FLATTENED PARALLEL DEVS SIMULATIONS ON GPU ARCHITECTURES

Guillermo G. Trabes[a], Alonso Inostrosa-Psijas[b], Verónica Gil-Costa[c] and Gabriel A. Wainer[a]

[a] Advanced Real-Time Simulation Laboratory, Carleton University, Canada
{guillermotrabes,gwainer}@sce.carleton.ca
[b] Escuela de Ingeniería Informática, Universidad de Valparaíso, Chile
alonso.inostrosa@uv.cl
[c] Universidad Nacional de San Luis, Argentina
gvcosta@email.unsl.edu.ar

ABSTRACT

Discrete Event System Specification (DEVS) is a modeling and simulation of discrete event systems formalism. Most DEVS-based simulators are implemented as sequential programs. However, simulating large-scale complex models in a sequential simulator is impractical (if possible), as simulations may take a long time to execute. A usual technique to speed up simulations is the parallel execution of the simulator. Most parallel discrete-event simulation efforts focus on logical process approaches, resulting in complex simulation architectures. Recent parallelizing efforts lean towards executing the simulators in multicore architectures. Despite promising results, they are limited to the amount of CPU processing cores. In this work, we propose an algorithm to accelerate the execution of DEVS simulations on Graphical Processing Units (GPU) architectures. We show different case studies where the proposed algorithm achieved speedups of up to 12.29 and 16.53 compared to a sequential version.

Keywords: parallel DEVS simulations, GPU architectures, flattened parallel DEVS.

1 INTRODUCTION

The Discrete Event System Specification (DEVS) formalism [1] provides a theoretical framework for developing M&S based on discrete events. Parallel DEVS (or PDEVS, a DEVS extension) allows dealing with simultaneous events while adding possibilities of parallel execution to its implementations [2] (from here, this paper will focus on PDEVS, which will be called simply DEVS). Nevertheless, most DEVS implementations use sequential simulation and their computing capabilities are restricted by the hardware of the computer that executes them. However, there is a growing need to simulate complex, large-scale DEVS models that either incur long execution times or require too much memory to run on sequential simulators. A solution is to simulate such models using parallel discrete-event simulation (PDES).

Most efforts in PDES for DEVS simulations were mainly focused on distributed memory architectures based on Logical Processes such as [3-6]. However, complex synchronization algorithms are required to guarantee global time-stamp order in event execution. More recently, researchers have gained interest in multicore shared memory architectures and Graphical Processing Units (GPU) for DEVS simulations. Authors of [7] developed an algorithm to execute DEVS simulations on multicore architectures, where the acceleration is restricted by the reduced number of cores at each chip and by the synchronization and communication overheads among the threads. Due to the practical constraints of multicore CPUs and the increasing incorporation of hardware accelerators to modern high-performance computers for increased parallelism and speedups, the next step is to continue to develop methods to execute DEVS simulations in hardware accelerators like GPUs.

Proc. of the 2024 Annual Simulation Conference (ANNSIM'24), May 20-23, 2024, American University, DC, USA
C. Ruiz-Martin, B. Oakes and R. Cárdenas, eds.
©2024 Society for Modeling & Simulation International (SCS)

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

GPUs are hardware accelerators initially devised for 3D graphics rendering. Due to their relatively low cost, they are now used in a broader range of applications, such as data-intensive numerical computations, and have become an attractive piece of hardware for high-performance computing. Architecturally, GPUs are manycore multiprocessors designed with a high degree of parallelism, containing up to tens of thousands of simpler independent processor cores. NVIDIA and AMD produce most GPUs for high-performance computing. NVIDIA GPUs are composed of several stream multiprocessors (SM), each of which comprises several computing cores (also called CUDA cores or stream processors), where each executes computational units known as warps. Each warp consists of several execution threads operating over different data elements in a Single Program Multiple Data (SPMD) fashion.

NVIDIA GPUs can have several gigabytes of global memory that any thread can access. There is also memory associated with each thread block (faster than global memory), which only their threads can access. Multiple threads can access different banks of memory concurrently. GPUs do not share the same address space as CPUs. Thus, they cannot access each other memory directly, but it must be explicitly copied from/to GPU/CPU memory. However, CUDA (NVIDIA library for GPU programming) provides a unified memory model that transparently accesses GPU or CPU memory [8].

As GPUs have become more readily available, they have been proposed them for running simulations. However, they have not been widely used for DEVS simulations. In [9], authors propose accelerating hybrid DEVS simulations by using GPUs to compute the continuous section of the simulation while the discrete portion is computed in the CPU, resulting in accelerated simulations with high energy consumption. In [10], authors propose using GPU to simulate cellular models with a parallel version of the DEVS protocol, resulting in accelerated executions when simulating homogeneous DEVS cellular models. Therefore, performing parallel simulations in GPUs remains an open topic [11].

Here, we propose an algorithm extending our previous multicore approach in [7] to accelerate the execution of DEVS simulations on GPU architectures. To validate the performance of our approach, we present experimental evaluations achieving speedups up to 12.29 and 16.53 on GPU architectures.

2 BACKGROUND

Following the DEVS philosophy of separation of concerns to distinguish the simulation model of its implementation [12], DEVS defines its components' semantics and proposes an implementation known as the DEVS abstract simulator [1], which represents the processing needed to execute DEVS models [13]. For the simulation of a DEVS model (Figure 1(a)), the abstract simulator creates an equivalent representation of the model using a structure that consists of a *root coordinator* (added at the top), *simulators* associated to each atomic model, and a *coordinator* associated with each coupled model (as shown in Figure 1(b)).

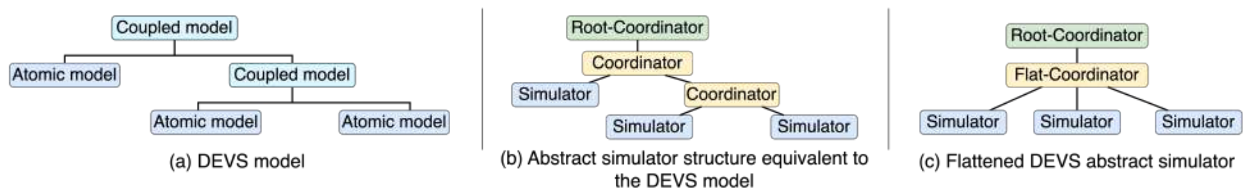

Figure 1: Different DEVS abstract simulator structures for a DEVS model [7].

Once the abstract simulator structure is created, the DEVS protocol is executed to perform the simulation. It works by exchanging messages between the components to orchestrate the simulation execution, including messages for initialization, compute output, executing a state transition, and sending outputs. In addition, a done message is used to signal the step completion. During the transmission of these events, the time for the current or next event is also transmitted.

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

There are various sequential implementations of the abstract simulator [14,15], as well as parallel algorithms to simulate DEVS models [12,13]. Additionally, [2] presents an approach for the parallel execution of DEVS models by executing independent simultaneous events in parallel. Parallel execution is specifically performed i) when multiple output functions must be computed at the same time and ii) when multiple transition functions must be computed at the same time. Their authors claim that the proposal can—in theory—achieve an average execution time of 60% higher than the best possible parallel execution.

A non-trivial issue with executing the DEVS simulation protocol is the overhead incurred by the messages passing through the different levels of the structure to coordinate its components. Authors in [16,17] proposed a flattening algorithm to reduce this message exchange, removing intermediate coordinators. Every flat abstract DEVS simulator has the same basic structure: one root-coordinator, one coordinator (called flat-coordinator) and several simulators. The root-coordinator oversees the simulation, performs advances to global simulation time and determines when the simulation ends. The flat coordinator is in charge of the message exchange among simulators, executing the computation of events on the simulators and determining the minimum time for the next event. The simulators are responsible for the behavior of the atomic models, executing the output, transitions, and time advance functions. The corresponding flattened abstract DEVS simulator for the model shown in Figure 1(a) can be observed in Figure 1(c).

The above approach effectively reduces the number of transmitted messages, resulting in accelerated executions of simulations in practical implementations [17-20]. Authors in [21] performed a complexity analysis explaining the performance improvement of the algorithm.

2.1 Cadmium for Multicore Architectures

Cadmium [15] is an open-source C++ simulation software designed to execute a sequential algorithm of the DEVS simulation protocol [14] that guarantees correct simulation results. In addition, [14] showed that Cadmium outperforms ADEVS [22] when there are numerous simultaneous events. In Cadmium, all message passing is performed through function calls and returns, reducing overhead caused by exchanging messages and synchronizations among components. More recently, a new version of Cadmium allowed the efficient execution of flattened DEVS simulations and was later extended for execution in shared-memory multicore architectures [7].

To allow parallel flattened DEVS simulations, the Root-Coordinator, Flat-coordinator, and Simulator algorithms were adapted from their sequential version. PARALLEL-ROOT-COORDINATOR is similar to its sequential counterpart. Once started, it creates threads that share the *next_time* variable and the PARALLEL-FLAT-COORDINATOR (PFC) component in the *execute_simulation* function. The threads concurrently execute the *initialize* and *next_time* functions from the PFC. Then, threads run the *execute_output_functions*, *route_messages*, *execute_state_transitions*, and *next_time* functions in the PFC component (in parallel). After the end of the simulation, threads are destroyed, and simulation results are stored. A highly detailed definition of this component is shown in Figure 2.

```
PARALLEL-ROOT-COORDINATOR {
 variables: next_time,
            PFC; //PARALLEL-FLAT-COORDINATOR

execute_simulation(initial_time, sim_time) {

 Create threads, shared variables: PFC,next_time;

 PFC.initialize(initial_time);
 next_time = PFC.next_time(next_time);
 while(next_time<sim_time){
   PFC.execute_output_functions(next_time);
   PFC.route_messages();
   PFC.execute_state_transitions(next_time);
   next_time = PFC.next_time(next_time);
  }// end while
  Destroy threads
  Save simulation results
  } // end execute-simulation
} // end PARALLEL-ROOT-COORDINATOR
```

Figure 2: PARALLEL-ROOT-COORDINATOR component [7].

The PARALLEL-FLAT-COORDINATOR definition is presented in Figure 3. Its main idea is to execute subcomponents in parallel by assigning subsets of them to threads.

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

```
PARALLEL-FLAT-COORDINATOR {
 variables: outbox, inbox, next_time, ic,
   sub_comps; // internal-coupling, components
 initialize(initial_time){
   parallel-for each PAR_SIM in sub_comps:
       PAR-SIM.initialize(initial_time);
   end-parallel-for
   synchronization
 } //end init;

 execute_output_functions(next_time) {
   parallel-for each PAR-SIM in sub_comps:
     PAR-SIM.execute_output_function(next_time);
   end-parallel-for
   synchronization
 } //end execute_output_functions;

 execute_state_transitions(next_time){
   parallel-for each PAR-SIM in sub_comps:
     PAR-SIM.execute_state_transition(next_time);
   end-parallel-for
   synchronization
 } //end execute_state_transitions

 route_messages() {
  parallel-for each coupling in ic:
    outbox = coupling.origin.get_outbox();
    if(coupling.origin.get_outbox() != empty)
       coupling.destiny.add_to_inbox(outbox);
  end-parallel-for
  synchronization
 }

 next_time(){
  next_time = subcomponents.first.next_time();
  parallel-for-reduction each PAR-SIM in
     sub_comps:
  if(PAR-SIM.next_time() < next_time)
      next_time = PAR-SIM.next_time();
  end-parallel-for-reduction
  synchronization
  return next_time;
 }//end next_time

}//end PARALLEL-FLAT-COORDINATOR
```

Figure 3: PARALLEL-FLAT-COORDINATOR component [7].

Each function can be executed independently in parallel, and synchronized when finished. The element has a *next_time* variable, inbox, outbox, internal coupling (*ic*), and subcomponents (*sub_comps*). First, at the *initialize* function, the PARALLEL-SIMULATOR (called PAR_SIM) components are assigned to different threads and initialized (in parallel). Then, *execute_output_functions* runs the *execute_outputs* of each PAR_SIM component (in parallel), assigning subsets of such components to threads. After execution, the threads perform a synchronization. Similarly, *route_messages* routes messages among PAR_SIM components (in parallel) and synchronize when done. In order to route messages to a subcomponent (*sub_comps*), each thread checks on every sender to see if its output bag is not empty. If it is not, then it inserts that output bag into the input bag of the component. Again, *execute_state_transition* runs the *execute_state_transition* function of each PAR-SIM component. Finally, the *next_time* function executes in parallel for each PAR-SIM subcomponent. Unlike the previous functions, the threads perform a reduction to obtain the lowest *next_time* value, synchronizing when done.

Finally, since PARALLEL-SIMULATOR components execute in threads running in parallel, the only modification made to the sequential equivalent corresponds to ensuring exclusive access to adding messages to the inbox employing a lock at the *add_to_inbox* function. The definition is shown in Figure 4.

3 A PARALLEL GPU-BASED ALGORITHM TO EXECUTE DEVS SIMULATIONS

The algorithm presented here is based on the one described in section 2.1, but adapted for use in NVIDIA GPU systems, which we chose due to their availability and extensive set of programming tools. Unlike the multicore version of the algorithm, this version starts the execution sequentially on the CPU and then delegates all tasks required to execute the DEVS simulation protocol to a GPU kernel programmed with CUDA. On the GPU, each kernel executes the computations needed to perform the tasks. In addition, after each kernel finishes, the CPU waits to guarantee the complete execution of a task before starting the next.

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

```
PARALLEL-SIMULATOR {
  variables: inbox, outbox,last, next, model;

  init(t){
    last = t;
    next = t + model.time_advance();
  }

  execute_output_function(t) {
    if (next==t)
      outbox = model.output_function();
  }

  state_transition(t) {
    if (t == next) {
      if (inbox == empty)
        model.internal_transition();
      else
        model.confluent_transition(t-last,inbox);
      last = t;
      next = last + model.time_advance();
   } else {
      if (inbox != empty) {
        model.external_transition(t-last,inbox);
        last = t;
        next = last + model.time_advance();
      }
    }
    Inbox = outbox = empty;
  }

  next_time() {
    return next;
  }

  get_outbox(){
    return outbox;
  }
  add_to_inbox(message){
    lock(inbox);
    inbox.add(message);
    unlock(inbox);
  }
}//end PARALLEL-SIMULATOR
```

Figure 4: PARALLEL-SIMULATOR component [7].

The components GPU-ROOT-COORDINATOR, GPU-FLAT-COORDINATOR and GPU-PARALLEL-SIMULATOR below are specialized versions of those used in the flattened DEVS simulation algorithm in Cadmium [7, 22]. In Figure 5, we can see a definition for GPU-ROOT-COORDINATOR.

```
GPU-ROOT-COORDINATOR {
  variables: next_time,
             GFC; //DEVS GPU-FLAT-COORDINATOR

  execute_simulation(initial_time, sim_time) {
    Store GFC in Unified Memory
    GFC.initialize(initial_time);
    wait for GPU to complete kernel
    next_time = GFC.next_time(next_time);
    while(next_time<sim_time){
      GFC.execute_output_functions(next_time);
      wait for GPU to complete kernel
      GFC.route_messages();
      wait for GPU to complete kernel
      GFC.execute_state_transitions(next_time);
      wait for GPU to complete kernel
      n_time = GFC.next_time(next_time);
    }// end while
    Save simulation results
  } // end execute-simulation

} // end GPU-ROOT-COORDINATOR
```

Figure 5: GPU-ROOT-COORDINATOR component.

Unlike the parallel version for multicores presented in [7], *execute_simulation* starts by storing the GPU-FLAT-COORDINATOR component (GFC in Figure 5) in the unified memory defined by the CUDA programming tool so both CPU and GPU programs can access this component. After this, the CPU offloads the tasks that can be parallelly executed to the GPU, just as in the multicore version. The difference is that, in this version, all tasks are performed by calling kernel functions that execute on the GPU. First, the program calls *initialize* and *next_time* functions on the GPU-FLAT-COORDINATOR for their execution on the GPU. Then, a simulation loop starts, and in each cycle, the program executes functions: *execute_output_functions*, *route_messages*, *execute_state_transitions*, and *next_time*. Once the simulation loop finishes, the program stores the simulation results and ends its execution. The first component executed by the algorithm is the GPU-FLAT-COORDINATOR—the definition for this component is shown in Figure 6.

This component comprises several GPU kernel functions that maintain a similar structure, thus working similarly. To execute them, as many threads as GPU-SIMULATORS are in the simulation structure must be created. First, each GPU thread calculates its unique global ID, which will determine the index of the GPU-SIMULATOR it will execute on. The global ID is computed based on the block the thread belongs

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

to, the number of dimensions the grid has and the ID of the thread on its block. To ensure the thread is computing a valid memory location, they check if their global ID is less than the number of GPU-SIMULATORs on the structure. Besides this, the rest of each function works according to its purpose.

```
GPU-FLAT-COORDINATOR {
  variables: subcomponents, internal_couplings, outbox, inbox, next_time;
  __global__ initialize(initial_time)  {
    int i = blockId.x * blockDim.x + threadIdx.x;
    if (i<n_subcomponents)
          subcomponents[i].initialize(initial_time);
  }
  __global__ execute_output_functions(n_time) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n_subcomponents)
      if (subcomponents[i].next_time == n_time){
        subcomponents[i].execute_output_function(next_time);
  }
  __global__ route_messages() {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n_subcomponents)
      for each coupling in internal_couplings[i] {
        outbox = coupling.sender.get_outbox();
        if (outbox != empty)
          SIMULATOR.add_to_inbox(outbox);
      }
  }
  __global__ execute_state_transitions(next_time) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n_subcomponents)
      subcomponents[i].execute_state_transition(next_time);
  }
  __host__ next_time(next_time){
    partial_next_times[numBlocks];
    GPU_reduction(*partial_next_times)
    cudaDeviceSynchronize();
    next_time = partial_next_times[0];
    for(int i=0; i<numBlocks; i++) {
      if(partial_next_times[i] < next_time)
        next_time = partial_next_times[i];
    }
    return next_time;
  }
  __global__ GPU_reduction(*partial_next_times){
    __shared__ blockCache[threadsPerBlock]
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int blockIndex = threadIdx.x;
    int jump = blockDim.x/2;
    //init blockCache values
    if(i < n_subcomponents)
      blockCache[blockIndex] = subcomponents[i].next_time();
    __syncthreads();
    while(jump > 0){
      if(blockIndex < jump)
        if(blockCache[blockIndex] > blockCache[blockIndex+jump])
          blockCache[blockIndex] = blockCache[blockIndex+jump];
      __syncthreads();
      jump = jump/2;
    }
    if(blockIndex == 0)
      partial_next_times[blockIdx.x] = blockCache[0];
  }//end GPU_reduction
}//end GPU-FLAT-COORDINATOR
```

Figure 6: GPU-FLAT-COORDINATOR component.

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

The *initialize* function initializes each simulator by taking the initial time for the simulation as a parameter. This function calls the *initialize* function on each subcomponent in parallel, but unlike the multicore version, we do this in a GPU kernel function. Each GPU thread is responsible for executing the function on one GPU-SIMULATOR. These functions are independent and, therefore, can be executed in parallel.

The *execute_output_functions* routine calls the *execute_outputs* function on every GPU-SIMULATOR subcomponent in parallel. Each GPU thread is responsible for executing on one GPU-SIMULATOR. As these functions are also independent tasks, they are executed in parallel.

The *route_messages* function routes all messages among GPU-SIMULATORs in parallel. Here, each thread is responsible for routing messages to one GPU-SIMULATOR. The GPU thread checks on every sender if the output bag is not empty for routing messages to a GPU-SIMULATOR. If this is not the case, then it inserts that output bag into the input bag of the component. Once again, these functions are also independent and, therefore, can be executed in parallel. The function *execute_state_transitions* calls the *execute_state_transition* function on every GPU-SIMULATOR component in parallel. Like in the previous functions, each thread is responsible for executing on one GPU-SIMULATOR (in parallel). Finally, the *next_time* function obtains the minimum value of *next_time* among each GPU-SIMULATOR subcomponent in parallel through a GPU reduction operation. This starts by calling a kernel function called *GPU_reduction*. This function computes one minimum partial result per block of threads, because of the limitation of performing global synchronizations across all threads running in the GPU. CUDA only allows thread synchronization within the same block, thus making it impossible to coordinate threads on different blocks to obtain a single result. In order to perform a partial reduction on each group of threads, each thread starts by obtaining the next time of one GPU-SIMULATOR and storing it on a cache variable defined in shared GPU memory. This kind of GPU memory is only shared by threads of the same block and is faster than the main GPU memory, which is shared by all threads (even of different blocks). After initializing the cache variable, each group of threads performs a synchronization. Then, an iterative process starts on each thread block to compute a partial minimum. The general idea is that each thread will compare two cache values and store their minimum in the same cache data structure. Since each thread compares two values, each iteration completes with half as many values as it started. In the following iteration, we do the same over the remaining half values. Before we can read the values we just stored in the cache on each iteration, we need to ensure that every thread that needs to write in the cache has already done so. A synchronization among the threads in a block ensures this. We continue for $\log_2$(threads per block) iterations until we have the minimum of every value in the cache. In Figure 7, we can see an example of this reduction. After finishing this loop, each block has a single value stored in the first entry of the cache variable and corresponds to the minimum value that the threads in that block compared.

To finish the reduction, we must compute the minimum value from all partial minimum values computed by each block of threads. To do this, the partial results are compared by the CPU to obtain the minimum.

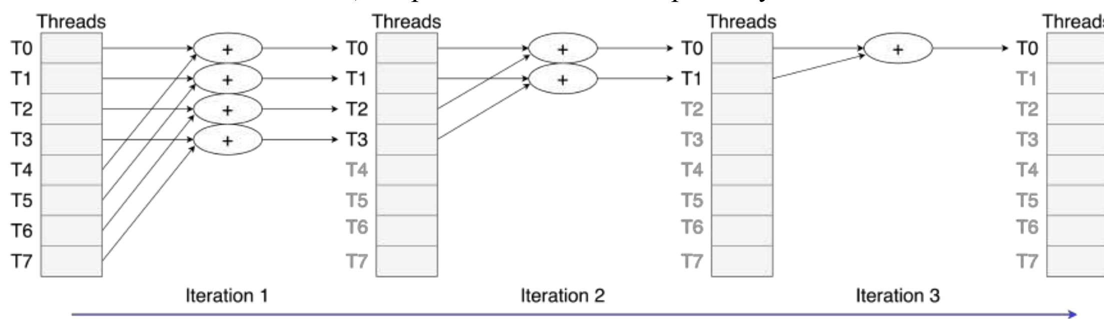

Figure 7: Reduction on GPU example.

Finally, the definition for the GPU-SIMULATOR component is shown in Figure 8. In this component, the variables and functions are like the ones defined for their equivalent in the sequential and parallel multicore versions. The difference in this version is that all functions are registered with the keyword __device__,

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

which indicates CUDA that these functions will be called from GPU functions and will execute on the GPU. Like on the sequential and multicore versions, this algorithm works by the interaction among a GPU-ROOT-COORDINATOR, a GPU-FLAT-COORDINATOR, and several GPU-SIMULATOR components. As mentioned, this definition extends previous work [7, 14,15] by implementing the DEVS simulation protocol [12,13] for execution on GPU devices.

```
GPU-SIMULATOR {
  variables: inbox, outbox, last, next, model;
  __device__ initialize(t) {
    last = t;
    next = t + model.time_advance();
  }
  __device__ execute_output_function(t) {
    if (next==t)
      outbox = model.output_function();
  }
  __device__ state_transition(t) {
    if (t == next) {
      if (inbox == empty)
        model.internal_transition();
      else
        model.confluent_transition(t-last,inbox);
      last = t;
  next = last +model.time_advance();
  } else {
   if (inbox != empty) {
        model.external_transition(t-last,inbox);
        last = t;
        next = last + model.time_advance();
      }
    }
    Inbox = outbox = empty;
  }
  __device__ next_time() { return next; }
  __device__ get_outbox(){ return outbox;
  }
  __device__ add_to_inbox(message){
    inbox.add(message);
  }
}//end GPU-SIMULATOR
```

Figure 8: GPU-SIMULATOR component.

4 EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of the proposed algorithm. First, we present the experimental setup and then the results we obtained.

4.1 Experimental Setup

For our experimental evaluation, we used two independent Cell-DEVS-based models. The first model corresponds to a synthetic benchmark characterized by being communication intensive. The second model is computationally intensive and consists of a SIR-based [23] epidemiological model to simulate the spreading of infectious diseases. The synthetic benchmark consists of a two-dimensional cell space containing a one-digit value as the state of each cell. Considering a Moore neighborhood, the rules state that each cell's initial value is zero and is changed to one in the following step. Then, they change their value back to zero, repeating the process until the end of the simulation. The SIR-based model divides the population into three compartments related to the stages of the disease that an individual passes. As such, individuals can be susceptible (S), infected (I) or recovered (R). The model is also a 2D cell space, where each cell contains a portion of the population in each S, I or R compartment. The set of rules describing the behavior of the SIR model is described in [24].

Both models were executed on two different computers with GPUs. The first one, which we call "NVIDIA 1050", is a computer with a quad-core AMD Ryzen 5 3550H and 16 GB of RAM, and a NVIDIA 1050 Q-MAX with 640 CUDA cores and 3 GB of global memory. The second computer, which we call "NVIDIA 1660", is equipped with a hexa-core Intel i5-9400 and an NVIDIA 1660 with 1408 CUDA cores and 6 GB of global memory. All experiments were run in 30 independent iterations, and we present the average results. The code of the models and simulator can be accessed in our GitHub repository.

4.2 Experimental Results

This section shows the proposed GPU algorithm's performance results (in terms of execution time and speedup). Two sets of execution are performed. First, we compare our algorithm in terms of execution times to a simpler version devised for execution in GPU architectures, which we refer to as "previous approach on GPU". Unlike the algorithm presented in this work, in the previous algorithm, the output and transition

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

functions execute on GPU kernels. In contrast, the rest of the tasks execute sequentially on the CPU. However, the algorithm of this proposal executes all tasks on the DEVS simulation protocol in parallel in the GPU. Then, both algorithms are compared to a CPU sequential version [7] to evaluate their speedup.

Figure 9 shows the results obtained for the synthetic benchmark problem. As expected, as the problem dimension grows—for both versions—time increases since there are more computations to execute. In Figure 9(a), executed on NVIDIA 1050, our algorithm improves performance over the previous GPU approach by up to 23.6 times because of the extra parallelism our algorithm introduces. In addition, to execute the previous approach, there is an additional overhead in transferring data between CPU and GPU memory. Each time the algorithm must execute on the GPU, data structures must be moved to the GPU memory. Then, after the GPU finishes and the computations continue on the CPU, data structures must be copied back to the CPU memory. This process occurs several times on each simulation step, creating a significant overhead. Even though the data transfers occur transparently to the user—using the unified model provided by CUDA—and do not involve additional effort by the programmer, the overhead in execution is significant, unlike our algorithm, which executes entirely on the GPU. Data structures are copied to the GPU memory once, and we only access data from the CPU to store simulation results once the simulation finishes.

Similarly, In Figure 9(b), executed on NVIDIA 1660, shows the results obtained for the synthetic benchmark problem executed on the NVIDIA 1660 computer. In this architecture, we can also see that, as expected, for both algorithms, the time increases as the problem size grows. Once again, this is because there are more cells when the problem size increases, and simulations require more computations. In addition, we can see that our algorithm improves performance over the previous version. In this architecture, our algorithm improves performance by up to 12.7X. As we explained in the previous results, this is because of our algorithm's additional parallelism, combined with the overhead of copying data between CPU and GPU memories. Next, we conducted the second set of experiments on the SIR model.

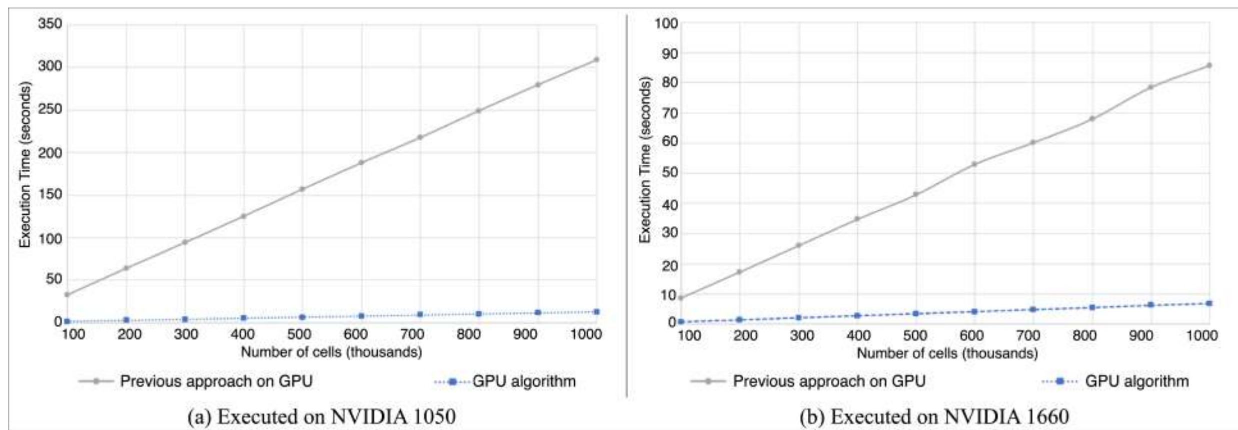

Figure 9: Execution time results for synthetic benchmark problem

Figure 10(a) shows the time results for the benchmark problem executed on NVIDIA 1050. As in the previous results, time increases when the problem dimension increases, as expected. Here, we can also see that our algorithm improves the previous approach, accelerating simulation by up to 8.8X. Finally, Figure 10(b) shows the times results for the SIR problem executed on NVIDIA 1660. Once again, as expected, the time increases when the problem dimension increases. In addition, like in the synthetic problem, we can see that our algorithm improves over the previous approach by 19.2X.

Results show the benefits of an algorithm that performs all tasks of the DEVS simulation protocol in parallel on the GPU. Our approach introduces additional parallelism and avoids the overhead of memory transfers

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

between CPU and GPU memories. This way, we improve performance and make the algorithm suitable for offloading GPU computations.

Next, we evaluate our GPU algorithm in terms of speedup. As mentioned earlier, we consider the results of a CPU sequential version described in [7], like in the multicore version of the same article. The speedup in the next set of results corresponds to the ratio of the sequential execution time to the GPU algorithm execution time, defined as *speedup = sequential algorithm time / GPU algorithm time*.

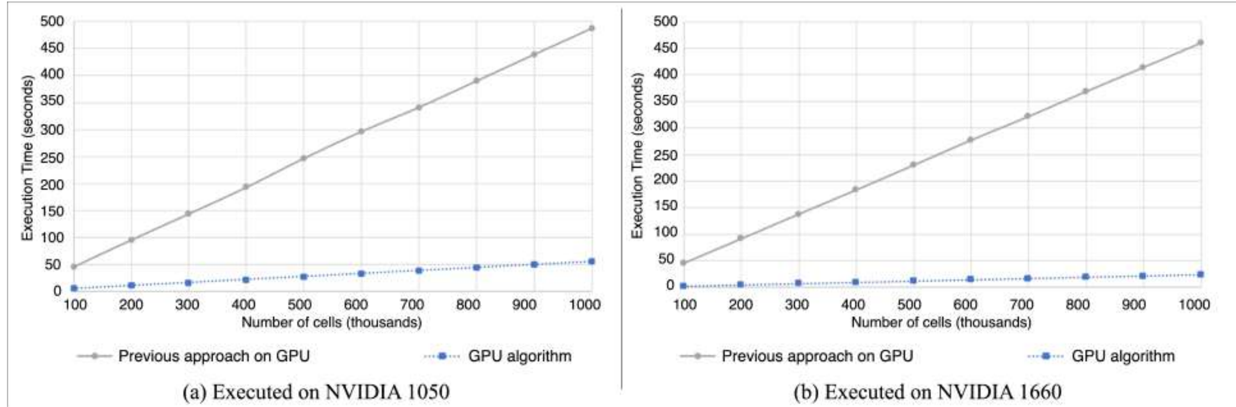

Figure 10: Execution time results for SIR.

In Figure 11(a), we can see the speedup results for the benchmark problem. We can see that our GPU algorithm accelerates several times the execution compared to the CPU sequential version. In addition, in both architectures, the speedup increases with the number of cells. The maximum speedups we obtained were 8.73 and 16.53 on NVIDIA 1050 and NVIDIA 1660, respectively. We should also note that the speedup on NVIDIA 1660 is close to twice as in the NVIDIA 1050 architectures. This is because NVIDIA 1660 is newer hardware with higher computing capabilities and a larger number of CUDA cores. Figure 11(b) shows the results obtained with our GPU algorithm on the SIR problem. The speedup obtained for the benchmark problem is higher than that obtained for the SIR problem on both architectures. This is because of the primary data type values used by the simulation models. In the benchmark, the state is represented by a single digit, whereas in the SIR model, we represent the state by a double-precision floating point value.

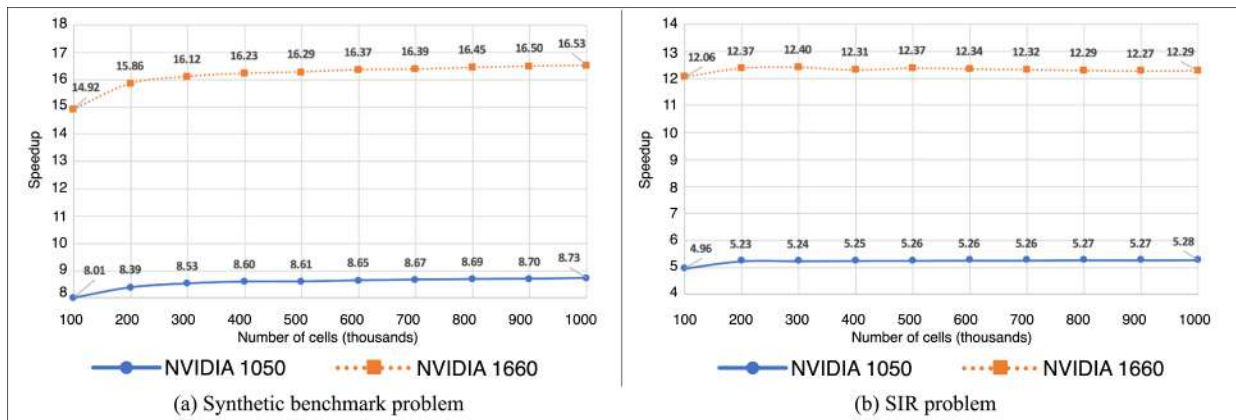

Figure 11: Speedup of GPU algorithms compared with Intel multicore sequential version.

On NVIDIA GPUs, performance varies depending on the datatype used. NVIDIA architectures are classified according to their CUDA computing capability. NVIDIA 1050 has CUDA capability 6.1, and in this type of architecture, the throughput for double precision floating point operations is 1/32 of the

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

throughput for integer values. Similarly, NVIDIA 1660 has a CUDA capability of 7.5, and in this type of architecture, double precision operations are 1/16 of the throughput for integer value operations [25]. For this reason, the speedup is lower on the SIR problem. Nevertheless, we achieved significant acceleration over the sequential version for both problems on both architectures.

Figure 12 shows a summary of the maximum speedup obtained with our GPU algorithms compared to a multicore algorithm running on an Intel multicore with 16 cores [7]. Our GPU algorithm improves the performance over the sequential and parallel versions on multicore architectures, achieving speedups up to 16.53 and 12.29 depending on the simulation model.

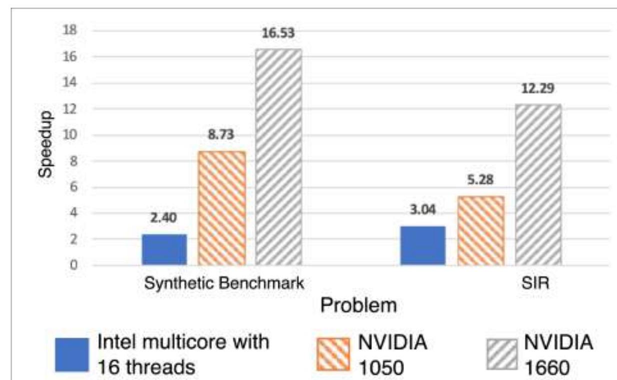

Figure 12: Summary of speedups achieved with multicore and GPU algorithms.

5 CONCLUSIONS

In this work, we presented an algorithm to execute DEVS simulations on heterogeneous computers with GPU architectures. This work extended a parallel algorithm for multicore architectures, whose approach is to offload all tasks on the DEVS simulation protocol on GPU architectures. Each of these tasks is distributed among the GPU cores, and this way, accelerating the simulations. As with the parallel multicore algorithm, its main advantage is that it allows parallel executions simply, delivering the same correct simulation results as the sequential version in all scenarios.

In addition, the experimental results show an important performance improvement. Executions accelerate several times for a synthetic benchmark and a real-world problem in two independent computers with different hardware characteristics. We showed that our algorithm performs better as we increase the number of components on both problems. Finally, we showed that our algorithm improves the performance over the sequential version executed on a CPU, achieving speedups up to 12.29 and 16.53 with the models used for its tests.

ACKNOWLEDGMENTS

Alonso Inostrosa-Psijas acknowledges support from Fondecyt de Iniciación 11230961 from ANID, Chile. The research has been partially funded by NSERC, Canada.

REFERENCES

[1] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press, 2018.

[2] B. P. Zeigler, "Using the parallel DEVS protocol for general robust simulation with near optimal performance," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 68-77, 2017.

[3] Y. Van Tendeloo and H. Vangheluwe, "An evaluation of DEVS simulation tools," *Simulation*, vol.

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

93, no. 2, pp. 103-121, 2017.

[4] Q. Liu and G. Wainer, "A performance evaluation of the lightweight time warp protocol in optimistic parallel simulation of DEVS-based environmental models," in *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 2009, pp. 27-34.

[5] S. Jafer and G. Wainer, "Conservative synchronization methods for parallel DEVS and Cell-DEVS," in *Proceedings of the 2011 Summer Computer Simulation Conference*. Citeseer, 2011, pp. 60-67.

[6] A. Inostrosa-Psijas, V. Gil-Costa, M. Marin, and G. Wainer, "Semi-asynchronous approximate parallel DEVS simulation of web search engines," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 7, p. e4149, 2018.

[7] G. G. Trabes, G. A. Wainer, and V. Gil-Costa, "A parallel algorithm to accelerate DEVS simulations in shared memory architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1609-1620, 2023.

[8] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1-6.

[9] S. Kim, J. Cho, and D. Park, "Accelerated large-scale simulation on DEVS based hybrid system using collaborative computation on multi-cores and GPUs," *Journal of the Korea Society for Simulation*, vol. 27, no. 3, pp. 1-11, 2018.

[10] M. G. Seok and T. G. Kim, "Parallel discrete event simulation for DEVS cellular models using a GPU," in *Proceedings of the 2012 Symposium on High Performance Computing*, 2012, pp. 1-7

[11] R. M. Fujimoto, "Research challenges in parallel and distributed simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 26, no. 4, pp. 1-29, 2016.

[12] A. C. Chow, B. P. Zeigler, and D. H. Kim, "Abstract simulator for the parallel DEVS formalism," in *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*. IEEE, 1994, pp. 157-163.

[13] J. Nutaro, "Parallel and distributed discrete event simulation," in *Theory of Modeling and Simulation*, B. P. Zeigler, A. Muzy, and E. Kofman, Eds. San Diego, CA, USA: Academic Press, 2019, ch. 14, pp. 339-372.

[14] D. Vicino, D. Niyonkuru, G. Wainer, and O. Dalle, "Sequential DEVS architecture," in *Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, 2015.

[15] L. Belloli, D. Vicino, C. Ruiz-Martín, and G. Wainer, "Building DEVS models with the cadmium tool," in *2019 Winter Simulation Conference (WSC)*. IEEE, 2019, pp. 45-59.

[16] K. Kim, W. Kang, B. Sagong, and H. Seo, "Efficient distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one," in *Proceedings 33rd Annual Simulation Symposium (SS 2000)*. IEEE, 2000, pp. 227-233.

[17] E. Glinsky and G. Wainer, "Definition of real-time simulation in the CD++ toolkit," in *Proceedings of SCS Summer Computer Simulation Conference*, 2002.

[18] R. Franceschini and P.-A. Bisgambiglia, "Decentralized approach for efficient simulation of DEVS models," *in IFIP International Conference on Advances in Production Management Systems.* Springer, 2014, pp. 336-343.

[19] G. Zacharewicz, M. E.-A. Hamri, C. Frydman, and N. Giambiasi, "A generalized discrete event system (G-DEVS) flattened simulation structure: Application to high-level architecture (HLA) compliant simulation of workflow," *Simulation*, vol. 86, no. 3, pp. 181-197, 2010.

[20] P.-A. Bisgambiglia and P. Bisgambiglia, "DecDEVS: New simulation algorithms to improve message handling in DEVS," *Open Journal of Modelling and Simulation*, vol. 9, no. 2, pp. 172 197, 2021.

[21] G. G. Trabes, V. Gil-Costa, and G. A. Wainer, "Complexity analysis on flattened DEVS simulations," in *2021 Winter Simulation Conference (WSC)*. IEEE, 2021, pp. 1-12.

[22] J. Nutaro, "A discrete event system simulator," https://web.ornl.gov/~nutarojj/adevs/, 2014, accessed Jan. 10, 2024.

[23] W. O. Kermack and A. G. McKendrick, "A contribution to the mathematical theory of epidemics," *Proceedings of the royal society of london. Series A, Containing papers of a mathematical and physical character*, vol. 115, no. 772, pp. 700-721, 1927

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.

[24] Hoya White, S., A. Martín del Rey, and G. Rodríguez Sánchez. "Modeling epidemics using cellular automata," *Applied mathematics and computation* vol. 186, pp. 193-202, 2007.

[25] NVIDIA, “CUDA C programming guide,” https://docs.nvidia.com/cuda/cuda-c-programming-guide /index.html#arithmetic-instructions, November 2023, accessed December 28, 2023.

AUTHOR BIOGRAPHIES

GUILLERMO G. TRABES received his Ph.D. in Electrical and Computer Engineering (Carleton University) and Computer Science (Universidad Nacional de San Luis). His research interests include high-performance computing applied to scientific applications, simulations and artificial intelligence. His email address is guillermotrabes@gmail.com.

ALONSO INOSTROSA-PSIJAS received a Ph.D. degree from Universidad de Santiago, Chile. Since 2021, he has been an Adjunct Professor at the School of Informatics Engineering at Universidad de Valparaiso, Chile. His research interests are related to parallel/distributed discrete-event simulation. He can be contacted at alonso.inostrosa@uv.cl.

VERONICA GIL-COSTA received her Ph.D. in Computer Science, both from Universidad Nacional de San Luis (UNSL), Argentina. She is a former researcher at Yahoo! Labs Santiago. She is currently an Associate Professor at the University of San Luis and a researcher at the National Research Council (CONICET) of Argentina. Her email address is gvcosta@unsl.edu.ar.

GABRIEL A. WAINER received the Ph.D. degree from Université d’Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada), where he is now a Full Professor. His current research interests are related to modelling methodologies and tools, parallel/distributed simulation, and real-time systems. His e-mail is gwainer@sce.carleton.ca. His website is http://www.sce.carleton.ca/faculty/wainer.

Authorized licensed use limited to: Carleton University. Downloaded on November 03,2024 at 17:36:22 UTC from IEEE Xplore. Restrictions apply.