Multi-level Parallelism for Time- and Cost-efficient Parallel Discrete Event Simulation on GPUs

Georg Kunz*, Daniel Schemmel‡ James Gross‡, Klaus Wehrle*
*Communication and Distributed Systems, ‡Mobile Network Performance Group
RWTH Aachen University
{kunz,wehrle}@comsys.rwth-aachen.de, {schemmel,gross}@umic.rwth-aachen.de

***Abstract*—Developing complex technical systems requires a systematic exploration of the given design space in order to identify optimal system configurations. However, studying the effects and interactions of even a small number of system parameters often requires an extensive number of simulation runs. This in turn results in excessive runtime demands which severely hamper thorough design space explorations.**

In this paper, we present a parallel discrete event simulation scheme that enables cost- and time-efficient execution of large scale parameter studies on GPUs. In order to efficiently accommodate the stream-processing paradigm of GPUs, our parallelization scheme exploits two orthogonal levels of parallelism: *External parallelism* among the inherently independent simulations of a parameter study and *internal parallelism* among independent events within each individual simulation of a parameter study. Specifically, we design an event aggregation strategy based on external parallelism that generates workloads suitable for GPUs. In addition, we define a pipelined event execution mechanism based on internal parallelism to hide the transfer latencies between host- and GPU-memory. We analyze the performance characteristics of our parallelization scheme by means of a prototype implementation and show a 25-fold performance improvement over purely CPU-based execution.

I. Introduction

Complex technical systems inherently provide a multitude of tuning parameters. In order to find an optimal configuration of a system (in terms of a specific goal), a thorough exploration of the given design space is necessary. This design space exploration is typically performed by means of detailed simulation models and elaborate parameter studies, often involving a large number of simulation runs. However, even if a single simulation run finishes quickly, the total combined runtime needed to complete a parameter study can become considerably large, thereby severely hampering the design space exploration process.

Despite existing techniques for reducing the number of relevant system parameters, e. g., factorial design [1], studying only a few parameters quickly results in large amounts of simulations. For example, a study over 5 parameters, each with 5 distinct values of interest, requires simulating $5^5 = 3125$ different parameter sets. Additionally, in order to obtain statistically credible results, all parameter sets need to be repeatedly executed with varying random seeds. Considering the 3125 parameter sets and 30 repetitions, a total of nearly 100,000 distinct simulations runs are necessary.

In this paper, we present a parallel discrete event simulation scheme that enables a cost- and time-efficient execution of large scale parameter studies on GPUs. Our approach leverages the massively parallel processing power of GPUs to *concurrently* execute all individual runs of a parameter study. The overall goal of our work is to provide a cost-efficient alternative to the traditional approach of distributing the simulations of a parameter study to multiple CPUs[1]. We argue that conducting large scale parameters studies on a single GPU might in fact be slower than running them on a *large* number of CPUs, yet purchasing one consumer level GPU is significantly cheaper than buying and maintaining a large number of CPUs. Hence, our approach constitutes a trade-off between cost and processing power.

In order to successfully exploit the massively parallel GPU-architecture, we need to address two particular challenges: i) GPUs implement a Single Instruction, Multiple Thread (SIMT) processing paradigm in which groups of threads execute in lockstep, and ii) due to limited onboard memory, data needs to be continuously transferred between host- and GPU-memory, hence imposing large memory access latencies. Our key contribution in this paper is a multi-level parallelization scheme that overcomes these challenges. In particular, the scheme builds upon two orthogonal levels of parallelism.

i) *External parallelism* given by the inherently independent simulations of a parameter study: Assuming that the simulations of a parameter study behave similarly, external parallelism enables an event aggregation scheme that generates SIMT-compatible workload.
ii) *Internal parallelism* given by the independent events within each individual simulation of a parameter study: By interleaving the transfer to and from GPU-memory with the execution of independent events on the GPU, internal parallelism allows for establishing a pipelined event execution that hides memory access latencies.

Based on a proof-of-concept prototype, we analyze the performance characteristics of the proposed scheme by means of synthetic benchmarks. Moreover, we conduct a case study using a wireless network model. We show that our parallelization scheme reduces the runtime demand in this case study by a factor of 25 over an equivalent CPU-based implementation.

The remainder of this paper is structured as follows: After analyzing the GPU-related challenges in Section II, we present the design of our multi-level parallelization scheme in

[1]We use the terms CPU and CPU-core interchangeably in this paper.

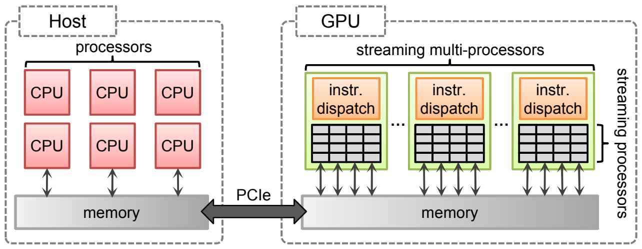


Fig. 1. Simplified illustration of a typical GPU architecture. Each streaming multi-processor contains multiple streaming processors which share a common instruction dispatch unit.

```
void gpuKernel() {
    if (threadId % 2 == 1)
        doSomething();
    else
        doSomethingElse();
    doSomethingTogether();
}
```

Fig. 2. Example for the lockstep execution problem. Due to a shared instruction dispatch unit, all threads in a warp alternate executing the diverging code paths of the `if`-statement. This effectively doubles the total runtime.

Section III. We then detail on our prototype implementation in Section IV and evaluate our scheme in Section V. Finally, we discuss related efforts in Section VI and conclude in Section VII.

II. Challenges of Integrating GPUs with PDES

Motivated by intriguing parallel performance and supported by general purpose programming frameworks like NVIDIA's CUDA or OpenCL, GPUs have become an invaluable tool for high-performance computing. For instance, the NVIDIA GTX 580 graphics card provides a total of 512 processing cores, organized in 16 multi-processors, which again consist of 32 stream processors each. Supported by hardware assisted thread scheduling, an even greater number of threads can concurrently execute on a single GPU. However, in order to fully exploit the available processing power, two peculiarities of the GPU architecture need to be taken into account: i) Threads are organized in groups which execute in lockstep, and ii) limited onboard memory necessitates copying data to and from host-memory. We discuss both aspects in the following and show why they pose a particular challenge in the context of Parallel Discrete Event Simulation (PDES).

A. *Lockstep Execution of Threads*

In NVIDIA's Fermi architecture, a group of 32 threads forms a so called warp [2]. Since the threads in a warp execute on a common multi-processor, they share a single instruction dispatch unit (see Fig. 1). As a result, all threads in a warp execute the same instructions in lockstep, implementing the Single Instruction, Multiple Thread (SIMT) processing paradigm. Inspired by z-culling, it is nevertheless possible for each thread to ignore single instructions from the common instruction stream. By selectively masking instructions, the threads in a warp can in principle follow different code paths. However, since the masked instructions are part of a common instruction stream, no other instructions can be executed in the meantime. Fig. 2 illustrates the resulting problem by means of a simple example. Assume `threadId` is a variable holding the unique ID of the threads in a warp. The conditional `if`-statement then causes all threads with an odd ID to process the function `doSomething` while all threads with an even ID process the function `doSomethingElse`. Since both functions comprise different sets of instructions from the common instruction stream, both groups of threads alternately execute one instruction from each set, thereby effectively doubling the runtime. Fortunately, the CUDA runtime automatically synchronizes both groups after executing diverging code paths, so that all threads jointly execute the function `doSomethingTogether`.

In stark contrast to the processing paradigm of GPUs, parallel discrete event simulations execute *independent* events in parallel. In general, those events do not execute the same code but instead model completely unrelated aspects of the simulated system. Consequently, when mapped to the same streaming multi-processor, independent yet unrelated events are still executed sequentially by the GPU. Concluding, in order to efficiently integrate the lockstep execution paradigm of GPUs with parallel discrete event simulations, we need to design an event processing strategy that provides SIMT workload to the streaming multi-processors of the GPU.

B. *Memory Size and Latency*

In comparison to the amount of main memory provided by a typical desktop computer (~8 GB) or server (~32 GB), the size of GPU memory is relatively small (~1.5 GB). Hence, data is commonly held in host memory and transferred to GPU memory on demand, modified there, and finally copied back to host memory (see Fig. 1). Unfortunately, such memory transfers across the PCIe bus suffer from significant latencies and can easily become a performance bottleneck. A common approach towards mitigating the adverse performance effects of these memory transfers is latency hiding [3]. The key idea is to perform three operations *concurrently*: i) Memory transfers to the GPU, ii) memory transfers from the GPU, and iii) actual GPU processing. This three-stage pipelining keeps the PCIe bus as well as the streaming processors of the GPU busy. In order to achieve a fully pipelined execution of events in parallel discrete event simulations, all events in the three stages of the pipeline need to be independent. Thus, the event scheduler has to identify independent events and synchronize their execution accordingly. Moreover, each memory transfer over the PCIe bus involves a control overhead that favors large transfer units (1-1000 K) over small ones (1-1000 bytes) [3]. However, the memory footprint of a single event in a discrete event simulation is typically small. Hence, transferring individual events to and from GPU-memory suffers from the control overhead inherent to small transfer sizes.

III. Multi-level Parallelization on GPUs

The goal of this work is to leverage the massively parallel processing power of GPUs to provide a cost-efficient reduction of the execution time of parameter studies. Based on the previous analysis of the peculiarities of GPUs, we state two key design requirements for an efficient utilization of GPUs in parallel discrete event simulations. Specifically, the parallel simulation framework has to

i) *provide SIMT-compatible workload* to the GPU to accommodate the lockstep execution paradigm, and
ii) *hide the latency of memory transfers* between host- and GPU-memory.

In order to meet those two design requirements, our proposed solutions are based on two orthogonal levels of parallelism. The first level, *external parallelism*, exploits the fact that individual simulations in a parameter study are trivially independent and can hence execute in parallel. External parallelism thus lays the foundation for an event aggregation scheme specifically designed for generating SIMT-compatible workload. The second level of parallelism, *internal parallelism*, makes use of the observation that within an individual simulation, groups of events may be independent and thus allow for parallel processing. We exploit internal parallelism to hide the latencies involved in memory transfers.

Both levels of parallelism are not new in themselves but have been used in parallel simulation frameworks before. Instead, we claim that the combination of both schemes results in a novel parallelization scheme which unlocks the massively parallel processing power of GPUs for parallel discrete event simulations. The following sections introduce our approach in greater detail.

A. SIMT-compatible workload using External Parallelism

In order to meet the first design requirement, our GPU-based parallelization framework needs to generate SIMT-compatible workload to match the lockstep execution paradigm of GPUs. To this end, we design an event aggregation scheme that exploits the fact that parameter studies comprise multiple independent and self-contained simulations.

1) Event Aggregation Scheme: In a parameter study, each individual simulation takes a specific combination of parameter values as input. Moreover, each combination of values is typically executed several times with different random seeds to obtain statistical confidence in the computed results. We argue that despite different parametrization, the individual simulations of a parameter study behave similarly since they encompass the same logic, i. e., model implementation. In particular, we expect the *order* of events in a simulation run to be similar across the individual simulations of a parameter study. Nevertheless, since each simulation uses a different set of parameters, the actual *state* of each simulation model, e. g., local variables, differs. Following the design of widely used simulators [4], [5], we assume that simulation models exhibit a modular structure. Hence, we define the state of an event to be the values of the local data structures and variables of the module the event takes place at. Based on these assumptions and definitions, our approach towards generating SIMT-compatible workload for GPUs is as follows.

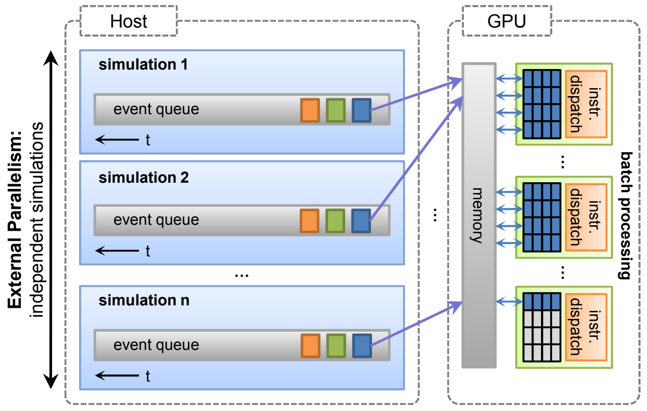
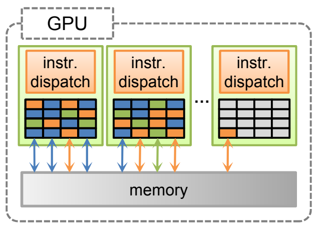


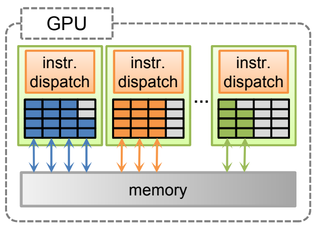
Fig. 3. By exploiting the external parallelism between independent simulations of a parameter study, our event aggregation scheme creates SIMT-compatible workload for the streaming multi-processors of the GPU.

Given a parameter study consisting of multiple individual simulations, our scheme executes all these simulations concurrently in a round-based fashion. In each round, it first dequeues from all simulations the event with the lowest timestamp. Since the simulations supposedly exhibit the same behavior, these events are of the same *event type*. We define the event type to be uniquely defined by the specific event handler called when executing an event. In contrast to the event type, the local state of each event is different. Hence, we aggregate and transfer all relevant states from host- to GPU-memory before executing the event handler on the GPU. As a result, we generate SIMT-compatible workload by executing one event handler (i. e., a single set of instructions) on a batch of aggregated event states (i. e., multiple data) by means of multiple threads. Fig. 3 gives a schematic overview of the event aggregation scheme. In this simplified example, all n simulations behave exactly the same, resulting in the same order and type of events in the corresponding event queues. Hence, by removing the first event from every event queue and aggregating the associated states in a batch, the stream processors of the GPU can modify multiple event states while executing the same instructions.

2) Divergent Simulations: Of course, we cannot expect all simulations of a parameter study to behave exactly the same in all scenarios. Instead, it is more realistic to assume a divergent event ordering among simulations. For instance, a successful packet transmission triggers an ACK while an unsuccessful transmission causes a NACK. In this case, the first events of the respective simulations will subsequently differ and the resulting event batches will contain different event types in an arbitrary ordering. As the ordering within a batch defines a mapping of events to the threads of a warp, divergent simulations therefore result in a heterogeneous mapping (see Fig. 4(a)). Consequently, the performance decreases due to the divergent code paths of different event handlers. In general, the degree of divergence between simulations heavily depends on the particular simulation model as well as the parameters under investigation. For example, a model of a strictly timed cellular network is less susceptible to divergent behavior than a model of a CSMA-based WiFi network.



(a) Unsorted event batch.



(b) Sorted and padded event batch.

Fig. 4. Diverging simulations cause a heterogeneous mapping of events to streaming multi-processors. By sorting and padding the batch of events, we create a homogeneous mapping that allows for truly parallel event processing.

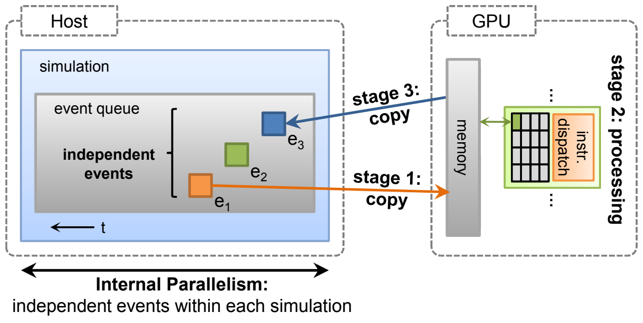


Fig. 5. Internal parallelism utilizes independent events within a simulation to establish a three-staged pipeline for event execution: i) copy event state to GPU memory, ii) execute event, iii) copy event state back to host memory.

To cope with divergent simulations, we exploit the fact that threads on *different* multi-processors can independently execute different instructions. Thus, our goal is to carefully map events of different types to independent multi-processors while assigning events of the same type to the same processor. Specifically, we propose two simple mapping techniques.

Sorting: Our first mapping technique sorts the events within a batch according to their type. The reasoning behind this simple approach is that a sorted batch of events increases the chances for assigning fewer different event types to one multi-processor. However, it cannot guarantee a clean 1-to-n mapping of event types to multi-processors and it generally performs poorly for highly heterogeneous event batches. Even in case of only few different event types, the simple sorting scheme cannot align events to the boundaries of the multi-processors. Thus a group of events of the same type might span two multi-processors despite actually fitting onto a one.

Padding: An extension of the sorting scheme is event padding. This scheme explicitly introduces gaps in a sorted event batch to achieve a clean 1-to-n mapping of event types to multi-processors and to align event types to the boundaries of multi-processors (see Fig. 4(b)). However, despite achieving a homogeneous mapping, the gaps effectively decrease the utilization of GPU resources since the processors mapping to a gap cannot perform useful work. Hence, padding involves a trade-off between a decrease in resource utilization and a performance gain through homogeneous event-to-processor mappings. We analyze the performance of both simple mapping schemes in Section V-A1 and show that they indeed are able to mitigate the performance impact of divergent simulations. Future efforts nevertheless will focus on the development and implementation of more sophisticated mapping algorithms.

Both padding and sorting algorithms aim for generating SIMT-compatible workload on the level of event types. Yet, even if executing a single event type (i. e., event handler), the potentially different states may still cause divergent code paths inside the corresponding event handlers (recall the example in Fig. 2). However, since this kind of divergence depends on the parametrization, we believe that its performance impact is on average less severe than that of entirely different event types. Nevertheless, future mapping algorithms may take a more fine grained view on the event batch, for instance by incorporating the actual state of the events.

B. Hiding Memory Latencies using Internal Parallelism

In addition to external parallelism, we exploit internal parallelism to hide the adverse performance effects of memory transfers to and from the GPU. Internal parallelism relies on the fact that simulation models typically exhibit a certain level of independence among the events of a simulation run. For instance, events representing packets in the protocol stack of different network nodes are independent of each other, hence allowing for parallel execution. Note that such independent events are the foundation of traditional PDES techniques [6], [7]. However, in contrast to traditional parallel discrete event simulation, we do not intend to actually *execute* such events concurrently. Instead, we exploit independent events to create a three-staged pipeline which allows for a concurrent *handling* of such events in order to hide memory transfer latencies.

Given three independent events e_1, e_2, e_3, the three pipeline stages perform the following tasks concurrently (see Fig. 5): The first stage copies e_1 from main- to GPU-memory. At the same time, stage 2 executes the previously transferred event e_2 on the GPU. Lastly, the third stage copies e_3 back from GPU- to main-memory. Interleaving all three stages allows for effectively hiding the memory latencies involved in stage 1 and 3, assuming a completely filled pipeline. However, if fewer than three independent events are available at any point in time during a simulation run, the pipeline stalls and some stages may remain empty. In order to determine independent events, our proposed parallelization scheme utilizes conservative synchronization [6], [7]. As our scheme does not impose special requirements on the conservative synchronization algorithm, any such algorithm can be used.

In addition to conservative synchronization, we moreover envision a speculative execution scheme that is inspired by modern CPUs. The scheme is based on the observation that the results of executing an event are two-fold and consist of i) a modified event state, and ii) newly generated events. Note that both results only take effect in the global simulation state after being transferred from GPU- to host-memory. Hence, in order to avoid stalls of the event pipeline due to a shortcoming of independent events, we can speculatively dispatch potentially independent events to the first ("copy state to GPU-memory") and second pipeline stage ("execute event on GPU"). If it turns out that executing the event was in fact correct, its state is copied back and newly generated events are enqueued in the corresponding future event set. If, however, the event should not have been executed, the modified event state and all new events are simply discarded. As a result, speculative execution may turn the performance overhead of transferring event states into a performance optimization.

A remaining limitation of our approach is the relatively small size of onboard memory which restricts the number and the size of event states that can reside in GPU-memory. For instance, assuming a parameter study with 500 simulations and a 3-staged pipeline, a maximum of 1500 events states need to be stored in GPU-memory. Assuming furthermore a typical consumer GPU with 1.5 GB of memory, each event state may not be larger than 1 MB. However, we argue that this limitation does not impose severe restrictions on model developers in practice. First, the event state contains only data which is transferred between the modules of a simulation model (see next section) and which is typically limited in size. Second, the growing popularity of general purpose computing on GPUs will likely foster a strong increase in the size of GPU memory.

IV. Implementation

We developed a proof-of-concept prototype to investigate the viability of our multi-level parallelization scheme. The prototype does not directly build on an existing simulation framework, yet its architecture and the modular structure of its simulation models are inspired by OMNeT++ [4]. The prototype consists of a CPU-bound simulation core and GPU-located simulation models. Furthermore, its implementation is based on CUDA 4 and makes heavy use of Unified Virtual Addressing (UVA). UVA provides a single virtual address space spanning host- and GPU-memory, thereby significantly increasing the memory available to the GPU. The drawback of UVA however is a severe performance penalty when accessing data in unified memory from the GPU as it needs to be fetched from host memory. Hence, in our implementation the event state solely comprises the "payload" of the events. As in OMNeT++, simulation models attach data to events to exchange information between modules. This gives model designers explicit control over the event state size and the corresponding memory transfer overhead. In contrast, data local to modules remains in UVA and can hence become (arbitrarily) large.

Our framework exports a typical discrete event simulation interface that abstracts from GPU-programming and CUDA. It primarily provides access to random number generators and allows for creating and scheduling new events. Hence, the model implementation effort is comparable to typical CPU-based simulators such as OMNeT++ or ns-3 [5] as the interface hides the complexities of GPU programming. For instance, to avoid the considerable overhead of dynamic memory allocation on the GPU, the framework creates new events in a specific buffer provided by each event state. After copying the event state and the buffer back to host memory, newly created events are removed from the buffer and enqueued in the event queues of the corresponding simulations. Furthermore, we utilize multiple CUDA streams to implement pipelined event execution. Each stream is part of a simulation driver which performs four tasks in a round based fashion. In each round, a simulation driver i) collects one event from each simulation, ii) writes the corresponding event states to its CUDA stream, iii) launches the event handling kernel, and iv) reads modified event states from the CUDA stream. By interleaving the rounds of multiple simulation drivers, our implementation establishes a pipelined event execution. In the current prototype, these CPU-based tasks execute sequentially in a single thread. We leave the natural extension of the prototype to a multi-threaded architecture for future work and focus instead on the GPU-related challenges in this paper. Similarly, our implementation does not yet provide speculative execution capabilities.

In order to handle divergent simulations, the prototype employs both simple sorting and padding schemes as outlined in Section III-A. Our implementation of the padding algorithm assumes the worst case in the sense that every simulation may contribute a different event type to the event batch. To still allow for a homogeneous mapping in this case, the size of the padded batch increases from $|S|$ to $|S| \cdot w$, where S denotes the set of simulations and w the warp size. The implementation hence trades off increased memory demands for performance.

V. Evaluation

This section presents an initial evaluation of the proposed multi-level parallelization scheme based on our proof-of-concept implementation. We analyze the performance properties of the proposed scheme in terms of the performance impact of divergent simulations and the event handling overhead. In order to precisely control these parameters, the evaluation employs a set of synthetic benchmarks we introduce separately in the following sections. Moreover, to get an impression of the possible performance gain in a real-world scenario, we complement the synthetic benchmarks with a case study based on a wireless mesh network model.

The benchmarking platform is a workstation PC providing an AMD Phenom II X4 945 4-core CPU with 8 GB of main memory and one NVIDIA GeForce GTX 470 GPU accommodating 1.28 GB of memory. The simulation framework runs on a 64 bit version of Ubuntu 10.10 with NVIDIA's proprietary drivers in version 290.10. Finally, we enable full optimizations using the NVIDIA CUDA compiler, nvcc, in version 4.0 and g++ in version 4.4. Furthermore, each data point shows the mean and the 99 % confidence intervals computed over 30 independent repetitions. Nevertheless, the confidence intervals are barely visible due to highly consistent performance results.

A. Synthetic Benchmarks

We utilize two different synthetic benchmarks to investigate i) the impact of divergent simulation behavior on parallel performance, and ii) the event handling overhead of our framework. The following sections introduce both benchmarks in detail and analyze their respective results.

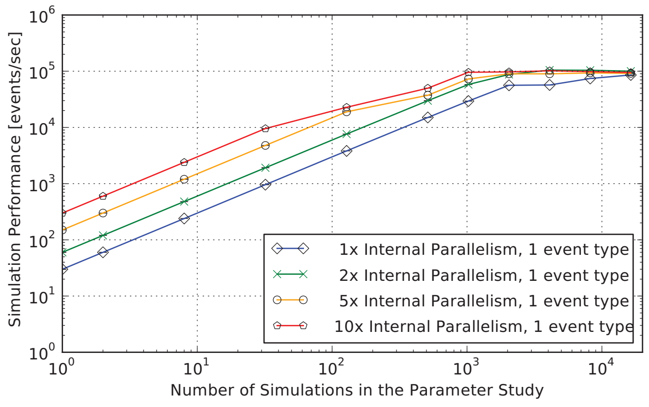


Fig. 6. Simulation performance using nondivergent and computationally complex simulations. The performance increases linearly along both levels of parallelism until the GPU is saturated for large numbers of simulations.

TABLE I
SYNTHETIC BENCHMARK PARAMETERS

Parameter	Values
Simulations (ext. par.)	$2^i, i \in \{0, 1, 3, 5, 7, 9, 10, 11, 12, 13, 14\}$
Modules (int. par.)	1, 2, 5, 10
Events per simulation	300
Event state size	72 bytes
LCG iterations per event	20000

1) Divergent Simulations: The first synthetic benchmark analyzes the performance impact of divergent simulations. It particularly investigates the effectiveness of the simple sorting and padding algorithms outlined in Section III-A.

Methodology: The performance impact of divergent simulations is most pronounced for computationally complex events which hide all other performance effects and overheads. Hence, the benchmark uses arithmetically dense Linear Congruential Generators (LCGs) for generating one pseudo random number per event. Each module of the benchmark model encapsulates one such LCG and continuously reschedules a single local event. To create divergent behavior, the benchmark employs different LCGs per module, each one resulting in a different code path and hence event type. Internal parallelism is controlled via the number of modules in the model as we consider concurrent events on different modules to be independent. The size of the event state is 72 bytes which includes the minimum set of meta-data required by the framework for event handling. Moreover, each simulation comprises a static workload by processing a fixed number of events. We measure the performance in terms of the average number of events processed per second. Table I gives an overview over the set of parameters and their values.

Results: In order to quantify the performance impact of divergent simulations, we first need to determine the maximal achievable performance in a nondivergent scenario as shown in Fig. 6. Focusing on an internal parallelism of 1, we observe a perfect linear performance increase up to 2048 simulations (note the double-log scale). Hence, this computationally dense benchmark indeed constitutes an ideal case for the massively parallel processing power of the GPU. Beyond 2048 simulations however, the GPU is fully saturated, achieving no additional speedup. In addition, the performance increases linearly with the level of internal parallelism as more events are available for processing on the GPU.

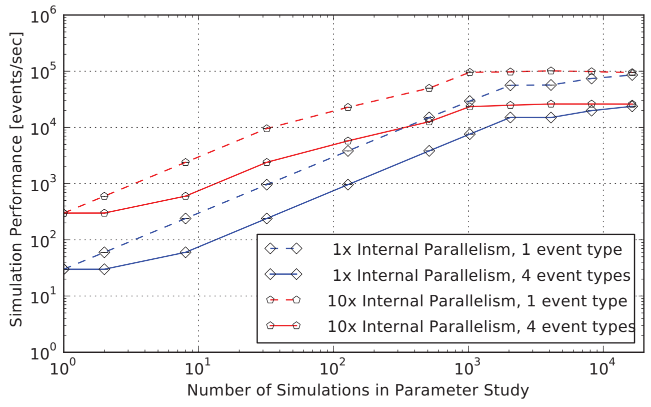


Fig. 7. Comparison of nondivergent (dashed lines) with 4-way divergent (solid lines) simulations w/o sorting and padding of events. The simulation performance drops by a factor of 4 due to divergent code paths.

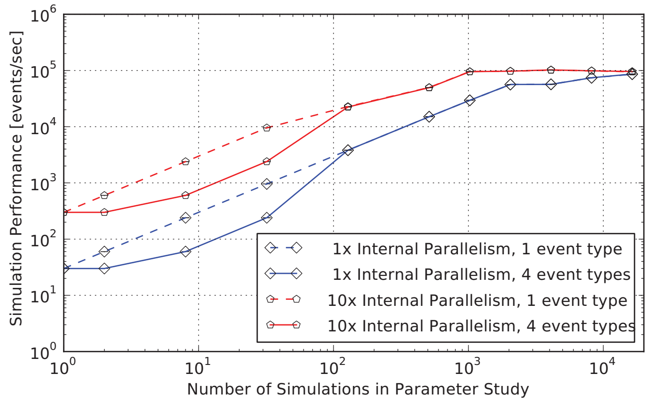


Fig. 8. Comparison of nondivergent (dashed lines) with 4-way divergent (solid lines) simulations with sorting but w/o padding of events. For more than 128 simulations, sorting achieves a perfect mapping of events to multiprocessors, resulting in a perfect performance recovery.

After establishing the optimal performance as baseline, we now consider 4-way divergent simulations. To this end, the modules of each individual simulation select one out of four different LCGs, yielding exactly four different event types per batch. In this context, Fig. 7 compares the baseline performance (dashed lines) with the performance of a 4-way divergent configuration (solid lines) without applying sorting or padding. We clearly observe a consistent 4-fold performance decrease over all values of internal and external parallelism larger than 2. This confirms the expected adverse effects of divergent simulations. Note that for 2 simulations, the performance drop corresponds to just a factor of 2 as there can only be 2 events in a batch anyway. The same holds true for just 1 simulation.

Sorting the events before execution yields the performance results shown in Fig. 8. Similarly to the nonsorted and nonpadded case discussed before, a 4-fold decrease in simulation performance remains for up to 64 simulations. However, starting with 128 simulations, the performance of the divergent

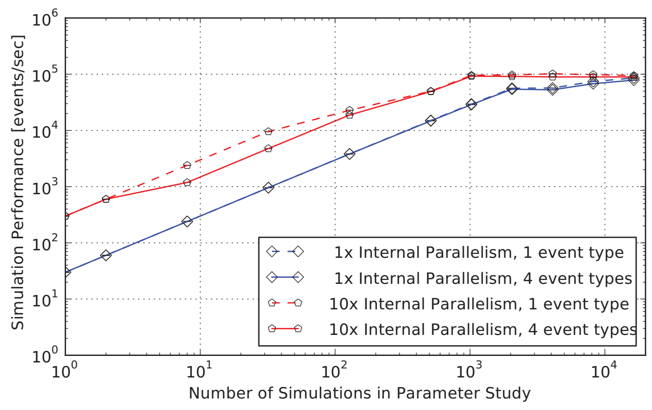


Fig. 9. Comparison of nondivergent (dashed lines) with 4-way divergent (solid lines) simulations with sorting and padding of events. Padding achieves a significant performance recovery but causes a noticeable runtime overhead.

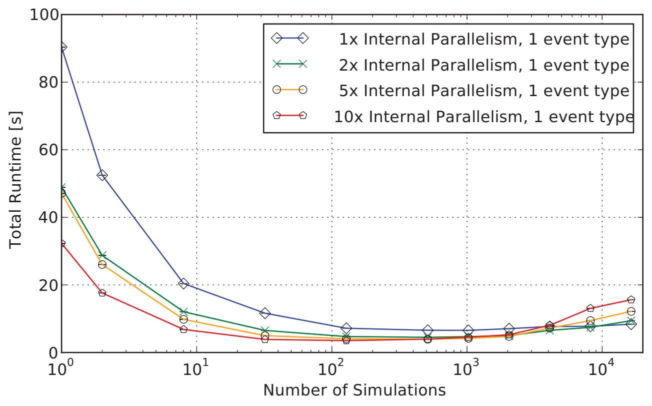


Fig. 10. Runtime of the synthetic overhead benchmark indicating that the event handling overhead is effectively parallelized with an increasing number of simulations. Up to 4096 simulations, a higher degree of internal parallelism furthermore reduces the overhead due to memory latency hiding. Beyond 4096 simulations the GPU is overloaded.

benchmark matches the ideal performance of the convergent one. We accredit this to the fact that the particular GPU used in the benchmarks utilizes a warp size of 32 threads. Therefore, in the case of 4 different event types and 128 simulations, the simple sorting scheme (coincidentally) achieves an ideal mapping of exactly 32 identical event types to each warp. The same also holds true for the remaining configurations as the number of simulations is divisible by 32 without remainder. Nevertheless, this constitutes an ideal case which rarely occurs in real scenarios. Hence, sorting is not sufficient to mitigate the performance loss inflicted by divergent simulations.

A more promising approach to realizing an ideal mapping of event types to warps is padding of the sorted event array as sketched in Section III-A. Applying this technique to the 4-way divergent benchmark yields the results presented in Fig. 9. The figure clearly illustrates that for an internal parallelism of 1, divergent and convergent behavior achieve the same performance. Furthermore, for an internal parallelism of 10, the results show a significant improvement over sorting, however, the performance still remains slightly below the nondivergent case. This is due to the additional computational overhead caused by the padding algorithm as well as a less efficient utilization of the multi-processors due to gaps in the event batch. We particularly blame the latter for the performance drop at 8, 32 and 128 simulations. Hence, future efforts towards improving the overall performance of the multi-level parallelization scheme will focus on developing more efficient event mapping algorithms.

2) Event Handling Costs: The event handling costs of a discrete event simulation framework comprise in general all management operations within the framework, such as creating and deleting, or enqueueing and dequeueing events. In the context of our GPU-based framework, the overhead additionally includes the costs of transferring event states between host- and GPU-memory over the PCIe bus.

Methodology: To measure the event handling costs, we use a synthetic benchmark model which does not perform any computations on the GPU apart from continuously re-scheduling new events. As a result, the runtime of this simulation model constitutes a direct measure for the event handling costs. In contrast to the benchmark model used in the previous section, this model employs a fixed workload for the *whole* parameter study. Since we use the runtime as a direct measure for the overhead, the workload has to remain constant when changing the benchmark parameters of interest. Thus, each parameter study executes a fixed number of events, which are equally distributed across all simulations and modules within the simulations. Based on this model, we analyze the event handling costs under consideration of the degree of internal and external parallelism as well as the event state size.

Event Scheduling Costs: In this benchmark, we again vary the level of external parallelism between 1 and 16384 and the level of internal parallelism between 1, 2, 5 and 10 while fixing the event state size to the minimum of 72 bytes. Fig. 10 shows the resulting total runtimes. Focusing on the results obtained for an internal parallelism of 1, we observe a nearly ideal decrease in the total runtime when increasing the number of simulations from 1 to 128. Above 128 simulations, however, the runtime converges towards a stable value. We conclude that the event handling overhead is effectively distributed across the growing number of parallel events (due to increasing external parallelism) and hence hidden up to a certain degree. More importantly, we conclude furthermore that the performance gain of parallel processing is larger than the event handling overhead, even for computationally insignificant events. Analyzing the results for the other values of internal parallelism indicates a similar behavior, yet even lower runtimes for up to 2048 simulations. This demonstrates that the pipelining approach for hiding memory latencies successfully reduces the effective event handling overhead as well. Nevertheless, the figure also reveals an increasing runtime for more than 4096 simulations and a degree of internal parallelism larger than 1. Specifically, the larger the internal parallelism, the longer it takes to complete the benchmark, thus resulting in an inversion of the performance results. We believe that the GPU is in fact over-saturated in these benchmark configurations due to contention on the CUDA-streams.

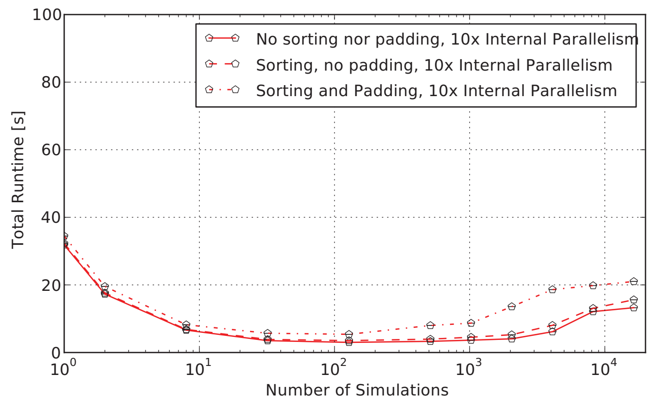


Fig. 11. Comparison of the overhead introduced by sorting and padding of events. Padding adds a considerable overhead over sorting which in turn adds only a negligible overhead.

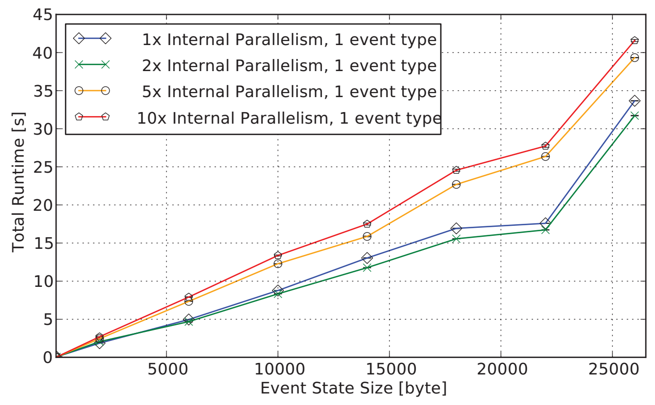


Fig. 12. Comparison of the overhead of different event sizes and levels of internal parallelism. The overhead grows with the size of the events as well as the internal parallelism due to an increased load on the PCIe bus. The number of simulations is fixed to 512 in this benchmark.

Event Mapping Costs: A type of overhead specific to our framework is caused by the event mapping algorithms. Fig. 11 compares the runtimes of the benchmark model using i) no sorting or padding, ii) only sorting, iii) sorting and padding. As expected, applying neither sorting nor padding results in the lowest runtimes, i. e., overhead. In comparison, sorting adds only slightly more overhead. This overhead, however, grows with an increasing number of simulations as the sorting operation becomes more complex. In contrast, the simple padding algorithm adds a considerable overhead, in particular for more than 1024 simulations. Nevertheless, as shown in the previous section, the performance gain of the padding scheme clearly outweighs its overhead.

Memory Transfer Costs: Lastly, we investigate the influence of the event state size on the event handling overhead. To this end, we fix the number of simulations to 512 and stepwise increase the event state size from the minimum of 72 bytes to 26000 bytes. Fig. 12 plots the resulting total runtimes for different values of internal parallelism. In contrast to the previous results, the figure illustrates that larger degrees of internal parallelism cause longer runtimes, which directly translates into more overhead. This is due to a trade-off between utilization and contention regarding the PCIe bus: For an increasing degree of internal parallelism, more CUDA streams allow for better utilizating the bus but also add more overhead due to contention. In Fig. 12, an internal parallelism of 2 achieves the best trade-off as indicated by the shortest runtime. An internal parallelism of 1 does not yet fully utilize the bus, while an internal parallelism of 5 and 10 cause too much contention. However, note that the events of this benchmark model are by design computational insignificant. Hence, they do not consume enough runtime to outweigh the memory transfer overhead inherent to large event states.

B. Case Study

In addition to the synthetic benchmarks, we conduct a case study by means of a wireless mesh-network model to get an impression of the user-perceived performance gain.

Methodology: The model simulates wireless transmissions based on an accurate and hence computationally complex channel and error model. For the sake of brevity, we do not introduce these models here but refer to Puñal et al. [8] for detailed information. Moreover, the simulated network comprises 5 nodes connected in a fully meshed topology, i. e., every transmission is received by all nodes in the network. Each node consists of three separate modules: i) An application module which broadcasts packets at a fixed sending rate. ii) A MAC module which implements the error model and a rudimentary MAC protocol: Each receiver sends an ACK or a NACK, depending on whether or not a transmission was successfully received. Note that sending different replies is a source of divergent behavior. iii) A PHY module which models the effects of the wireless channel. The model furthermore abstracts from the network and transport layer and disregards interference. Since events traverse the modules up and down the protocol stack, a total of 7 different event types occur in the model. Furthermore, due to the fully meshed topology, every transmission is received by 4 nodes, resulting an internal parallelism of at least 4. Finally, the event size is 268 byte.

In order to create a simple parameter study, the model provides two independent parameters. First, the application module allows for configuring the packet generation rate. Second, the fading model of the wireless channel considers different movement speeds in order to model a dynamic environment. We vary the former between 1, 5, and 10 packets/s and the latter between 1, 5, and 10 m/s. In addition, every combination of parameter values is repeated 30 times with different seeds to obtain statistical confidence. Altogether, the parameter study comprises 270 simulation runs.

To allow for a performance comparison between classic CPU-bound simulation and the proposed multi-level parallelization scheme, the case study utilizes two versions of the wireless network model: A CPU-based version which builds upon OMNeT++ and a GPU-based variant which builds on top of our prototype simulation framework. We explicitly aimed for keeping the implementation of both models as similar as possible despite the architectural differences of the underlying simulation frameworks. Hence, this case study aims at providing a rough impression of the potential performance gain

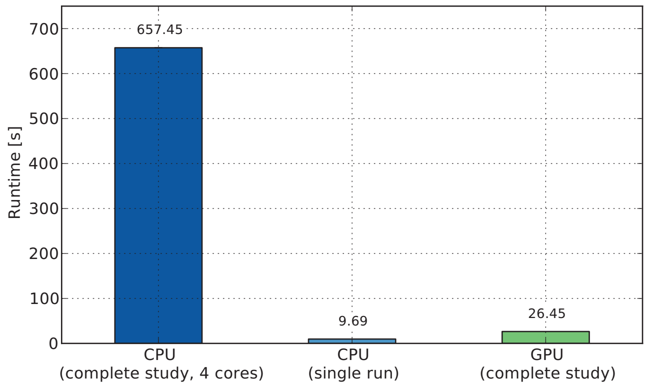


Fig. 13. Runtime of the complete parameter study on a 4-core CPU (left) and on a GPU (right), as well as the runtime of a single CPU-bound parameter run (middle). GPU-assisted multi-level parallelization achieves a 25-fold performance improvement over CPU-based execution.

while keeping the limitations of the particular comparison in mind. Moreover, the CPU benchmark does not utilize internal parallelization but solely employs external parallelism. We argue that any internal parallelization scheme using n CPUs is inherently slower than n independent sequential simulations because of additional synchronization overhead. Given the 4-core CPU of our benchmarking machine, we execute the CPU-bound parameter study in groups of 4 parallel simulations.

Results: Fig. 13 compares the resulting total runtime of the CPU-based parameter study (left bar) with the total runtime of the GPU-based parameter study (right bar). The figure shows that our GPU-assisted approach significantly outperforms the traditional CPU-bound implementation by a factor of more than 25. This confirms that our approach enables a time-efficient execution of parameter studies. Moreover, the bar in the middle of Fig. 13 illustrates the runtime of a single simulation on one CPU-core. It hence represents a distributed execution of the parameter study on 270 individual CPUs. This large scale setup achieves an additional 2.7-fold performance increase over our single GPU-based implementation. However, the costs of purchasing and maintaining computers with a total of 270 CPUs by far exceed the costs of a single computer providing one consumer level graphics card. Thus, our approach constitutes a truly cost-efficient alternative to purely CPU-oriented large scale parallelization.

VI. Related Work

GPUs lay the foundation for improving the performance of a wide range of different types of simulations, including simulations of physical processes [9], computer architectures [10], vehicular networks [11], Monte Carlo simulations [12], etc. A comprehensive survey by Owens et al. [13] gives an overview of the subject. In this paper, we focus particularly on parallel discrete event simulation. Since this field comprises a large body of research [7], we only discuss closely related efforts.

A. Integrating GPUs with PDES

Parallel discrete event simulations can either utilize GPUs solely as potent co-processors or execute the entire simulation on the GPU. We discuss both variants in the following.

1) GPUs as Co-processors: Similarly to our approach, the majority of related efforts integrate GPUs as co-processors onto which complex computations are offloaded. The core logic of the simulation framework, e. g., event schedulers and event queues, remain in host memory and run on the CPU. In this context, Bauer et al. [14] investigate the applicability of GPUs to combined simulations [15] in which the discrete component of the simulation (the event scheduler) executes on CPUs while the continuous component (the event handlers) runs on GPUs. Using a synthetic workload model, the authors report considerable speedups for simulations containing computationally complex events. However, memory I/O turns out to be the primary performance bottleneck of this work which does not consider memory latency hiding techniques. Following a similar approach, Xu et al. [16] identify sources of data- and task-parallelism within detailed network simulation models. In contrast to our work, this approach mainly relies on data-parallelism within complex events and does not explicitly attempt to achieve a high degree of task-parallelism between events. SCGPSim [17] is a simulation framework focusing on SystemC simulations. Using source-to-source compilation of SystemC to CUDA-enabled code, it automatically maps sequentially executing SystemC threads to parallel threads on a GPU. This approach achieves a considerable speedup, but it inherently relies on the processing model of SystemC and hence cannot be applied in general to discrete event simulation.

2) Purely GPU-based Simulation: In contrast to limiting GPUs to mere co-processors, Perumalla [18] explores the challenges of a purely GPU-based simulation framework. To account for the streaming-oriented processing model of GPUs, the traditional event scheduling loop underlying discrete event simulation is replaced by an event-streaming algorithm. Despite a low-level GPU implementation, the approach indeed achieves a parallel speedup in a specific heat diffusion simulation. Although the paper shows the feasibility of an entirely GPU-based simulation framework, the central question of general applicability remains unanswered. Moreover, this pioneering work suffers from the absence of general-purpose programming environments such as CUDA. Park et al. [19], [20] extend the previous work by developing a GPU-based event aggregation and execution scheme based on the concept of approximate time [21]. While the proposed event aggregation scheme can indeed generate considerable performance improvements, it results in numerical errors. Although error analysis and approximation techniques allow for mitigating the amplitude of these numerical errors, this approach is as well not generally applicable. Finally, Chatterjee et al. [22] propose a fully GPU-based simulator for evaluating hardware designs on the gate level. In order to make efficient use of the GPU, the authors introduce a dedicated compilation phase in which the typically monolithic hardware model is segmented in smaller parallelizable tasks. Nevertheless, this approach heavily depends on the specifics of hardware logic simulations. In contrast, our proposed multi-level parallelization scheme is readily applicable to any discrete event simulation.

B. *Efficient Execution of Parameter Studies*

Simulation cloning [23], [24], [25], [26] reduces the amount of common computations across all simulations of a parameter study. Instead of executing a separate simulation for each parameter set, simulation cloning conducts only a single simulation which represents all possible execution paths within a parameter study. To branch into diverging execution paths, the simulation clones its current state at so-called decision points, i.e. events causing diverging behavior, and subsequently follows each path separately, but in parallel. As a result, the path segment up to the decision point is shared among both resulting simulation paths and thus only computed once. While simulation cloning can significantly reduce the total runtime of a parameter study, the primary drawback of this technique is its complexity and overhead due to state saving and maintenance.

VII. Conclusion and Future Work

We presented a novel parallel discrete event simulation scheme that utilizes the massively parallel processing power of GPUs to enable a cost- and time-efficient execution of large scale parameter studies. In order to efficiently bridge the substantially different processing paradigms of GPUs and discrete event simulations, the proposed scheme exploits two levels of parallelism: i) External parallelism between the independent simulations of a parameter study for generating SIMT-compatible workload, and ii) internal parallelism among the events within each simulation to hide the transfer latency between host- and GPU-memory. Based on a proof-of-concept implementation, we obtain early performance results that underline the viability of the multi-level parallelization scheme.

Nevertheless, as this paper covers only an initial investigation, future efforts target three specific goals. First, we aim for further mitigating the negative performance impact of lockstep execution on simulation performance. The simple padding algorithm sketched in this paper trades memory for performance by introducing artificial gaps between events of different types. In order to better utilize the limited memory resources on the GPU, we will develop and investigate enhanced algorithms for mapping events to multi-processors. Second, we will extend the current prototype implementation to make use of multiple CPUs. In contrast to the current single-threaded implementation, a multi-threaded architecture can improve the simulation performance by means of distributed data management and preprocessing of event batches. Third, we will substantiate our preliminary performance results by porting and thoroughly investigating further real-world simulation models.

Acknowledgments

This research was funded by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

References

[1] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons New York, 1991, vol. 182.

[2] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Whitepaper, V1.1. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[3] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-Accelerated Software Router," in *Proc. of the ACM SIGCOMM conference*, 2010.

[4] A. Varga, "The OMNeT++ Discrete Event Simulation System," in *Proc. of the 15th European Simulation Multiconference (ESM)*, 2001.

[5] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley, "ns-3 Project Goals," in *Proc. of the 2006 Workshop on ns-2: The IP Network Simulator*, 2006.

[6] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, 1990.

[7] K. S. Perumalla, "Parallel and Distributed Simulation: Traditional Techniques and Recent Advances," in *Proc. of the 38th Winter Simulation Conference*, 2006.

[8] O. Puñal, H. Escudero, and J. Gross, "Performance Comparison of Loading Algorithms for 80 MHz IEEE 802.11 WLANs," in *Proc. of the 73rd IEEE Vehicular Technology Conference*, 2011.

[9] J. Yang, Y. Wang, and Y. Chen, "GPU Accelerated Molecular Dynamics Simulation of Thermal Conductivities," *Journal of Computational Physics*, vol. 221, no. 2, pp. 799–804, Feb. 2007.

[10] M. Moeng, S. Cho, and R. Melhem, "Scalable Multi-cache Simulation Using GPUs," in *Proc. of the 19th Intern. Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 2011.

[11] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal, "GPU-based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios," in *Proc. of the 23rd Workshop on Principles of Advanced and Distributed Simulation*, Washington, DC, USA, 2009.

[12] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU Accelerated Monte Carlo Simulation of the 2D and 3D Ising Model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468–4477, Jul. 2009.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[14] D. W. Bauer, M. McMahon, and E. H. Page, "An Approach for the Effective Utilization of GP-GPUs in Parallel Combined Simulation," in *Proc. of the 40th Winter Simulation Conference*, 2008.

[15] B. Zeigler, H. Praehofer, and T. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Complex Dynamic Systems*. Academic Press, 2000.

[16] Z. Xu and R. Bagrodia, "GPU-Accelerated Evaluation Platform for High Fidelity Network Modeling," in *Proc. of the 21st Intern. Workshop on Principles of Advanced and Distributed Simulation*, 2007, pp. 131–140.

[17] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: A fast SystemC Simulator on GPUs," in *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2010.

[18] K. S. Perumalla, "Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)," in *Proc. of the 20th Workshop on Principles of Advanced and Distributed Simulation*, 2006.

[19] H. Park and P. A. Fishwick, "A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation," *Simulation*, vol. 86, pp. 613–628, Oct. 2010.

[20] ——, "An Analysis of Queuing Network Simulation using GPU-based Hardware Acceleration," *ACM Trans. Model. Comput. Simul.*, vol. 21, no. 3, Feb. 2011.

[21] R. M. Fujimoto, "Exploiting Temporal Uncertainty in Parallel and Distributed Simulations," in *Proc. of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[22] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven Gate-level Simulation with GP-GPUs," in *Proceeding of the 46th ACM/IEEE Design Automation Conference (DAC)*, 2009.

[23] M. Hybinette and R. Fujimoto, "Cloning: A Novel Method for Interactive Parallel Simulation," in *Proceedings of the 29th Winter Simulation Conference*, 1997, pp. 444–451.

[24] M. Hybinette and R. M. Fujimoto, "Cloning Parallel Simulations," *ACM Transactions on Modeling and Computer Simulation*, vol. 11, no. 4, pp. 378–407, Oct. 2001.

[25] P. Peschlow, M. Geuer, and P. Martini, "Logical Process Based Sequential Simulation Cloning," in *Proc. of the 41st Annual Simulation Symposium*, 2008, pp. 237–244.

[26] P. Peschlow, P. Martini, and J. Liu, "Interval Branching," in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, 2008, pp. 99–108.