Parallel Simulation of Equation-based Object-Oriented Models with Quantized State Systems on a GPU

Martina Maggio [*] [‡], Kristian Stavåker [†], Filippo Donida [*], Francesco Casella [*], Peter Fritzson [†]

Abstract

This work focuses on the use of parallel hardware to improve the simulation speed of equation-based object-oriented Modelica models. With this intention, a method has been developed that allows for the translation of a restricted class of Modelica models to parallel simulation code, targeted for the Nvidia Tesla architecture and based on the Quantized State Systems (QSS) simulation algorithm. The OpenModelica Compiler (OMC) has been extended with a new back-end module for automatic generation of the simulation code that uses the CUDA extensions to the C language to be executable with a General Purpose Graphic Processing Unit (GPGPU). Preliminary performance measurments of a small example model have been done on the Tesla architecture.

Keywords: Parallel Simulation, QSS algorithm, CUDA architecture, OpenModelica compiler, GPGPU

1 Introduction

Recent increases in the continuing growth of computing power predicted by Moore's law are mainly due to increased parallelism, rather than to increased clock frequency [16]. A challenge in the field of dynamic system simulation is to exploit this trend, reducing computation time via the use of parallel architectures [3, 12, 13].

Traditionally the majority of the parallel programming techniques are based on multi-CPU architectures. Recently, parallel execution of general purpose code has become cheaply available through the use of Graphic Processing Units (GPUs) that allows for general code execution, also known as General Purpose Graphic Processing Units (GPGPUs). The use of this particular hardware has been widely encouraged in recent years; in fact many applications have been developed, see for example [4, 10, 15].

The aim of this work is two-fold; as a first point the possibility of parallelization of the QSS algorithm *per se* together *with the chosen architecture* is investigated, while, as a second step, the parallel performance of the QSS integration method via automatically generated CUDA code is studied and some test are conducted to evaluate the chosen approach.

Since the Modelica language is used to describe many different classes of systems, in this work the test models have been restricted only to a subset:

- continuous time, time-invariant systems (with no events),
- index-1 DAE (if the index is greater than 1 the index reduction algorithm should be used before processing the model),
- initial values of states and values of parameters known at compile time, and inserted into the generated code as numbers,
- no implicit systems of nonlinear equations to be solved numerically.

The QSS integration method is a Discrete Event System (DEVS) method that was introduced in [5, 8], where the author suggested that it could be suitable for parallel execution. However, to the best of the authors' knowledge, no attempts have previously been made to deeply investigate the possibility of parallel implementations. In this work, a general discussion on the parallel QSS algorithm is done and a possible implementation for a particular shared-memory parallel architecture is presented.

The generated code, in fact, has been targeted for the Nvidia Tesla architecture, that "is useful to manage general purpose computation" [2]. To obtain speed improvement through fine-grained parallelism, the C language extension CUDA has been used, taking low level implementation details into account.

[*]Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy

[†]PELAB - Programming Environment Lab, Dept. Computer Science Linköping University, S-581 83 Linköping, Sweden

[‡]Corresponding author email: maggio@elet.polimi.it

This work is structured as follows. Section 2 describes the parallel architecture chosen to test the approach, highlighting the advantages and the disadvantages of the particular hardware. In section 3 the subset of the Modelica models targeted for the automatic generation of CUDA code is defined. The implementation of the parallel simulation through the available language and the strategies used to parallelize the Quantized State Systems simulation algorithm are treated in section 4. Section 5 describes the changes applied to the OpenModelica compiler to enable the code generation. Experimental results from model simulations are described in section 6 while in section 7 the conclusions of this work are explained and some proposals for future developments are sketched.

2 Parallelism with a Graphic Card

CUDA stands for *Compute Unified Device Architecture*, it is a C language extension developed by Nvidia with the intention of making it possible to exploit the massive parallelism found in GPUs for general purpose computing. Beyond the large number of computing cores available with the GPU architecture, the most interesting advantage is the presence of fast threads and a fast shared memory region, that leads to improved performance in memory writes and readbacks to and from the GPU.

On the other hand there are also some strong limitations: first of all, the language is a recursion-free, function-pointer-free subset of the C language, plus some simple extensions for managing the parallelism and allowing a single process run spread across multiple disjoint memory spaces. The memory management has to be taken in serious consideration, since there are strong limitations on the available address space. The bus bandwidth and the latency between the CPU and the GPU may be a bottleneck. Moreover, threads should be run in groups of at least 32 for best performance, with the total number of threads numbering in the thousands. Branches in the program code do not impact the performance significantly, provided that each of 32 threads (in a group) takes the same execution path. The SIMD (single instruction, multiple data) execution model of all thread in a group becomes a significant limitation for every inherently divergent task, in fact when taking a diverging branch the code execution will be significantly slowed down since each different code variant has to be executed in sequence.

The SIMT (single instruction, multiple thread) architecture inserts a new element in the Flynn taxonomy [6], since the groups of threads execute the same code among the core components in a single cluster and not among all the processing units as in the classical SIMD method; in fact MIMD parallelism can be achieved with a careful allocation of the threads to the clusters. Nonetheless, the parallelism exploitation is not trivial since the code needs to be designed *ad hoc* for the specific hardware to limit diverging branches. Specifically, the objective is to make threads run as long as possible over the same portion of the code. This is somehow in contrast with the concept of "parallel architecture" where every processing component can perform different operations, on the same or on different data. As stated, some code portions should be processed with a MIMD (multiple instruction, multiple data) method and CUDA partially allows it through the thread distribution to the available multiprocessors.

In the following, the architecture is described in detail; each graphic card is made up of common core components. The Tesla architecture, see figure 1, is based on a scalable processor array (SPA), with a certain number of streaming-processor (SP) cores. These SP cores are organized in sets of streaming multiprocessors (SMs) and in processor clusters (TPCs), i.e., independent processing units.

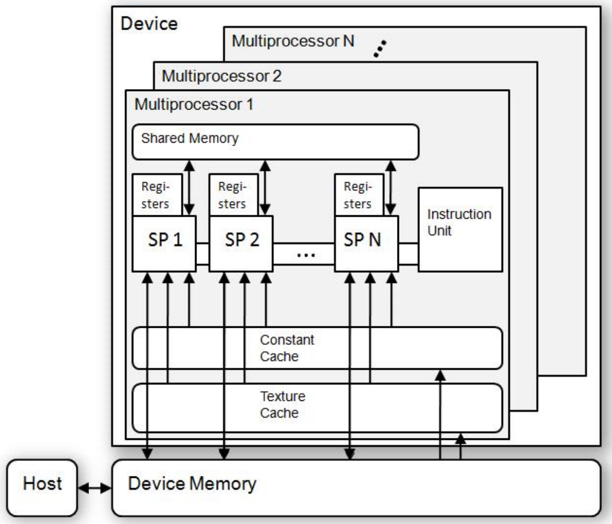

Figure 1: The Nvidia Tesla architecture.

The graphic unit interface communicates with the host processor, replying to commands, fetching data from system memory, checking command consistency, and performing context switching. The work distribution units forward the input assembler's output stream to the array of processors. The processor array executes thread programs and provides thread control and management. The number of clusters determines the processing performance and scales from one processor cluster in a small graphic card to twenty or more in high-level hardware.

The streaming multiprocessors consist of eight streaming-processor cores, a multi threaded instruction fetch and issue unit, an instruction cache, a read-only constant cache, and some shared memory. The shared memory holds input buffers or shared data for parallel computing. A low-latency interconnect network between the streaming-processors and the shared-memory banks provides shared memory access.

In this work we are using two different graphic cards, the more powerful one is the *Nvidia Tesla C1060*, which features 240 stream processors organized in 30 clusters of 8 SIMD processors, supports single and double precision, has 4 GB of memory and a memory bandwidth of 102 GB/s. According to the specification [2] this hardware has a “Compute Capability 1.3”, this means that the maximum number of threads per block is 512, the maximum number of active blocks per multiprocessor is 8 and each multiprocessor is composed of eight processors, so that a multiprocessor is able to process the 32 threads of a warp in four clock cycles. It also supports some features like warp voting, that are not used in this work.

The results obtained are compared with data obtained from an *Nvidia GeForce 8600*, which has just 32 stream processors, organized in 4 clusters, only supports single precision, has 512 MB of memory and a memory bandwidth of 57.6 GB/s. Its “Compute Capability” is just 1.1; this means that this hardware does not support double precision and has not the additional features of the previous. In order to compare the behaviour with different numbers of clusters, single precision numbers are used for both tests.

The memory management instructions access three read/write memory spaces:

- local memory for per-thread, private, temporary data (implemented in external DRAM);
- shared memory for low-latency access to data shared by cooperating threads in the same SM;
- global memory for data shared by all threads of a computing application (implemented in external DRAM).

A more detailed survey of the architecture features can be found in [11].

In order to exploit parallelism with the CUDA architecture, a programmer has to write a serial program that calls parallel kernels, which can be simple functions or full programs. The CUDA program executes serial code on the CPU and executes parallel kernels across a set of parallel threads on the GPU. The programmer has to organize these threads into a hierarchy of thread blocks in order to obtain SIMD, SIMT and MIMD parallelism. In fact, when a CUDA program on the host CPU invokes a kernel parallel execution, the thread blocks are enumerated and distributed to free multiprocessors on the device. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. Figure 2 shows the execution flow of the code.

3 A Restricted set of Models: the Parallelizable Modelica Models

Even if the long term goal is to be able to automatically generate parallel code for every possible Modelica model, in this work the study is restricted to a subset of purely continuous-time, time-invariant systems with time-varying external inputs.

This subclass of systems can be brought into standard form by applying index reduction and BLT algorithms, thereby generating code corresponding to:

$$\left\{ \begin{array}{lcl} \dot{x} & = & f(x,u) \\ y & = & g(x,u) \end{array} \right. \tag{1}$$

In order to simulate the system (1) it is necessary to implement functions for calculating the derivative of each state variable, as well as the output variables. This should be done within the graphic card kernel space. A function for the thread management is also needed: this function should be able to start a new thread and assign it to one of the core components preserving the load balance. For the current implementation, the load balance could be improved considering for example the estimated load of each new thread.

In addition some structural information about the mathematical representation of the model is required, i.e., the number of the state variable of the index reduced model and the number of outputs. It is also important to stress that the output computation should be executed within the card kernel space, thus resulting in a minimal overhead.

Another important aspect for simulation of the model is the QSS integration step. In this work we used a constant quantization step, unchanged for all

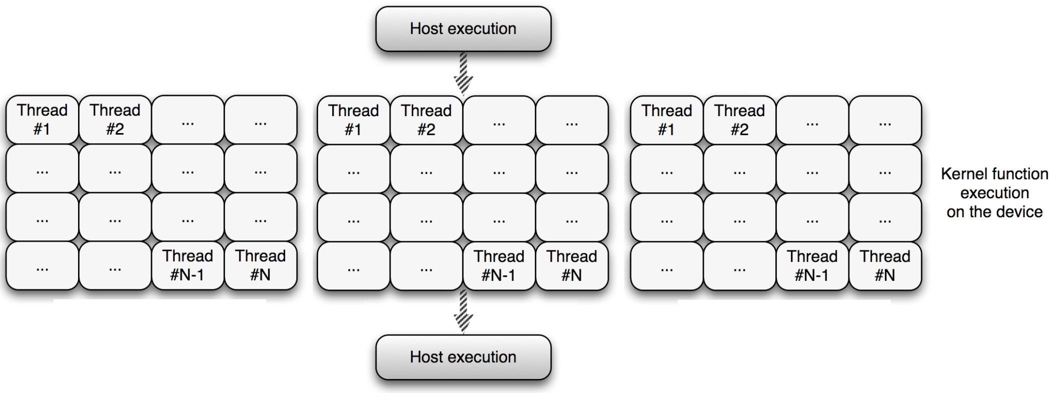

Figure 2: Execution flow example; the host machine asks the graphic device to compute the parallel execution of some CUDA threads, divided in blocks and assigned to the clusters of multiprocessors. After all thread terminations the control is returned to the host which can run the next instruction.

the state variables (for more detail see section 4) but a different quantization step can be used for each state variable, with minor modification to the code. Finally, input variables should be known *a priori* for the correct QSS algorithm execution. The input signals are pre-processed to compute the QSS inputs, expressed as piecewise constant trajectories.

4 Quantized State System Simulation and Parallelism

The QSS algorithm is a method to solve ODE systems; there are different ODE solvers, varying in approximation orders or in time slicing (i.e., how often they compute new state values). Moreover, there are explicit and implicit algorithms to compute the values of the state variables at the next discrete time instant, given current and past state and derivative information. Rather than making use of the concept of time slicing to reduce a continuous-time problem to an (in some way equivalent) discrete-time problem, the QSS method employs the concept of state quantization for the same purpose.

Given the current value of a state variable, $x_i = Q_i \in X_i$ where X_i is an ordered increasing set of discrete values that the state variable may assume; the QSS algorithm calculates when is the earliest time instant at which this state variable shall reach either the next higher or the next lower discrete level in the set.

The algorithm transforms a continuous time system in a Discrete Event System (DEVS) [18, 17]. The QSS algorithm has been studied in depth, and it has been proved by mathematical theorems that a limited boundary error exists when transforming a continuous time system into a DEVS one, i.e.:

$$\dot{x} = f(x,u) \longrightarrow \dot{x} = f(q,u) \qquad (2)$$

where the state vector x becomes a "quantized state vector" q where state values are in the corresponding set. The quantized state vector is a vector of discretized states where each state varies according to an hysteretic quantization function [8]. Suppose u are described by a piecewise constant trajectories (i.e., are described by events that at a certain time makes the value of u_i change from $u_{i\ old}$ to $u_{i\ new}$).

Simulate a system with the QSS algorithm means applying a variable-step techniques. The algorithm adjusts the time instant at which the state variable is re-evaluated to the speed of change of that state variable, and it is naturally asynchronous. This means that different state variables update their state values separately and independently of each other at different instants of time.

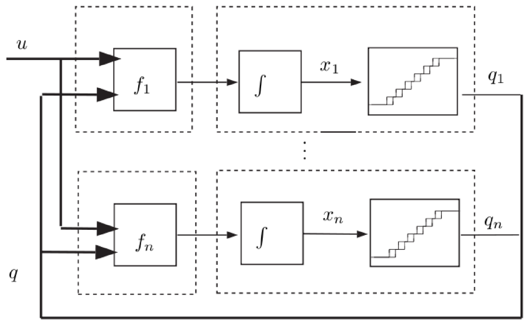

Figure 3: The scheme of a QSS model.

Specifically, the QSS algorithm consists in the creation of a coupled DEVS model, similar to the one in figure 3, where each state variable has an associated DEVS subsystem and the subsystem interconnection is based on the dependency between state variables and derivative equations. The events of the DEVS model are fired when the hysteretic quantization threshold are reached. The simulation therefore consists of three different steps:

- Search the DEVS subsystem that is the next to perform an internal transition, according to its internal time and to the derivative value. Suppose that the event time is t_{next} and the associated state variable is x_i. If $t_{next} > t_{inputevent}$ than set $t_{next} = t_{inputevent}$ and perform the input change.

- Advance the simulation time from current time to t_{next} and execute the internal transition function of the model associated to x_i or the input change associated to u_i.

- Propagate the new output event produced by the transition to the connected state variable DEVS models.

This approach is very interesting for parallel simulation since "*due to the asynchronous behavior, the DEVS models can be implemented in parallel in a very easy and efficient way*" [8]. As noticed, the QSS algorithm is naturally keen to be parallelized, because of the possibility to separately compute the derivatives state variables and the time events schedule; however some considerations are indeed.

The interested reader can refer to [9] for a detailed treatment on the matter, however, for the purposes of this work, the QSS integration method can be briefly described as follows.

For the first step, assume that the initial values of the state variables are known, the derivative of the state variables are computed using the model equations; this part of the code advantages from the MIMD execution model. After that calculation, the time of the new event is calculated; this code section exploits completely SIMD parallelism because all the computing threads execute the same code on different data portion. The second step consists in the time advance, a new event is registered if the values of one of the inputs changes or if one of the bounds of the quantized state function is reached. To verify the second possibility the minimum time advance for the state variable vector is taken into account. When an event occurs, each value of the state variables is re-computed, according to the new values of the inputs and/or the state variables and the quantized integrators are updated. Here the SIMD parallelism is exploited as well as in the previous part, due to the same reason (the same code executes on different data element). The last algorithm step does not need further explaination within the chosen architecture due to the fact that data are saved in the shared memory without need for propagation.

As shown, the specific architecture cannot be neglected when trying to asses the parallelization performance. A very careful analysis is needed to exploit the architecture dependent features. The first difference to be considered is the one between a *message passing* and a *shared memory* architecture. In [7] the authors make a comparison between these different architecture models.

For our application a message passing architecture would be interesting, but has some limitations. Each processor can manage a single or a group of DEVS subsystems, receiving events from the connected one. This is not particularly flexible, in fact, while the number of processors is fixed, the number of subsystems depends on the particular model. The grouping itself should be performed according to subsystems connection; in order to minimize the number of exchanged messages.

A shared memory architecture, as the Nvidia Tesla is, is more flexible but much attention has to be given to the algorithm definition. Since the Nvidia TESLA architecture requires all the computing cores in the same group to compute the same instruction at the same time, good performance can be achieved via the definition of a state vector array. Each derivative state value is calculated within a separate thread.

In this case the code to compute such values is different for each state variable, therefore the SIMD model is not performing well. A MIMD-fashion code should be produced. The speed-up is limited from the number of clusters present in the architecture, the execution is in fact parallel for each group of clusters. When all threads finish, the derivative values have been calculated and the threads execute the same portion of code (therefore speeding up) to calculate the next time event for each variable. This part of the code should gain an advantage from the SIMD model as every thread execute the same code on different data portion (i.e., following the *single instruction, multiple data* technique). After doing that the next time event of the QSS simulation is determined and processed.

In summary, for the particular architecture and programming technique, the *derivative calculation* part of the code is not completely parallel, while the *system advance* part takes full advantage of the hardware possibilities.

5 Extracting the Model from Modelica Code

The OpenModelica Compiler (OMC) [1] is an open source compiler and development environment for the Modelica language that can be used for, among other things, research in language technology and code generation. In this work we have extended the back-end of the compiler with a new module GPUpar that generates simulation code according to the specifications given in this paper. This module can be turned on and off with a compiler flag. If this module is instructed to run it will take the equation system right after the matching and index reduction phases and generate CUDA C-code.

5.1 Overview

A brief overview of the interesting internal call chain in the compiler can be seen in figure 4.

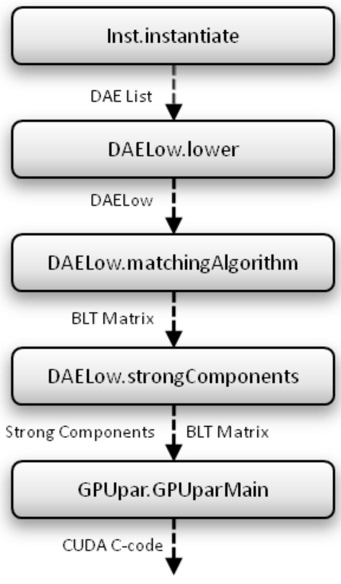

Figure 4: Internal call chain in the OpenModelica compiler to obtain parallel CUDA code.

The flattening phase takes the abstract syntax representation of the initial code and instantiates it (flattening, type checking, etc.) and the result is a list of so-called DAE elements. Here we are only interested in DAE elements that are equations. The list of DAE elements/equations is then transformed into a more suitable form called DAELow by DAELow.lower. The DAELow form contains the equations as well as all the variables and parameters. After this sorting, index reduction, strong component gathering, etc. is performed. The resulting data structures - BLT Matrix, strong components, DAELow form, etc. - are then passed into our new GPUpar module (in the normal case with serial simulation code we would call the module Simcodegen instead at this point).

5.2 GPUpar Module

In this module different kernel and header files are generated in succession. In order to generate the model-specific files, some data have to be computed from the DAELow form. The most important things to consider are:

- A *derivative* function which contains the algorithm for the time derivative computation is generated in the CUDA C-code for each state variable. If the time derivative calculation relies on other equations, they are also added to the *derivative* function.
- An *output* function for computing the output values is generated in the CUDA C-code for each output variable. As for the *derivative* function, each of them can also contain other equations if necessary.
- Initial variable (and parameter) values must be gathered from the list of variables in the DAELow form.

The additional equations necessary for the single *derivative*/*output* functions, where present, form a subtree having the main equation as the root node. An existing function (DAELow.markStateEquations) was slightly modified to handle with this problem. All the equations are also brought into solved form (explicit form) by calling Exp.solve and the equations are sorted by using information obtained in the sorting phase (which was run before GPUpar was called). The initial values are gathered in a rather straight-forward manner by traversing the list of variables. Finally, in the generated code some of the variables are stored in different arrays: xd (derivatives), x (state variables), y (output variables), u (input variables) and p (parameters). At the beginning of the GPUpar module an environment is created that contains a mapping between each variable/parameter and the array name plus the index number in this array. This environment is then used when the CUDA C-code is generated to find the correct array and index to print for a given variable.

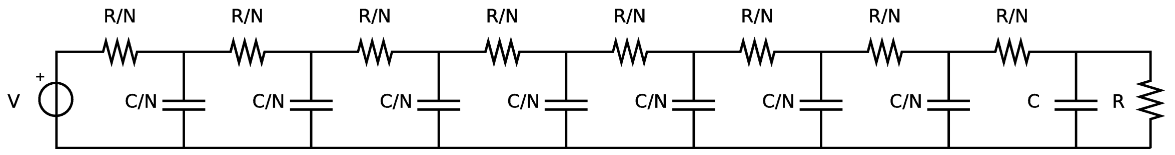

Figure 5: Test case example.

Appendix A contains a Modelica model and a part of the code necessary for simulating that model with the Nvidia architecture. In particular the missing files are model independent and can be found in [14].

6 Experimental Results

In this section a test case is presented, to evaluate the CUDA code performances. The two mentioned graphic cards are tested and a summary of the comparison between them is reported. The execution times are measured using the `clock()` function provided by the CUDA library. The initial time is obtained at the beginning of the program, before the memory allocation, in order to evaluate the architecture properly. The end time is measured when the simulation stops with the same function call and the difference between them is divided by the `CLOCKS_PER_SEC` constant, to compare architectures with different clock periods. The *parallel* algorithm is compared to the *sequential* one, where a single thread is executed on the graphic card and takes care of the computation sequentially.

The code for the circuit model of figure 5 is generated and executed. The depicted model has eight state variables that stands for the voltages in the eight capacitors. The model is then extended to sixteen, thirty-two and sixty-four state variables while keeping the same structure to prove the method scalability.

The following considerations apply to the model with N state variables. The circuit consists of a generator voltage that comprises $N-1$ different branches; each of them is composed by a resistor with resistance R/N and of a capacitor with capacitance C/N. The last branch is made up of the resistor with resistance R/N and a capacitor with capacitance C together with a resistor with resistance R in parallel. The only input of the system, in the following referred as u, is the voltage V, that is supposed to be a square wave with rise time and fall time of 1s and voltage of 1Volt. The equation model with $N=8$ is therefore

$$\begin{cases} \dot{x}_0(t) &= \frac{N^2}{RC}(-2x_0(t)+u+x_1(t)) \\ \dot{x}_1(t) &= \frac{N^2}{RC}(-2x_1(t)+x_0(t)+x_2(t)) \\ \dot{x}_2(t) &= \frac{N^2}{RC}(-2x_2(t)+x_1(t)+x_3(t)) \\ \dot{x}_3(t) &= \frac{N^2}{RC}(-2x_3(t)+x_2(t)+x_4(t)) \\ \dot{x}_4(t) &= \frac{N^2}{RC}(-2x_4(t)+x_3(t)+x_5(t)) \\ \dot{x}_5(t) &= \frac{N^2}{RC}(-2x_5(t)+x_4(t)+x_6(t)) \\ \dot{x}_6(t) &= \frac{N^2}{RC}(-2x_6(t)+x_5(t)+x_7(t)) \\ \dot{x}_7(t) &= \frac{N}{RC}(-R(\frac{N+1}{N})x_7(t)+x_6(t)) \end{cases} \tag{3}$$

and can be easily generalized to $N = 8\times i$ with i being an integer value ($i = 1,2,3,\dots$). The tests are conducted with $R = 1k\Omega$, $C = 1m$F and with a quantum of 0.001 for the QSS algorithm execution.

The results with the Nvidia Tesla GeForce 8600 can be seen in Table 1. Table 2 contains the results with the Nvidia Tesla C1060 when just one cluster is used to compute the derivative values, while Table 3 reports the data with the same graphic card when all the available clusters are used.

	parallel [s]	**sequential** [s]	**speed-up**
8-statevar	6.26	7.07	1.129
16-statevar	8.04	10.27	1.277
32-statevar	27.02	45.55	1.685
64-statevar	103.18	507.38	4.917

Table 1: Execution times and speed-up with the GeForce 8600.

	parallel [s]	**sequential** [s]	**speed-up**
8-statevar	1.06	5.71	5.387
16-statevar	8.11	9.07	1.118
32-statevar	22.91	47.30	2.065
64-statevar	208.76	711.00	3.406

Table 2: Execution times and speed-up with the C1060 using one cluster for the derivative calculation.

In figure 6 a summary of the obtained speed-up values is presented.

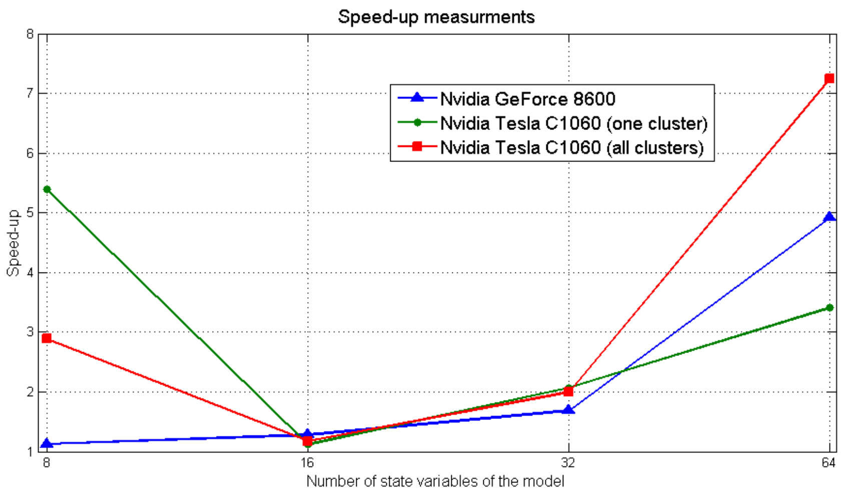

Figure 6: Speed-up measurements: comparison between a GeForce 8600 and an Nvidia Tesla C1060 when the number of state variables in the model changes.

The results presented are promising but should investigated further in order to understand the scalability of the proposed solution. Moreover, the new tools available from Nvidia (e.g., a profiler) give the possibility of a more careful analysis of the performances.

7 Conclusions and Future Research

This work considers the parallelization of the QSS algorithm using a GPGPU. As shown in section 4 the implementation of the algorithm can not neglect the particular hardware architecture. In this case the difficulties are essentially related to the fact that the Nvidia Tesla GPGPU is not a completely general parallel architecture. The memory consumption should also be taken into account. In particular, a problem with 256 state variables requires more than $\frac{(5\times64+1\times32)\times256}{8}[Bytes] = 11[Mb]$, while a case with 1024 state variables would require $43[Mb]$.

Surely, the side effects of the diverging branches has to be furthermore reduced. A comparison between the code that uses just one cluster of multiprocessors and a complete has been performed; however, further studies are still necessary to investigate possible extensions, e.g. for exploiting the computational power of each processor within the cluster.

	parallel [s]	**sequential** [s]	**speed-up**
8-statevar	1.98	5.71	2.884
16-statevar	7.73	9.07	1.173
32-statevar	23.73	47.30	1.993
64-statevar	98.09	711.00	7.248

Table 3: Execution times and speed-up with the C1060 using all the clusters for the derivative calculation.

Despite this, the results are promising albeit preliminary. Future work will compare QSS-based parallel method with other parallel implementations and investigate how the Tesla architecture thread manager allocates threads to the different computing cores. A profiling analysis is needed too, in order to understand if the limitations in the speed-up are caused by physical limits of the architecture or due to the non-exploitable hardware facilities.

References

[1] The OpenModelica project webpage: http://www.openmodelica.org.

[2] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2008.

[3] P. Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Linköping University, Department of Computer and Information Science, 2006.

[4] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *Processing the 2009 International Conference on Parallel (ICPP)*, 2009.

[5] F.E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, 2006.

[6] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948–960, 1972.

[7] A. C. Klaiber and H. M. Levy. A comparison of message passing and shared memory architectures for data parallel programs. *SIGARCH Comput. Archit. News*, 22(2):94–105, 1994.

[8] E. Kofman. *Discrete Event Based Simulation and Control of Continuous Systems*. PhD thesis, School of Electronic Engineering - FCEIA Universidad Nacional de Rosario, 2003.

[9] Ernesto Kofman and Sergio Junco. Quantized-state systems: a DEVS approach for continuous system simulation. *Trans. Soc. Comput. Simul. Int.*, 18(3):123–132, 2001.

[10] H. Li and L. Petzold. Efficient parallellization of stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. Technical report, Dept. Computer Science, University of California, Santa Barbara, 2008.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[12] H. Lundvall. Automatic paralleliztion using pipelining for equation-based simulation languages, 2008. Lic. Thesis.

[13] H. Lundvall, K. Stavåker, P. Fritzson, and C. Kessler. Automatic parallelization of simulation code for equation-based models with software pipelining and measurements on three platforms. *Computer architecture news, Special issue MCC08 - Multicore computing 2008*, 36(5), 2008.

[14] M. Maggio. Simulazione di modelli orientati agli oggetti su architetture parallele tramite algoritmo QSS. Master thesis. Politecnico di Milano, Dipartimento di Elettronica ed Infomazione, 2008.

[15] Michael Schwarz and Marc Stamminger. Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum*, 28(2):365–374, 2009.

[16] H. Shutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3).

[17] Bernard P. Zeigler, Tag G. Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, London, January 2000.

[18] Bernard P. Zeigler, Hae Sang Song, Tag Gon Kim, and Herbert Praehofer. DEVS framework for modelling, simulation, analysis, and design of hybrid systems. In *In Proceedings of HSAC*, pages 529–551. Springer-Verlag, 1996.

Appendix A: Code References

The following code example contains the model dependent part of the code for the generation of the experiment presented in Section 6 with 8 state variables, where three output variables are defined.

```
model Test_Model
 parameter Integer N = 8;
 input Real inputVars[1](start = 0.0);
 Real stateVars[N](start = 0.0);
 output Real outputVars[3];
equation

 der(stateVars[1]) = N*N * (-2.0*stateVars[1] +
   stateVars[2] + inputVars[1]);

 for i in 2:(N-1) loop
 der(stateVars[i]) = N*N * (-2.0*stateVars[i] +
   stateVars[i-1] + stateVars[i+1]);
 end for;

 der(stateVars[N]) = N * (stateVars[N-1] -
   1000 * ((N+1)/N) * stateVars[N]);

 outputVars[1] = stateVars[1];
 outputVars[2] = stateVars[4];
 outputVars[3] = stateVars[N];
end Test_Model;
```

The translation phase produces two output files: `model.h` and `model.cu`. The first one is the C-CUDA header and contains the function prototypes of the routine contained in the second one.

```
/*********************************
 * MODEL.H
 ********************************/
#ifdef _MODEL_H
#define _MODEL_H

#define NUMBER_STATES 8
#define NUMBER_INPUTS 1
#define NUMBER_OUTPUT 3
#define NUMBER_EVENTS 10
#define SIMULATION_TIME 10
#define SIMULATION_STEP 0.001

/* Initializations */
void initializeSystem(float* x, float* u);

void initializeEvents(float* t, unsigned* i, float* v);
```

```
/* Derivative calculation */
__global__ void derivative
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx7
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx6
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx5
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx4
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx3
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx2
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx1
 (float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx0
 (float* dx, float* x, float* u, float* t, unsigned* c);

 /* Output calculation */
__global__ void output
 (float* y, float* x, float* u, float* t, unsigned* c);
__device__ void y2
 (float* y, float* x, float* u, float* t, unsigned* c);
__device__ void y1
 (float* y, float* x, float* u, float* t, unsigned* c);
__device__ void y0
 (float* y, float* x, float* u, float* t, unsigned* c);

#endif

/*********************************
 * MODEL.CU
 ********************************/
#include "inclusion.h"
#include "model.h"

/* Initializations */
void initializeSystem(float *x, float* u) {
  int i;
  u[0]=0.0;
  for(i=0;i<NUMBER_STATES;i++) x[i]=0.0;
}

void initializeEvents(float* t, unsigned* i, float* v) {
  t[0] = 1; i[0] = 0; v[0] = 1;
  t[1] = 2; i[1] = 0; v[1] = 0;
  t[2] = 3; i[2] = 0; v[2] = 1;
  t[3] = 4; i[3] = 0; v[3] = 0;
  t[4] = 5; i[4] = 0; v[4] = 1;
  t[5] = 6; i[5] = 0; v[5] = 0;
  t[6] = 7; i[6] = 0; v[6] = 1;
  t[7] = 8; i[7] = 0; v[7] = 0;
  t[8] = 9; i[8] = 0; v[8] = 1;
  t[9] = 10;i[9] = 0; v[9] = 0;
}

/* Derivative calculation */
__global__ void derivative
(float* dx, float* x, float* u, float* t, unsigned* c) {
int i = threadIdx.x;
switch(i) {
  case 7: dx7(dx, x, u, t, c); break;
  case 6: dx6(dx, x, u, t, c); break;
  case 5: dx5(dx, x, u, t, c); break;
  case 4: dx4(dx, x, u, t, c); break;
  case 3: dx3(dx, x, u, t, c); break;
  case 2: dx2(dx, x, u, t, c); break;
  case 1: dx1(dx, x, u, t, c); break;
  case 0: dx0(dx, x, u, t, c); break;
  }
}
__device__ void dx7
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[7] = 8.0 * (x[6] - 1000 * 1.0625 * x[7]);
}
__device__ void dx6
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[6] = 16384.0 * (-2.0 * x[6] + x[5] + x[7]);
}
__device__ void dx5
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[5] = 16384.0 * (-2.0 * x[5] + x[4] + x[6]);
}
__device__ void dx4
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[4] = 16384.0 * (-2.0 * x[4] + x[3] + x[5]);
}
__device__ void dx3
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[3] = 16384.0 * (-2.0 * x[3] + x[2] + x[4]);
}
__device__ void dx2
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[2] = 16384.0 * (-2.0 * x[2] + x[1] + x[3]);
}
__device__ void dx1
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[1] = 16384.0 * (-2.0 * x[1] + x[0] + x[2]);
}
__device__ void dx0
(float* dx, float* x, float* u, float* t, unsigned* c) {
dx[0] = 16384.0 * (-2.0 * x[0] + x[1] + u[0]);
}
/* Output calculation */
__global__ void output
(float* y, float* x, float* u, float* t, unsigned* c) {
int i = threadIdx.x;
switch(i) {
  case 2: y2(y, x, u, t, c); break;
  case 1: y1(y, x, u, t, c); break;
  case 0: y0(y, x, u, t, c); break;
  }
}
__device__ void y2
(float* y, float* x, float* u, float* t, unsigned* c) {
y[2] = x[7];
}
__device__ void y1
(float* y, float* x, float* u, float* t, unsigned* c) {
y[1] = x[3];
}
__device__ void y0
(float* y, float* x, float* u, float* t, unsigned* c) {
y[0] = x[0];
}
```