $f_{max}/2$ events for one of the new lists it is to be split to, an

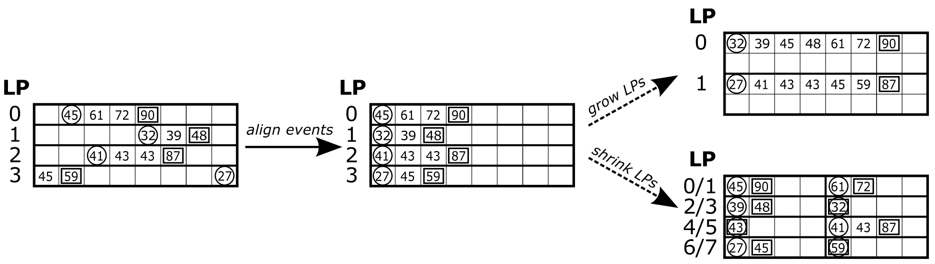


Figure 4: To resize LPs, FELs are first aligned to the start of their respective memory area boundaries and subsequently relocated according to the new boundaries.

overflow would occur. Hence, prior to a decrease in the number of nodes per LP, a check is performed to guarantee that the new FELs will not exceed the memory bounds reserved for each ring buffer.

At runtime, each time the adaptation process is triggered, LPs are resized to handle one peer each. Then, the simulator iterates over LP sizes up to a configured limit, for each LP size resuming simulation and measuring the number of events executed per second of wall-clock time. Once measurements for all configured LP sizes have been performed, LP size is adapted according to the largest measured number of events per second until the next adaptation is triggered, e.g., after a fixed number of executed events.

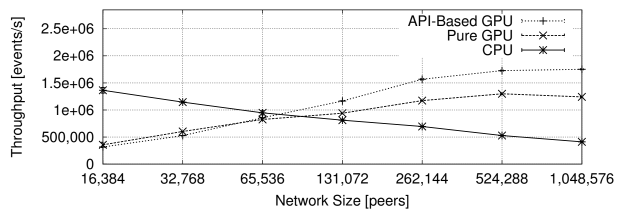
4.3 Model Implementation

The Kademlia network model used for the evaluation of the GPU-based simulation engine was developed for the reference CPU implementation first, and subsequently ported to the GPU. No efforts were made to maximize GPU utilization by exposing data parallelism or to increase memory access efficiency through reordering of data structures (NVIDIA Corporation 2013). Executing the model on the GPU required two minor modifications: first, in the sequential simulator, global variables used to gather statistics about the simulated network can be accessed directly from the event handling code. In the parallel case, multiple threads may attempt to modify global variables concurrently. We achieve consistency by replacing write accesses to global statistics variables with calls to corresponding atomic operations provided by CUDA. Second, random numbers are required to generate lookups and to determine link latencies in the simulated network. In the sequential case, random numbers are drawn from a single random number stream, leading to a deterministic simulation and identical simulation results between runs when using the same random number seed. In the parallel case, when employing a single random number stream, different random numbers will be assigned to different threads depending on timing. Hence, as the memory footprint of each random number stream is low, we create one random number stream per peer.

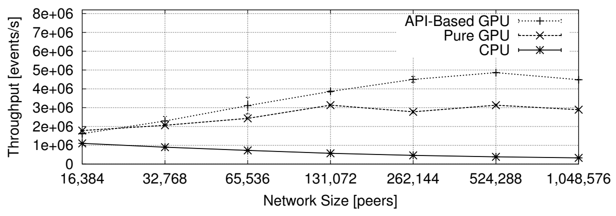
Apart from the changes for accessing global variables atomically and the separation of random number streams, the network model code is identical between the CPU and the parallel GPU-based variants.

5 SIMULATOR EVALUATION

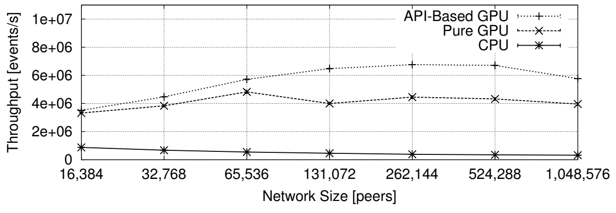
We evaluate the performance of the implementation of the proposed simulator engine with respect to simulations of Kademlia-based networks by comparing three simulator variants: a GPU-based approach using the CUDA API for memory access synchronization, a purely GPU-based approach using software-based synchronization, and an optimized CPU implementation as a baseline. As processing time per event in the evaluation network model is quite low at about 1μs or less depending on the scenario, a large portion of simulation time is spent handling the FEL in the sequential variant. Hence, a meaningful comparison requires an efficient FEL implementation. We used the *map* container class from the C++ standard library to implement the FEL, which is the default in the well-known network simulator ns-3. Our test system contains an NVIDIA GTX 660Ti graphics card with 1344 cores in 7 SMs, allowing us to assign 7168 threads to the purely GPU-based simulator variant (cf. Section 2.2). In the API-based simulator variant,



(a) Low amount of traffic ($d_{max} = 10m$).



(b) Moderate amount of traffic ($d_{max} = 1m$).



(c) Large amount of traffic ($d_{max} = 10s$).

Figure 5: Events executed per second of wall-clock time depending on the simulator variant and the amount of traffic in the simulated network, varying the number of peers in the network.

we measured highest performance for 256 threads per block for any sufficiently large number of blocks. In our experiments, we used $\lceil l_{init}/256 \rceil$ blocks, l_{init} being the initial number of LPs.

We demonstrate the efficiency of the LP size adaptation mechanism by comparing the runtimes of simulations using fixed LP sizes with simulations under the adaptation scheme. The performance plots show averages over three runs per configuration and include 95% confidence intervals.

5.1 Speedup Measurements

Figure 5 shows the average number of events executed per second of wall-clock time, denoted as *throughput*, for the three simulator variants depending on the number of peers in the simulated network. We vary the computational load in the simulation by configuring different amounts of traffic: each peer executes lookups with a delay in ms drawn from a uniform distribution on $[0, d_{max}]$ between lookups. With smaller d_{max}, the computational load of the simulation increases as more messages are generated per unit of simulated time. In Figure 5(a), with $d_{max} = 10m$, we see that with 65,536 or fewer peers, the throughput of the CPU simulator is higher than that of both GPU variants. However, with larger network sizes, the throughput of the CPU variant decreases. Conversely, as we have seen in Section 3, simulations of larger networks are associated with higher parallelism, achieving larger throughput on the GPU by better utilization of the GPU's resources. For a network of 1,048,576 peers, a simulation speedup of 4.3 was achieved by the API-based GPU variant. With d_{max} at 1m and 10s, the GPU variant performed better than the CPU variant in all scenarios. For $d_{max} = 1m$, the largest speedup was 13.5 with 1,048,576 peers. With $d_{max} = 10s$, the largest speedup was 19.5 with 524,288 peers, with a throughput of $6.71 * 10^6$ events per second.

The event density and in consequence, the amount of simulated time covered by the simulator in a given amount of wall-clock time, depends on the scenario configuration. With 1,048,576 peers and $d_{max} = 10s$, simulated time progressed at a factor of 0.31 of wall-clock time. At $d_{max} = 60s$, the network could be simulated in real time with a factor of 1.08. At $d_{max} = 10m$, about 2265s were simulated in 571s of wall-clock time, a factor of 3.97. The largest factor measured was 52.93 with 16,384 peers and $d_{max} = 10m$.

For the GPU variants, there is a decrease in event throughput for networks of 1,048,576 peers incurred by the time required for initially populating the simulated peers' routing tables. Of course, the relative impact of the initialization phase diminishes for runs covering larger periods of simulated time.

In almost all cases, the API-based GPU variant was more efficient than the purely GPU-based variant. To determine whether the software-based synchronization itself is more inefficient than API-based synchro-

Table 1: Percentage of runtime spent on simulation steps for $d_{max} = 10s$.

Network Size	*CPU*		*API-Based GPU*				*Pure GPU*			
	Handle	Other	MinTs	Handle	Insert	Other	MinTs	Handle	Insert	Other
16,384 Peers	29.4%	70.6%	7.4%	52.7%	39.7%	4.7%	2.9%	46.5%	48.1%	0.0%
131,072 Peers	27.5%	72.5%	2.3%	60.1%	36.9%	0.7%	1.4%	45.1%	53.5%	0.0%
1,048,576 Peers	40.0%	60.0%	1.5%	71.4%	26.8%	0.3%	0.3%	36.3%	61.1%	0.2%

nization, or whether the limited number of blocks allowed in the purely GPU-based variant is insufficient to effectively hide memory access latencies, we configured the same number of threads for both GPU-based simulator variants and studied the resulting throughput for all network sizes with $d_{max} = 10s$. Even though the throughput of the API-based variant dropped by up to 12.7%, the API-based variant still achieved higher throughput than the pure variant in almost all cases. Hence, we conclude that in our setup, software-based memory access synchronization on the GPU is less efficient than API-based synchronization.

Table 1 lists the percentage of runtime spent on the different simulation steps for 16,384, 131,072 and 1,048,576 peers and $d_{max} = 10s$. For the CPU-based simulator, we distinguish two steps: event handling (Handle) and overheads (Other), including, and dominated by, FEL management. For the GPU-based simulator variants, there are four steps corresponding to the execution procedure described in Section 4.1: calculation of the smallest global timestamp (MinTs), event handling (Handle), insertion of events into FELs (Insert), and overheads (Other). While the CPU-based simulator spent 29.4% of its runtime executing events with 16,384 peers, with 1,048,576 peers, this value increased to 40%. As total runtime increased from 1134s to 3141s while the number of executed events is constant, we can see that both the processing time per event as well as the FEL management overhead increased for larger networks. In the GPU-based simulator, in addition to the benefits of the large number of cores of the GPU, a larger portion of runtime was spent executing events than was the case for the CPU-based simulator. On the GPU, the results clearly show the superiority of the API-based variant: while in the purely GPU-based variant, the relative overhead for inserting events into FELs increases with larger network size, in the API-based variant, a larger portion of runtime was spent on event execution with larger network sizes. In all cases, finding the global minimum timestamp comprised only a small portion of the total runtime.

5.2 Optimal Logical Process Size

When assigning only a single peer to each LP, the parallelism of the Kademlia network is fully exploited so that hundreds or thousands of events can be executed in each round (cf. Section 3). However, overheads due to idle GPU cores and for event selection increase with larger LP counts. To show that the proposed simulator successfully balances parallelism and overhead at runtime, Table 2 compares the throughput of simulation runs with fixed LP size to runs using adaptive LP size. In each run, the LP size was adapted a single time after initialization of the simulated network. The optimal fixed number of peers per LP varied between 2 and 16. In general, the lower the traffic in the simulated network and the fewer events there are per unit of simulated time, the more peers need to be aggregated in each LP to achieve best performance. In almost all cases, the adaptive simulator implementation was able to select an efficient LP size and hence closely approximated the largest throughput among the runs with fixed LP size. With 16,384 peers and $d_{max} = 1m$, the adaptive simulator even slightly outperformed the best fixed-LP run. With 1,048,576 peers and $d_{max} = 10m$, however, due to high variance of runtime performance, the chosen LP size achieved only 84.3% of the highest-throughput run. When increasing the duration of each performance measurement from 10^5 to 10^6 events, 97.2% of the highest throughput was achieved. In the simulation run with the largest throughput of $6.7 * 10^6$ events per second, due to the large traffic configured using $d_{max} = 10s$, even more events can be executed in each round than in the configuration used in the critical path analysis of Section 3. On average, about 15,600 events were executed per simulation round, while achieving an overall speedup of 19.5. We expect to reduce the remaining gap between the immense parallelism in the network model and the resulting simulation speedup by the optimizations to the simulator discussed in Section 5.3.

Table 2: Comparing simulator throughput [$10^6 events/s$] using fixed-sized and adaptive LPs.

Network Size	16,384 Peers			131,072 Peers			1,048,576 Peers		
d_{max}	*10s*	*1m*	*10m*	*10s*	*1m*	*10m*	*10s*	*1m*	*10m*
2 Peers per LP	3.82	1.58	0.30	6.78	3.31	0.74	5.90	3.27	0.77
4 Peers per LP	3.51	1.61	0.32	6.49	3.98	1.02	5.79	4.18	1.25
8 Peers per LP	2.22	1.29	0.32	4.39	3.87	1.18	4.49	4.50	1.78
16 Peers per LP	1.09	0.81	0.26	2.34	2.59	1.18	2.86	3.50	2.08
Adaptive LP Size	***3.51***	***1.62***	***0.32***	***6.48***	***3.86***	***1.17***	***5.77***	***4.49***	***1.75***
Percentage of Best	**91.8%**	**100.3%**	**99.4%**	**95.5%**	**96.9%**	**99.0%**	**97.7%**	**99.7%**	**84.3%**

5.3 Discussion

While the proposed simulator exposes a large portion of the parallelism of the network model, a number of challenges remain to be addressed: *first*, per-LP FELs are represented as ring buffers, incurring time complexity in O(n) and large memory access overhead when inserting events into FELs, whereas access to the FEL head is in O(1). Currently, the runtime adaptation of LP size implicitly determines the average number of events in each FEL so that insertion overhead remains acceptable. We plan to explore the performance of different data structures when accessed on a per-thread basis on the GPU in our future work. *Second*, we have shown that the performance of the purely GPU-based variant of the simulator is lower than that of the API-based variant. The dynamic parallelism feature of CUDA devices of compute capability 3.5 allows for synchronization of memory accesses between all threads on the GPU. Hence, if synchronization using dynamic parallelism is more efficient than API-based synchronization, higher performance in the purely GPU-based simulator may be feasible. *Third*, the maximum size of the network state and FELs is limited by the graphics memory available. Our current execution model allows for a dynamic resizing of LPs to resolve FEL overflows. For large-scale network simulations, the simulator could be extended to support multiple GPU devices while incurring some overhead for data transfers over the PCI-E bus. *Finally*, the simulation performance depends on properties of the network model: as graphics memory is optimized for high bandwidth instead of low latency, if there are sequences of scattered memory accesses during event handling, large numbers of parallel events are required to allow for efficient hiding of memory access latencies, limiting the benefit of GPU-based simulation for small-scale networks. Additionally, as all threads of each warp operate in lock-step, heavy branching in the model code depending on node states will impede performance. Hence, GPU-based simulations of models with large variation in node behavior, such as state machine models of TCP connections, should be studied in future work.

6 CONCLUSION

In this paper, we proposed a GPU-based simulator enabling substantial runtime reductions for simulations of large-scale networks. All steps of the simulation are performed on the GPU so that only minimal interaction between the CPU and GPU execution contexts is required. To balance parallelism and simulation overheads, the simulator adapts its configuration to the runtime conditions in the simulated network. We compared two implementations of the simulator based on NVIDIA CUDA: in our experiments, higher performance was achieved by a variant using the CUDA API for synchronization of memory accesses than when using a software-based synchronization mechanism on the GPU. An analysis of a widely used peer-to-peer network protocol showed that thousands of processor cores are necessary to fully exploit the parallelism inherent in the network. Using a single commodity GPU, the proposed simulator enabled a simulation runtime reduction by a factor of up to 19.5 compared with an efficient sequential simulator implementation.

Our future work will focus on exploring data structures to reduce event management overhead and on alternative mechanisms for memory access synchronization.

ACKNOWLEDGEMENT

We acknowledge the valuable work of P. Pfaffe who explored parts of the GPU-based simulator design space in his diploma thesis.

REFERENCES

Andelfinger, P., J. Mittag, and H. Hartenstein. 2011. "GPU-Based Architectures and Their Benefit for Accurate and Efficient Wireless Network Simulations". In *19th Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2011*, 421–424.

Berry, O., and D. Jefferson. 1985. "Critical Path Analysis of Distributed Simulation". In *Proceedings of the 1985 SCS Multiconference on Distributed Simulation*.

Carothers, C. D., R. LaFortune, W. D. Smith, and M. R. Gilder. 2006. "A Case Study in Modeling Large-Scale Peer-to-Peer File-Sharing Networks Using Discrete-Event Simulation". In *Proceedings of the International Mediterranean Modeling Multiconference*, 617–624.

Dinh, T. T. A., M. Lees, G. Theodoropoulos, and R. Minson. 2008. "Large Scale Distributed Simulation of P2P Networks". In *16th Euromicro Conf. on Parallel, Distr. and Network-Based Processing*, 499–507.

Jünemann, K., P. Andelfinger, and H. Hartenstein. 2011. "Towards a Basic DHT Service: Analyzing Network Characteristics of a Widely Deployed DHT". In *Proceedings of the 20th International Conference on Computer Communications and Networks (ICCCN)*, 1–7.

Kunz, G., D. Schemmel, J. Gross, and K. Wehrle. 2012. "Multi-Level Parallelism for Time- and Cost-Efficient Parallel Discrete Event Simulation on GPUs". In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation*, 23–32.

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

Maymounkov, P., and D. Mazières. 2002. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In *Peer-to-Peer Systems*, Volume 2429 of *Lecture Notes in Computer Science*, 53–65.

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

NVIDIA Corporation 2013. *NVIDIA CUDA C Programming Guide*. Version 5.5, NVIDIA Corporation.

Park, A., R. M. Fujimoto, and K. S. Perumalla. 2004. "Conservative Synchronization of Large-Scale Network Simulations". In *Proc. of the 18th Workshop on Parallel and Distributed Simulation*, 153–161.

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

Perumalla, K. S. 2006. "Discrete-Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)". In *Proc. of the 20th Workshop on Principles of Adv. and Distributed Sim.*, 74–81.

Quinson, M., C. Rosa, and C. Thiery. 2012. "Parallel Simulation of Peer-to-Peer Systems". In *CCGrid 2012 – The 12th IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing*, 668–675.

Sengupta, S., M. Harris, and M. Garland. 2008. "Efficient Parallel Scan Algorithms for GPUs". *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003* (1): 1–17.

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

Xiao, S., and W.-c. Feng. 2010. "Inter-Block GPU Communication via Fast Barrier Synchronization". In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 1–12.

AUTHOR BIOGRAPHIES

PHILIPP ANDELFINGER is a PhD candidate at the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. His research subject is the performance of parallel and distributed simulations of large-scale computer networks. His email address is philipp.andelfinger@kit.edu.

HANNES HARTENSTEIN is a professor of computer science and a director of the Steinbuch Centre for Computing at the Karlsruhe Institute of Technology (KIT), Germany. His research interests include mobile and virtual networks, security, and information technology management. His email address is hannes.hartenstein@kit.edu.