Parallel simulation of equation-based models on CUDA-enabled GPUs

Conference Paper · October 2010

DOI: 10.1145/2039312.2039317

CITATIONS
8

READS
151

3 authors, including:

Per Ostlund
Linköping University
2 PUBLICATIONS **21** CITATIONS

SEE PROFILE

Peter Fritzson
Linköping University
415 PUBLICATIONS **5,716** CITATIONS

SEE PROFILE

Parallel Simulation of Equation-Based Models on CUDA-Enabled GPUs

Per Ostlund
Department of Computer and Information Science
Linkoping University
SE-58183 Linkoping, Sweden
per.ostlund@liu.se

Kristian Stavaker
Department of Computer and Information Science
Linkoping University
SE-58183 Linkoping, Sweden
kristian.stavaker@liu.se

Peter Fritzson
Department of Computer and Information Science
Linkoping University
SE-58183 Linkoping, Sweden
peter.fritzson@liu.se

ABSTRACT

Our contributions with this work are methods and a prototype implementation for compiling and executing a limited set of equation-based mathematical models (written in the object-oriented equation-based modeling language Modelica) on CUDA-enabled GPUs. We look at methods of finding parallelism in Modelica models, that can be used on the massively parallel CUDA architecture. The methods have been implemented in a new back-end module of the OpenModelica compiler (an open-source Modelica compiler). This paper shows that it is possible to automatically generate simulation code for pure continuous-time models that can be reduced to an ordinary differential equation system without algebraic loops and where the initial values of all variables and parameters are known at compile time. It is possible to get some speedup compared with simulation on a single CPU core, a (approximated) relative speedup of 4.6 was for instance obtained for one model.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms

Algorithms

Keywords

CUDA, GPU, GPGPU, Modelica, Mathematical Simulation

1. INTRODUCTION

Graphics Processing Units (GPUs) have in recent years become increasingly programmable, giving rise to an emerging field of General-Purpose computation on Graphics Processing Units (GPGPU) [7]. The theoretical processing power of GPUs have far surpassed that of CPUs due to the highly parallel structure of GPUs. [14] Harnessing the performance of GPUs, however, is problematic. GPUs are in general good at solving certain massive parallel problems, for instance problems that exists in the field of graphic computation where computations are applied pixel by pixel. In this work, however, we are dealing with a more irregular problem, as we shall see. Compute Unified Device Architecture (CUDA) is a software platform for Nvidia GPUs that simplifies GPGPU.

Modelica is a language for equation-based object-oriented mathematical modeling that is being developed through an international effort [6]. Compilation of Modelica models is a complex process that involves many steps. The end result of the compilation process is primarily a system of equations that is linked with run-time simulation code. Generation of parallel executable code from Modelica models has been a research topic for several years at our research group, see for instance [1], [5]. For other work on parallel differential equation solver implementations, see for instance [10], [11], [4].

Our contributions with this work are methods and a prototype implementation for compiling and executing a limited set of equation-based mathematical models, written in Modelica, on CUDA-enabled GPUs. Since Modelica is a large and complex language we restrict our models in this work to pure continuous-time models (discrete time and hybrid models are also possible in Modelica) that can be reduced to an explicit Ordinary Differential Equation (ODE) system without algebraic loops and where the initial values of all variables and parameters are known at compile time. The methods have been implemented in a new back-end module of the OpenModelica compiler, an open-source Modelica compiler developed by the Open Source Modelica Consortium (OSMC) [9].

We obtained, for instance, a relative speedup of 4.6 for the WaveEquationSample model (presented in section 3) with 3840 sections (about 7680 state variables), comparing the Nvidia GeForce 8800 GTS to the Intel E6600 CPU using single precision calculations. See section 5 for more details about this measurement. A different and less general approach to parallelism that has potential for better speedup for some models, for instance the WaveEquationSample model introduced in section 3, is to compile data parallel Modelica array computations into CUDA code. A first feasibility study of this approach can be found in [12]. This paper is mainly based on [13] which is an application of the work in [1] and [5].

POOSC'10 17-OCT-2010, Reno, USA

The rest of the paper is organized as follows. Section 2 contains background information on the languages and tools involved in this work. Section 3 introduces one of the models we wish to generate code for and simulate. In section 4 we describe our implementation and in section 5 we provide measurements of generating and executing code with our new implementation. Finally in section 6 we discuss our results and in section 7 we give some general conclusions and discuss future work.

2. BACKGROUND TECHNOLOGIES

This section contains background information on the languages and systems involved in this work.

2.1 The Modelica Modeling Language

Modelica is a language for equation-based object-oriented mathematical modeling and it is being developed through an international effort [6]. Modelica allows the user to write his/her equations directly in the model code without having to manually transform the equations into computational low-level form in a standard imperative programming language. Instead this is done by the Modelica compiler in question. The user can also make use of high-level concepts such as object-oriented programming and component composition. The multi-domain capability of Modelica gives the user the possibility of combining electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model.

2.2 OpenModelica

OpenModelica is an open source implementation of a Modelica compiler, simulator and development environment for research, education and industrial purposes. OpenModelica is developed and supported by an international effort, the Open Source Modelica Consortium (OSMC) [9]. It consists of a Modelica compiler as well as other tools that form an environment for creating and simulating Modelica models.

2.3 Compute Unified Device Architecture

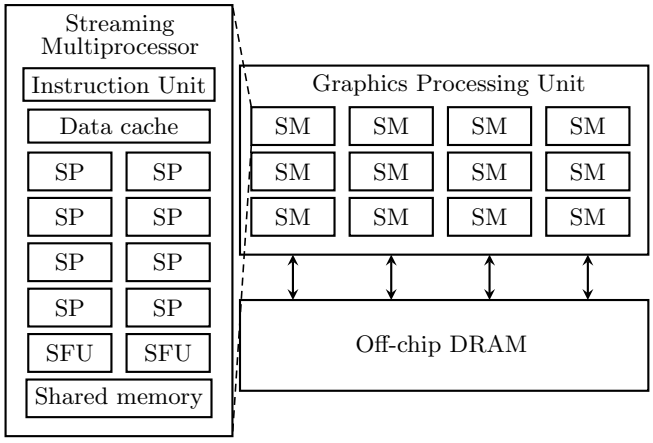

Figure 1: Simplified schematic of a GPU with 96 SPs.

For a more detailed description the reader is referred to [7]. A CUDA-enabled GPU consists of several Streaming Multiprocessors (SMs) whom each contain several Scalar Processors (SPs). See Figure 1. In each clock cycle, each SP can produce a result. All the SPs in one SM execute the same instruction. The SMs execute asynchronously without communication and no formal consistency model exists between them. One could say that we have SIMD execution (or SIMT execution, see [7]) inside one SM and MIMD execution across several SMs. Some synchronization between the SMs is possible via the global GPU memory. The programmer launches blocks of threads from the host (normally a CPU) to be executed on the GPU. The blocks are automatically scheduled onto the different SMs. Each block is divided into several warps (typically 32 threads). One warp is run at a time on each SM and if one warp is stalled (due to memory latency, etc.) another warp may execute. If any threads in a warp take divergent execution paths, then each of these paths will be executed separately, and the threads will then converge again when all paths have been executed.

There are several different types of memories on a CUDA-enabled GPU.

- Each SP has a set of registers. Access typically requires no extra clock cycles per instruction, except for some special cases.
- Each SM has a shared memory (16 Kbytes on the Tesla C1060) which is shared by all the SPs. Accessing the shared memory is typically as fast as a accessing a register and the shared memory can be used for cooperation between the different threads in a block.
- Each SM also has a constant and a texture cache. We don't use the constant and texture caches in this work.
- Finally we have an off-chip Dynamic Random Access Memory (DRAM) (4 Gbytes on the Tesla C1060). This memory is accessible from all SMs as well as externally e.g. from the CPU with DMA transfers. The DRAM is much slower for the threads to access than the shared memory, typically it takes 400 to 600 clock cycles for a memory access.

3. CASE STUDY

Here we introduce one of the models we wish to simulate in order to introduce the Modelica language and give the user an idea of the problem at hand. This model is taken from [3](page 584).

The one-dimensional wave equation is given by a partial differential equation of the following form:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{\partial^2 p}{\partial x^2} \,. \tag{1}$$

where $p = p(x,t)$ is a function of both space and time. We consider a duct of length 10 where we let $-5 \leq x \leq 5$. We discretize the problem in the spatial dimension and approximate the spatial derivatives using difference approximations using the approximation:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{(p_{i-1} + p_{i+1} - 2p_i)}{\Delta x^2} \tag{2}$$

where $p_i = p(x_1 + (i-1) \cdot \Delta x)$ on an equidistant grid and Δx is a small change in distance. We get the following Modelica model, where the initial pressure is 0.0 and where n can be altered to increase/decrease the number of sections and thus the number of state variables:

```
model WaveEquationSample
  parameter Real L = 10 "Length of duct";
  parameter Integer n = 30 "Number of sections";
  parameter Real dL = L/n "Section length";
  parameter Real c = 1;
  Real[n] p(start = fill (0,n));
  Real[n] dp(start = fill (0,n));
equation
  p[1] = exp(-(-L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp = der(p);
  for i in 2:n-1 loop
    der(dp[i])=c^2*(p[i+1]-2*p[i]+p[i-1])/dL^2;
  end for;
end WaveEquationSample;
```

This model consists of two sections, a section with parameters and variables and a section with equations. Parameters are constants that can only change their values between simulation runs. The vectors p and dp are vectors with state variables since these variables occur with the derivative operator. The for loop is a for-equation that will be expanded into scalar equations. All of the equations in the model are merged together in one system and handled as described in section 4.1.

4. IMPLEMENTATION

In this section we start with a short overview of compilation and simulation of Modelica code and then we discuss our new back-end module for the OpenModelica Compiler that generates CUDA code.

4.1 Compilation and Simulation of Modelica Models

Due to the special nature of Modelica, the compilation process of Modelica code differs quite a bit from imperative programming languages such as C, C++, and Java. For a more detailed description the reader is referred to for instance [2], [3]. The output from the OpenModelica front-end is an internal data structure with separate lists for variables, equations, functions and algorithm sections. The mapping of time-invariant parts of a model into executable code is done in a relatively straightforward manner. The handling of the equations is much more complex though and involves among other things symbolic index reduction, topological sorting (according to the causal dependencies between the equations), conversion into assignment form, etc. Simulation corresponds to solving the resulting equation system with respect to time using a numerical solver method. The compiled Modelica model (the final equation system and the compiled Modelica functions) is linked with a simulation runtime system that contains among other things an integration method. Fourth-order Runge-Kutta is the numerical integration method used in this paper. Initial values are given in a data file (these have been extracted from the original model) and a final plot file is produced (containing values for the state and algebraic variables at different time intervals).

4.2 CUDA Code Generation

In [1] it was investigated how parallel executable code from equation-based models could be generated, which resulted in a OpenModelica back-end module. This work was then continued in [5] by inlining the numerical solver and by introducing so-called software pipelining. In this work we have tried to adopt some of these techniques, especially from [1], for generating CUDA code. The reader is referred to [13] for a more in-depth description. See Figure 2 for the different compilation phases.

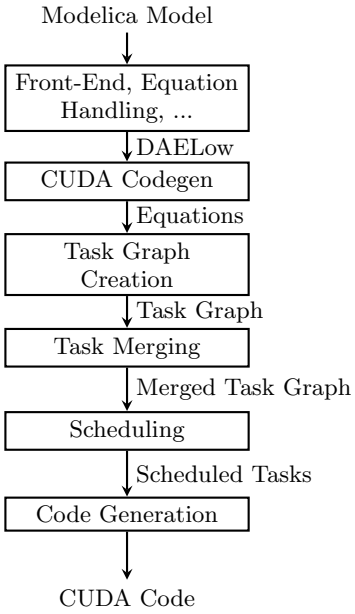

Figure 2: The process of compiling a Modelica model to CUDA code.

4.2.1 Task Graph Creation and Task Merging

The CUDA code generation module is invoked in the back-end of the compiler with a sorted equation system as input. As a first step a task graph is generated from the equation system. A task graph is a directed acyclic graph where each vertex represents a task (a scalar operation) and the edges of the graph represents precedence constraints on the tasks. There is a cost associated with each edge and task. The cost of an edge is the cost of sending data between two tasks and the cost of a task is the cost of executing the task. So far we have only used crude costs for the tasks: the cost of unary and binary operations are set to 1 and the cost of special functions are set to 4 (this should reflect the fact that a SM has eight SPs but only two special function units). The cost of communication is set to 100, which should reflect the latencies introduced by the global memory. A task graph is a convenient structure to work with for detecting parts of an ODE system that can be computed in parallel. Because the initial task graph is extremely fine grained, a merging algorithm is applied to it. Scalar tasks are merged together into larger nodes.

4.2.2 Scheduling

When the task graph has been merged it is necessary to determine in which order the tasks should be executed, and whether they should be executed in parallel on different SMs or not. The scheduling is done in two main steps. In the first step the nodes in the merged task graph is scheduled with the so-called critical path algorithm and then the tasks

inside each node are scheduled. The critical path algorithm selects the critical path in a graph, which is the path with the longest execution time. The critical path algorithm is run over and over again until the whole graph has been covered. The tasks inside one node are scheduled using a first in, first out queue with the tasks to be scheduled. An example can be seen in figure 3.

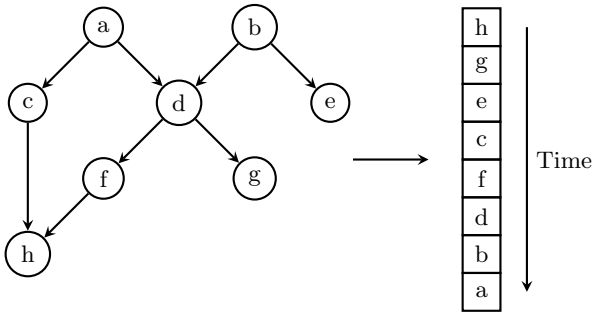

Figure 3: An example of the task scheduling algorithm.

In addition to these two steps the scheduler also tries to find nodes that are operationally equivalent to other nodes, and schedules those nodes to be executed in parallel on the same SM (SIMD style execution). As of now we use a relatively crude test for operational equality, see [13] for more details.

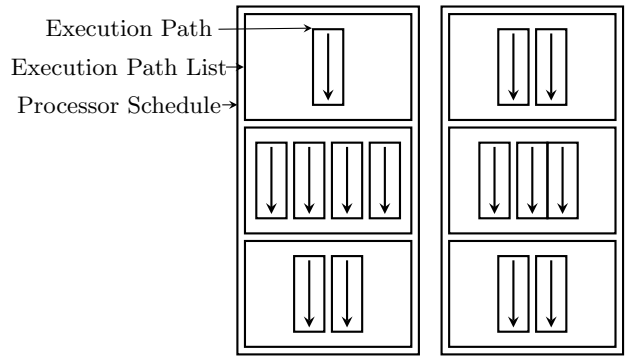

Figure 4: An example of a schedule for two streaming multiprocessors.

The result of the scheduling is a processor schedule. See figure 4. The processor schedule contains execution paths and execution path lists. An execution path is a list of task executed in order. One SM executes one execution path list, that hopefully contains several execution paths, at a time. Inside one SM we can, if there are several execution paths, execute in SIMD mode. Running several SMs in parallel and synchronizing via the global memory, corresponds to MIMD execution. The blocks of threads are scheduled automatically on the different SMs and we do not know in which order the different blocks are going to execute. However, we never execute more blocks than there are SMs (to avoid dead-locks) and we synchronize via the global memory, thus we can run several blocks in parallel. Communication between processors is necassary in many cases. Each node has to be inspected to see if any of the task has a dependency that is scheduled on another processor. If this is the case the scheduler inserts signals and locks into the schedule and determines what data should be sent where. Special execution paths for communication are inserted into the schedule.

4.2.3 Code Generation

Code generation is done by iterating through the schedule one processor at a time and one execution path list at a time. If the execution path list is a special execution path for communication, communication code is generated. If the execution path list contains several execution paths code for SIMD is generated. Otherwise if it is a normal execution path list, code for each task is generated one by one.

A technique to reduce long off-chip DRAM latencies is memory coalescing. The DRAM memory can access whole chunks of memory at a time. We use coalesced memory reads that read 16 variables at a time from the device memory, the size of a coalesced read of 32-bit variables, and then we move those variables to where they should be in the shared memory on an SM. It does not matter if not all 16 variables are used since it costs as much to read one variable as it does to read 16 variables. By coalescing memory accesses and having several warps active at the same time it is thus possible to mask the latencies associated with the off-chip memory somewhat.

4.2.4 Generated Code

In the code listning below the main simulation function is given. A fourth order Runge-kutta integration scheme is used. This integration scheme was used both for the GPU implementation and the normal sequential CPU implementation, that we used for comparison. This integration method is relatively easy to implement yet gives decent enough results.

$$\begin{aligned}
\mathbf{k_1} &= \mathbf{f}\left(\mathbf{x}\left(t\right), \mathbf{u}\left(t\right), t\right) \\
\mathbf{k_2} &= \mathbf{f}\left(\mathbf{x}\left(t + \tfrac{h}{2} \cdot \mathbf{k_1}\right), \mathbf{u}\left(t + \tfrac{h}{2}\right), t + \tfrac{h}{2}\right) \\
\mathbf{k_3} &= \mathbf{f}\left(\mathbf{x}\left(t + \tfrac{h}{2} \cdot \mathbf{k_2}\right), \mathbf{u}\left(t + \tfrac{h}{2}\right), t + \tfrac{h}{2}\right) \\
\mathbf{k_4} &= \mathbf{f}\left(\mathbf{x}\left(t + h \cdot \mathbf{k_3}\right), \mathbf{u}\left(t + h\right), t + h\right)
\end{aligned}$$

$$\mathbf{x}(t+h) \approx \mathbf{x}(t) + h \cdot \tfrac{1}{6} \cdot (\mathbf{k_1} + 2 \cdot \mathbf{k_2} + 2 \cdot \mathbf{k_3} + \mathbf{k_4})$$

The first four steps in the method evaluates $\mathbf{f}$ at different points in time, and in the last step a weighted sum of these values is calculated. The next value of $\mathbf{x}$ is then calculated using the weighted sum. The execution of tasks and incrementation of steps are done in parallel by launching kernels.

```
//Determine the size of the shared memory needed.
int shmem_size = 100 * sizeof(real);

for(int step = 0; step < steps; ++step)
{
//Move the pointers of the result arrays forward.
r_dx += DERIVATIVES;
r_x += STATES;
r_y += ALGEBRAICS;

//Execute the tasks, call integration kernel.
execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
 d_y, d_c, d_l, t);
step_and_increment1<<<2, 32>>>(d_x, d_old_x,
 d_dx, d_k, half_h);
//Increment the time by half a time step.
```

	GeForce 8800 GTS	Tesla C1060
Streaming Multiprocessors	12	30
Scalar Processors	96	240
Scalar Processor Clock (MHz)	1200	1300
Single Precision GFLOPS	346	933
Double Precision GFLOPS	N/A	78
Memory Amount (MB)	320	4096
Memory Interface	320-bit	512-bit
Memory Clock (MHz)	800	800
Memory Bandwidth (GB/s)	64	102
PCIe Version	1.0	2.0 (1.0 used)
PCIe Bandwidth (GB/s)	4	8 (4 used)
CUDA Compute Capability	1.0	1.3

Table 1: GPU Specifications

```
t += half_h;

//Do two more steps of the RK4 method.
execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
 d_y, d_c, d_l, t);
step_and_increment2<<<2, 32>>>(d_x, d_old_x,
 d_dx, d_k, half_h);
execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
 d_y, d_c, d_l, t);
step_and_increment3<<<2, 32>>>(d_x, d_old_x,
 d_dx, d_k, h);
//Increment the time again with half a time step.
t += half_h;

//Do the final integration.
execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
 d_y, d_c, d_l, t);
step_and_integrate<<<2, 32>>>(d_x, d_old_x,
 d_dx, d_k, h_div_6);

//Save the new values.
cudaMemcpy(r_x, d_x, STATES * sizeof(real),
cudaMemcpyDeviceToHost);
cudaMemcpy(r_dx, d_dx, DERIVATIVES * sizeof(real),
cudaMemcpyDeviceToHost);
cudaMemcpy(r_y, d_y, ALGEBRAICS * sizeof(real),
cudaMemcpyDeviceToHost);
}
```

5. EXPERIMENTAL RESULTS

In this section we present measurements of executing CUDA code from one model on two GPU architectures and on one CPU architecture. For more measurement data see [13]. Table 1 shows the specifications of the two GPU architectures used. Figure 5 shows the time-measurements obtained. We alter the parameter n for each simulation run. The CPU used was an Intel Core 2 Duo E6600 (2.4 GHz clock frequency) but only one core was used (since there were no code generator that generates parallel code from OpenModelica available). The model used, WaveEquationSample, was introduced in section 3 and section 6 contains a discussion of the results. An approximation was done that gives an upper limit of the performance, see [13].

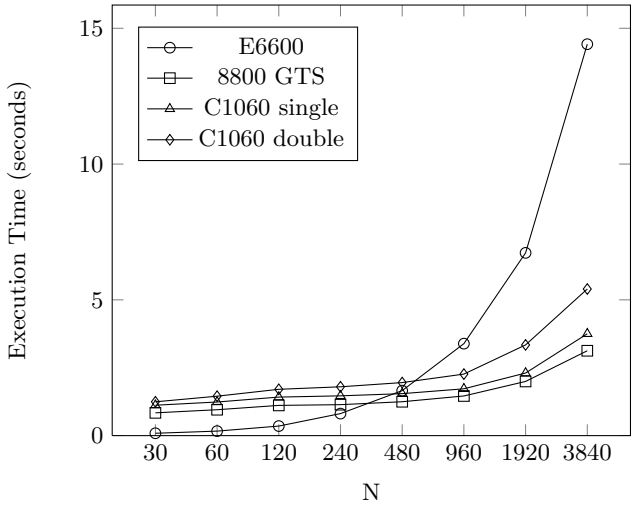

Figure 5: Execution Time for the WaveEquationSample Model as a Function of the Number of Sections.

	8800 GTS	C1060 single	C1060 double
Task Execution	0.164	0.592	0.389
Shared Mem Allocation	1.440	1.426	2.287
Integration	0.417	0.400	0.445
Memory Transfers	1.104	1.332	2.278

Table 2: Seconds spent in the different parts of the simulation of the WaveEquationSample model.

6. DISCUSSION OF RESULTS

- We obtained a relative speedup of 4.6 with n=3840 for WaveEquationsSample (comparing the GeForce 8800 GTS to the Intel E6600 CPU using single precision calculations). As noted before, this was an approximation.

- For small models (few state variables) the GPU performs badly but as the number increases the GPU retain a relatively flat curve, while the simulation times on the CPU rises much faster. The reason for this is likely that the GPUs need many thread blocks with many threads to fully utilize their power, and smaller models therefore do not use all SMs available. The simulation times on the CPUs instead behaves as could be expected by approximately doubling when the model size is doubled.

- It was shown that memory transactions take most of the time, and the actual computations take a smaller amount of time. See table 2.

- While the models used in this work are parallel enough to easily spawn many threads, the computations per variable ratio is quite low. Models with more computations per variable should see an even larger performance increase when using a GPU.

7. CONCLUSIONS

The goal of this work was to evaluate the feasibility of simulating Modelica models on CUDA-enabled GPUs. We have demonstrated that it is possible to simulate at least a subset of typical models. Nvidia will soon release the new Fermi architecture [8]. This architecture has several improvements. It has a cache hierarchy, improved support for double precision operations, more shared memory on the SMs, support for running several kernels at a time (better utilization of idle SMs), etc. A potential future work could therefore be to compile CUDA code for the Fermi architecture using the methods outlined in this paper. A different approach of using GPUs and CUDA for simulating Modelica models (or a subset of Modelica models) is to compile Modelica array-equations into CUDA code. Modelica models containing for-equations operating over large arrays are for instance common in models derived from discretizations of partial differential equations. This is recently initialized work at our research group, see [12].

8. ACKNOWLEDGMENTS

Funded by the European ITEA2 OPENPROD Project (Open Model-Driven Whole-Product Development and Simulation Environment) and CUGS Graduate School.

9. REFERENCES

[1] P. Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Linköping University, Sweden, 2006. Dissertation No. 1022.

[2] E. K. Francois Cellier. *Continuous System Simulation*. Springer, 2006.

[3] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. IEEE Press, 2004.

[4] M. Korch and T. Rauber. Scalable parallel rk solvers for odes derived by the method of lines. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 830–839. Springer, 2003.

[5] H. Lundvall. *Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, Licentiate thesis 1381*. Linköping University, 2008.

[6] Modelica and the Modelica Association, accessed January 8 2011. http://www.modelica.org.

[7] CUDA Zone, accessed January 8 2011. http://www.nvidia.com/object/cuda_home_new.html.

[8] Nvidia Fermi, accessed January 8 2011. http://www.nvidia.com/object/fermi_architecture.html.

[9] OpenModelica, accessed January 8 2011. http://www.openmodelica.org.

[10] T. Rauber and G. Rünger. Iterated runge-kutta methods on distributed memory multiprocessors. In *PDP*, pages 12–19. IEEE Computer Society, 1995.

[11] T. Rauber and G. Rünger. Parallel execution of embedded and iterated runge-kutta methods. *Concurrency - Practice and Experience*, 11(7):367–385, 1999.

[12] K. Stavåker, D. Rolls, J. Guo, P. Fritzson, S. Scholz. Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on CUDA-enabled GPUs. EOOLT 2010, Oslo, Norway, October 3, 2010.

[13] P. Östlund. *Simulation of Modelica Models on the CUDA Architecture. Master Thesis. LIU-IDA/LITH-EX-A–09/062–SE*. Linköping University, 2009.

[14] M. Wolfe, The Portland Group, Inc: Compilers and More: GPU Architecture and Applications, September 10 2008.