Experiences with Implementing Parallel Discrete-event Simulation on GPU

Janche Sang
Cleveland State University, J.SANG@csuohio.edu

Che-Rung Lee
National Tsinghua University

Vernon Rego
Purdue University

Chung-Ta King
National Tsinghua University

Experiences with implementing parallel discrete-event simulation on GPU

Janche Sang · Che-Rung Lee ·
Vernon Rego · Chung-Ta King

Abstract Modern graphics processing units (GPUs) offer much more computational power than recent CPUs by providing a vast number of simple, data-parallel, multithreaded cores. In this study, we focus on the use of a GPU to perform parallel discrete-event simulation. Our approach is to use a modified service time distribution function to allow more independent events to be processed in parallel. The implemen-tation issues and alternative strategies will be discussed in detail. We describe and compare our experience and results in using Thrust and CUB, two open-source paral-lel algorithms libraries which resemble the C++ Standard Template Library, to build our tool. The experimental results show that our implementation can be two orders of magnitude faster than the sequential simulation for large-scale simulation models.

Keywords Parallel simulation · Discrete-event simulation · GPU · CUDA · Thrust/CUB

1 Introduction

Discrete-event simulation (DES) is a widely used technique that allows an analyst to study the dynamic behavior of a complex system [3]. DES exploits a computer to model a system stochastically at discrete points in simulated time. A simulation program operates on a model's state variables during each of a sequence of time-ordered events and schedules future events during such processing. However, simulation is usually computationally intensive and time-consuming. Typical simulation applications often execute for hours or even days. Therefore, exploiting the availability and the power of multiprocessors to speed up the simulation execution is of considerable interest.

Parallel discrete-event simulation (PDES) attempts to speed up a simulation's execution by partitioning the simulation model into several distinct simulation objects, each of which has its own event set and is executed by a *Logical Process* (LP) on a different processor. That is, the traditional PDES is usually executed using the multiple-instruction, multiple-data (MIMD) style on a cluster of workstations or on a multicore machine. To guarantee the distributed events will be executed in an appropriate order, two main types of synchronization mechanisms among LPs have been proposed: conservative and optimistic [4]. Conservative mechanisms do not allow an LP to process an event until it is certain that causality violation will not occur. This means that an LP will not receive an event with a smaller timestamp than its current clock from another LP. However, An LP may wait for events that never arrive. Therefore, LPs may send null messages to other LPs to avoid deadlocks [2]. Optimistic mechanisms ignore inter-process synchronization issues, but make compensations by performing rollbacks to a checkpointed consistent state when a causality error occurs [7]. This requires periodic state saving of the simulator.

With the advance of graphics hardware technology, programming and executing general applications on GPUs is more feasible [8]. In recent years, the GPU with hundreds or even thousands of processing cores has been used for improving the performance of various computational intensive applications [9,12,18]. Figure 1 shows an overview of the modern GPU architecture [14]. It consists of a scalable number of streaming multiprocessors (SMs) and each SM contains a group of streaming processors (SPs). The kernel function, which is executed on the device, is composed of a grid of threads to be executed on the SPs. More precisely, a grid is divided into a set of blocks and each block contains multiple warps of threads. Blocks are distributed evenly to different SMs to run. The warp is the scheduling unit, and there are 32 threads in a warp. These 32 threads are executed using the single-instruction, multiple-data (SIMD) style. The GPU device has its own off-chip device memory (i.e., global memory). Therefore, data need to be transferred from the host CPU before executing the kernel function. Furthermore, shared memory and registers in a SM are on-chip memory which can be accessed much faster than the global memory. They are per-block resources and will be released when all the threads in the block finish execution.

In this paper, we focus on the use of a GPU to perform parallel discrete-event simulation. To meet the SIMD processing style on GPU, we restructure the sequential simulation implementation. That is, instead of using a priority queue to maintain the order of events, we use the parallel reduction to find the smallest event timestamp $min\_timestamp$. To allow more events to be processed in parallel, our approach is to use a modified service time distribution function with a predefined interval d. This guarantees that the events with the timestamp less than or equal to $min\_timestamp + d$ are independent of each other. These events can be extracted by using the parallel selection method and then be processed simultaneously without any causality errors. In other words, our method can be treated as a conservative approach from certain

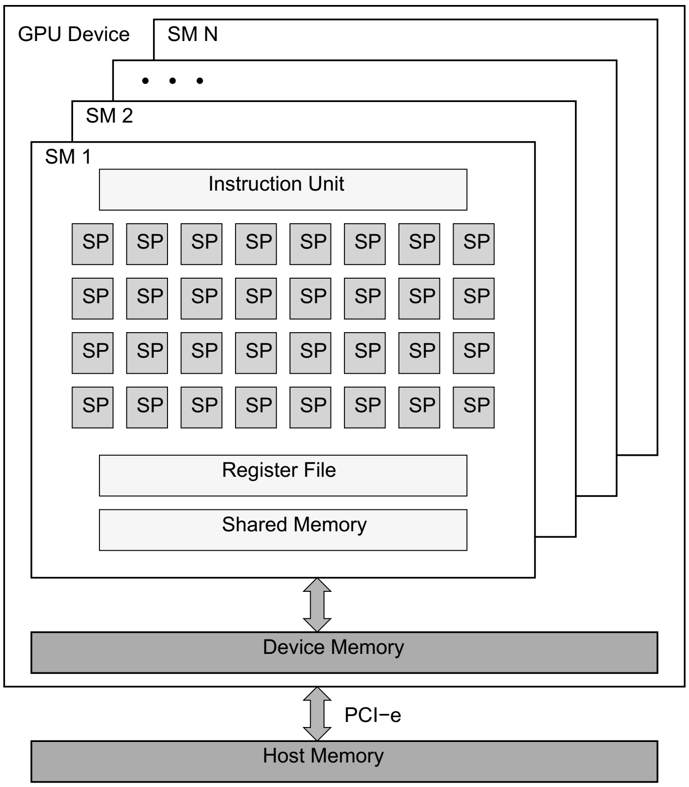

Fig. 1 GPU architecture

viewpoint. Furthermore, the event handling routines also need to be modified so the events can be processed in the SIMD style.

Our implementations are realized by using the Thrust [6] and the CUB [13] packages on the NVIDIA Compute Unified Device Architecture (CUDA) platform. Thrust and CUB are CUDA libraries of parallel algorithms with user-friendly interfaces resembling the C++ Standard Template Library (STL). They hide the details of low-level CUDA function calls and provides highly optimized implementation of standard algorithms, such as searching, sorting, reduction and compaction, which greatly enhances developer productivity. Therefore, using Thrust/CUB, an application that runs on a GPU can be more readable and concise. Furthermore, CUB utilizes the warp shuffle intrinsics [5] and the atomic operations [10] to make the execution of the library routines much faster. In this paper, we describe our experience using Thrust and CUB and compare their performance in our implementations. So other researchers can benefit from our experience for improving the performance of PDES with GPUs.

The organization of this paper is as follows. Section 2 describes the related work in parallel simulation and the old simulation algorithm on GPU. Section 3 presents our improved implementation strategies. In Sect. 4, the experiments and the results for performance evaluation are presented. We give a short conclusion in Sect. 5.

2 Related work

In the area of practical parallel simulation, two apparently orthogonal streams of effort have developed over the past decades. The *replication*-based effort entails the natural parallelism of statistical sampling by executing several replications of a sequential simulation on different processors independently. The *EcliPSe* toolkit described in [20,23] has proved to be a very successful system for replication-based simulations. The *distribution*-based effort emphasizes functional decomposition of a model across processors. Examples of systems supporting distributed simulation include ModSim [24], Sim++ [1], *ParaSi* [21] and *ParaSol* [11]. An inherent difference between the two approaches is that *replication* exploits statistical sampling to speed up the generation of multiple (typically, but not necessarily independent) sample paths, while *distribution* exploits model partitioning to speed up the generation of a single sample path. In this paper, we focus on the *distribution*-based approach.

Because of its massively data-parallel computing power, GPU has been used by more and more researchers for simulating large-scale models over the past few years. For example, a discrete-event simulation of heat diffusion performed on GPU can be found in [17]. The algorithm selects the minimum among all update times and uses it as a timestep to perform a synchronous update of state across all elements in the grid. Another work reported in [27] focuses on a high-fidelity network modeling and uses the GPU as a coprocessor to distribute computation-intensive workloads. Our approach is similar to the work in [15,16] which develop an event clustering and execution scheme based on the concept of approximation time. In these two papers, the former illustrates practical implementation strategies, while the latter presents an analysis of the approximation error in their algorithm. Our algorithm borrows some ideas from their algorithm for updating service facilities.

Note that the work in [15,16] introduced a time-synchronous/event algorithm using a time interval instead of a precise time. Figure 2 shows the pseudo-code of the old algorithm in [15]. To achieve a higher degree of parallel processing, their algorithm clusters events within a time interval. That is, the simulation time is divided into many fixed-sized time slots which is similar to the time-based simulation, a methodology usually used for continuous physics/dynamics simulation [25]. However, unlike the pure time-based simulation which advances the time slot by slot, the old algorithm directly moves the clock to the slot which contains the event with the minimum times-

```
while ( current_time  <  simulation_time ) {
   min_timestamp = find_min(future_event_set);
   current_step = the smallest multiple of time interval greater than or
                  equal to min_timestamp;
   parallel for each event e in future_event_set
       if (the timestamp of e <= current_step) {
          extract e from future_event_set;
          process e and generate new events into future_event_set;
       }
   end parallel for
}
```

Fig. 2 Pseudo-code of the old algorithm in [15]

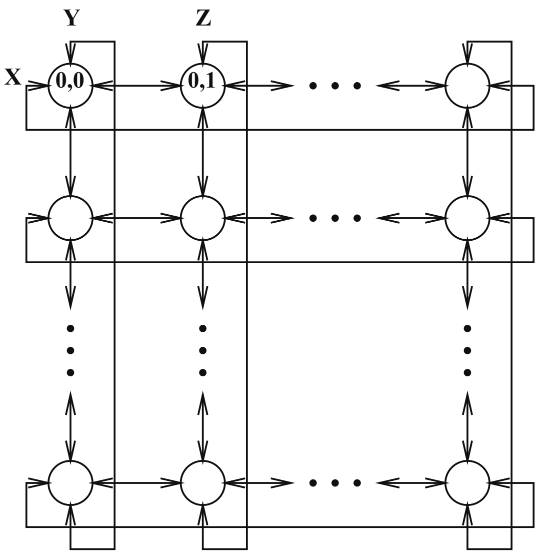

Fig. 3 Torus queueing network

tamp in the future event set. This could reduce the execution time if a slot doesn't have any events to be processed. Therefore, as shown in Fig. 2, all events with timestamps less than or equal to the time slot boundary (i.e., the smallest multiple of time interval greater than or equal to the minimum timestamp) can be extracted from the future event set and then be executed.

However, the old algorithm cannot be directly used in the precise-time PDES. Note that the PDES should handle the events in a causal consistent way exactly as the sequential DES does. Let us use the simulation of a torus queueing network as an example. As shown in Fig. 3, a torus consists of service facilities arranged in a two-dimensional mesh. Each facility has four outgoing and four incoming channels. When a token arrives at a service facility, it gets the service for some random amount of time if the server is idle. Otherwise, the token has to wait in the server's waiting queue. After being served, the token moves to one of the four neighbors. For simplicity, we assume that the probabilities of a token leaving a facility on any given outgoing channel are equal (i.e., 0.25).

Assume that there are three tokens X, Y and Z in the torus network (see Fig. 3). The token X and the token Y enter the service facility[0,0] at time 0.6 and 0.7, respectively. The token Z will arrive at the facility[0,1] at time 0.9. Also assume that the service time for the token X being served at the facility[0,0] is 0.2. Using the old algorithm with the time interval $d = 0.5$, all of these three events can be processed in parallel at the time 1.0 (i.e., the smallest multiple of d which is greater than 0.6). The scenario is depicted in Fig. 4a. Note that an event E in Fig. 4 represents a combined departure/arrival event.

Since both X and Y enter the facility[0,0], the old algorithm uses the original timestamps to keep the causal order. That is, the token X will get the service immediately, while the token Y will stay in the waiting queue. However, if we use the original timestamp for the token X to calculate its departure/arrival time, the token X should enter the facility[0,1] at time 0.8. As shown in Fig. 4b, a causality error occurs because

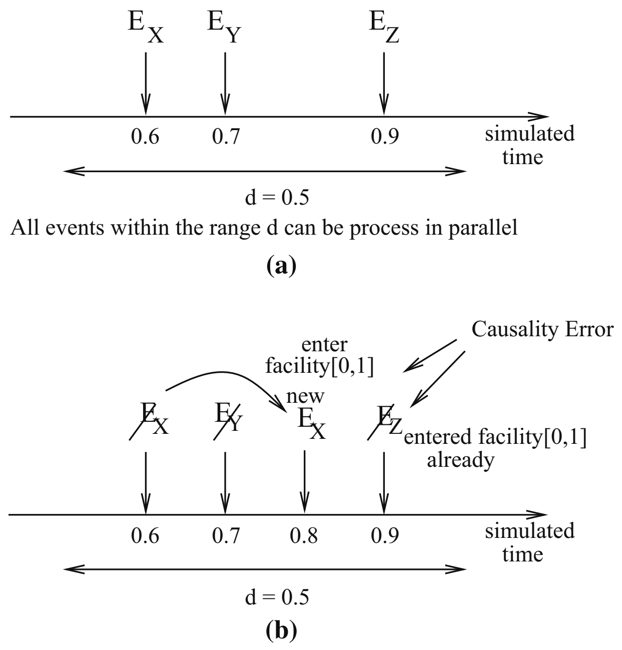

Fig. 4 Causality error. **a** Before parallel processing and **b** after parallel processing

Table 1 Number of causality errors with varying the number of facilities

Number of facilities	64×64	128×128	256×256	512×512	1024×1024
$d = 0.25$	363	1122	3665	12,101	36, 466
$d = 0.5$	1177	3738	11, 450	36,184	124, 951

the token Z, with the arrival time at 0.9, has been served in the facility[0,1] already. Therefore, the old algorithm cannot process the events exactly as the causal order in the sequential DES. We also conducted an experiment to verify this. We recorded the last arrival time for each service facility. If the timestamp of a new arrival is smaller than the last arrival time, a causality error is detected. Table 1 shows that the larger the interval, the more causality errors occurred in the simulation.

3 The improved implementations

Our algorithm for PDES is based on the exact timestamp order, not on the approximation time as in [15,16]. The first issue we need to deal with is preventing the potential causality error occurred as discussed in the previous section. To solve the problem, we let the service time for each token contain the constant time interval *d* and subtract the constant *d* from the mean service time in the invocation of the service time distribution function. More precisely, if the service time is exponentially distributed, we change the expression of calling exponential distribution function from

$$\texttt{expon(M)}$$

to

$$\texttt{expon(M} - \texttt{d)} + \texttt{d}$$

where M is the mean service time. Note that in the modified formula, the mean service time is still M, but the service time for any token is always greater than d. Therefore, the aforementioned causality error will not occur. For example, the timestamp of the new departure/arrival event for the token X in Fig. 4 will be at least $0.6 + d = 1.1$ which is after the token Z enters the facility[0,1].

Another advantage of using the modified formula for the service time is that the full time interval can be used to cluster events for parallel processing. Our algorithm extracts any event which has the timestamp less than or equal to

$$\texttt{minimum\_timestamp} + \texttt{d}$$

and hence will include more events than the old algorithm. The more the parallel events be executed, the faster the program runs. For example, assume that $d = 0.5$ and the minimum timestamp in the future event set is 1.42, the events with the timestamp between 1.42 and 1.50 (i.e., the smallest multiple of 0.5 which is greater than 1.42) can be processed concurrently in the old algorithm. The effective range size is only 0.08. Using our algorithm, the range is between 1.42 and 1.92. In general, giving the same interval d, the average effective range size of the old algorithm is half of the range size in our algorithm. However, our method still has its disadvantage. The biased distribution function will yield a small difference as compared with the result of using the original distribution function. The empirical evaluation of the difference will be reported in the next section.

Figure 5 shows our implementation on the host using the Thrust library. As mentioned before, Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). One of the reasons we use Thrust is that it abstracts away the details of low-level CUDA function calls, such as cudaMalloc, cudaMemcpy and kernel launch. For example, it provides the device pointer which allows programmers access the device memory without calling cudaMemcpy explicitly. The `*mptr` in Fig. 5 is such a case. For interoperability with C, the device pointer can be converted into a raw pointer, and then, the users can use it as a parameter to launch a CUDA C kernel.

As shown in Fig. 5, the host launches five kernel calls on GPU: `sim_init`, Thrust `min_element`, thrust `copy_if`, `process_departure` and `process_arrival`. That is, after initialization of the simulation environment, it starts simulation by finding the event with the minimum timestamp. Then, it advances the clock and selects the parallel events within the interval d based on the minimum timestamp. Next, it will process the selected departure events and then the arrival events. This procedure will be repeated until the simulated clock reaches the limit. Note that our implementations are motivated by the support of the fancy functions, such as the warp shuffle, warp voting and atomic operations, from the newer NVIDIA

```
thrust::device_ptr<FACTYPE> all_fac = thrust::device_malloc<FACTYPE>(N*N);
FACTYPE *facp = thrust::raw_pointer_cast(all_fac);

thrust::device_ptr<TOKENTYPE> all_tkn = thrust::device_malloc<TOKENTYPE>(1);
TOKENTYPE *tknp = thrust::raw_pointer_cast(all_tkn);

thrust::device_ptr<int> key = thrust::device_malloc<int>(N*N);
  int *kp = thrust::raw_pointer_cast(key);

thrust::device_ptr<float> events = thrust::device_malloc<float>(N*N);
float *evp = thrust::raw_pointer_cast(events);

thrust::device_ptr<int> chsn = thrust::device_malloc<int>(N*N);
int  *chp = thrust::raw_pointer_cast(chsn);

...

int gridSize = (N*N + blocksize-1) / blocksize;
// initialize facilities, tokens, events, and let key[i]=i used in copy_if
sim_init<<<gridSize,blocksize>>> (facp,tknp,ep,kp);

while (clock < SIMTIME ) {

   // find the event with the minimum timestamp
   thrust::device_ptr<float> mptr=thrust::min_element(events, events+N*N);

   clock = *mptr + d;

   // select the parallel events with the timestamp <= clock
   thrust::device_ptr<int> chsn_last=thrust::copy_if(key, key+N*N, events,
                                                      chsn,leq(clock));
   int chsn_num = chsn_last - chsn; // number of chosen events

   gridSize = (chsn_num+blocksize-1)/blocksize;

   // process the departure events
   process_departure<<<gridSize,blocksize>>> (facp,tknp,evp,chp,chsn_num);

   // process the arrival events
   process_arrival<<<gridSize,blocksize>>> (facp,tknp,evp,chp,chsn_num);
}
```

Fig. 5 Implementation using the Thrust library

architectures since Kepler. More detailed implementation issues about the kernel functions on GPU are discussed below.

3.1 Finding the minimum timestamp

Note that we used the Thrust library in our first implementation because it provides the functions we needed. So we did not need to write our own, and hence, the programming effort could be saved greatly. Furthermore, these functions have been tuned particularly for the NVIDIA GPU architecture. For example, the code used in the old algorithm

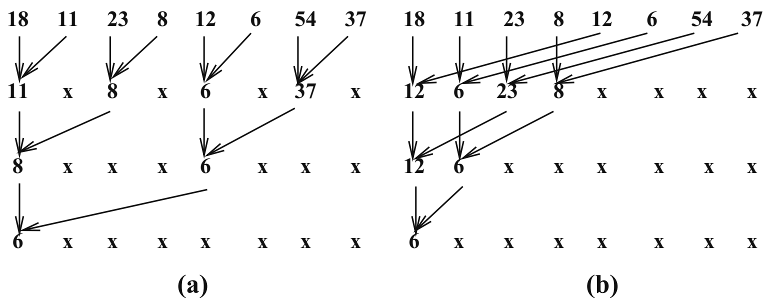

Fig. 6 Parallel reduction steps using **a** interleaved addressing and **b** sequential addressing

[15] to find the minimum element based on the parallel reduction method is out of date and inefficient. The general ideas of how the parallel reduction steps are performed in the old algorithm and in the Thrust library are shown in Fig. 6a, b, respectively. The former uses the interleaved addressing approach, in which the distance between the two elements to be compared in the array is doubled for each reduction step. The latter adopts the sequential addressing approach, in which the distance is reduced half in every step. In theory, there is no difference between these two methods because both need $O(\log n)$ steps to find the minimum value among n elements. In practice, the latter is bank conflict-free and takes advantage of the CUDA memory coalescing within a warp to improve performance [8].

Recently, we found that the NVIDIA CUB library also provides a function which can find the minimum value in an array. Note that CUB is similar to the Thrust library, but its abstractions are slightly lower level than Thrust. That is, unlike Thrust which supports several device backends such as CUDA, TBB and OpenMP, CUB is specific to CUDA C++ and its interfaces explicitly accommodate CUDA-specific features. Furthermore, CUB utilizes the fancy warp shuffle functions to perform parallel reductions. For example, the function __shfl_up(int v, int d) will let the lane k thread read the variable v held by the lane $(k - d)$ thread. Note that the lane is the thread's index within a warp, ranging from 0 to 31. For another example, a thread which invokes the function __shfl(int v, int srcLane) will get the value of the variable v held by the thread with the lane ID srcLane. That is, if every thread in the warp copies from the same source lane, it behaves the same as broadcasting. These shuffle functions enable threads within the same warp to exchange variables (i.e., registers) without using shared memory, so the performance can be improved. Therefore, our revised implementation adopts the `Min()` function in CUB to find the smallest event timestamp.

3.2 Selection of the parallel events

Another important implementation issue is how to extract the aggregated events from the future event set. It is straightforward that the comparison of each event's timestamp with the interval's upper bound can be done in parallel on each thread. The issue here is the management of the chosen events to run after the comparison. The way how this is implemented is not discussed in [15]. The simplest approach is to let the thread

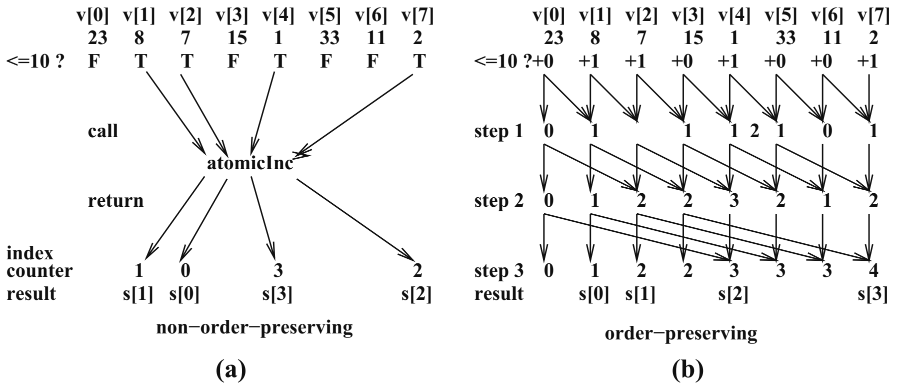

Fig. 7 Selection of events using **a** atomicInc, **b** Thrust copy_if (parallel prefix sum)

discontinue to run if the selection criteria is not met, while the thread which gets TRUE in the comparison will continue to execute the event, i.e., handling the departure/arrival, updating the facility and generating new events. However, based on our experience, only a small portion of events will be selected in a large-scale simulation. Hence, this approach will cause many threads idle and only two or three threads in a warp can run.

The better approach is to use two phases of kernel launches. In the first phase, the parallel events are collected into an array which stores the identifiers of the selected events. Therefore, the number of the chosen events can be known and then we can run that many of threads to execute the events in the second phase. For collecting the chosen events into an array, each thread needs to figure out the correct position to be stored in the array. There are two implementation methods for this. One method is that we can use an index counter which will be incremented by one for each newly selected event. Since the index counter will be shared and accessed by many threads, the addition has to be an atomic operation. This can be done by using the CUDA `atomicInc()` function. However, due to the high degree of the competition for the mutex lock, this method cannot guarantee the input–output ordering will be preserved for the selected elements. Another method, which is used in the Thrust `copy_if` function, adopts the list ranking algorithm with the parallel prefix sum operation [26] to obtain the position of each selected event. As shown in Fig. 7, the selected elements keep the same relative order as in the input.

Note that there is a hybrid approach which utilizes both of the atomic and the parallel prefix sum operations [19] to perform selection on GPUs. This approach divides the input elements into many 1024-element groups and each group will be further divided into 32 subgroups. Each group will be processed by a warp using three steps, as shown in Fig. 8. In the first step, it uses the intra-warp voting functions such as __ballot() and __popc() to get the number of selected elements for each subgroup. Note that the __ballot(int p) function returns a 32-bit integer in which bit n is set if and only if the predicate p provided by the thread with lane ID n is nonzero. Namely, the __ballot() intrinsic collects the predicates from all threads in a warp into a 32-bit integer and returns this integer to every thread. The __popc(int v) function performs the population count operation by returning the number of bits which are set to 1 in the 32-bit integer

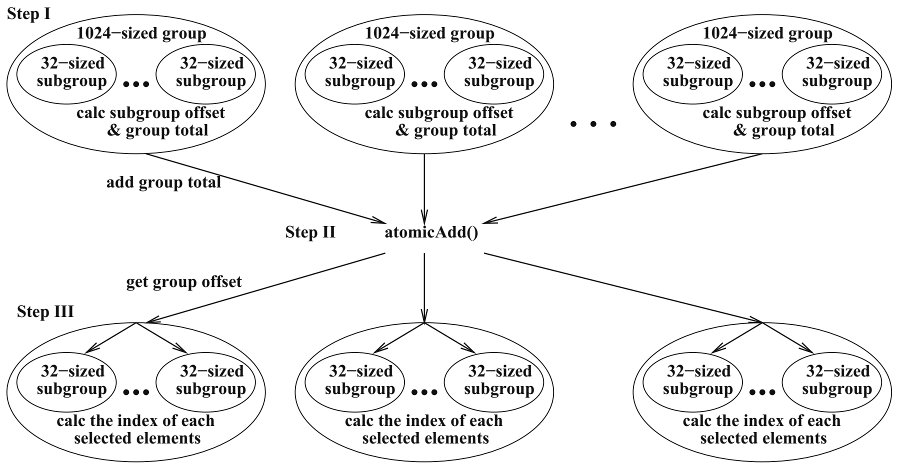

Fig. 8 Hybrid approach

v, where v is the returned value from the __ballot(). Therefore, using both __ballot() and __popc() functions, each thread in a warp can cast a one-bit vote and count the votes quickly. Next, the hybrid method adopts the warp shuffle function __shfl_up() to obtain the offset for each subgroup and the total of selected elements in a group. So in the second step, the group total can be added to the global counter via atomicAdd() to get the group offset. In the third step, the group offset can be broadcasted to all of the threads in a warp by using the __shfl() function. Furthermore, the intra-subgroup offset for a wanted element can be obtained by using bit masking and __popc(). So a thread can find the chosen element's location in the destination array by adding the group, the subgroup and the intra-subgroup offsets together. Because this hybrid will invoke atomicAdd() at most once for each 1024-element group, the execution time depends on the input number of elements, not on the number of selected elements. Furthermore, the use of intra-warp voting, atomic operation and warp shuffle functions makes this approach very efficient.

3.3 Processing departure and arrival events

Figure 9 shows the pseudo-code of event execution. When a token leaves a facility, the first token, if any, in the waiting will get its service and a departure event will be scheduled for it. For the leaving token, an uniform random variable will be generated to determine its destination and its token identifier and timestamp will be put into the next service facility's incoming port. For processing the arrivals at a facility, we append all of the incoming tokens to the waiting queue if the service facility is busy. Otherwise, the newly arrived token with the smallest timestamp can start the service, while the rest of incoming tokens will be put in the waiting queue based on their timestamp order. Note that processing the departures and the arrivals should be launched from the kernel, respectively. This is because we have to wait until all of the threads finishes the process of departures and then start the process of arrivals. Otherwise, the incoming port data

```
__global__ void process_departure(facp,tknp,evp,chp,chsn_num)
{
   calculate the statistics;
   If the facility's waiting queue is empty {
      set the state of the facility to be idle;
   } else {
      remove the front token from the waiting queue and put it in service;
      schedule a departure event for the token;
   }
   determine destination for the leaving token;
}

__global__ void process_arrival(facp,tknp,evp,chp,chsn_num)
{
   sort the incoming tokens (at most 4) by their timestamps;
   if the state of the facility is idle {
      let the first incoming token get the service and schedule a departure
              event for it;
      put the rest of incoming tokens into the waiting queue;
   } else
      append the incoming tokens to the waiting queue;
}
```

Fig. 9 Pseudo-code for event processing

will not be consistent. Furthermore, the CUDA function `__syncthreads()` cannot be used here because it can only synchronize the threads within a warp, not all of the threads.

4 Experimental results

In this section, we compare our PDES implementation on the GPU with a sequential heap-based DES on the CPU. The experimental platform, supported by Ohio Supercomputing Center, has one Dell PowerEdge R730 with two Intel Xeon E5-2680 version 4 processors (2.40 GHz, 128 GB memory) running 64-bit Linux OS. The GPU used in the experiments is a cutting-edge NVIDIA Tesla P100 (Pascal), which contains 56 multiprocessors (3584 CUDA cores in total) and 16GB GDDR5 memory. The device programs use CUDA compiler driver 8.0, Thrust version 1.8.3 and CUB version 1.6.4. The parallel algorithm runs on the host and the device, while the sequential algorithm runs on the host. The torus queueing network model mentioned in the earlier section was used for the simulation.

In the first experiment, we compared the selection performance of using Thrust `copy_if`, `atomicInc` and the hybrid selection algorithm proposed in [22]. We conducted a simple measurement by choosing a certain percentage, say p, of 16-million random numbers which are uniformly distributed between [0.0, 1.0) and stored in an array. That is, given a p, the numbers which are less than or equal to p will be selected and their corresponding array indices will stored in the output array. Figure 10 shows the kernel execution times, excluding the array transfer time, for these three methods by varying the percentages. It can be found that the selection times using

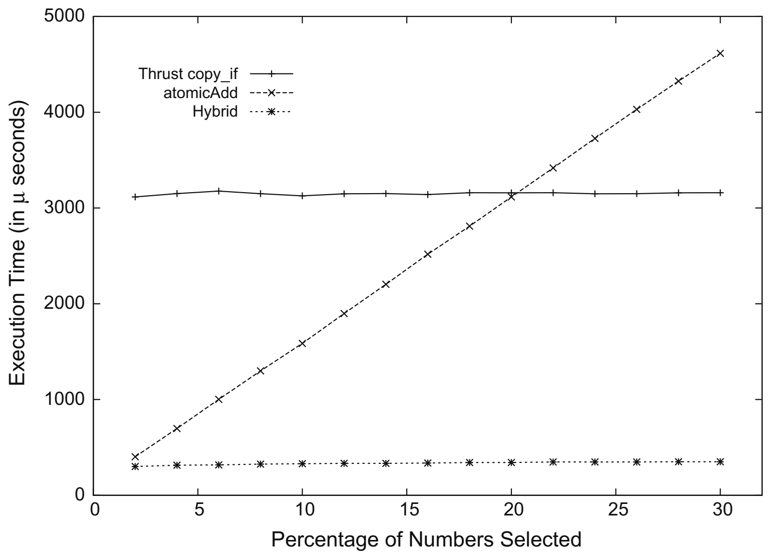

Fig. 10 Selection performance comparison of using atomicInc and Thrust copy_if

Thrust `copy_if` are unchanged even when the percentage increases. This is because Thrust `copy_if` uses the list ranking algorithm with the parallel prefix sum operation. Hence, its execution time depends on the input data size, not on the size of the selected output data. On the contrary, the execution time of using `atomicInc` grows linearly and proportional to the size of the selected random numbers. The reason is that any selected random number has to call the `atomicInc` function to find its position in the output array. The `atomicInc` function becomes the bottleneck. Note that the intersection of the two lines in Fig. 10 is the trade-off size of the selected numbers between `atomicInc` and Thrust `copy_if`. In other words, it is preferred using `atomicInc` if the size of the selected output is small, while using Thrust `copy_if` when the output size is large. Note that as mentioned before, the running times of the hybrid approach are also not dependent on the percentage *p*, because each group will issue only one invocation of atomicAdd(). Furthermore, it can be seen that the hybrid approach performs the best due to using the efficient intra-warp voting, warp shuffle functions and atomic operations.

As reported in [13], the function of finding the minimum in an array which is supported by the NVIDIA CUB library runs faster than the one in Thrust. So we replaced the Thrust `copy_if` with the CUB `Min()` function in our revised implementation. Table 2 shows the difference of the breakdown time for simulating a 2048×2048 torus network. In the simulation model, the mean service time (i.e., the parameter M in calling the function `expon()`) of the service facility is set to 10 and the simulation runs until the simulated clock reaches 50,000. It can be observed that our revised implementation using CUB `Min` and the hybrid selection method runs faster that our previous version which uses Thrust `min_element()` and `copy_if()` functions. Furthermore, the execution times of processing the departure and the arrival events

Table 2 Comparison of breakdown time (in seconds) using Thrust min, Thrust copy_if, CUB min and Hybrid Select (number of facilities: 2048 × 2048)

	Find min		Selection		Departure	Arrival	Total
$d = 0.25$	Thrust min	129	Thrust copy_if	163	92	59	444
	CUB min	11	Hybrid Select	22	98	64	195
$d = 0.5$	Thrust min	65	Thrust copy_if	82	95	59	301
	CUB min	6	Hybrid Select	11	94	60	171
$d = 1.0$	Thrust min	32	Thrust copy_if	41	86	54	213
	CUB min	3	Hybrid Select	6	87	53	149
$d = 2.0$	Thrust min	17	Thrust copy_if	21	80	49	167
	CUB min	1	Hybrid Select	3	81	49	134

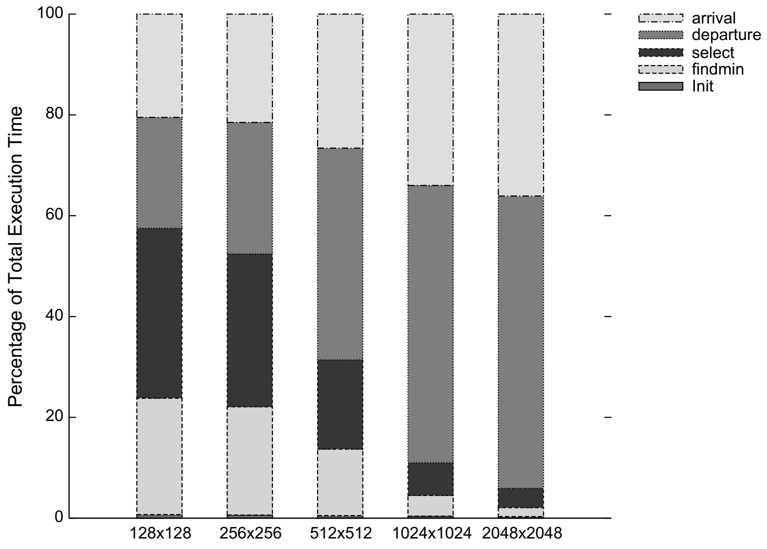

Fig. 11 Percentage of total execution time ($d = 1.0$)

are decreased when the interval d increases. This is because more independent events can be processed in parallel for a larger d.

In the next experiment, we measured the breakdown percentage of the total execution time of our revised implementation for the interval $d = 1.0$ by varying the number of facilities. Figure 11 shows that the percentages of the total execution time for processing the departure events and the arrival events increase when the number of facilities increases. The larger number of facilities, the more events needed to be processed.

We then measured the simulation execution times of our revised implementation by varying the number of facilities and the interval sizes. Figure 12 shows the performance improvement in the GPU experiments compared to the sequential simulation on the CPU. The speedups grow when the number of facilities increases. In particular, our

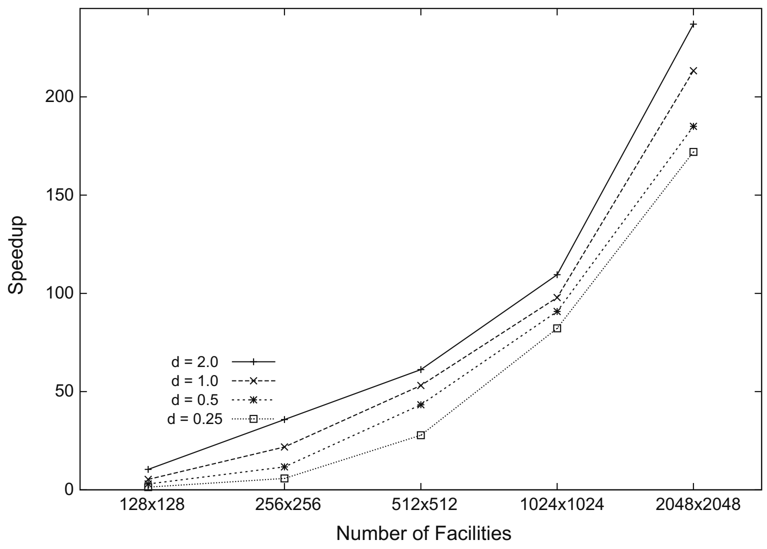

Fig. 12 Speedups

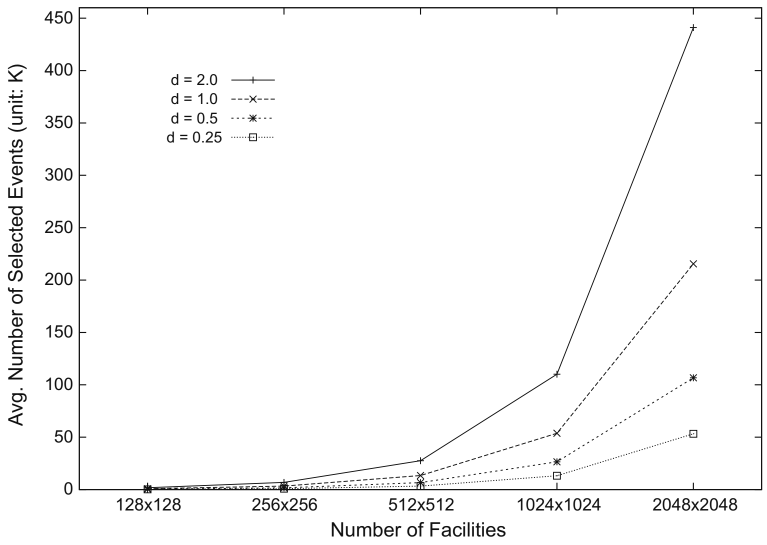

Fig. 13 Average number of parallel events

PDES implementation outperforms the sequential DES by more than 200× speedup for 2048×2048 facilities with $d = 2.0$. The curve increases as the data size increases which implies that the speedup could be increased further for simulating a larger-scale torus network. Figure 12 shows that the larger the interval value d, the larger the speedup obtained. This is because a larger interval allows more parallel events to run. To verify this, we also measured the average number of parallel events for different number of facilities and different interval sizes. The result is shown in Fig. 13.

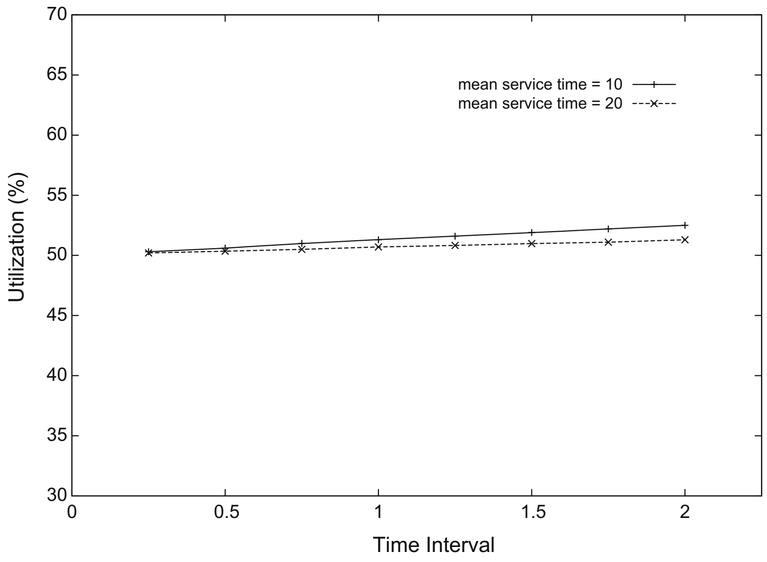

Fig. 14 Utilization with different time interval d

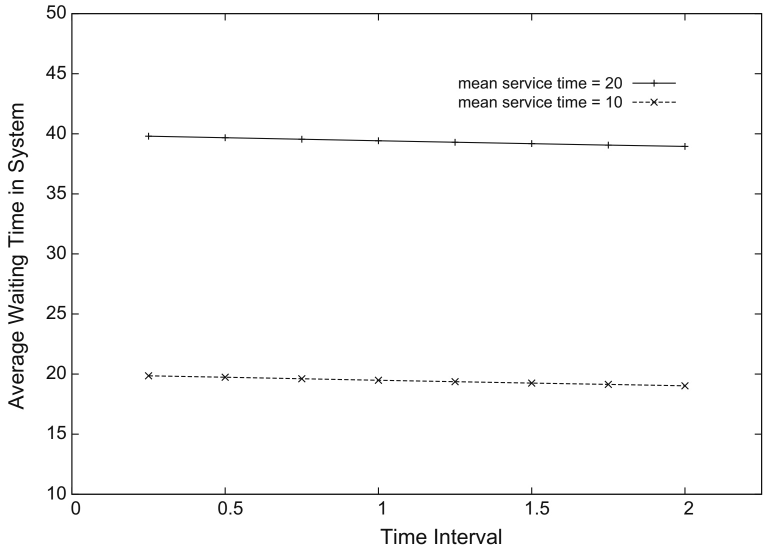

Fig. 15 System waiting time with different time interval d

In another experiments, we evaluated the accuracy in simulation summary statistics due to the use of the modified service time distribution function. Figure 14 shows the difference in the facility server utilization for varied intervals. The simulation with smaller time interval behaves closer to running the simulation with the original service distribution function, i.e., utilization rate is %50. As the interval d increases, the utilization also increases because the service time is at least large as d. Figure 15 shows similar effect on the system waiting time, which is the average time of a token staying in a service facility, including the service time and the waiting time in the queue. For the purpose of comparison, we used two mean service times 10 and 20, and the expected system waiting time will be 20 and 40, respectively, if the original service distribution function is used. Unlike utilization, the system waiting time drops as interval increases. For the same interval d, the larger mean service time has smaller difference in utilization and system waiting time because the interval d occupies a smaller portion in the service time.

5 Conclusion

We presented fast implementations of PDES on GPU by using the productivity-oriented Thrust and CUB libraries. Our scheme exploits a modified service distribution function to allow clustered events to be processed in parallel, while preserving timestamp ordering and causal relationships of events. CUB, which provides a collection of optimized data-parallel primitives such as reduce, stream compaction and prefix sums, makes our implementation much more efficient. The experimental results are encouraging. We were able to achieve 240$\times$ speedup using our implementation at the expense of accuracy in the results. This indicates that our implementation utilizes the massively data-parallel processing power of GPU and is suitable for large-scale simulation models.

Acknowledgements This research was supported in part by a Fulbright Senior Research Scholarship and by an allocation of computing time from the Ohio Supercomputer Center.

References

1. Baezner D, Lomow G, Unger BW (1990) Sim++: the transition to distributed simulation. In: Proceedings of the SCS Multiconference on Distributed Simulatio, pp 211–218
2. Chandy KM, Misra J (1979) Distributed simulation: a case study in design and verification of distributed programs. IEEE Trans Softw Eng 5(5):440–452
3. Evans JB (1988) Structures of discrete event simulation. Ellis Horwood Limited, Market Cross House
4. Fujimoto R (1990) Parallel discrete event simulation. CACM 33(10):30–53
5. Harris M, Garland M (2011) Optimizing parallel prefix operations for the fermi architecture. In: Chapter 3 of the book "GPU Computing Gems—Jade Edition". Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
6. Hoberock J, Bell N (2010) Thrust: a parallel template library. https://code.google.com/p/thrust/. Accessed 19 Mar 2017
7. Jefferson D (1985) Virtual time. ACM Trans Program Lang Syst 7:404–425
8. Kirk DB, Hwu WMW (2010) Programming massively parallel processors: a hands-on approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco
9. Lo SH, Lee CR, Chung IH, Chung YC (2013) Optimizing pairwise box intersection checking on GPUs for large-scale simulations. ACM Trans Model Comput Simul 23(3):19:1–19:22
10. Luitjens J Faster parallel reductions on Kepler. https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/. Accessed 19 Mar 2017
11. Mascarenhas E, Knop F, Pasquini R, Rego V (1998) Minimum cost adaptive synchronization: experiments with the parasol system. ACM Trans Model Comput Simul 8(4):401–430
12. Nobile MS, Cazzaniga P, Besozzi D, Mauri G (2014) GPU-accelerated simulations of mass-action kinetics models with cupSODA. J Supercomput 69:17–24
13. NVIDIA: CUB. https://nvlabs.github.io/cub/. Accessed 19 Mar 2017

14. NVIDIA (2012) CUDA Programming guide version 4.2
15. Park H, Fishwick PA (2010) A GPU-based application framework supporting fast discrete-event simulation. Simulation 86(10):613–628
16. Park H, Fishwick PA (2011) An analysis of queuing network simulation using GPU-based hardware acceleration. ACM Trans Model Comput Simul 21(3):18
17. Perumalla KS (2006) Discrete-event execution alternatives on general purpose graphical processing units (gpgpus). In: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation, PADS '06, Washington, DC, USA, pp 74–81
18. Rango AD, Macr M, D'Ambrosio D, Spataro W (2015) Accelerating lava flows simulations with gpgpu and opencl. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp 581–588
19. Rego V, Sang J, Yu C (2016) A fast hybrid approach for stream compaction on GPUs. In: Proceedings of the International Workshop on GPU Computing and Applications
20. Rego V, Sunderam VS (1992) Experiments in concurrent stochastic simulation: the eclipse paradigm. J Parallel Distrib Comput 14(1):66–84
21. Sang J, Mascarenhas E, Rego V (1996) Mobile-process based parallel simulation. J Parallel Distrib Comput 33:12–23
22. Schwetman HD (1978) Hybrid simulation models of computer systems. Commun ACM 21:718–723
23. Sunderam VS, Rego V (1991) Eclipse: a system for high performance concurrent simulation. Softw Pract Exp 21(11):1189–1219
24. West J, Mullarney A (1988) ModSim: a language for distributed simulation. In: Proceedings of SCS Multiconference on Distributed Simulation
25. Woods RL, Lawrence KL (1997) Modeling and simulation of dynamic systems. Prentice-Hall, Upper Saddle River
26. Wyllie JC (1979) The complexity of parallel computations. Technical report, Ph.D. thesis, Cornell University, Ithaca, NY, USA
27. Xu Z, Bagrodia R (2007) GPU-accelerated evaluation platform for high fidelity network modeling. In: PADS, pp 131–140