Parallel Discrete Event Simulation for DEVS Cellular Models using a GPU

Moon Gi Seok and Tag Gon Kim
Systems Modeling Simulation Laboratory
Korea Advanced Institute of Science and Technology (KAIST)
373-1 Kuseong-dong, Yuseong-gu, Daejoen, Korea
mgseok@smslab.kaist.ac.kr, tkim@ee.kaist.ac.kr

Keywords: Parallel discrete event simulation, cellular model, DEVS, GPU, CUDA

Abstract

This paper presents a parallel discrete event simulation (PDES) environment using graphics processing unit (GPU) to simulate cellular models described by the discrete event system specification (DEVS). The DEVS is a general system specification, and it helps the modeler to verify and validate the model. The parallel simulation algorithm for simulating the DEVS model has been well studied, and based on the simulation algorithm, the operations of DEVS cellular models and their relative simulators are processed in parallel using the GPU. The algorithm of managing the event-list and routing output events in the DEVS simulation is revised to be processed in parallel considering the features of the GPU. The performance analysis and the experimental result are provided to demonstrate the increased speed of the proposed PDES using the GPU as compared to the PDES using the CPU in a large-scale cellular model simulation.

1. INTRODUCTION

To analyze a physically or chemically homogeneous system, we describe the system as mathematical equations [1][2]. Unfortunately, the mathematical equations are sometimes too complex to solve, or too difficult to find in analyzing a complex system. In that case, a simulation-based approach could be an alternative method [3][4]. In a simulation-based approach to a homogeneous system, the system is expressed as cellular models instead of obtaining a general solution, and the description of each model is based on spatial and temporal behavior in a local space or entity. The entire simulation proceeds as each cellular model interacts with adjacent models, and one of the example following this approach is the cellular automata simulation [5].

To guarantee and verify the simulation's correctness, modeling of the target system should be based on a formal system specification, called formalism. Among several system specifications, the DEVS formalism is a general system specification from generic dynamic systems theory and has been applied to both continuous and discrete phenomena, so it is widely used and applicable [6]. Using the DEVS formalism, a homogeneous system can be described as the DEVS cellular models in Fig 1.

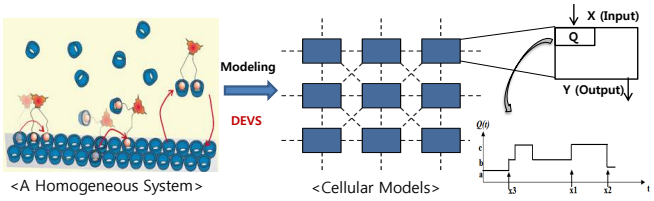

Figure 1. A homogeneous system is represented by cellular models using DEVS formalism. The DEVS cellular models interact by sending and receiving the output and input. The state of each cellular model is changed by its own decision or when receiving its neighbors' events

To describe a homogeneous system in more detail, a large number of cellular models should be employed which demands a high performance simulation technique. One of the most well-known techniques is parallel simulation[7]. In a traditional parallel simulation, the simulation is executed on multiple central processing units (CPUs) or a multi-core CPU, but in this paper, we simulate the cellular models using the graphics processing unit(GPU).

The GPU is primarily dedicated to computing the graphics calculation. Recently, the GPU has emerged as a powerful many-core processor as the performance of the GPU grows rapidly. GPU corporations also provide the better application programming interface (API), like computing unified device architecture (CUDA)[8] by NVIDIA or ATI stream by AMD. In turn, this causes developers be more attracted to the GPU when solving the compute-intensive problem.

In this paper, we present a parallel discrete event simulation (PDES) environment to simulate the DEVS cellular models using a GPU. The parallel simulation algorithm for the DEVS model has already researched [6][9], and following the verified DEVS simulation algorithm, the operations of the cellular models and their relative simulators, which are the components of the simulation engine are computed in parallel using the many-cores of a GPU. Moreover, the algorithm of routing events and event-scheduling in the DEVS simulation algorithm is revised to be processed in parallel using the GPU. To analyze and show the benefits of PDEVS using the GPU, we compared the simulation performance with PDEVS using several cores in the CPU through an experiment. Concerning

the experiment, the simulation target is a fire-spreading phenomenon, which is an example of a homogeneous system. A PDES environment using the GPU was developed using CUDA, and the PDES environment using the CPU was developed using OpenMP[10].

The rest of the paper is organized as follows. In Sections 2 and 3, we describe some works related to this paper and the backgrounds of the CUDA and DEVS simulation algorithm. Section 4 describes the parallel simulation using the GPU. In Section 5 and 6, we analyze and evaluate the performance of the PDES using the GPU compared to the CPU. Section 7 will conclude the paper.

2. RELATED WORK

The DEVS simulation of the cellular model has been studied to analyze complex homogeneous system[11]. In a typical DEVS cellular model simulation, the number of the cellular models is far too large for detailed simulation, and it requires a high-performance simulation technique. The technique to improve the performance of the DEVS simulation has been studied in two approaches. The first approach is improving the DEVS simulation algorithm, and the second approach is increasing the number of computing processors or machines through parallel and distributed DEVS simulation.

Examples of the first approach includes flattening the hierarchical structure of the models[12] or using a fast scheduling algorithm which deal with active models only by tracking the active models[13] or eliminating the unnecessary components during simulation execution[14].

In the second approach, both parallel and distributed DEVS simulations improve the simulation performance. Due to the interaction overhead on the network, distributed DEVS simulation is not as adequate for high-performance simulation as the parallel DEVS simulation. However, it has advantages to integrate geographically distributed simulators, so it has been actively researched using various middlewares[15],[16],[17]. In parallel DEVS simulation research, parallel simulation protocols have been mainly researched[18],[19].

Recently, the novel and powerful compute-intensive processors like the Cell processor has emerged. Qu and Gabriel[20][21] proposed a simulation environment for parallel DEVS simulation considering the architecture features of the Cell processor. In this paper, we proposes a simulation environment for parallel DEVS simulation using the GPU, which is one of the emerging compute-intensive processors. The purpose of both researches is to optimize and map the parallel DEVS simulation into the specific processor, but due to the different architecture features between the Cell processor and the GPU, the DEVS simulation algorithm is differently modified and mapped to each hardware.

Park and Paul[22] proposed the GPU-based framework for the discrete event simulation using the parallel event-list. The proposed PDES environment in this paper also utilizes the GPU. However, the environment follows the DEVS simulation algorithm, and in our event-scheduling approach of the environment, the parallel event-list is unnecessary.

3. BACKGROUND

3.1. CUDA Programming Overview

CUDA is the parallel computing framework developed by the NVIDIA corporation. CUDA provides the APIs, which allow the developers to easily access GPU programming without having technical GPU knowledge.

The CUDA kernel manages the threads in a two-level hierarchy, and assigns threads to GPU multiprocessors, as shown in Fig. 2.

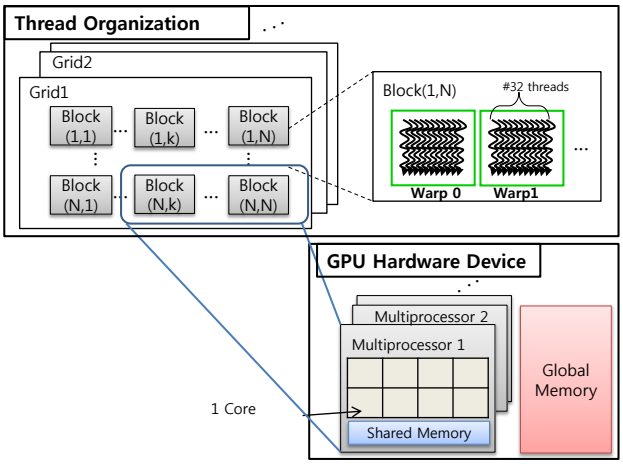

Figure 2. Developers manages the organization of threads by specifying the grid and block size. Several blocks are mapped to a multiprocessor, which consists of multiple cores, and the threads in a block are divided into 32 threads when the blocks are scheduled and executed in each multiprocessor.

The number of the assigned blocks per multiprocessor is determined by the CUDA kernel considering the amount of the memory needed for the simultaneous execution of the blocks, but the number is limited to 8 blocks. Once a block is assigned to a multiprocessor, the threads in the block are further divided into 32-threads units called warps. These warps are the unit of thread scheduling and execution in a multiprocessor for each clock cycle. Each multiprocessor is operated independently of the other multiprocessor; this enables the execution of CUDA to embrace the single instruction multiple thread (SIMT) paradigm[23], which is similar to the execution model of single instruction multiple data (SIMD). The SIMT-based paradigm allows divergence in the execution path between multiprocessors.

CUDA threads access multiple memory spaces, such as

shared, and global memory (Fig. 2) to read and write data. The shared memory is only accessible by threads in a block, and the global memory is accessible by all threads in CUDA. The shared memory space is much faster than the global memory spaces because the shared memory is on-chip, but global memory is on the device memory.

3.2. DEVS Simulation Algorithm Overview

Originally, the DEVS formalism began with classic DEVS. Several years later, a Parallel DEVS was introduced to remove the constraints of sequential processing and enable the parallel execution of models. In this paper, we used Parallel DEVS, not classical DEVS, for the parallel simulation. In DEVS formalism, there are 2 classes of models, the atomic model and the coupled model. In modeling a homogeneous system, cellular models are atomic models. The network of these cellular models constitutes a coupled model that maintains the information of the coupling relationships of cellular models.

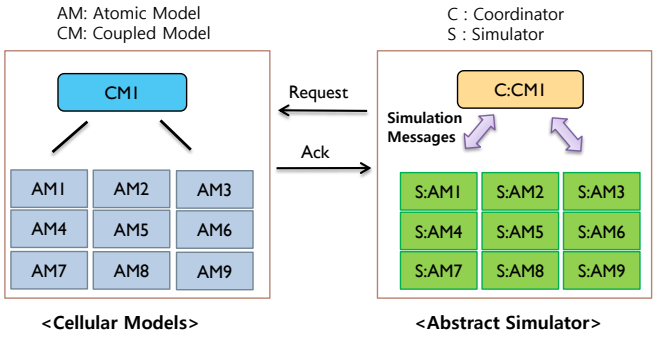

Figure 3. A homogeneous system is expressed as the cellular models, which correspond to the atomic model and one coupled model that has the information about the network of the cellular models. The atomic models and coupled model are carried out by simulators and a coordinator by abstract simulators, which consist of simulators and a coordinator.

There are abstract simulators called simulation engines that execute models to generate their dynamic behavior. The abstract simulators consist of the coordinator and simulators, and the relationship between the models and the abstract simulators is shown in Fig. 3.

Based on the DEVS simulation algorithm[9][13], the coordinator executes the following code every simulation cycle:

```
AllSimulator.GetNexttN();
tN = ComputeGlobalTN();
imminentSimualtor.GenerateOutput(tN);
influenceeSimulator.DeliverOutput();
AllSimulator.StateTransitionOrNot(tN);
```

Detailed descriptions of each procedure in the upper code are as follows:

1. The coordinator asks all simulators about each next scheduled time, denoted by tN, which is determined based on the state of its cellular model.

2. The coordinator proceeds with the global simulation time as much as the minimum tN among the received tN of the simulators.

3. The coordinator orders the imminent simulators to compute the output event of its cellular model. The imminent simulators are those whose tN is the same as current global simulation time.

4. The coordinator sends all received output-events to its destination simulators, which are called influencee simulators.

5. The coordinator orders all simulators to change their cellular model state based on the global tN and input event. The simulators whose tN is not the same as the global tN and whose input event is empty will do nothing.

4. PARALLEL DEVS SIMULATION USING THE GPU

4.1. Parallel Simulation of Simulators and Atomic Models using the GPU

The DEVS algorithm supports the parallel execution of atomic models and their relative simulators by handling the transition collision, and the external event collision [24]. When computing the simulators and atomic models in parallel using the GPU, each simulator and its atomic model is allocated in a thread and run in the GPU, as shown in Fig. 4.

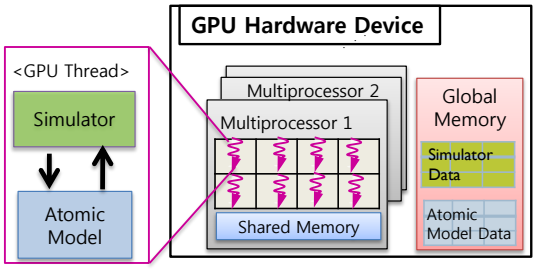

Figure 4. The whole cellular model and its relative simulators are assigned to a GPU thread, and the threads are scheduled and executed in multiprocessors. Each thread accesses the global memory to get simulators and model data.

To allocate an atomic model and its simulator to a thread in the GPU, we should consider the execution paradigm of

SIMT [23] in the CUDA. In the SIMT paradigm, the instructions for all threads should be identical which limits the target system from a general system to a homogeneous system. In a homogeneous system simulation, we express the system as the aggregated homogeneous cellular model with the same behavior algorithm. Since all the simulation algorithms of the simulators and the behavior algorithm of the homogeneous cellular models are identical, each simulator and atomic model can be allocated to a GPU thread following the SIMT paradigm. Even though the instructions of the simulators and the atomic model are the same, depending on the stored data, the atomic model and the simulator operates differently.

4.2. Parallel Simulation of a Coordinator using the GPU

To simulate the DEVS cellular models and their simulators in parallel, and to compute some coordinator operations in parallel using the GPU, some DEVS coordinator algorithms should be revised.

One of the algorithms is managing time events and imminent simulators. When the coordinator asks the simulators about their next scheduled time, which is denoted by tN, the simulators send the time event, which contains their tN information, to the coordinator. Generally, the DEVS coordinator manages these time events using an event-list, and the events are sorted in chronological order based on their tN information. Coordinator proceeds global simulation time as much as the minimum tN which is the tN of the first events in the event-list. After updating the current simulation time, the coordinator only activates imminent simulators to generate output events.

In the proposed simulation environment, the coordinator need not to manage the imminent simulators because of the property of the thread allocation in CUDA. Thread allocation in CUDA is determined at the program compilation, so the various number of threads cannot be dynamically allocated in runtime as many as the number of the imminent simulators. To generate output events in parallel, the coordinator should activate all simulators regardless of whether each simulator is imminent or not. It means that the even-list need not to be sorted unnecessarily.

Using tNs in the unsorted event-list, the coordinator should find minimum tN to proceed the simulation time. The methods to find minimum value using the multiple processors has been researched, and the parallel reduction technique[25] is one of them. It enables speed-up by utilizing the many-cores in GPU. After updating the current simulation time to minimum tN, all simulators and their cellular models run concurrently in GPU threads, but only imminent simulators and their cellular models generate output events.

This approach transfers the computation load from the coordinator to simulators by excluding the burden of sorting the event-list, and including the overhead of executing all simulators to generate output events. The overhead of activating all simulators in the GPU is not serious as much as the CPU because of low context switching overhead due to the warp-based thread management in CUDA[23].

The other revised algorithm is routing output events of the simulators. After receiving all output events of the simulators, the coordinator delivers output events to the destination simulators based on the coupling information. Each operation of routing output events is independent with others, but if each GPU thread transfers an output event to the destination simulator based on the coupling relation of the source simulator, there can happen collisions when two or more events whose destination simulator is same are delivered to the same simulator concurrently. To avoid collisions, we propose the routing event algorithm, as shown in Algorithm 1.

In Algorithm 1, Let N is the number of the simulators, $OutputEventBuf$ is the array of output events from imminent simulators, $CouplingData$ is the array of coupling relation data from the coupled model, and $SimulDatArray$ is the array of the simulator data. All arrays are stored in the GPU global memory, and each GPU thread can access these arrays.

Algorithm 1 Parallel Output Event Routing

Input : $OutputEventBuf[N]$,$CouplingData[N]$
Output: $SimulatorDatArray[N]$

```
t_ID : target simulator ID
ns_ID : neighbor simulator ID
outputEvent : temporal output event
inputEvent : temporal input event
coupDat :temporal coupling data which is the array of the ns_IDs
procedure Parallel-Output-Event-Routing
for each thread do
  Initialize t_ID based on the index of the grid and block
  coupDat ← CouplinData[t_ID]
  for each ns_ID in coupDat do
    outputEvent ← outputEventBuf[ns_ID]
    if the destination simulator ID of outputEvent is same with
    t_ID then
      inputEvent ← outputEvent
      Deliver inputEvent to simulDatArray[t_ID]
    end if
  end for
end for
```

In proposed routing event algorithm, each thread gets the coupling relation of a target simulator. Based on the coupling relation of the simulator, all output events from neighbors are examined. If the destination of some output events is the same with the target simulator, the output events are delivered to the target simulator as the input events one by one. Since the target simulator ID in each thread is not overlapped, this algorithm prevents collisions when delivering output events to

the same destination simulator without some mutual exclusion techniques.

5. SIMULATION PERFORMANCE EVALUATION USING THE GPU

One of the parameter to analyze simulation performance is simulation execution time. To analyze and show the benefits of the proposed environment, we compare the simulation performance using the multiple cores in the CPU. Using Amdahl's law, we get two speed-up equations using the multiple cores in the GPU and the CPU. Combining two equation, we can deduct the speed-up equation using the GPU compared to the CPU as follow:

$$Speedup = \frac{Exe.Time_{CPU}}{Exe.Time_{GPU}} = \frac{(1-(P_{cc}+P_{sa}))+\frac{P_{cc}+P_{sa}}{NC_{cpu}}}{(1-(P_{cc}+P_{sa}))+\frac{P_{cc}+P_{sa}}{\sigma_p \cdot \sigma_e \cdot NC_{gpu}}+P_{Overhead}} \quad (1)$$

$Exe.Time_{GPU}$ and $Exe.Time_{CPU}$ are the expected simulation execution times using multi-core in the GPU and the CPU, supposing that the expected simulation execution time using one core in a CPU is one. P_{cc} and P_{sa} are the proportions of the sum of the coordinator, coupled model operations and the sum of all simulator, atomic model operations that can be done in parallel in the whole simulation. NC_{cpu} and NC_{gpu} are the number of cores in the CPU and the GPU. σ_P is the ratio of the core performance in the GPU to the core performance in the CPU, and it is based on a clock speed. σ_e is the parameter considering other different features like different processing paradigm of the GPU and the CPU. In CUDA, the warp-based threads are executed in the GPU. When one thread in a warp is running and 31 threads are finished, 31 threads should wait until all threads are finished or replaced by the other warp, while the cores in the CPU can run independently. $P_{overhead}$ is the additional overhead of using the GPU. It occurs during the data communication between DRAM and the GPU global memory, different memory access time, and so on.

The operations that are computed in parallel are proportional to the number of the cellular models, but other sequential operations are less affected by number of cellular models. Thus, if the number of cellular models goes to infinity, $P_{cc}+P_{sa}$ goes one, and the $P_{Overhead}$ is negligible compared to the P_{cc}, P_{sa}. When the number of cellular models goes to zero, P_{cc} and P_{sa} also goes zero. According to this relation between P_{cc}, P_{sa}, and the number of cellular models, Eq. 1 can be transformed into Eq. 2 in a small-scale cellular model simulation, and into Eq. 3 in a large-scale cellular model simulation.

$$Speedup_{small-scale} \approx \frac{1}{1+P_{Overhead}} \quad (2)$$

$$Speedup_{large-scale} \approx \frac{\frac{P_{cc}+Psa}{NC_{cpu}}}{\frac{P_{cc}+P_{sa}}{\sigma_p \cdot \sigma_e \cdot NC_{gpu}}} = \frac{\sigma_p \cdot \sigma_e \cdot NC_{gpu}}{NC_{cpu}} \quad (3)$$

Following Eq. 2, in a small-scale simulation, the expected speed-up using GPU can be less than 1 due to the overhead of using the GPU. However, in a large-scale simulation, the expected speedup using the GPU is dependent on the value of the parameters in Eq. 3. Supposing the large-scale simulation is computed using an nVIDIA GTS250, which has 128 cores operating at 1.8GHZ, using an i7 Intel core, which has 4 cores operating at 2.8GHZ, and σ_e is about 0.2, we can roughly predict that the speedup of using the GPU will be about 4 times.

6. PERFORMANCE EVALUATION

To analyze the proposed simulation environment, we simulate a fire-propagation phenomenon, which is an example of a homogeneous system. Each unit space starts burning when the received heat energy is over a certain threshold value and is generating energy by fuel consumption that is delivered to neighbors based on the distance to the neighbors, diffusion velocity, and the wind velocity in the unit space. The parameters and propagation equations are based on Rothermel's research[26][27].

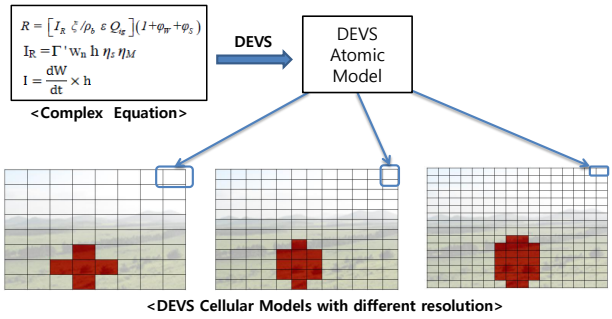

Figure 5. Using complex equations, a fire-spreading cellular model is generated using DEVS formalism. As the number of cellular models increases, more detailed simulations are possible.

We express a fire-spreading system using different numbers of DEVS cellular models. In Fig. 5, as the number of cellular models increases, more detailed simulation results can be obtained. Different numbers of cellular models are simulated using multi-cores in the CPU and the GPU, as shown in TABLE 1. The simulation environment using the GPU is developed using CUDA, and the simulation environment using

the CPU is developed using OpenMP[10]. The experiment result shows in Fig. 6

Table 1. Experiment Environment

i7 CPU	Intel Core i7 CPU 860 @ 2.80GHZ
i5 CPU	Intel Core i5 CPU M580 @ 2.67GHZ
GPU	nVIDIA GeForce GTS 250
Memory	6GB
CUDA	CUDA Driver API SDK v3.2
OpenMP	OpenMP v2.0

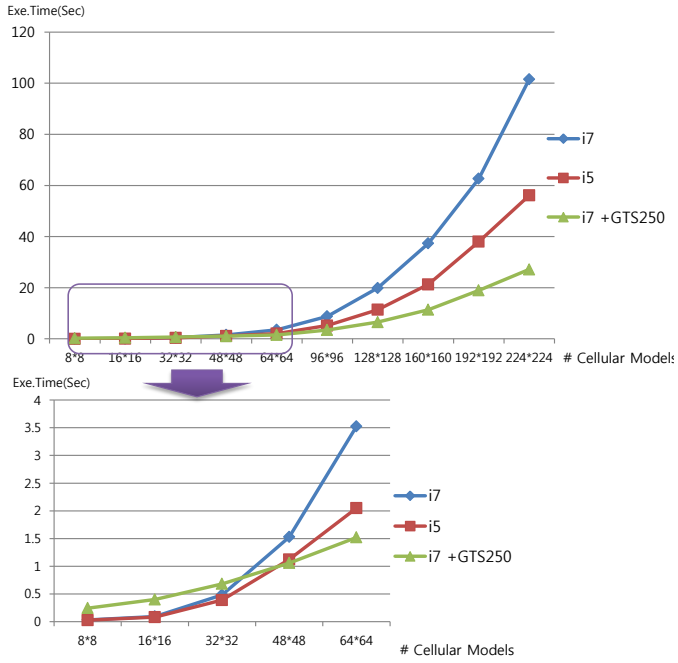

Figure 6. Experimental result shows that as the number of the cellular models goes larger, the simulation execution time is much more reduced when using the GPU

As expected in Eq. 2, in the small-scale cellular model simulation, the performance of proposed PDES environment is lower than the performance of the PDES environment using a CPU, but this is meaningless because the total simulation execution time is small in each case. As the number of cellular models grows, the PDE using the GPU outperforms the PDE using the CPU, and the difference of simulation execution time also increases.

7. CONCLUSION

In this paper, we present a parallel discrete event simulation environment using a GPU to simulate DEVS cellular models that represent a homogeneous system. The simulation algorithm in the proposed environment is based on the DEVS simulation algorithm. Using this algorithm, the operation of each cellular model and its relative simulator is allocated to a GPU thread, and all of the allocated threads are computed in parallel using the GPU's multiprocessors. To execute the atomic model and simulators run concurrently, and to process some coordinator operations using the GPU in parallel, the DEVS simulation algorithm should be revised. One of the algorithms is managing event-lists and imminent simulators. Due to the features of CUDA, we remove the process of sorting the event-list, and activate all simulators to generate output events. Using the unsorted event-list, we find the minimum next time information with the parallel reduction technique. The other revised algorithm is routing event algorithm to be processed in parallel and prevent collisions in the GPU. We expect the performance speedup of the proposed simulation environment using the GPU to the environment using the CPU based on Amdahl's law. The prediction of the speed-up is confirmed by the experiment result of the fire-spreading simulation. As the number of cellular models increases, the speed-up of the PDES using the GPU compared to the PDEVS using the CPU also grows.

In this paper, due to the SIMT execution paradigm in CUDA, we limit the target system from a general system to a homogeneous system. Our future research will concentrate on providing a DEVS parallel simulation environment for a general system simulation utilizing both a multiple cores in GPU and a CPU. By using the CPU and the GPU, the advantages of the CPU and GPU will be maximized.

REFERENCES

[1] K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwellâs Equations in Isotropic Media," *IEEE Transactions on Antennas and Propagation*, vol. 14, no. 3, 1966.

[2] F. Gelbard and J. H. Seinfeld, "Numerical Solution of the Dynamic Equation for Particulate Systems," *Journal of Computational Physics*, vol. 14, no. 3, pp. 357–375, 1978.

[3] B. Zeigler, H. Song, T. Kim, and H. Praehofer, "DEVS Framework for Modeling, Simulation, Analysis, and Design of Hybrid Systems," *Lecture Notes in Computer Science*, vol. 999, pp. 529–551, 1995.

[4] G. A. Wainer, "Modeling and simulation of complex systems with Cell-DEVS," in *Proceedings of the 36th conference on Winter simulation*, ser. WSC '04, 2004, pp. 49–60.

[5] B. Chopard and M. Droz, *Cellular automata modeling of physical systems*. United Kingdom, Cambridge: Cambridge University Press, 1998.

[6] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. San Diego, CA, USA: Academic Press Professional, Inc., 2000.

[7] R. M. Fujimoto, "Parallel simulation: parallel and distributed simulation syste," in *Proceedings of the 33nd conference on Winter simulation*, 2001, pp. 147–157.

[8] NVIDIA CORPORATION, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2007.

[9] A. Chow, B. Zeigler, and D. H. Kim, "Abstract simulator for the parallel DEVS formalism," in *Proceedings of the Fifth Annual Conference on the AI, Simulation, and Planning in High Autonomy Systems*, Dec. 1994, pp. 157–163.

[10] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, Jan.-Mar. 1998.

[11] J. Ameghino, A. Troccoli, and G. Wainer, "Models of complex physical systems using Cell-DEVS," in *Proceedings of the 34th Annual Simulation Symposium*, 2001, pp. 266–273.

[12] S. Jafer and G. Wainer, "Flattened Conservative Parallel Simulator for DEVS and CELL-DEVS," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 1, Aug. 2009, pp. 443–448.

[13] H. Xiaolin and B. P. Zeigler, "A high performance simulation engine for large-scale cellular DEVS models," in *Proceedings of the High Performance Computing Symposium (HPC '04)*, 2004.

[14] F. J. Barros, "Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation," in *Proceedings of the 27th conference on Winter simulation*, ser. WSC '95, Washington, DC, USA, 1995, pp. 781–785.

[15] Y. J. Kim and T. G. Kim, "A heterogeneous simulation framework based on the DEVS BUS and the high level architecture," in *Proceedings of the Winter Simulation Conference*, vol. 1, Dec. 1998, pp. 421–428.

[16] Y. Cho, B. Zeigler, and H. Sarjoughian, "Design and implementation of distributed real-time DEVS/CORBA," in *Proceedings of the IEEE International Conference on the Systems, Man, and Cybernetics*, vol. 5, 2001, pp. 3081–3086.

[17] B. Chen and X. G. Qiu, "MPI-Based Distributed in DEVS Simulation," in *Proceedings of the 3rd International Symposium on Intelligent Information Technology Application*, vol. 2, Nov. 2009, pp. 78–81.

[18] Y. R. Seong, S. H. Jung, T. G. Kim, and K. H. Park, "Parallel Simulation of Hierarchical Modular DEVS Models: A Modified Time Warp Approach," *International Journal in Computer Simulation*, vol. 5, no. 3, pp. 263–285, 1995.

[19] Y. Sun and J. Nutaro, "Performance Improvement Using Parallel Simulation Protocol and Time Warp for DEVS Based Applications," in *Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, Oct. 2008, pp. 277–284.

[20] Q. Liu and G. A. Wainer, "Exploring multi-grained parallelism in compute-intensive devs simulations," in *Proceedings of the 24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, May 2010.

[21] Q. Liu and G. A. Wainer, "Accelerating large-scale devs-based simulation on the cell processor," in *Proceedings of the 2010 Spring Simulation Conference (SpringSim10), DEVS Symposium*, April 2010.

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

[23] NVIDIA CORPORATION, NVIDIA CUDA C Programming Guide, 2010.

[24] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism," in *Proceedings of the 26th conference on Winter simulation*, ser. WSC '94, 1994, pp. 716–722.

[25] M. Harris, "Optimizing parallel reduction in cuda," *NVIDIA Developer Technology*, 2008. [Online]. Available: http://www.mendeley.com/research/optimizing-parallel-reduction-cuda/

[26] R. Rothermel, *A mathematical model for predicting fire spread in wildland fuels*. Ogden, UT: Research Paper INT-115, U.S. Department of Agriculture, 1972.

[27] L. Ntaimo, B. Zeigler, M. Vasconcelos, and B. Khargharia, "Forest Fire Spread and Suppression in DEVS," *SIMULATION*, vol. 80, pp. 479–500, 2004.