Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

SIMIAN INTEGRATED FRAMEWORK FOR PARALLEL DISCRETE EVENT SIMULATION ON GPUS

Guillaume Chapuis
Stephan Eidenbenz
Nandakishore Santhi
Eun Jung Park

Los Alamos National Laboratory
Bikini Atoll Rd., SM 30
Los Alamos, NM 87545, USA

ABSTRACT

Discrete Event Simulation (DES) allows the modelling of ever more complex systems in a variety of domains ranging from biological systems to road networks. The increasing need to model larger systems stresses the demand for efficient parallel implementations of DES engines. Recently, Graphics Processing Units have emerged as an efficient alternative to Central Processing Units for the computation of some problems. Although substantial speedups can be achieved by using GPUs, writing an efficient implementations of given suitable problems often requires in-depth knowledge of the architecture. We present a new framework integrated in the Simian engine, which allows to make efficient use of GPUs for computationally intense sections of code. This framework allows modellers to offset some or all handlers to the GPU by efficiently grouping and scheduling these handlers. As a case-study, we implement a population activity simulation that takes into account evolving traffic conditions in a simulated urban area.

1 INTRODUCTION AND MOTIVATION

GPUs are perhaps the most challenging among all hardware acceleration technologies that have started to emerge after traditional hardware scaling laws, such as Moore's Law, have broken down. Because they are nearly universally available in commodity hardware and relatively cheap, GPUs have the potential to offer computing power in terms of the traditional floating point operations (FLOP) measure that a few years ago was reserved for actual supercomputers and the inevitable HVAC systems and costs that come with them.

Alas, discrete event simulation appears overall ill-equipped to exploit GPUs efficiently because of a traditional focus on applications, such as communication network simulation, that place a focus on communication across entities and what little computation happens is mostly pointer chasing: neither focus ressembles the GPU paradigm of executing the identical operation on many different data elements. On the other hand, there are DES application areas, particularly agent-based simulation that are better suited for the GPU paradigm.

The DES community realized early on that exploiting the potential of GPUs would be challenging. A number of heroic efforts to include GPUs in DES applications have come out of the DES community. They are mostly application specific and we describe some of them in the related work section.

Our focus in this paper is to develop a general-purpose framework of how to use GPUs in DES and PDES systems. The framework is implemented in the Python-based parallel discrete simulation engine Simian, using PYCuda – the Python version of the GPU programming language Cuda. Our goals for the PDES GPU framework are (i) to make the PDES GPU framework accessible to novice programmers, domain scientists, and other users who do not enjoy writing near machine-level code that is sometimes required for

GPUs, and (ii) to enable a quickly implemented GPU solution for most application domains that achieves about 90% of the benefit that a completely customized and application specific GPU implementation could.

We achieve the general-purpose nature of our GPU simulation framework by making the event handlers the units of computation that can be – but do not have to be – sent to the GPU. The event scheduler in the main loop of the simulation engine provides support for the integration of GPU-handled and CPU-handled events. Thus, the user simply needs to implement an event handler code in PyCUDA if it is to be sent to the GPU.

We illustrate our framework first on a few small examples. We then describe a slightly more involved implementation of an agent-based activity simulation, in which agents optimize their daily schedules, in terms of activity types, such as work, sleep, eating, and leisure, and locations where these activities are to be performed. Based on traffic congestion, these agents may arrive late to their activities and then require re-planning of their schedules. The application includes a road network model with traffic, where congestion occurs if too many agents are trying to use a road link at the same time. Both the routing computations as well as the schedule planning optimization can be done on the GPU. Our GPU version of the application is written in less than 500 lines of Python and PyCuda code, whereas the corresponding CPU-only version is 300 lines of Python code – both easily manageable code complexity. We compare the running time of the two versions, clearly illustrating the benefit of GPUs in a parallel discrete event simulation for similar development times.

1.1 Related Work

Parallel discrete event simulations have already been shown to benefit from the use of GPUs for some types of computations. Perumalla and Aaby (2008) showed that, in some cases, GPUs could be used efficiently for PDES but also raised some concerns about modularity, ease of programmability and reusability. We aim to address some of these limitations in our framework.

Among GPU PDES implementations, a certain number implemented data structures for event queues directly on the GPU as well as methods to sort and insert events in those queues (Park and Fishwick 2009, Sang, Lee, Rego, and King 2013, Wenjie, Yiping, and Feng 2013, Andelfinger and Hartenstein 2014, Ventroux, Peeters, Sassolas, and Hoe 2014). We chose not to implement event queues on GPUs for two main reasons. First, event queues do not fit well with the GPU paradigm: for event queues to provide sufficient data locality for a GPU, they need to be contiguously allocated in memory, which makes inserting new elements less efficient. Second, not all computations are suitable for a GPU - accesses to the file system, inherently sequential tasks, tasks with random memory access patterns or high branch divergence etc.; in order to target a larger spectrum of simulations, we want to allow mixed GPU/CPU computations where only suitable handlers are offset to the GPU. In this context, maintaining and synchronizing event lists on both the CPU and the GPU would prove too cumbersome. Instead, we see the GPU as a co-processor that can be used to relieve the CPU of some of the computations; asynchronous GPU computations allow for concurrent event handling in this hybrid context.

Liu et al. (2014) proposed to use the GPU as a co-processor for network traffic models. Our framework can be seen as a generalization of this approach to any type of simulation where some handlers would benefit from execution on a GPU. In Kunz et al. (2012), the approach draws parallelism from the concurrent execution of multiple independent simulations. We aim to show that a single simulation may also benefit from GPUs. In Perumalla (2009), Aaby, Perumalla, and Seal (2010), authors propose several prototype implementations that demonstrate the tremendous potential offered by modern hybrid clusters, which we also aim to target.

2 METHODS

Conservative approaches to PDES allow the computation of events in a given time window to be executed concurrently. This time window is defined by the current simulation time - typically the timestamp of the last processed event - and a minimum delay or lookahead value. Events, whose timestamps lie within this window can safely be executed in any order and therefore concurrently because they may not generate new events in this time window and will not break causality.

Each individual handler associated to such safe-to-process-in-any-order events may not be enough work to efficiently exploit the parallel capabilities of a GPU. However, when offset simultaneously to the GPU, they have a better chance at providing a sufficient amount of independent tasks for a GPU. However, grouping these handlers and efficiently offsetting their computation to a GPU requires knowledge of the GPU hardware .

In this section, we first describe Simian, our PDES engine. We then describe our Simian integrated framework for offsetting PDES event handlers to one or more available GPUS. Two approaches are possible, depending on the nature of the handler: if the handler provides enough work for multiple threads, they can be scheduled on the GPU as a single computation block - we later refer to such handlers as multithreaded handlers - and asynchronous computations allows our framework to run these handlers simultaneously on the GPU; if the handler only provides work for a single thread, these handlers need to be batched into a GPU blocks - we later refer to such handlers as batched handlers.

2.1 Simian Just In Time Parallel Discrete Event Simulation (JIT PDES) Engine

Simian (Santhi 2015) is a general purpose; process oriented, Just In Time (JIT) compiled, conservative Parallel Discrete Event Simulator (PDES). Simian aims to introduce JIT compilation techniques to the already mature PDES community in order to speed up simulations. Simian simultaneously introduces a structured form of distributed memory parallelism to the JIT community. It is written in a mix of the scripting languages Lua and Python. It extensively uses the C Foreign Function Interface (CFFI) to interface with external code such as the Message Passing Interface (MPI) library. Simian has been written from the start with JIT compilers in mind, so that it works very well with LuaJIT, PyPy, etc., in addition to regular non-JIT capable cPython. The current design of Simian puts heavy emphasis on low complexity for the code base, which makes it easy for beginners to pick up and modify. The event loop and queuing mechanism are minimal; yet offer a completely general-purpose PDES functionality. Simpler event handling code is also considerably more JIT compiler friendly.

With Simian, the user defines a discrete event simulation model at an appropriate level of abstraction for any complex system they wish to evaluate. The PDES model is usually represented as a graph of Entity nodes, which communicate with each other by sending information packets at discrete time epochs. The entities themselves reside on logical processes on individual compute nodes. In Simian, each Entity is represented as a Class. The packets are sent to user-defined methods on entity classes, which are called Service Methods (or just Services). In addition, one can have Processes running locally on the entities, which are special methods called co-routines or micro-threads that can yield and restart at arbitrary states within the method. The processes can further be started and stopped in response to either local or remote requests. Every event is processed at discrete simulated time epochs, making the overall simulation a DES.

Simian can operate in a serial fashion without MPI support; in fact this is usually the preferred operating mode at the initial modeling stage. For scalability however, it is often required that the user-defined model be able to run on distributed memory clusters unaltered. This is especially useful when the simulation model has millions of entity nodes. Simian can make use of MPI libraries to run on distributed memory computers without any change in the user-defined model. This is as easy as changing a user-provided flag to the simulation engine initialization call.

Another notable feature of the Simian engine is the ability to speed up PDES simulations by making effective use of JIT compilers. JIT compilers are able to perform additional optimizations at runtime by virtue of being aware of the dynamic data dependencies, which a static compiler cannot hope to make use of. By relying on proven JIT compiler optimization techniques, Simian event loop and data structures can afford to be significantly simpler. This is because specially coded cases, which can often speed up event

Algorithm 1 Main processing event loop on each LP for CPU only events.

```
1 while now < end_of_simulation:
2     base_time = now
3     while !empty(eventQueue) and eventQueue[0] < base_time + look_ahead:
4         (time, event) = eventQueue.pop() # Get next event
5         now = time # Advance time
6         event.handle() # Call event handler
7     receive_events() # Receive events from other LPs
8     now = MPIallreduce(eventQueue[0]) # Set now to lowest timestamp
```

processing when static-compiled, may simply be auto-discovered at runtime by well-designed tracing JIT compilers. As a result, the JIT compiled version of Simian is typically more than 20 times faster in serial mode over the non-JIT interpreted version, even while written in a scripting language such as Python or Lua.

Simian has been designed to speed up the development of application simulators, by being user friendly and minimalistic in approach. Typical application codes are often much shorter and easier to code in dynamic scripting languages. By allowing for JIT compilation, code written in these dynamic languages can even approach the execution efficiency of statically compiled code in languages such as C and FORTRAN. While only recently developed, Simian has already been successfully used to model various computational physics, social network modeling and performance prediction codes at LANL.

2.2 Offsetting Handlers to the GPU

Each Logical Process (LP) is represented as an MPI rank in Simian. As a conservative PDES engine, Simian allows events, whose timestamps lie in the interval $[now, now + look\_ahead]$ (where *now* is the current simulation time and *look_ahead* is the minimum delay after which an event is able to generate a new event), to be processed in parallel. These events are later referred to as safe events. On a single LP however, these events are processed sequentially - lines 2 to 5 in algorithn 1. Parallelism at this point occurs by running multiple LP in parallel. Once all safe events in a given time window have been processed on all LPs, a rendez-vous is performed between MPI ranks in the form of an MPI allreduce call - line 7 of algorithm 1 - to determine the remaining event with lowest timestamp accross all LPs.

CPU and GPU handlers are treated the same way but in order to overlap computations on both processing units, GPU computations are made asynchronous. This way, handling a GPU event does not block the processing of remaining safe events until the result is obtained. We however need to make sure that GPU computations are completed before the MPI rendez-vous with other LPs as GPU events may in turn generate new events - see line 6 of algorithm 2.

In order to also overlap the computation of multiple GPU events, successive events are queued in different streams. A stream in the Nvidia Cuda terminology is defined as a sequence of operations that need to be processed in order. Two streams can however be computed in parallel if the hardware permits it - ie. if computational resources are available.

Algorithm 2 Main processing event loop on each LP for CPU only events.

```
while now < end_of_simulation:
    base_time = now
    while !empty(eventQueue) and eventQueue[0] < base_time + look_ahead:
        (time, event) = eventQueue.pop() # Get next event
        now = time # Advance time
        event.handle() # Call event handler
    wait_for_gpu_computations()
    receive_events() # Receive events from other LPs
    now = MPIallreduce(eventQueue[0]) # Set now to lowest timestamp
```

Each GPU handler will therefore generate the following sequence in a stream:

1. Asynchronous transfer of input data from host to device;
2. Kernel call;
3. Asynchronous transfer of results from device to host.

A Cuda event is used to determine when results for a given handler are ready to be used. To avoid the overhead of creating a new stream for every event, streams are statically created during an initialization phase. The total number of streams is computed as follows $nb\_sm * max\_block\_per\_sm$, where nb_sm is the number of available streaming processors on the GPU and $max\_block\_per\_sm$ is the maximum number of blocks that can run concurrently on a single streaming processor. This ensures that in the case where each GPU event generates the smallest possible task of one GPU block - with limited number of threads, register use, shared memory use etc. - maximum occupancy of the GPU can potentially be achieved. If multiple GPUs are available, the total number of streams is increased accordingly and context switches occur to allow both GPUs to work simultaneously. Though possible, using multiple GPUs with a single LP is not the preferred approach. When manual load balancing of the GPUs is achievable - ie. the amount of computation required for each task can be reasonably estimated, it is preferable to assign a single GPU to a given LP in order to avoid the overhead of a context switch to another GPU. In the case where heterogenous GPUs are available on a single node, load balancing can still be achieved by assigning different numbers of entities to each GPU.

Multiple LPs can also be assigned to the same GPU. GPU tasks from different LPs will also be able to execute concurrently on the device - see figure 1. The order in which GPU tasks are executed is undefined and determined by the internal scheduler of the device.

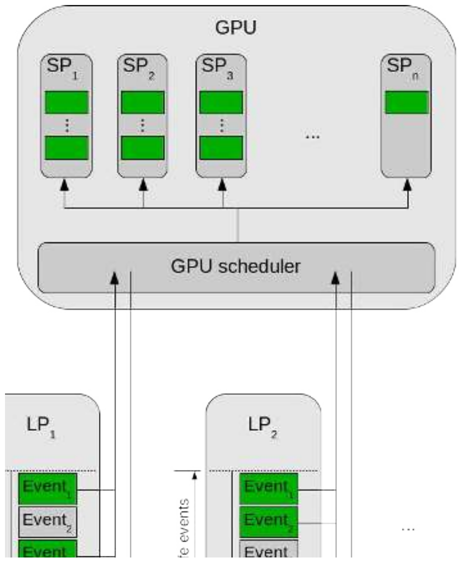

Figure 1: An example of two LPs offsetting some handlers to the same GPU. Events that generate GPU tasks are represented in green in each LP's event list.

2.3 Multithreaded vs. Batched Handlers

Our framework is written in python and uses Pycuda (Klöckner, Pinto, Lee, Catanzaro, Ivanov, and Fasih 2012) for GPU computations. Pycuda is a python package that creates a python interface for Cuda. Pycuda provides a higher level of abstraction than Cuda and yet also gives access to most of the functionalities offered by the low-level driver api from Nvidia. Some functionalities from Pycuda, in particular the elementwise kernel approach, were particularly useful to us in order to aggregate multiple handlers into a single GPU task.

We define two types of GPU handlers. In case where sufficient parallelism can be drawn for the task to offset to the GPU, users can define what we refer to as multithreaded handlers. Though not a requirement, such handlers should provide work for a number of threads at least equal to the size of a warp. In Nvidia Cuda terminology a warp is a group of threads that execute concurrently on a single streaming processor. The current size of a warp on Nvidia GPUs is 32. To achieve the best performance, the number of threads declared in such a handler should be a multiple of the size of a warp.

Algorithm 1 shows an example of a multithreaded handler. In order to define such a handler, the user must first write the Cuda code to be executed on the GPU - lines 1 to 7 of algorithm 1. The user then provides a brief description of the arguments for the GPU kernel - lines 12 to 14. "IN" and "OUT" parameters are used to determine how arguments are to be transferred to and from the device or if they simply need to be allocated. The callback function passed at line 15 is called once the results are available. By default, the block size is set to the size of a warp and the grid is composed of a single GPU block; users can however specify different values for more specific cases.

When no parallelism can be drawn from a single handler, multiple handlers can be batched together to provide sufficient work for the GPU. Algorithm 4 shows an example of a batched handler. Note that the GPU code defined at lines 3 to 7 only provides work for a single thread. When describing the parameters - lines 12 to 14 - the "element" attribute is used to determine whether the argument should be aggregated or

Algorithm 3 Example of a multithreaded handler. This example simply squares each item of an array *source* and writes the results in array *dest*.

```
src = """
__global__ void square_them(float *a, float *dest)
{
    const int i = threadIdx.x;
    float tmp = a[i];
    dest[i] = tmp * tmp;
} """

def handler():
    a = numpy.random.randn(size).astype(numpy.float32)
    dest = numpy.zeros_like(a)
    a = {'name':"a",'data':a,'IN':True,'OUT':False}
    dest = {'name':"dest",'data':dest,'IN':False,'OUT':True}
    args = [a, dest]
    send_MT_handler(args, src, "square_them", call_back)

def call_back(results):
    # This function is called when results are available
```

Algorithm 4 Example of a batched handler. This example takes the sin of each element of an array multiple times.

```
n_iter = numpy.int(10000)
decl = "int n_iter, float *a"
src = """
for(int n = 0; n < n_iter; n++)
{
    a[i] = sin(a[i]);
}"""
kernel_name = "gpu_sin"""

def handler():
    a = numpy.random.rand()
    a = {'name':"a",'data':a,'IN':True,'OUT':True,'element':True}
    niter = {'name':"n_iter",'data':n_iter,\
        'IN':True,'OUT':False,'element':False}
    args = [niter,a]
    send_batched_handler(args,decl,src,kernel_name,call_back)

def call_back(results):
    # This function is called with each data item
    # when results are available
```

is simply a scalar that should hold the same value for each thread. In the case of algorithm 4, "n_iter" is a scalar whereas "a" is to be aggregated. Each handler provides a single piece of data for array "a"; once a sufficient number of calls to this handler have been made (or the last safe event has been processed), values for array "a" are aggregated and transfers and computations are asynchronously launched. Once the results are available, each callback function is called with its associated data item.

3 CASE-STUDY: POPULATION ACTIVITY SIMULATION WITH ROAD TRAFFIC

In order to test our GPU framework for PDES, we provide a case-study for agent-based activity simulation that also includes a road network for a simulated urban area. Each simulated agent plans his daily activities as a combination of four activity types: eating, sleeping, working and leasure. Each agent is randomly assigned a home location and a work location in the graph representing the urban area. Agents always sleep at their home location and always work at their work location. Eating and leasure activities are performed at random locations.

3.1 Activity Planning

Activity planning is presented as an optimization problem. The overall score of planning a given activity is the sum of a positive effect for performing a given activity, represented as a utility function, and a negative effect for not performing other activities in the form of a priority function. These functions depend on the time these activities have last been performed and are described in Galli et al. (2009).

A daily schedule is composed of four activities chosen among the four activity types. Each activity is assigned a duration. Optimizing a daily schedule is done using a simple genetic-like algorithm - see algorithm 5. We first generate a population of random schedules - line 5. Then, at each generation, each random schedule undergoes two types of mutations. The first type of mutation is swapping two random activities and their associated durations. the second type of mutation consists in changing a random activities and assigning it a new random duration. At the end of a generation iteration, a new generation is created consisting of copies of the best schedule found so far.

Algporithm 5 is implemented on the GPU by having each GPU thread handle an individual in the schedule population. Communication between threads is performed using on device shared memory. once an agent has planned a new schedule, the first activity is performed. A new planning event is created at simulation time $now + transport + duration$, where now is the current simulation time, $transport$ is the time it takes to travel from the agent's current location to the location for the activity - this time takes traffic conditions into account - and $duration$ is the duration of the planned activity. After each activity, agents thus replan their schedules as varying traffic conditions may have affected their previous schedules.

3.2 Road Traffic Simulation

The simulated urban area is represented as a random planar graph, where vertices correspond to possible locations and edges correspond to road links. Each road link possesses a random weight that correpsonds to the time it takes to travel through it under optimal traffic conditions and a random capacity to describe how much it is affected by potential traffic; ie. road links with higher capacity can support a larger number of cars. Each LP has a traffic controller entity that collects traffic information for agents belonging to the same LP. Road traffic conditions are updated at regular simulation time intervals by a single traffic controller. The main traffic controller computes on the GPU the All-Pairs Shortest Path problem in the planar graph using the FLoyd-Warshall algorithm (Floyd 1962). Each road link receives a weight equal to its initial weight under optimal traffic conditions plus an additional weight depending on the number of cars that have used the road link in the previous interval and the capacity of the road link. The results of this operation are a routing matrix used to retrieve the shortest path between any two locations in the graph and a distance matrix to retrieve the transport duration of such a path. These results are transmitted to all LPs via event messages.

Algorithm 5 Algorithm for optimizing a daily schedule.

```
INPUT: Timestamps at which activities were last performed
   (last_performed), size of the schedule population (pop_size),
   number of generations to run (nb_generations).
def plan_activities(last_performed, pop_size, nb_generations):
   schedules = generate_random_schedules(pop_size)
   for i in range(nb_generations):
      for each schedule in schedules:
         score = evaluate(schedule)
         new_schedule = swap_2_activities(schedule)
         new_score = evaluate(new_schedule)
         if new_score > score:
            schedule = new_schedule
            score = new_score
         new_schedule = change_random_activity(schedule)
         new_score = evaluate(new_schedule)
         if new_score > score:
            schedule = new_schedule
            score = new_score
      schedules = copy_best(schedules)
   return best(schedules)
```

Whenever an agent travels between two locations, the number of cars on each road link composing the current shortest path between these locations is incremented. This information is sent and reset at regular intervals from traffic controllers on all LPs to the main traffic controller. This way, the shortest path between two given locations may change at different time intervals depending on traffic conditions.

4 EXPERIMENTS AND RESULTS

In order to test both our GPU framework and our proposed case-study, we design three experiments. All runs simulate a total of 48 hours, with traffic updates every simulated half-hour; activity planning handlers generate a population of 32 schedules over a 100 generations. Experiments 1 and 2 are executed on three nodes with an Intel Xeon E5-2660 @2.2GHz and an NVIDIA Tesla K40c each. Experiment 3 is executed on a cluster of nodes with an Intel Xeon E5-2670 8C @2.6GHz and two NVIDIA 2090 cards each.

The first experiment aims at assessing the influence on the total execution time of the size of the graph representing the simulated urban area. In this experiment, the population size if fixed to 500 agents and the graph size varies from 256 to 1024; a comparaison is given with a CPU implementation of the same case-study. Note that neither the CPU nor GPU implementations have been carefully optimized; the given speedups should therefore not be interpreted as a fair comparison between a CPU and a GPU. Instead, we aim to show that with the same programming time for both implementations, our framework allows us to obtain a substantial speedup for a given problem that fits the GPU paradigm.

Figure 2 shows the run times for the first experiment. Updating traffic induces an unbalanced task that consists in computing the shortest paths in the entire graph. We can see that the CPU version is most affected by the change in graph size. Increasing the graph size renders traffic updates computations more dominant on the CPU. Since these computations are performed by a single CPU thread, remaining resources are mostly idle for larger graphs.

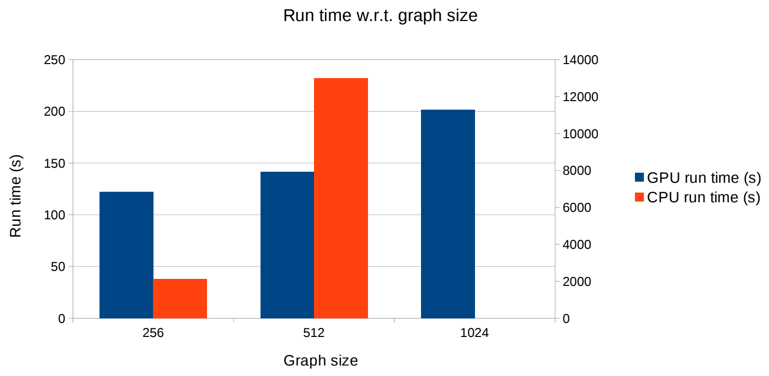

Figure 2: Results for the first experiment. GPU run times are to be read on the left y-axis; CPU run-times on the right y-axis. The CPU run time for the largest urban area is not given here as computations exceeded the time limit.

In a second experiment, we aim to assess the influence on the total execution time of the population size. In this second experiment, the graph size is fixed to 1024 and the population increases from 500 to 4000. Figure 3 shows the results for this experiment. Results indicate that optimal performances are obtained with a population size of 4000 and more, as events per second reaches a plateau. This indicates that about 1300 agents per GPU (4000 for three GPUs) are required to fully exploit the devices. Finally, a third experiment aims at determining the scalability of our GPU example. For this last experiment, we compute a fixed size problem of 32000 agents with a graph size of 1024 on an increasing number of nodes ranging from 6 to 16. Figure 4 shows the results for this experiment. This simulation scales well up to 12 cluster nodes and thus 24 GPUs as this experiment was run on a cluster of nodes each equipped with two devices. Although the GPUs used in experiments 2 and 3 are different, both experiments indicate that performances are suboptimal when the number of agents per GPU is less than about 1300. Simulations with larger populations would most likely benefit from additional GPUs.

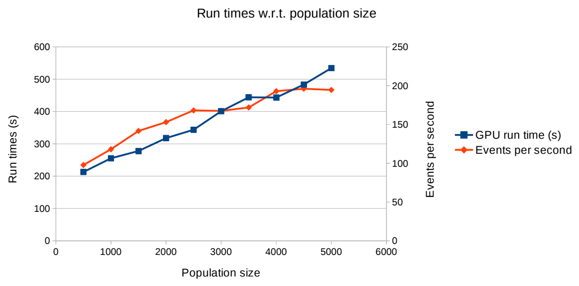

Figure 3: Evolution of run times and events per second with increasing population size. The average number of handled events per second (in orange) reaches a plateau after 4000 agents.

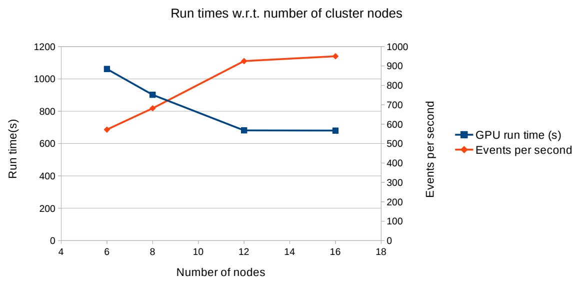

Figure 4: Evolution of run times and events per second with increasing number of cluster nodes. This simulation scales up to 12 cluster nodes (ie. 24 GPUs on this cluster).

Figure 5 shows traffic conditions at various simulated times, from lightest (left) to heaviest (right) traffic. Congestion is indicated by the color of road links - from green, least congested, to red, heavy traffic.

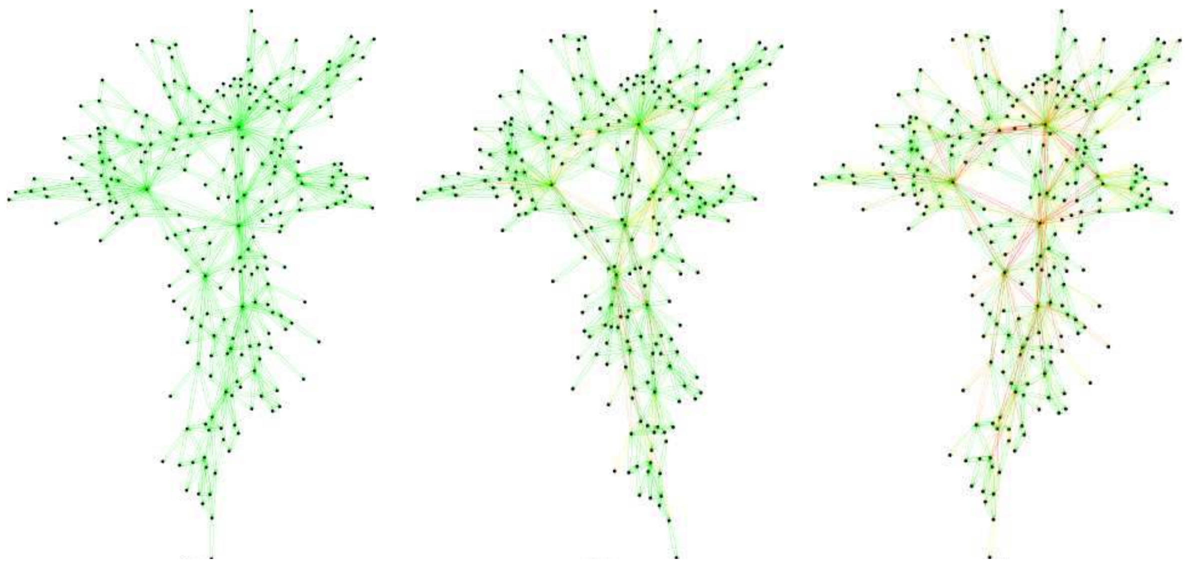

Figure 5: Traffic conditions at various simulation times (12am left, 10:30am center, and 9am right). The color of a road link indicates the amount of traffic in the previous time interval (from lightest traffic in green to heaviest traffic in red). The slight variation in vertex positioning is induced by the plotting algorithm used.

5 CONCLUSION

We proposed a new framework for PDES on GPUs. This framework allows modellers to quickly exploit the potential of GPUs for PDES with event handlers that fit the GPU paradigm. Two types of approaches are offered: an approach where parallelism can be extracted from a single handler and a second approach where sequential handlers can be aggregated to fit the SIMT GPU paradigm. Our agent-based simulation case-study shows that in very few lines of code, complex simulations can be developed and can benefit from computations on GPUs. The comparison between a GPU version and a CPU version of our case-study shows that for the same programming time, our GPU version offers a substantial speedup.

Our future work includes improving our framework in order to allow for a broader range of simulations to exploit GPU computations as well as offer easier ways to automatically balance tasks between devices.

REFERENCES

Aaby, B. G., K. S. Perumalla, and S. K. Seal. 2010. "Parallel Agent-Based Simulations on Clusters of GPUs and Multi-Core Processors". Technical report, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences.

Andelfinger, P., and H. Hartenstein. 2014. "Exploiting the Parallelism of Large-Scale Application-Layer Networks by Adaptive GPU-Based Simulation". In *Proceedings of the 2014 Winter Simulation Conference*, 3471–3482. IEEE Press.

Floyd, R. W. 1962. "Algorithm 97: Shortest Path". *Communications of the ACM* 5 (6): 345.

Galli, E., L. Cuéllar, S. Eidenbenz, M. Ewers, S. Mniszewski, and C. Teuscher. 2009. "ActivitySim: Large-Scale Agent-Based Activity Generation for Infrastructure Simulation". In *Proceedings of the 2009 Spring Simulation Multiconference*, 16. Society for Computer Simulation International.

Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. 2012. "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". *Parallel Computing* 38 (3): 157–174.

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 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, 23–32. IEEE Computer Society.

Liu, J., Y. Liu, Z. Du, and T. Li. 2014. “GPU-Assisted Hybrid Network Traffic Model”. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of Advanced Discrete Simulation*, 63–74. ACM.

Park, H., and P. A. Fishwick. 2009. “A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation”. *Simulation*.

Perumalla, K. S. 2009. “Switching to High Gear: Opportunities for Grand-Scale Real-Time Parallel Simulations”. In *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, 3–10. IEEE Computer Society.

Perumalla, K. S., and B. G. Aaby. 2008. “Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs”. In *Proceedings of the 2008 Spring simulation multiconference*, 116–123. Society for Computer Simulation International.

Sang, J., C.-R. Lee, V. Rego, and C.-T. King. 2013. “A Fast Implementation of Parallel Discrete-Event Simulation on GPGPU”.

Santhi, N. 2015. “Simian Just In Time Parallel Discrete Event Simulation Engine”. Technical report, Los Alamos National Laboratory. LANL CODE-2015-48 and CODE-2015-50. Simian was funded under the LDRD DR 20150098.

Ventroux, N., J. Peeters, T. Sassolas, and J. C. Hoe. 2014. “Highly-Parallel Special-Purpose Multicore Architecture for SystemC/TLM Simulations”. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, 250–257. IEEE.

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

AUTHORS BIOGRAPHIES

GUILLAUME CHAPUIS is a post-doctoral Research Associate with the Information Sciences Group (CCS3) at Los Alamos National Laboratory, New Mexico, United States of America. He holds a PhD degree in computer science from ENS Cachan (France) and a computer engineering degree from INSA Rennes (France). His research interests include Parallel Discrete Event Simulation, graph theory, High Performance Computing, General-Purpose Graphics Processing Units and bioinformatics. His email address is gchapuis@lanl.gov.

STEPHAN EIDENBENZ is a computer scientist at Los Alamos National Laboratory. He leads research projects in cyber security, computational codesign, performance prediction of super computers, and process modelling spanning a range of government and commercial sponsors as well as internal research grants. Stephan obtained his PhD from the Swiss Federal Institute of Technology, Zurich (ETHZ) in Computer Science. He has made research contributions in many areas of computer science, including cyber security, computational codesign, communication networks, scalable modelling and simulation, and theoretical computer science. His email address is eidenben@lanl.gov.

NANDAKISHORE SANTHI is a computer research scientist with the Information Sciences Group (CCS3) at Los Alamos National Laboratory, New Mexico, United States of America. He holds a PhD in Electrical and Computer Engineering from the University of California San Diego. His areas of research interests include parallel discrete event simulation, performance modeling of HPC systems, applied mathematics, communication systems and computer architectures. He has also worked for the semiconductor industry, designing communication and memory chips. His email address is nsanthi@lanl.gov.

EUN JUNG PARK is a Postdoctoral Research Associate of Information Sciences (CCS-3) at Los Alamos National Laboratory. She holds a Ph.D. degree in Computer Science from the University of Delaware. Her research interests include parallel discrete event simulation, performance prediction, compiler optimizations, and machine learning. Her email address is ejpark@lanl.gov.