Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units

by

Kristian Stavåker
Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units

by

Kristian Stavåker

December 2011
Linköping Studies in Science and Technology
Thesis No. 1507
ISSN 0280--7971
LiU--Tek--Lic--2011:46

ABSTRACT

In this thesis we investigate techniques and methods for parallel simulation of equation-based, object-oriented (EOO) Modelica models on graphics processing units (GPUs). Modelica is being developed through an international effort via the Modelica Association. With Modelica it is possible to build computationally heavy models; simulating such models however might take a considerable amount of time. Therefore techniques of utilizing parallel multi-core architectures for simulation are desirable. The goal in this work is mainly automatic parallelization of equation-based models, that is, it is up to the compiler and not the end-user modeler to make sure that code is generated that can efficiently utilize parallel multi-core architectures. Not only the code generation process has to be altered but the accompanying run-time system has to be modified as well. Adding explicit parallel language constructs to Modelica is also discussed to some extent. GPUs can be used to do general purpose scientific and engineering computing. The theoretical processing power of GPUs has surpassed that of CPUs due to the highly parallel structure of GPUs. GPUs are, however, only good at solving certain problems of data-parallel nature. In this thesis we relate several contributions, by the author and co-workers, to each other. We conclude that the massively parallel GPU architectures are currently only suitable for a limited set of Modelica models. This might change with future GPU generations. CUDA for instance, the main software platform used in the thesis for general purpose computing on graphics processing units (GPGPU), is changing rapidly and more features are being added such as recursion, function pointers, C++ templates, etc.; however the underlying hardware architecture is still optimized for data-parallelism.

This work has been supported by the European ITEA2 OPENPROD project (Open Model-Driven Whole-Product Development and Simulation Environment) and by the National Graduate School of Computer Science (CUGS).

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Acknowledgements

First of all I would like to express my greatest gratitude to my main supervisor Professor Peter Fritzon for accepting me as a PhD student and for giving me guidance through the years. I would like to thank my colleagues at PELAB for a nice working environment and for their support and help, especially my secondary supervisor Professor Christoph Kessler, Professor Kristian Sandahl, Usman Dastgeer, and Per Östlund, Dr Adrian Pop and Martin Sjölund (for giving me advice on OpenModelica implementation issues). I would also like to thank Jens Frenkel (TU Dresden). I am very thankful for the kind help I have been given when it comes to administrative issues from Bodil Mattsson Kihlström, Åsa Kärrman, and the other administrators at IDA. Finally, I would like to thank my sister Kirsti Stavåker.
# Contents

1 Introduction 1
   1.1 Motivation ............................... 1
   1.2 Research Question ....................... 3
   1.3 Research Process ....................... 3
   1.4 Contributions .......................... 4
   1.5 Delimitations .......................... 4
   1.6 List of Publications ................... 5
   1.7 Thesis Outline ......................... 6

2 Background 8
   2.1 Introduction ........................... 8
   2.2 The Modelica Modeling Language ........ 8
   2.3 The OpenModelica Development Environment .................. 10
   2.4 Mathematical Concepts .................. 10
      2.4.1 ODE and DAE Representation ........ 10
      2.4.2 ODE and DAE Numerical Integration Methods ........ 11
   2.5 Causalization of Equations ............. 12
      2.5.1 Sorting Example ................... 12
      2.5.2 Sorting Example with Modelica Model ........ 15
      2.5.3 To Causal Form in Two Steps .......... 16
      2.5.4 Algebraic Loops .................... 17
   2.6 Compiler Structure ..................... 17
   2.7 Compilation and Simulation of Modelica Models .......... 19
   2.8 Graphics Processing Units (GPUs) ........ 20
      2.8.1 The Fermi Architecture .......... 21
      2.8.2 Compute Unified Device Architecture (CUDA) ...... 22
      2.8.3 Open Computing Language (OpenCL) ........ 24

3 Previous Research 25
   3.1 Introduction ............................ 25
   3.2 Overview ................................ 25
   3.3 Early Work with Compilation of Mathematical Models to Parallel Executable Code ........ 26
   3.4 Task Scheduling and Clustering Approach .......... 26
CONTENTS

3.4.1 Task Graphs ........................................... 27
3.4.2 Modpar ............................................... 28
3.5 Inlined and Distributed Solver Approach ................. 28
3.6 Distributed Simulation using Transmission Line Modeling . . 29
3.7 Related Work in other Research Groups .................. 31

4 Simulation of Equation-Based Models on the Cell BE Processor Architecture 32
4.1 Introduction ............................................. 32
4.2 The Cell BE Processor Architecture ......................... 33
4.3 Implementation ......................................... 33
4.4 Measurements ........................................... 34
4.5 Discussion ............................................... 35

5 Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units 37
5.1 Introduction ............................................. 37
5.2 Restricted Set of Modelica Models ........................ 38
5.3 Quantized State Systems (QSS) .......................... 38
5.4 Implementation ......................................... 40
5.5 Measurements ........................................... 42
5.6 Discussion ............................................... 43

6 Simulation of Equation-Based Models with Task Graph Creation on Graphics Processing Units 46
6.1 Introduction ............................................. 46
6.2 Case Study .............................................. 47
6.3 Implementation ......................................... 47
6.4 Run-time Code and Generated Code ......................... 48
6.5 Measurements ........................................... 51
6.6 Discussion ............................................... 53

7 Compilation of Modelica Array Computations into Single Assignment C (SAC) for Execution on Graphics Processing Units 54
7.1 Introduction ............................................. 54
7.2 Single Assignment C (SAC) ................................ 55
7.3 Implementation ......................................... 56
7.4 Measurements ........................................... 57
7.5 Discussion ............................................... 59

8 Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-core Simulation 60
8.1 Introduction ............................................. 60
8.2 Implementation ......................................... 60
  8.2.1 Parallel Variables ................................. 61
CONTENTS

8.2.2 Parallel Functions .................................. 61
8.2.3 Kernel Functions .................................. 61
8.2.4 Parallel For-Loops .................................. 62
8.2.5 OpenCL Functionalities .............................. 63
8.2.6 Synchronization and Thread Management ............ 63
8.3 Measurements .......................................... 63
8.4 Discussion ............................................. 64

9 Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units 67
9.1 Introduction ........................................... 67
9.2 Problems with Expanding Array Equations ............... 68
9.3 Splitting For-Equations with Multiple Equations in their Bodies .................................. 69
  9.3.1 Algorithm ......................................... 69
  9.3.2 Examples ......................................... 69
9.4 Transforming For-Equations and Array Equations into Slice Equations .......................... 70
  9.4.1 Algorithm ......................................... 70
  9.4.2 Examples ......................................... 71
9.5 Matching and Sorting of Unexpanded Array Equations ...... 72
  9.5.1 Algorithm ......................................... 72
  9.5.2 Examples ......................................... 74
9.6 Implementation ........................................ 77
  9.6.1 Instantiation - Symbolic Elaboration .................. 81
  9.6.2 Lowering ......................................... 81
  9.6.3 Equation Sorting and Matching ....................... 82
  9.6.4 Finding Strongly Connected Components ............... 82
  9.6.5 CUDA Code Generation .............................. 82
9.7 Discussion ............................................. 82

10 Discussion ............................................... 84
10.1 What Kind of Problems are Graphics Processing Units Suitable for? .......................... 84
10.2 Are Graphics Processing Units Suitable for Simulating Equation-Based Modelica Models? ............... 85
  10.2.1 Discussion on The Various Approaches of Simulating Modelica Models on GPUs .................. 86
10.3 Summary and Future Work .............................. 87

A Quantized State Systems Generated CUDA Code 89
Chapter 1

Introduction

In this chapter we start by giving a motivation to the research problem we are investigating in this thesis work. We then continue by stating the research question followed by the research process taken, the contributions of this work together with the delimitations. Finally we provide a list of publications on which this thesis is based and an outline of the rest of the thesis.

1.1 Motivation

By using the equation-based object-oriented modeling language Modelica [25], [15] it is possible to model large and complex physical systems from various application domains. Large and complex models will typically result in large differential equation systems. Numerical solution of large systems of differential equations, which in this context equals simulation, can be quite time consuming. Therefore it is relevant to investigate how parallel multi-core architectures can be used to speedup simulation. This has also been a research goal in our research group Programming Environment Laboratory (PELAB) at Linköping University for several years, see for instance [4], [21], [3], [40]. This work involves both the actual code generation process and modifying the simulation run-time system. Several different parallel architectures have been targeted, such as Intel multi-cores, STI\(^1\) Cell BE, and Graphics Processing Units (GPUs). In this thesis the main focus is on GPUs. GPUs can be used to do general purpose scientific and engineering computing in addition to their use for graphics processing. The theoretical processing power of GPUs has surpassed that of CPUs due to the highly parallel structure of GPUs. GPUs are, however, only good at solving certain problems which are primarily data-parallel. Mainly four methods for harnessing the power of GPUs for Modelica model simulations are discussed in this thesis:

\(^{1}\)An alliance between Sony, Toshiba, and IBM
Chapter 1. Introduction

- compiling Modelica array equations for execution with GPUs;
- creating a task graph from the model equation system and scheduling the tasks in this task graph for execution;
- simulation of Modelica models using the QSS integration method on GPUs;
- extending the algorithmic subset of Modelica with parallel constructs.

These four approaches will be described in the various chapters of this thesis.

Handling Modelica array equations unexpanded through the compilation process is discussed in this thesis to some extent. This is necessary in order to compile data-parallel, array-based models efficiently. This is in contrast to the usual way of compiling Modelica array equations used by almost all Modelica tools, which is to expand the array equations into scalar equations early in the compilation process and then these equations are handled just like normal scalar equations. Keeping the array equations unexpanded involves modifications of the whole compilation process, from the initial flattening, the middle parts with equation sorting and the code generation. Keeping the array equations unexpanded opens up possibilities for generating efficient code that can be executed with GPUs. This work is mainly being carried out by the author and is work in progress, see Chapter 9.

The second approach of creating a task graph of the equation system and then scheduling the tasks in this task graph has been used earlier for other architectures [4], [21]. We have now investigated this approach for GPUs which is described in the master thesis of Per Östlund [41] and in Paper 4 [33]. That work is summarized in this thesis, some conclusions are drawn and we relate this approach to the other approaches.

In Chapter 5 and Paper 2 [34] a joint work with Politecnico di Milano is discussed regarding simulation of Modelica models using the QSS integration method on GPUs.

The fourth approach of extending the algorithmic subset of Modelica with parallel programming constructs is also discussed and summarized in this thesis. This has been investigated in the master thesis of Mahder Gebremedhin [16] and in Paper 6 [23] and supervised by the author. That work is summarized in this thesis, some conclusions are drawn and we relate this approach to the other approaches.

Moreover several other pieces of work are presented in this thesis. Simulating Modelica models with the Cell BE architecture is discussed in Chapter 4 and in Paper 1 [6]. In Chapter 7 and Paper 3 [37] work with compiling...
Modelica array operations into an intermediate language, Single Assignment C (SAC) is described. That work is related to the work with compiling unexpanded Modelica array constructs and is a joint work with University of Hertfordshire. In Chapter 3 some previous work on simulating Modelica models on parallel multi-core architectures is presented.

An important part of this thesis is the last chapter where we relate the various approaches and draw some conclusions regarding whether or not GPUs are suitable for simulating Modelica models. That last chapter is related to Section 2.8 with GPUs, where we give a description of this architecture.

All the implementation work in this thesis have been done in the open-source OpenModelica development environment [30]. OpenModelica is an open-source implementation of a Modelica compiler, simulator and development environment and its development is supported by the Open Source Modelica Consortium (OSMC). See Section 2.3 for more information.

1.2 Research Question

The main research question of this work is given below.

‘Is it possible to simulate Modelica models with GPU architectures; will such simulations run with sufficient speed compared to simulation on other architectures, for instance single- and multi-core CPUs, are GPUs beneficial for performance; and what challenges are there in terms of hardware limitations, memory limitations, etc.’

1.3 Research Process

The following research steps have roughly been taken for carrying out this work.

- Literature study of background theory.
- Literature study of earlier work.
- Theoretical derivations and design sketches on paper.
- Implementations in the open-source OpenModelica compiler and runtime system and then measurements of execution times for various models.
- Presentation of papers at workshops and conferences for comments and valuable feedback.
The research methodology used in this work is the traditional system oriented computer science research method, that is, in order to validate our hypotheses prototype implementations are built. The prototypes are used to simulate Modelica models both with serial and parallel architecture runs and then compare the simulation times. In this way we can calculate the speedup. As noted in [5], the ACM Task Force on the core of computer science has suggested three different paradigms for conducting research within the discipline of computing: theory, abstraction (modeling), and design. The first discipline is rooted in mathematics, the second discipline is rooted in experimental scientific methods, and the third discipline is rooted in engineering and consist of stating requirements, defining the specification, designing the system, implementing the system and finally testing the system. All three paradigms are considered to be equally important and computer science and engineering consist of a mixture of all three paradigms.

1.4 Contributions

The following are the main contributions of this work:

- A survey and summary of various techniques by the author and co-workers for simulating Modelica models with GPUs and an analysis of the suitability of this architecture for simulating Modelica models.

- Enhancement to the open-source OpenModelica compiler with methods and implementations for generating GPU-based simulation code as well as extensions to accompanying run-time system.

- Some preliminary theoretical results and a partial implementation in the OpenModelica compiler of unexpanded array equation handling in the equation sorting phase of the compilation process.

1.5 Delimitations

The main delimitations concern the models we generate code for. We have (mainly) only looked at a subset of possible Modelica models:

- Models that are purely continuous with respect to time.

- Models that can be reduced to ordinary differential equation systems (ODE systems). This will be described in more details later.

- Models where the values of all constants and parameters are known at compile time.
1.6 List of Publications

This thesis is mainly based on the following publications.


- **Paper 2** Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, Peter Fritzson. Parallel Simulation of Equation-based Object-Oriented Models with Quantized State Systems on a GPU. In Proceedings of the 7th International Modelica Conference (Modelica’2009), Como, Italy, September 20-22, 2009. [34]

- **Paper 3** Kristian Stavåker, Daniel Rolls, Jing Guo, Peter Fritzson, Sven-Bodo Scholz. Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on CUDA-enabled GPUs. 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, Oslo, Norway, October 3, 2010. [37]

- **Paper 4** Per Östlund, Kristian Stavåker, Peter Fritzson. Parallel Simulation of Equation-Based Models on CUDA-Enabled GPUs. POOSC Workshop, Reno, Nevada, October 18, 2010. [33]


Other publications (pre-PhD) by the author not covered in this thesis.


1.7 Thesis Outline

Since this work contains contributions by several people it is important to state which parts have been done by the author of this thesis and which parts have been done by others.

- **Chapter 2** This chapter contains a summary of existing background knowledge not by the author. We introduce the Modelica modeling language, the OpenModelica development environment, some mathematical concepts as well as the GPU architecture and the CUDA computing architecture.

- **Chapter 3** This chapter contains a summary of earlier work mainly from the same research group (PELAB).

- **Chapter 4** This chapter is mainly based on paper 1 which contains (updated) material from Håkan Lundvall’s licentiate thesis [21] as well as new material with targeting the Cell BE architecture for simulation of equation-based models. The author did the actual mapping to the Cell BE processor. The author also co-authored the actual paper.

- **Chapter 5** This chapter is mainly based on paper 2. In this paper ways of using the QSS simulation method with NVIDIA GPUs were investigated. The author implemented the OpenModelica backend QSS code generator. The author also co-authored the actual paper.

- **Chapter 6** This chapter is a summary of paper 4. This chapter describes work of creating a task graph of the model equation system and then scheduling this task graph for execution. The implementation work was done by Per Östlund and is described in his master thesis [41]. The author co-authored the actual paper, held the paper presentation and supervised the master thesis work.

- **Chapter 7** This chapter is mainly based on paper 3. The chapter discusses compiling Modelica array constructs into an intermediate language, Single Assignment C (SAC), from which highly efficient code can be generated for instance for execution with CUDA-enabled GPUs. The author has been highly involved with this work.

- **Chapter 8** This chapter is mainly based on paper 6. This chapter addresses compilation and benchmarking of the algorithmic subset of Modelica, primarily to OpenCL executed on GPUs and Intel multi-cores. The implementation and measurements were done by two master students (Mahder Gebremedhin and Afshin Hemmati Moghadam), supervised by the author.

- **Chapter 9** (Work In Progress) This chapter describes preliminary results of ways to keep the Modelica array equations unexpanded
through the compilation process. The author is highly involved with the actual implementation that is being carried out in the OpenModelica compiler, which at the moment is only a partial implementation.

- **Chapter 10** Chapter has been completely written by the author of this thesis.
Chapter 2

Background

2.1 Introduction

In this chapter we begin by introducing the Modelica modeling language and the open-source OpenModelica compiler. We then continue by introducing some mathematical concepts and describing the general compilation process of Modelica code. The final section contains a description of GPUs and the CUDA and OpenCL software architectures, with focus on CUDA.

2.2 The Modelica Modeling Language

Modelica is a modeling language for equation-based, object-oriented mathematical modeling which is being developed through an international effort \cite{25}, \cite{15}. Since Modelica is an equation-based language it support modeling in acausal form using equations. This is in contrast to a conventional programming language where the user would first have to manually transform the model equations into causal (assignment) statement form but with Modelica it is possible to write equations directly in the model code and the compiler in question will take care of the rest. When writing Modelica models it is also possible to utilize high-level concepts such as object-oriented modeling and component composition. An example of a Modelica model is given below in Listing 2.1.

The model in Listing 2.1 describes a simple circuit consisting of various components as well as source and ground. Several components are instantiated from various classes (Resistor class, Capacitor class, etc.) and these are then connected together with the connect construct. The connect is an equation construct since it expands into one or more equations. Subsequently a Modelica compiler can be used to compile this model into code that can be linked with a runtime system (where the main part consists of a solver, see Section 2.4.2) for simulation. All the object-oriented structure is removed.
at the beginning of the compilation process and the connect equations are expanded into scalar equations.

```
model Circuit
    Resistor R1(R=10);
    Capacitor C(C=0.01);
    Resistor R2(R=100);
    Inductor L(L=0.1);
    VsourceAC AC;
    Ground G;

equation
    connect(AC.p, R1.p);
    connect(R1.n, C.p);
    connect(C.n, AC.n);
    connect(R1.p, R2.p);
    connect(R2.n, L.p);
    connect(L.n, C.n);
    connect(AC.n, G.p);
end Circuit;
```

Listing 2.1: A Modelica model for a simple electrical circuit.

Modelica and Equation-Based Object-Oriented (Eoo) languages in general support the following concepts:

- **Equations**
- **Models/Classes**
- **Objects**
- **Inheritance**
- **Polymorphism**
- **Acausal Connections**

Continuous-time differential or algebraic equations make it possible to model continuous-time systems. There are also discrete equations available for modeling hybrid systems, i.e., systems with both continuous and discrete parts. Modelica has a uniform design meaning that everything, e.g., models, packages, real numbers, etc. in Modelica are classes. A Modelica class can be of different kinds denoted by different class keywords such as model, class, record, connector, package, etc. From the Modelica class, objects can be instantiated. Just like in C++ and Java classes can inherit behavior and data from each other. To conclude, Modelica supports imperative, declarative and object-oriented programming thus resulting in a complex compilation process that places a high burden on the compiler constructor.
Chapter 2. Background

2.3 The OpenModelica Development Environment

OpenModelica is an open source implementation of a Modelica compiler, simulator and development environment for research, education and industrial purposes and it is developed and supported by an international effort, the Open Source Modelica Consortium (OSMC) [30]. OpenModelica consists of several parts namely a Modelica compiler and other tools that form an environment for creating and simulating Modelica models. The OpenModelica Compiler is easily extensible; a different code generator can for instance be plugged-in at a suitable place. The OpenModelica User Guide [31] states:

- The short-term goal is to develop an efficient interactive computational environment for the Modelica language, as well as a rather complete implementation of the language. It turns out that with support of appropriate tools and libraries, Modelica is very well suited as a computational language for development and execution of both low level and high level numerical algorithms, e.g. for control system design, solving nonlinear equation systems, or to develop optimization algorithms that are applied to complex applications.

- The longer-term goal is to have a complete reference implementation of the Modelica language, including simulation of equation based models and additional facilities in the programming environment, as well as convenient facilities for research and experimentation in language design or other research activities. However, our goal is not to reach the level of performance and quality provided by current commercial Modelica environments that can handle large models requiring advanced analysis and optimization by the Modelica compiler.

2.4 Mathematical Concepts

Here we will give an overview of some of the mathematical theory that will be used later on in the thesis. For more details see for instance [8].

2.4.1 ODE and DAE Representation

Central concepts in the field of equation-based languages are ordinary differential equation (ODE) systems and differential algebraic equation (DAE) systems. A DAE representation can be described as follows.

\[ 0 = f(t, \dot{x}(t), x(t), y(t), u(t), p) \]

- \( t \) time
- \( \dot{x}(t) \) vector of differentiated state variables
2.4. Mathematical Concepts

- $x(t)$ vector of state variables
- $y(t)$ vector of algebraic variables
- $u(t)$ vector of input variables
- $p$ vector of parameters and/or constants

The difference between an ODE and DAE system is that with an ODE system the vector of state derivatives $\dot{x}$ is explicitly stated. In the compilation process, as a middle step we will typically arrive to a DAE system from the transformed Modelica model after all the object-oriented structure has been removed and expansions have been made, see section 2.7.

2.4.2 ODE and DAE Numerical Integration Methods

In this section we describe some of the numerical integration methods that are available for numerically solving an ODE or a DAE systems.

Euler Integration Method

The simplest method for numerically solving an ODE system is the Euler method which is derived below, where $x$ is the state vector, $u$ is the input vector, $p$ is a vector of parameters and constants, and $t$ represents time.

$\dot{x}(t_n) \approx \frac{x(t_{n+1}) - x(t_n)}{t_{n+1} - t_n} \approx f(t_n, x(t_n), u(t_n), p)$

The derivative is approximated as the difference of the state values between two time points divided with the difference in time (this can easily be derived by studying a graph). The above equation gives the following iteration scheme.

$x(t_{n+1}) \approx x(t_n) + (t_{n+1} - t_n) \cdot f(t_n, x(t_n), u(t_n), p)$

Runge-Kutta Integration Method

The explicit Runge-Kutta numerical integration method is a multi-stage scheme. The generic s-stage explicit Runge-Kutta method is given below, where $\Delta t$ represents a time step.

$k_1 = f(t, x(t_n))$

$k_2 = f(t + c_2 \cdot \Delta t, x(t_n) + \Delta t a_{21} k_1)$

$k_3 = f(t + c_3 \cdot \Delta t, x(t_n) + \Delta t (a_{31} k_1 + a_{32} k_2))$

...  

$k_s = f(t + c_s \cdot \Delta t, x(t_n) + \Delta t (a_{s1} k_1 + ... + a_{s,s-1} k_{s-1}))$

$x(t_{n+1}) = b_1 k_1 + ... + b_s k_s$
Chapter 2. Background

The values of the constants are given by the Runge-Kutta table below (given a value of $s$).

<table>
<thead>
<tr>
<th>$c_2$</th>
<th>$a_{21}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_3$</td>
<td>$a_{31}$ $a_{32}$</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$c_s$</td>
<td>$a_{s1}$ $a_{s2}$ ... $a_{s,s-1}$</td>
</tr>
<tr>
<td></td>
<td>$b_1$ $b_2$ ... $b_{s-1}$ $b_s$</td>
</tr>
</tbody>
</table>

We also have the following necessary condition.

$$c_j = \sum_{i=1}^{j-1} a_{ij}$$

DASSL Solver

DASSL stands for Differential Algebraic System Solver and it implements the backward differentiation formulas of orders one through five. The nonlinear system (algebraic loop) at each time-step is solved by Newton’s method. This is the main solver used in the OpenModelica compiler. Input to DASSL are systems in DAE form $F(t,y,y')=0$, where $F$ is a function and $y$ and $y'$ are vectors, and initial values for $y$ and $y'$ are given. [10]

2.5 Causalization of Equations

As mentioned earlier in this chapter, systems that consist of a mixture of implicitly formulated algebraic and differential equations are called DAE systems. Converting an implicit DAE system to equivalent explicit sorted ODE system if possible (we know in which order and by which equation a variable should be computed) is an important task for a compiler of an equation-based language.

Two simple rules can determine which variable to solve from which equation:

- If an equation only has a single unknown variable then that equation should be used to solve for that variable. It could be a variable for which no solving equation has been found yet.
- If an unknown variable only appears in one equation, then use that equation to solve for it.

2.5.1 Sorting Example

We use $f_1,\ldots,f_5$ to denote expressions containing variables. Initially all equations are assumed to be on acausal form. This means that the equal sign should be viewed as an equality sign rather than an assignment sign. The structure of an equation system can be captured in a so-called incidence
2.5. Causalization of Equations

matrix. Such a matrix lists the equations as rows and the unknowns in these equations as columns. In other words if equation number \(i\) contains variable number \(j\) then entry \((i,j)\) in the matrix contains an 1 otherwise 0. The best one can hope for is to be able to transform the incidence matrix into Block-Lower-Triangular (BLT) form, that is a triangular form but with "squares" on the diagonal representing sets of equations that needs to be solved together (algebraic loops).

\[
\begin{align*}
  f_1(z_3, z_4) &= 0 \\
  f_2(z_2) &= 0 \\
  f_3(z_2, z_3, z_5) &= 0 \\
  f_4(z_1, z_2) &= 0 \\
  f_5(z_1, z_3, z_5) &= 0
\end{align*}
\]

The above equations will result in the sorted equations with the solved for variables underlined:

\[
\begin{align*}
  f_2(z_2) &= 0 \\
  f_4(z_1, z_2) &= 0 \\
  f_3(z_2, z_3, z_5) &= 0 \\
  f_5(z_1, z_3, z_5) &= 0 \\
  f_1(z_3, z_4) &= 0
\end{align*}
\]

Note that we have an algebraic loop since \(z_3\) and \(z_5\) have to be solved together. The corresponding matrix transformation is given below. The matching of the variables with an equation to compute that variable is shown in Figure 2.1 and Figure 2.2.

![Figure 2.1: Equation system dependencies before matching.](image-url)
Chapter 2. Background

Figure 2.2: Equation system dependencies after matching.

\[
\begin{pmatrix}
 z_1 & z_2 & z_3 & z_4 & z_5 \\
 f_1 & 0 & 0 & 1 & 1 & 0 \\
 f_2 & 0 & 1 & 0 & 0 & 0 \\
 f_3 & 0 & 1 & 1 & 0 & 1 \\
 f_4 & 1 & 1 & 0 & 0 & 0 \\
 f_5 & 1 & 0 & 1 & 0 & 1 \\
\end{pmatrix}
\]

This algorithm is usually divided into two steps: 1. solve matching problem and 2. find strong components (and sort equations).
2.5. Causalization of Equations

2.5.2 Sorting Example with Modelica Model

An example Modelica model is given in Listing 2.2.

```model NonExpandedArray1
Real x;
Real y;
Real z;
Real q;
Real r;
equation
2.3232\ast y + 2.3232\ast z + 2.3232\ast q + 2.3232\ast r = \text{der}(x);
der(y) = 2.3232\ast x + 2.3232\ast z + 2.3232\ast q + 2.3232\ast r;
der(z) = 2.3232\ast x + 2.3232\ast y + 2.3232\ast q + 2.3232\ast r;
der(r) = 2.3232\ast x + 2.3232\ast y + 2.3232\ast z + 2.3232\ast q;
der(q) = 2.3232\ast x + 2.3232\ast y + 2.3232\ast z + 2.3232\ast r = \text{der}(q);
end NonExpandedArray1;
```

Listing 2.2: Modelica model used for sorting example.

The above model will result in the following matrix.

\[
\begin{pmatrix}
x & y & z & q & r \\
eq1 & 1 & 0 & 0 & 0 \\
eq2 & 0 & 1 & 0 & 0 \\
eq3 & 0 & 0 & 1 & 0 \\
eq4 & 0 & 0 & 0 & 1 \\
eq5 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

In sorted form:

\[
\begin{pmatrix}
x & y & z & q & r \\
eq1 & 1 & 0 & 0 & 0 \\
eq2 & 0 & 1 & 0 & 0 \\
eq3 & 0 & 0 & 1 & 0 \\
eq5 & 0 & 0 & 0 & 1 \\
eq4 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]
2.5.3 To Causal Form in Two Steps

Here the algorithms commonly used are described in more details.

Step 1: Matching Algorithm

Assign each variable to exactly one equation (matching problem), find a variable that is solved in each equation. Then perform the matching algorithm, which is the first part of sorting the equations into BLT form. See Listing 2.3.

```plaintext
assign(j) := 0, j=1,2,...,n
for <all equations i=1,2,...,n>  
vMark(j) := false, j=1,2,...,n;
eMark(j) := false, j=1,2,...,n;
if not pathFound(i), "singular";
end for;

function success = pathFound(i)
  eMark(i) := true;
  if <assign(j)=0 for one variable j in equation i> then
    success := true;
    assign(j) := i;
  else
    success := false;
    for <all variable j of equation i with vMark(j) = false>
      vMark(j) := true;
      success := pathFound(assign(j));
      if success then
        assign(j) := i;
        return
      end if
    end for
  end if
end
```

Listing 2.3: Matching algorithm pseudo code.

Step 2: Tarjan’s Algorithm

Find equations which have to be solved simultaneously. This is the second part of the BLT sorting. It takes the variable assignments and the incidence matrix as input and identifies strong components, i.e. subsystems of equations. See Listing 2.4.
2.6. Compiler Structure

```
i = 0; % global variable
number = zeros(n,1); % global variable
lowlink = zeros(n,1); % root of strong component
<empty stack> % stack is global

for w = 1:n
    if number(w) == 0 % call the recursive procedure
        strongConnect(w); % for each non-visited vertex
    end if
end for

procedure strongConnect(v)
i = i+1:
    number(v) = i:
    lowlink(v) = i:
    <put v on stack>
    for <all w directly reachable from v>
        if number(w) == 0 @(v,w) is a tree arc
            strongConnect(w):
            lowlink(v) = min(lowlink(v),lowlink(w));
        else if number(w) < number(v) % is a root of a strong component
            if <w is on stack>
                lowlink(v) = min(lowlink(v),number(w));
            end if
        end if
    end for
if lowlink(v) == number(v) % v root of a strong component
    while <w on top of stack satisfies number(w) >= number(v)> % v root of a strong component
        <delete w from stack and put w in current component>
    end while
end if
end
```

Listing 2.4: Tarjan’s algorithm pseudo code.

2.5.4 Algebraic Loops

An algebraic loop is a set of equations that cannot be causalized to explicit form, they need to be solved together using some numerical algorithm. In each iteration of the solver loop this set of equations has to be solved together, i.e. a solver call is made in each iteration. Newton iteration could for instance be used if the equations are nonlinear.

2.6 Compiler Structure

In this section we will outline the basic principles behind compilers and compiler construction. Basically a compiler is a program that reads a program written in a source language and translates it to a program written in a target language. Before a program can be run it must be transformed by
a compiler into a form which can be executed by a computer\(^1\). A compiler should also report any errors in the source program that it detects during the translation process. See Figure 2.3 for the various compilation steps.

- **Front-end.** The front-end typically includes lexical analysis and parsing. That is, from the initial program code an internal abstract syntax tree is created by collecting groups of characters into tokens (lexical analysis) and building the internal tree (syntax analysis).

- **Middle-part.** The middle-part typically includes semantic analysis (checking for type conflicts etc.), intermediate code generation and optimization.

- **Code Generation.** Code generation is the process of generating code from the internal representation. Parallel executable code generation is the main focus of this thesis.

\[\text{Figure 2.3: Main compilation phases in a typical compiler.}\]

\(^1\)There are also interpreters that execute a program directly at run-time.
2.7 Compilation and Simulation of Modelica Models

The main translation stages of the OpenModelica compiler can be seen in Figure 2.4. The compilation process of Modelica code differs quite a bit for instance from typical imperative programming languages such as C, C++ and Java. This is because Modelica is a complex language that mixes several programming styles and especially due to the fact that it is a declarative equation-based language. Here we give a brief overview of the compilation process for generating sequential code, as well as the simulation process. For a more detailed description the interested reader is referred to for instance [8]. The Modelica model is first parsed by the parser, making use of a lexer as well; this is a fairly standard procedure. The Modelica model is then elaborated/instantiated by a front-end module which involves among other things removing of all object-oriented structure, type checking of language constructs, etc. The output from the front-end is a lower level intermediate
form, a data structure with lists of equations, variables, functions, algorithm sections, etc. This internal data structure will then be used, after several optimization and transformation steps, as a basis for generating the simulation code.

There is a major difference between the handling of time-dependent equations and the handling of time-independent algorithms and functions. Modelica assignments and functions are mapped in a relatively straightforward way into assignments and functions respectively in the target language. Regarding the equation handling, several steps are taken. This involves among other things symbolic index reduction (that includes symbolic differentiation) and a topological sorting according to the data flow dependencies between the equations and conversion into assignment form. In some cases the result of the equation processing is an ODE system in assignment form. But in general, we will get an DAE system as the result. The actual runtime simulation corresponds mainly of solving this ODE or DAE system using a numerical integration method, such as the once described earlier (Euler, Runge-Kutta or DASSL). Several C-Code files are produced as output from the OpenModelica compiler. These files will be compiled and linked together with a runtime system, which will result in a simulation executable. One of the output files is a source file containing the bulk of the model-specific code, for instance a function for calculating the right-hand side f in the sorted equation system. Another source file contains the compiled Modelica functions. There is also a makefile generated and a file with initial values of the state variables and of constants/parameters along with other settings that can be changed at runtime, such as time step, simulation time, etc..

2.8 Graphics Processing Units (GPUs)

This section is based on [14], [28], [29] and [9]. The main goal of GPUs was initially to accelerate the rendering of graphic images in memory frame buffers intended for output to a display, graphic rendering in other words. A GPU is a specialized circuit designed to be used in personal computers, game consoles, workstations, mobile phones, and embedded systems. The highly parallel structure of modern GPUs make them more effective than general-purpose CPUs for data-parallel algorithms. The same program is executed for each data element. In a personal computer there are several places where a GPU can be present. It can for instance be located on a video card, or on the motherboard, or in certain CPUs, on the CPU die. Several series of GPU cards have been developed by NVIDIA, the three most notable are mentioned below.

- **The GeForce GPU computing series.** The GeForce 256 was launched in August 1999. In November 2006 the G80 GeForce 8800 was released, which supported several novel innovations: support of C, the single-
2.8. Graphics Processing Units (GPUs)

instruction multiple-thread (SIMT) execution model (more on this later in Section 2.8.2), shared memory and barrier synchronization for inter-thread communication, a single unified processor that executed vertex, pixel, geometry, and computing programs, etc..

- The Quadro GPU computing series. The goal with this series of card was to accelerate digital content creation (DCC) and computed-aided design (CAD).
- The Tesla GPU computing series. The Tesla GPU was the first dedicated general purpose GPU.

The appearance of programming frameworks such as CUDA (Compute Unified Device Architecture) from NVIDIA minimizes the programming effort required to develop high performance applications on these platforms. A whole new field of general-purpose computation on graphics processing units (GPGPU) has emerged. The NVIDIA Fermi hardware architecture will be described together with CUDA in Section 2.8.2, since they are closely related. Another software platform for GPUs (as well as for other hardware architectures) is OpenCL that will be described in Section 2.8.3.

2.8.1 The Fermi Architecture

A scalable array of multi-threaded Streaming Multiprocessors (SMs) is the most notable feature of the architecture. Each of these streaming multiprocessors then contains Scalar Processors (SPs), resulting in a large number of computing cores that can compute a floating point or integer instruction per clock for a thread. Some synchronization between the streaming multiprocessors is possible via the global GPU memory but no formal consistency model exists between them. Thread blocks (will be described later in Section 2.8.2) are distributed by the GigaThread global scheduler to the different streaming multiprocessors. The GPU is connected to the CPU via a host interface. Each scalar processor contains a fully pipelined integer arithmetic unit (ALU) and floating point unit (FPU). Each streaming multiprocessor has 16 load/store units and four special function units (SFU) that executes transcendental instructions (such as sin, cosine, reciprocal, and square root). The Fermi architecture supports double precision.

Memory Hierarchy

There are several levels of memory available as described below.

- Each scalar processor has a set of registers and accessing these typically requires no extra clock cycles per instruction (except for some special cases).
- Each streaming multiprocessor has an on-chip memory. This on-chip memory is shared and accessible by all the scalar processors on the
streaming multiprocessor in question, which greatly reduces off chip traffic by enabling threads within one thread block to cooperate. The on-chip memory of 64KB can be configured either as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache.

- All the streaming multiprocessors can access a L2 cache.
- The Fermi GPU has 6 GDDR5 DRAM memory of 1 GB each.

### 2.8.2 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture or CUDA is a parallel programming model and software and platform architecture from NVIDIA [28]. It was developed in order to overcome the challenge with developing application software that transparently scales the parallelism of NVIDIA GPUs but at the same time maintains a low learning curve for programmers familiar with standard programming languages such as C. CUDA comes as a minimal set of extensions to C. CUDA provides several abstractions for data-parallelism and thread parallelism: a hierarchy of thread groups, shared memories, and barrier synchronization. With these abstractions it is possible to partition the problem into coarse-grained subproblems that can be solved independently in parallel. These subproblems can then be further divided into smaller pieces that can be solved cooperatively in parallel as well. The idea is that the runtime system only needs to keep track of the physical processor count. CUDA, as well as the underlying hardware architecture has become more and more powerful and increasingly powerful language support has been added. Some of the CUDA release highlights from [28] are summarized below.

- **CUDA Toolkit 2.1** C++ templates supported in CUDA kernels as well as debugger support using gdb.
- **CUDA Toolkit 3.0** C++ class and template inheritance and support for the Fermi architecture.
- **CUDA Toolkit 3.1** Support for function pointers and recursion for the Fermi architecture as well as support to run up to 16 different kernels at the same time (with the Fermi architecture).
- **CUDA Toolkit 4.0** Support for sharing GPUs across multiple threads and a single united virtual address space (from a single host thread it is possible to use all GPUs in the system concurrently). No-copy pinning of system memory, no need to explicitly copy data with cudaMallocHost().
2.8. Graphics Processing Units (GPUs)

CUDA Programming Model

The parallel capabilities of GPUs are well exposed by the CUDA programming model. Host and device are two central concepts in this programming model. The host is typically a CPU and the device is one or more NVIDIA GPUs. The device operates as a coprocessor to the host and CUDA assumes that the host and device operate separate memories, host memory and device memory. Data transfer between the host and device memories takes place during a program run. All the data that is required for computation by the GPUs is transferred by the host to the device memory via the system bus. Programs start running sequentially on the host and kernels are then launched for execution on the device. CUDA functions, kernels, are similar to C functions in syntax but the big difference is that a kernel, when called, is executed N times in parallel by N different CUDA threads. It is possible to launch a large number of threads to perform the same kernel operation on all available cores at the same time. Each thread operates on different data. The example in Listing 2.5 is taken from the CUDA Programming Guide [9].

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
}
```

Listing 2.5: CUDA kernel example, taken from [9].

The main function is run on the host. The global keyword states that the `vecAdd` function is a kernel function to be run on the device. The special `<<<..., ...>>>` construct specifies the number of threads and thread blocks to be run for each call (or an execution configuration in the general case). Each of the threads that execute `vecAdd` performs one pair-wise addition and the thread ID for each thread is accessible via the `threadIdx` variable. Threads are organized into thread blocks (which can be organized into grids). The number of threads in a thread block is restricted by the limited memory resources. On the Fermi architecture a thread block may contain up to 512 threads. After each kernel invocation, blocks are dynamically created and scheduled onto multiprocessors efficiently by the hardware. Within a block threads can cooperate among themselves by synchronizing their execution and sharing memory data. Via the `syncthreads` function call it is possible to specify synchronization points. When using this barrier all threads in a block must wait for the other threads to finish.

Thread blocks are further organized into one-dimensional or two-dimensional
Chapter 2. Background

grids. One important thing to note is that all thread blocks should execute independently, in other words they should be allowed to execute in any order. On the Fermi architecture it is possible to run several kernels concurrently in order to occupy idle streaming multiprocessors; with older architectures only one kernel could run at a time thus resulting in some streaming multiprocessors being idle.

CUDA employs an execution mode called SIMT (single-instruction, multiple-thread) which means that each scalar processor executes one thread with the same instruction. Each scalar thread executes independently with its own instruction address and register state on one scalar processor core on a multiprocessor. On a given multiprocessor the threads are organized into so called warps, which are groups of 32 threads. It is the task of the SIMT unit on a multiprocessor to organize the threads in a thread block into warps, and this organization is always done in the same way with each warp containing threads of consecutive, increasing thread IDs starting at 0. Optimal execution is achieved when all threads of a warp agree on their execution path. If the threads diverge at some point, they are executed in serial and when all paths are complete they converge back to the same execution path.

2.8.3 Open Computing Language (OpenCL)

OpenCL (Open Computing Language) is a framework that has been developed in order to be able to write programs that can be executed across heterogeneous platforms. Such platforms could consist of CPUs, GPUs, Digital Signal Processors (DSPs), and other processors. It has been adopted into graphics card drivers by AMD, ATI and NVIDIA among others. OpenCL consists of, among other things, APIs for defining and controlling the platforms and a language for writing kernels (C-like language). Both task-based and data-based parallelism is possible with OpenCL. OpenCL shares many of its computational interfaces with CUDA and is similar in many ways. [29]
Chapter 3

Previous Research

3.1 Introduction

This chapter describes earlier research that has been conducted mainly at our research group PELAB with simulations of equation-based models on multi-core architectures.

3.2 Overview

From a perspective there are three main approaches to parallelism with equation-based models.

- **Explicit Parallel Programming Constructs in the Language**
  The language is extended with language constructs for expressing parts that should be simulated/executed in parallel. It is up to the application programmer to decide which parts will be executed in parallel. This approach is touched upon in chapter 9 and in [27].

- **Coarse-Grained Explicit Parallelization Using Components**
  The application programmer decides which components of the model can be simulated in parallel. This is described in more details in section 3.6 below.

- **Automatic (Fine-grained) Parallelization of Equation-Based Models**
  It is entirely up to the compiler to make sure that parallel executable simulation code is generated. This is the main approach that is investigated in this thesis.

The automatic parallelization approach can further be divided using the following classification.
Chapter 3. Previous Research

- **Parallelism over the method** With this approach one adopts the numerical solver to exploit parallelism over the method. But this can lead to numerical instability.

- **Parallelism over time** The goal of this approach is to parallelize the simulation over the simulation time. This method is difficult to implement though, since with a continuous time system each new solution of the system depends on preceding steps, often the immediately preceding step.

- **Parallelism of the system** With this approach the model equations are parallelized. This means the parallelization of the right-hand side of an ODE system.

### 3.3 Early Work with Compilation of Mathematical Models to Parallel Executable Code

In [3] some methods of extracting parallelism from mathematical models are described. In that work search for parallelism was done on three levels.

- **Equation System Level** Equations are gathered into strongly connected components. The goal is to try to identify tightly coupled equation systems within a given problem and separate them and solve them independently of each other. A dependency analysis is performed and an equation dependence graph is created using the equations in the ordinary differential equation system as vertices where arcs represents dependencies between equations. From this graph the strongly connected components are extracted. This graph is then transformed into an equation system task graph. A solver is attached to each task in the equation system task graph.

- **Equation Level** Each equation forms a separate task.

- **Clustered Task Level** Each subexpression is viewed as a task. This is the method that has been used extensively in other research work. See subsection 3.4 below on task scheduling and clustering.

### 3.4 Task Scheduling and Clustering Approach

In [4] the method of exploiting parallelism from an equation-based Modelica model via the creation and then the scheduling of a task graph of the equation system was extensively studied.
3.4 Task Scheduling and Clustering Approach

3.4.1 Task Graphs

A task graph is a directed acyclic graph (DAG) for representing the equation system. There are costs associated with the nodes and edges. It can be described by the following tuple.

\[ G = (V, E, c, \tau) \]

- \( V \) is the set of vertices (nodes) representing the tasks in the task graph.
- \( E \) is the set of edges. An edge \( e = (v_1, v_2) \) indicates that node \( v_1 \) must be executed before \( v_2 \) and send data to \( v_2 \).
- \( c(e) \) gives the cost of sending the data along an edge \( e \in E \).
- \( \tau(n) \) gives the execution cost for each node \( v \in V \).

An example of a task graph is given in Figure 3.1.

![Task Graph Example](image)

**Figure 3.1:** An example of a task graph representation of an equation system.

The following steps are taken.

- **Building a Task Graph** A fine grained task graph is built, at the expression level.
- **Merging** An algorithm is applied that tries to merge tasks that can be executed together in order to make the graph less fine grained. Replication might also be applied to further reduce execution time.
• **Scheduling** The fine grained task graph is then scheduled using a scheduler for a fixed number of computation cores.

• **Code Generation** Finally code generation is performed. The code generator takes the merged tasks from the last step and generates the executable code.

### 3.4.2 Modpar

Modpar is the name of the OpenModelica code generation back-end module that was developed in [4] which performs for automatic parallelization of (a subset) Modelica models. Its use is optional, via flags one can decide whether to generate serial and parallel executable code. The internal structure of Modpar consist of a task graph building module, a task merging module, a scheduling module, and a code generation module.

### 3.5 Inlined and Distributed Solver Approach

In [21] the work with exploiting parallelism by creating an explicit task graph was continued. A combination of the following three approaches was taken:

![Figure 3.2: Centralized solver running on one computational core with the equation system distributed over several computational cores.](image)
3.6 Distributed Simulation using Transmission Line Modeling

Technologies based on transmission line modeling, TLM, have been developed for a long time at Linköping University, for instance in the HOPSAN simulation package for mechanical engineering and fluid power, see for instance [17]. It is also used in the SKF TLM-based co-simulation package [1]. Work has also been done on introducing distributed simulation based on TLM technology in Modelica [40]. The idea is to let each component
Figure 3.4: Two-stage inlined Runge-Kutta solver distributed over three computational cores [21].
in the model solve its own equations, in other words we have a distributed
solver approach where each component is numerically isolated from the other
components. Each component and each solver can then have its own fixed
time step which gives high robustness and also opens up potential for parallel
execution over multi-core platforms. Time delays are introduced between
different components to encounter for the real physical time propagations
which gives a physically accurate description of wave propagation in the
system. Mathematically, a transmission line can be described in the frequency
domain by the four pole equation. Transmission line modeling is illustrated
in Figure 3.5.

![Figure 3.5: Transmission Line Modeling (TLM) with different solvers and step
sizes for different parts of the model [40].]

### 3.7 Related Work in other Research Groups

For work on parallel differential equation solver implementations in a broader
context than Modelica see for instance [36], [19], [20], [35].

In [26] and [12] a different and more complete implementation of the
QSS method for the OpenModelica compiler was described. That work
interfaces the OpenModelica compiler and enables the automatic simulation
of large-scale models using QSS and the PowerDEVS runtime system. The
interface allows the user to simulate by supplying a Modelica model as input,
even without any knowledge of DEVS and/or QSS methods. In that work
discontinuous systems were also handled, something that we do not handle
in our work in Chapter 5.

SUNDIALS from the Center for Applied Scientific Computing at Lawrence
Livermore National Laboratory has been “implemented with the goal of
providing robust time integrators and nonlinear solvers that can easily be
incorporated into existing codes” [42]. PVODE is included in the SUNDIALS
package for equation solving on parallel architectures. Interfacing this solver
with OpenModelica could be a promising future work.
Chapter 4

Simulation of Equation-Based Models on the Cell BE Processor Architecture

4.1 Introduction

This chapter is based on paper 1 which mainly covered two areas: a summary of old approaches of extracting parallelism from equation-based models (this we have covered somewhat in Chapter 2 of this thesis) and an investigation of using the STI\(^1\) Cell BE architecture [7] for simulation of equation-based Modelica models. A prototype implementation of the parallelization approaches with task graph creation and scheduling for the Cell BE processor architecture was presented for the purpose of demonstration and feasibility. It was a hard-coded implementation of an embarrassingly parallel flexible shaft model. We manually re targeted the generated parallel C/C++ code (from the OpenModelica compiler) to the Cell BE processor architecture. Some speedup was gained but no further work has been carried out since then. This work is covered in this thesis since it holds some relevance with the work on generating code for NVIDIA GPUs.

This chapter is organized as follows. We begin by describing the Cell BE processor architecture. We then discuss the above mentioned hard-coded implementation. We provide the measurements that was given in the paper. Finally, we conclude with a discussion section where we discuss the measurements results, suitability of the Cell BE architecture for simulation of

\(^1\)An alliance between Sony, Toshiba, and IBM
4.2 The Cell BE Processor Architecture

The Cell BE Architecture is a single-chip multiprocessor consisting of one so-called Power Processor Element (PPE) and 8 so-called Synergistic Processor Elements (SPE). The PPE runs the top level thread and coordinates the SPEs which are optimized for running compute-intensive applications. Each SPE has its own, small local on-chip memory for both code and data but the SPEs and PPE do not share on-chip memory. To transfer data between the main memory and the SPEs and between the different SPEs DMA transfers (which can run asynchronously in parallel) are used. To conclude, the main features of the Cell BE processor architecture are the following.

- One main 64-bit PPE processor (PowerPC) Power Processor Element, 2 hardware threads good at control tasks, task switching, and OS-level code SIMD unit VMX.
- 8 SPE processors (RISC with 128bit SIMD operations). Good at compute-intensive tasks, small local memory 256KB (code and data).
- No direct access to main memory, DMA transfers used.
- Internal communication: Signals, Mailboxes Interface to main memory (off chip, 512 MB and more).

4.3 Implementation

Here the hard-coded implementation for demonstration and feasibility studies is described. The equations on statement form to be calculated are divided into 6 different subsets and in the PPE 6 pthreads are created and loaded with 6 different program handlers. The PPE then uses the mailbox facility to send out a pointer to a control block in main memory to each SPE which is then used to transfer a copy of the control block via DMA to its local store. The SPEs will use the pointers in control block to fetch and store data from main storage and when sending and synchronizing between different SPEs. Next the initial data is read by each SPE for the different vectors $\dot{x}$ (state variable derivatives), $x$ (state variables), $u$ (input variables) and $p$ (constants and parameters) into local store. Then comes the actual iteration of the solver (that runs on the PPE) in $N$ time steps where new values of the state variables $x(t+h)$ are calculated in each step (the values $x(t+h)$ associated with each SPE). DMA transfers are used if SPEs needs to send and receive data between them. Data is sent back from the SPEs to the main memory buffer at the end of each iteration step (or at the end of some iteration steps, in a periodic manner). After all threads have finished the PPE will write this data to a result file. In order to exploit the full performance
potential of the Cell BE processor, the SIMD instructions of the SPEs need to be leveraged (but only inter-SPE parallelism and DMA parallelism was utilized). This requires vectorization of the (generated) code (or for instance keeping the array equations unexpanded throughout the compilation process). DMA transfers have the advantage that an SPE in some cases can continue to execute while the transfer is going on. For large examples code, data distribution across a cluster of several Cell BE processors is needed. An other alternative is time-consuming overlay of multiple program modules in SPE local store.

```model ShaftElement
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.TwoFlanges;
  Rotational.Inertia inertia;
  SpringDamperNL springDamper1(c=5,d=0.11);
  equation
    connect(inertia.flange_b,springDamper1.flange_a);
    connect(inertia.flange_a,flange_a);
    connect(springDamper1.flange_b,flange_b);
end ShaftElement;

model FlexibleShaft
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.TwoFlanges;
  parameter Integer n(min=1) = 3;
  ShaftElement shaft[n];
  equation
    for i in 2:n loop
      connect(shaft[i-1].flange_b,shaft[i].flange_a);
    end for;
    connect(shaft[1].flange_a,flange_a);
    connect(shaft[n].flange_b,flange_b);
end FlexibleShaft;

model ShaftTest
  FlexibleShaft shaft(n=100);
  Modelica.Blocks.Sources.Step c;
  equation
    connect(shaft.flange_a,src.flange_b);
    connect(c.y,src.tau);
end ShaftTest;
```

Listing 4.1: SimpleShaft Modelica model.

### 4.4 Measurements

The Modelica model used for the measurements is given in Listing 4.1. Running the whole flexible shaft example 100000 iteration steps on the Cell BE processor (with 6 SPUs as mentioned earlier) took about 31.4 seconds (from start of the PPU main function to end of the PPU main function). The final writing of the result to result files is not included in this measurement.
4.5 Discussion

Table 4.1: Measurements of running the flexible shaft model on six threads (on Cell BE).

<table>
<thead>
<tr>
<th>Thread</th>
<th>$T_{tot}$ (s)</th>
<th>$T_{DMA}$ (s)</th>
<th>% DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31.39</td>
<td>2.49</td>
<td>7.9%</td>
</tr>
<tr>
<td>2</td>
<td>31.39</td>
<td>12.28</td>
<td>39.1%</td>
</tr>
<tr>
<td>3</td>
<td>31.39</td>
<td>11.10</td>
<td>35.4%</td>
</tr>
<tr>
<td>4</td>
<td>31.39</td>
<td>12.25</td>
<td>39.0%</td>
</tr>
<tr>
<td>5</td>
<td>31.39</td>
<td>11.04</td>
<td>35.2%</td>
</tr>
<tr>
<td>6</td>
<td>31.38</td>
<td>4.39</td>
<td>13.9%</td>
</tr>
</tbody>
</table>

The relative speedup is shown in Figure 4.1, comparing running with 6 SPUs to one SPU. However, it is not straightforward to define relative speedup since the Cell BE architecture is a heterogenous architecture, and this measurement should be taken with some caution.

Figure 4.1: Relative speedup of running the flexible shaft model on the Cell BE architecture with 6 threads.

4.5 Discussion

From table 4.1 we can conclude several things. Thread 2 to 5 spent more than a third of the execution time on DMA transfers, but thread 1 and 6 did not spend much time doing DMA transfers. The total execution time of about 31.4 seconds is pretty bad. On a 4 core Intel Xeon with hyper-threading the same example took 11.35 seconds (using one core) and on SGI Altix 3700 Bx2 it took 22.59 seconds (using one processor). Another issue is the fact that on our Cell BE version double precision calculations takes about 7 times...
more time than single precision (this was improved in the next version of the Cell BE processor). To conclude, this implementation was a crude one and it seems the memory transfers eat up any performance gains. A model with larger and more heavier calculations in each time step might have worked better. We will discuss this further in Chapter 10 of this thesis.
Chapter 5

Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units

5.1 Introduction

This chapter discusses the use of the Quantized State Systems (QSS) [18] simulation algorithm as a way to exploit parallelism for simulating Modelica models with NVIDIA GPUs. This chapter is mainly based on paper number 2. In that paper a method was developed that made it possible to translate a restricted class of Modelica models to parallel QSS-based simulation code. The OpenModelica compiler was extended with a new back-end module that automatically generated CUDA-based simulation code. Some performance measurements of an example model on the Tesla architecture [44] was performed. The QSS method replaces the classical time slicing by a quantization of the states in an ODE system. It is an alternative way for computing ODE systems. The QSS integration method is a Discrete Event System (DEVS) method. No further work on the implementation was done after the paper was presented. The goal was two-fold: to investigate the possibility of parallelization of the QSS algorithm per se together with the chosen architecture and to investigate the parallel performance of the QSS integration method via automatically generated CUDA code. It was first suggested in [18] that QSS could be suitable for parallel execution. The set of models have been restricted only to a subset: continuous time, time-invariant
systems (with no events); index-1 DAE; initial values of states and values of parameters known at compile time, and inserted into the generated code as numbers; and no implicit systems of nonlinear equations to be solved numerically.

This chapter is organized as follows. We begin by describing the restricted set of Modelica models we generate code from. We then introduce the quantized state systems integration method. In the section after this we describe why the QSS method is suitable for parallel execution. We then make a summary of the implementation work that was described in the paper. We continue with a measurements section and finally conclude with a discussion section. We will not cover the NVIDIA GPU architecture nor the CUDA programming model however although this was done in the paper, since we have already done this in Chapter 2 of this thesis.

5.2 Restricted Set of Modelica Models

The set of models were restricted to a subset. This is mainly because the Modelica language is used to describe many different classes of systems and it was deemed suitable to limit the work. The restricted set of models are described below.

- 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.

A constant QSS integration step was used, unchanged for all the state variables but a different quantization step can be used for each state variable, with minor modification to the code.

5.3 Quantized State Systems (QSS)

The QSS integration method is a discrete event system (DEVS) method that was introduced in [8], where the authors suggested that it could be suitable for parallel execution. Here we will describe the main characteristics of this method.

Time slicing is by far the most commonly used approach for numerically solving a set of ODEs on a digital computer. But instead of discretization of the time it is possible to discretized the state values. This is what is done in
5.3. Quantized State Systems (QSS)

the QSS method. QSS is actually a set of algorithms that have in common that they aim to discretize the state values. The classical approach is as follows.

Given that the state value at time $t_k$ is equal to $x$, what is the state value at time $t_{k+1} = t_k + \Delta t$?

With QSS one instead tries to answer the following question.

Given that the state has a value of $x_k$ at time $t$, what is the earliest time instant, at which the state assumes a value of $x_{k\pm1} = x_k \pm \Delta x$?

In other words, a 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 a set of values. The currently available QSS algorithms are not yet, since the QSS method is relatively new, as sophisticated as the classical numerical ODE solvers.

A limited boundary error exists when transforming a continuous time system into a DEVS one, i.e.:

$$x = f(x, u) \rightarrow \dot{x} = f(q, u)$$

Here 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 and each state varies according to an hysteretic quantization function. When simulating a system with the QSS algorithm a variable-step technique is applied. The algorithm is asynchronous: it adjusts the time instant at which the state variable is re-evaluated to the speed of change of that state variable. In other words, different state variables will update their values independently of each others. This approach can be seen in Figure 5.1. Each state variable has an associated DEVS subsystem. The dependency between state variables and derivative equations decides the interconnection between subsystems. When the hysteretic quantization threshold is reached the events of the DEVS model are fired.

The simulation method consists of three main 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 $t_{\text{next}}$ and the associated state variable is $x_i$. If $t_{\text{next}} > t_{\text{input event}}$ than set $t_{\text{next}} = t_{\text{input event}}$ and perform the input change.

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

Chapter 5. Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units

Figure 5.1: Updating state variable values with the QSS method [34].

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

5.4 Implementation

It was noted in [18] that "due to the asynchronous behavior, the DEVS models can be implemented in parallel in a very easy and efficient way". Because of the possibility to separately compute the derivatives of the state variables and the time events schedule the QSS algorithm is naturally keen to be parallelized. We shall first describe the method in general terms and then go on to discuss the actual code that is generated. A Modelica model and the generated CUDA code can be found in Appendix A.

- The derivatives of the state variables are computed using the model equations (assuming that the initial values of the state variables are known), MIMD execution model.

- The time of the new event is calculated. Since all the computing threads execute the same code on different data we can completely exploit SIMD parallelism.

- A time advance is done. If the values of one of the inputs changes or if one of the bounds of the quantized state function is reached a new event is registered. Each value of the state variables is then re-computed and the quantized integrators are updated. Since the same code executes on different data elements SIMD parallelism can be exploited here as well.

Good performance can be achieved via the definition of a state vector array since the NVIDIA Tesla architecture requires all the computing
5.4. Implementation

cores in the same group to compute the same instruction at the same time. Each derivative state value is calculated within a separate thread but the SIMD model is not performing well here since the code to compute such values are different for each state variable. Instead a MIMD style code is needed.

When all threads finish, the derivative values have been calculated. The next time event for each variable is then calculated by the threads, executing the same portion of code. Since every thread execute the same code on different data portion this part of the code should be able to execute in SIMD fashion. Finally, next time event of the QSS simulation is determined and processed.

To conclude, the system advancement part takes full advantage of the hardware possibilities but the derivative calculation part of the code is not completely parallel with this approach.

The OpenModelica compiler was extended with a new back-end module that generates QSS, CUDA-based simulation code. The module took the equation system right after the matching and index reduction phases and generated the CUDA code.

Figure 5.2: Internal call chain in the OpenModelica compiler to obtain CUDA code.

Figure 5.2 shows the internal call chain in the compiler to obtain CUDA code. The new module is the GUPar module, the other phases were described in the background chapter of this thesis. In the GUPar module different kernel and header files are generated. As input GUPar takes the DAELow form as well as the BLT matrix and strong components information from the equation sorting phase. Some data has to be computed from the
Chapter 5. Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units

DAELow form in order to generate the model-specific files.

- For each state variable a derivative function is generated. This function contains the algorithm for the time derivative computation. If there is a dependency with other equations they are also added to the derivative function (in statement form of course).

- For each output variable an output function for computing the output values is generated. If there is a dependency with other equations they are also added to this function (in statement form).

- From the list of variables in the DAELow form initial variable (and parameter) values must be gathered

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

5.5 Measurements

The test model used for the measurements is depicted in Figure 5.3. The circuit consists of a generator voltage that comprises \( N \) different branches; each of them is composed of 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 initial time is obtained at the beginning of the program, before the memory allocation. The end time is measured when the simulation stops with the same function call and the difference between them is divided by a \( \text{CLOCKSPERSEC} \) constant, to compare architectures with different clock periods. The parallel algorithm is compared to the sequential one, where a
5.6. Discussion

In this work methods of simulating equation-based models using the QSS method with NVIDIA GPUs were investigated which was a first attempt at this, with some minor success. However, it is worth noting that the work done in [26] and [12] is more promising for the future. The simulations in this paper were very slow, compared to simulating the same model on for instant with a normal CPU. This has to do with memory latency of copying data back and forth to the GPU device.
Chapter 5. Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units

Table 5.1: Execution times and speed-up with the GeForce 8600.

<table>
<thead>
<tr>
<th></th>
<th>parallel [s]</th>
<th>sequential [s]</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 state variables</td>
<td>6.26</td>
<td>7.07</td>
<td>1.129</td>
</tr>
<tr>
<td>16 state variables</td>
<td>8.04</td>
<td>10.27</td>
<td>1.277</td>
</tr>
<tr>
<td>32 state variables</td>
<td>27.02</td>
<td>45.55</td>
<td>1.685</td>
</tr>
<tr>
<td>64 state variables</td>
<td>103.18</td>
<td>507.38</td>
<td>4.917</td>
</tr>
</tbody>
</table>

Table 5.2: Execution times and speed-up with the C1060 using one cluster for the derivatives of the state variables calculation.

<table>
<thead>
<tr>
<th></th>
<th>parallel [s]</th>
<th>sequential [s]</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 state variables</td>
<td>1.06</td>
<td>5.71</td>
<td>5.387</td>
</tr>
<tr>
<td>16 state variables</td>
<td>8.11</td>
<td>9.07</td>
<td>1.118</td>
</tr>
<tr>
<td>32 state variables</td>
<td>22.91</td>
<td>47.30</td>
<td>2.065</td>
</tr>
<tr>
<td>64 state variables</td>
<td>208.76</td>
<td>711.00</td>
<td>3.406</td>
</tr>
</tbody>
</table>

Table 5.3: Execution times and speedup with the C1060 using all the clusters for the derivatives of the state variables calculation.

<table>
<thead>
<tr>
<th></th>
<th>parallel [s]</th>
<th>sequential [s]</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 state variables</td>
<td>1.98</td>
<td>5.71</td>
<td>2.884</td>
</tr>
<tr>
<td>16 state variables</td>
<td>7.73</td>
<td>9.07</td>
<td>1.173</td>
</tr>
<tr>
<td>32 state variables</td>
<td>23.73</td>
<td>47.30</td>
<td>1.993</td>
</tr>
<tr>
<td>64 state variables</td>
<td>98.09</td>
<td>711.00</td>
<td>7.248</td>
</tr>
</tbody>
</table>
A problem with 256 state variables requires more than \( (5 \times 64 + 1 \times 32) \times \frac{256}{8} \text{ Bytes} = 11 \text{ Mb} \), while a case with 1024 state variables would require 43 Mb. The side effects of the diverging branches has to be furthermore reduced.
Chapter 6

Simulation of Equation-Based Models with Task Graph Creation on Graphics Processing Units

6.1 Introduction

This chapter is mainly based on paper number 4 which was based on [41]. In that paper it was showed 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. A new back-end module was implemented for the OpenModelica compiler and measurements were performed: a relative speedup of 4.6 was obtained for one of the models. The method for finding parallelism in this work is by creating a large task graph from the equation system, merge this coarse grained task graph into larger tasks, and then schedule this task graph for execution with NVIDIA GPUs. Methods of efficiently using the available memory spaces on the architecture were also presented, which is an important issue that we shall discuss in Chapter 10. Other ways of using the CUDA architecture more efficiently were also discussed.
6.2 Case Study

The model is taken from [15](page 584). The model models the one-dimensional wave equation that is given by a partial differential equation of the following form:

\[
\frac{\partial^2 p}{\partial t^2} = c^2 \frac{\partial^2 p}{\partial x^2}
\]

Here \( p = p(x, t) \) is a function of both space and time and 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 the difference approximation:

\[
\frac{\partial^2 p}{\partial t^2} = c^2 \frac{p_{i-1} + p_{i+1} - 2p_i}{\delta x^2}
\]

where \( p_i = p(x_1 + (i - 1)\delta x) \) on an equidistant grid and \( \delta x \) is a small change in distance. The Modelica model in Listing 6.1 is obtained.

```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;
```

Listing 6.1: Modelica model WaveEquationSample.

6.3 Implementation

The general compilation and simulation process of Modelica models was described in chapter 2. In this implementation some changes of the normal compilation process of Modelica models was done. This is depicted in Figure 6.1. A new module was implemented as a small MetaModelica package that exports a task graph to an external C++ module, which then manipulates the task graph and finally generates the CUDA code. This module is invoked with a sorted equation system as input. A task graph is then created from the equation system. A task graph was described in chapter 3. Crude costs
for the tasks were used: the cost of unary and binary operations are set to 1 and the cost of special functions are set to 4 (which should reflect the fact that a streaming multiprocessor has eight scalar processors but only two special function units). The cost of communication is set to 100, which should reflect the latencies to the global memory. After the task graph has been created a merging algorithm is applied to it. This merging algorithm is further described in [41].

The merged task graph is then sent to the scheduling phase. We need to determine in which order the tasks should be executed and whether they should be executed in parallel on different streaming multiprocessors. This is a two step process: 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. Inside one node the tasks are scheduled using a first in, first out queue with the tasks to be scheduled. An example of this approach can be seen in Figure 6.2. There is also a third approach used by the scheduler. The scheduler tries to find nodes that are operationally equivalent to other nodes. If they are operationally equivalent they are scheduled to be executed in parallel on the same streaming multiprocessor (SIMD style execution). A processor schedule is the result of the scheduling, an example can be seen in Figure 6.3. From the figure we can see that the processor schedule contains execution paths and execution path lists where an execution path is a list of task executed in order. The goal is to execute one execution path on one streaming multiprocessor and inside one streaming multiprocessor we should hopefully (if there are several execution paths) execute in SIMD mode. We can run several blocks in parallel by using the following technique (remember that we do not know in which order the different blocks are going to execute): we never execute more blocks than there are streaming multiprocessors (to avoid dead-locks) and we synchronize via the global memory. If it is the case that a task has a dependency with a task that is scheduled on another processor, the scheduler inserts signals and locks into the schedule and determines what data should be sent where. In addition to this, special execution paths for communication are inserted into the schedule. From the schedule code is then generated. This is done by iterating through the processors schedule one processor at a time one execution path at a time. Memory coalescing is used to reduce long off-chip DRAM latencies. 16 variables are read at a time from the device memory (the size of a coalesced read of 32-bit variables). These variables are then moved to where they should be in the shared memory on a streaming multiprocessor.

6.4 Run-time Code and Generated Code

The actual simulation function is given in the code Listing 6.2. A fourth-order Runge-Kutta method was used, both for the GPU-based implementation
6.4. Run-time Code and Generated Code

Modelica Model

OMC Front-end

DAELow

CudaCodegen

Equations

Task Graph

Task Merger

Merged Task Graph

Scheduler

Scheduled Tasks

Code Generator

CUDA Code

Figure 6.1: The process of compiling a Modelica model to CUDA code.

as well as the normal CPU implementation. In the code below there is a main for-loop that corresponds to the main simulation loop. The function execute\_tasks corresponds to the computation of one of the stages in the Runge-Kutta solver scheme. This is done in parallel by launching kernels for the GPU. The various step\_and\_increment functions takes care of advancing the step and adding vectors together. Note that we have four calls to execute\_tasks since we have a fourth-order Runge-Kutta solver scheme. In each iteration device to host copying is done of the vectors with state variables, which is time-consuming of course.

Figure 6.2: An example of the task scheduling algorithm.
Chapter 6. Simulation of Equation-Based Models with Task Graph Creation on Graphics Processing Units

Figure 6.3: An example of a schedule for two processors.

```c
// 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.
    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);
```
6.5 Measurements

Listing 6.2: Main CUDA simulation loop, based on a 4-stage Runge-Kutta solver.

```c
// 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);
```

Figure 6.4: Execution time for the WaveEquationSample Modelica model as a function of the number of sections.

6.5 Measurements

The specification for the two GPU cards used can be seen in table 6.1. The CPU used was an Intel Core 2 Duo E6600 with 2.4 GHz clock frequency. Note though that only one core was used. Table 6.2 shows seconds spent in different parts of the simulation of the test model WaveEquationSample and the graph in Figure 6.4 shows the execution time for the sample model as a function of the number of sections.
Table 6.1: Specifications for the GPUs used.

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

Table 6.2: Seconds spent in the different parts of the simulation of the WaveEquationSample Modelica model.

<table>
<thead>
<tr>
<th>Feature</th>
<th>8800 GTS</th>
<th>C1060 single</th>
<th>C1060 double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Execution</td>
<td>0.164</td>
<td>0.592</td>
<td>0.389</td>
</tr>
<tr>
<td>Shared Mem Allocation</td>
<td>1.440</td>
<td>1.426</td>
<td>2.287</td>
</tr>
<tr>
<td>Integration</td>
<td>0.417</td>
<td>0.400</td>
<td>0.445</td>
</tr>
<tr>
<td>Memory Transfers</td>
<td>1.104</td>
<td>1.332</td>
<td>2.278</td>
</tr>
</tbody>
</table>
6.6 Discussion

Some general conclusions can be drawn from the measurements.

- A relative speedup of 4.6 with 3840 sections was obtained using single precision calculations and comparing the GeForce 8800 GTS to the Intel E6600 CPU.

- Actual computations take a small amount of time and memory transactions take most of the time.

- The simulation times on the CPU approximately doubles when the model size is doubled. The simulation times for the GPUs instead rise slower. However, the GPUs need many thread blocks with many threads to fully utilize their power.

- The computation per variable is low for the model used. If the model would have had more computations per variable we would most likely have seen a larger performance increase when using a GPU.
Chapter 7

Compilation of Modelica Array Computations into Single Assignment C (SAC) for Execution on Graphics Processing Units

7.1 Introduction

This chapter is mainly based on paper 3. In that paper the possibility of compiling Modelica array equations into an intermediate language, Single Assignment C (SAC) [39], was investigated. SAC is a language with C-like syntax but that allows Matlab-style programming on n-dimensional arrays. The SAC2C SAC compiler can generate highly efficient code and several auto-parallelizing back-ends have been developed. These back-ends include POSIX-thread based code for shared memory multi-cores and CUDA-based code for GPUs. A future plan was to enhance the OpenModelica compiler with capabilities to detect and compile data-parallel Modelica array equations and/or algorithmic array operations into SAC WITH-loops. A SAC WITH-loop is the most important construct in the SAC language. In the paper however, only a feasibility study was conducted. As a first step calls to SAC array operations in the code generated from OpenModelica were inserted manually and as a second step parts of the OpenModelica runtime system was rewritten in SAC code. The paper was about unifying three technologies OpenModelica, SAC2C and CUDA. Measurements of this new integrated runtime system with and without CUDA were performed as well as stand-alone measurements of CUDA code generated with SAC2C.
This chapter is organized as follows. We begin by describing SAC and its main characteristics. We then continue with the actual implementation work that was done in the paper. After this we provide some of the measurements from the paper and conclude the chapter with a discussion section.

### 7.2 Single Assignment C (SAC)

SAC combines a C-like syntax with Matlab-style programming on n-dimensional arrays. The highly optimizing compiler SAC2C can generate high performance code from SAC due to its functional underpinnings. Most constructs in SAC, however, are inherited from C and the overall design policy is that a C style construct should behave in the same way as it does in C. The support of non-scalar data structures is a major difference between SAC and C however. In C the programmer has to allocate and deallocate memory as needed and sharing of data structures is explicit via the use of pointers. In SAC there is no notion of pointers and n-dimensional arrays are stateless data structures. Allocation, reuse and deallocation of memory is handled by the compiler and runtime system and arrays can be passed to and returned from functions in the same way as scalar values can. Memory is reused as soon as possible and array updates are performed in place whenever possible, this is ensured by the compiler and runtime system.

SAC comes with a very versatile data-parallel programming construct, the WITH-loop. Here we will just briefly discuss this construct. A modarray WITH-loop take the general form seen in Listing 7.1.

```
with {
  (lower1 <= idx_vec < upper1) : expr1;
  ....
  (lowern <= idx_vec < upperr) : exprn;
} : modarray(array)
```

**Listing 7.1: General form of SAC modarray WITH-loop.**

Here idxvec is an identifier and lower_i and upper_i denote expressions for which for any i lower_i and upper_i should evaluate to vectors of identical length. expri denote arbitrary expressions that should evaluate to arrays of the same shape and the same element type. Such a WITH-loop defines an array of the same shape as array is, whose elements are either computed by one of the expressions or copied from the corresponding position of the array. As we shall see, the goal is to map Modelica array equations into SAC WITH-loops.
Chapter 7. Compilation of Modelica Array Computations into Single Assignment C (SAC) for Execution on Graphics Processing Units

7.3 Implementation

We shall discuss the implementation work that was done by using an example model and show the various compilation steps by use of this model. The model used is the WaveEquationSample model introduced in Section 6.2 of this thesis. After instantiating this model the system of equations in Listing 7.2 is obtained.

\[
\begin{align*}
  p[0] &= \exp(-(-L / 2.0) \cdot 2.0); \\
  p[n-1] &= \exp(-(-L / 2.0) \cdot 2.0); \\
  \text{der}(p[0]) &= p[0]; \\
  \text{der}(p[n-1]) &= p[n-1]; \\
  \text{der}(dp[0]) &= 0; \\
  \text{der}(dp[n-1]) &= 0; \\
  \text{der}(dp[1]) &= c^2.0 \cdot ((p[2]+(-2.0*p[1]+p[0])) \cdot dL^\cdot-2.0); \\
  \text{der}(dp[n-2]) &= c^2.0 \cdot ((p[n-1]+(-2.0*p[n-2]+p[n-3])) \cdot dL^\cdot-2.0); \\
  \text{der}(dp[n-1]) &= 0; \\
  \end{align*}
\]

Listing 7.2: Instantiated equation code from the WaveEquationSample Modelica model.

In Listing 7.3 define four expressions (where \(0 \leq Y \leq n-1\) and \(2 \leq X \leq n-3\)). The expressions corresponds to the various right-hand side expressions present in the equation system in Listing 7.2.

\[
\begin{align*}
  \text{Expression 1} & . \ p[Y] \\
  \text{Expression 2} & . \ c^\cdot2.0\cdot((p[2]+(-2.0*p[1]+p[0]))\cdot dL^\cdot-2.0) \\
  \text{Expression 3} & . \ c^\cdot2.0\cdot((p[X+1]+(-2.0*p[X]+p[X-1]))\cdot dL^\cdot-2.0) \\
  \text{Expression 4} & . \ c^\cdot2.0\cdot((p[n-1]+(-2.0*p[n-2]+p[n-3]))\cdot dL^\cdot-2.0) \\
\end{align*}
\]

Listing 7.3: Definition of expressions from the instantiated equation code from the WaveEquationSample Modelica model.

The generated C++ code from OpenModelica will then have the structure as seen in the pseudo code in Listing 7.4.

```cpp
void functionODE(...) {
    // Initial code
    tmp0 = exp((-pow((L / 2.0), 2.0))); \\
    tmp1 = exp((-pow((-L / 2.0), 2.0))); \\
    stateDers[0 ... (NX/2)-1] = Expression 1; \\
    stateDers[NX/2] = Expression 2; \\
    stateDers[(NX/2 + 1) ... (NX - 2)] = Expression 3;
}
```
7.4 Measurements

The code in `functionODE` is rewritten into SAC code which can be seen in Listing 7.5.

```sac
with {  
  ([0] <= iv < [NX/2]) : Expression 1;  
  ([NX/2] <= iv <= [NX/2]) : Expression 2;  
  ([NX/2] < iv < [NX - 1]) : Expression 3;  
  ([NX-1] <= iv <= [NX-1]) : Expression 4;  
} : modarray(stateVars);
```

Listing 7.5: SAC with-loop corresponding to the generated equation code from the WaveEquationSample model.

A second approach that was tried in the paper was to rewrite the actual simulation loop in SAC. In the first approach we make at least one call to a SAC binary in each time step, which we shall see in the measurements section is very time consuming. A simple Euler loop was written in SAC, which can be seen in Listing 7.6, where `functionODE` as the same as earlier.

```sac
while (time < stop)  
{  
  states = states * timestep * derivatives;  
  derivatives = functionODE(states, c, 1, dL);  
  time = time + timestep;  
}
```

Listing 7.6: A main Euler simulation loop written in SAC.

7.4 Measurements

All the experiments were run on CentOS Linux with Intel Xeon 2.27GHz processor and 24Gb of RAM, 32kb of L1 cache, 256kb of L2 cache per core and 8Mb of processor level 3 cache. SAC2C measurements were run with version 16874, Gcc 4.5 and version 5625 of OpenModelica was used. Figure 7.1 shows time measurements for the modified generated code from the WaveEquationSample Modelica model with functionODE implemented purely in C and in SAC respectively.

Figure 7.2 shows time measurements for the WaveEquationSample Modelica model with the modified solver loop.
Chapter 7. Compilation of Modelica Array Computations into Single Assignment C (SAC) for Execution on Graphics Processing Units

Figure 7.1: The WaveEquationSample model run for different number of sections \(n\) with functionODE implemented as pure OpenModelica-generated C++ code as OpenModelica-generated C++ code with functionODE implemented in SAC. Start time 0.0, stop time 1.0, step size 0.002 and without CUDA.

Figure 7.2: The WaveEquationSample model run for different number of sections \(n\) with functionODE and euler loop implemented as pure OpenModelica-generated C++ code and as OpenModelica-generated C++ code with functionODE and euler loop implemented in SAC. Start time 0.0, stop time 10.0, step size 0.002 and without CUDA.
7.5 Discussion

In this work ways to make use of the efficient execution of array computations that SAC and SAC2C offer, in the context of Modelica and OpenModelica, were investigated. It is common to have large arrays of state variables in Modelica code, as in the model used in this chapter. It was shown that it is possible to generate C++ code from OpenModelica that can call compiled SAC binaries for execution of heavy array computations. It was also shown that it is possible to rewrite the main simulation loop of the runtime solver in SAC. To conclude, the potential for the use of SAC as an intermediate language and runtime language was shown. At least it has potential for code fragments that the OpenModelica compiler can identify as potentially data-parallel. With the new implementation of handling of unexpanded Modelica arrays in the OpenModelica compiler, this work has some future promise.
Chapter 8

Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-core Simulation

8.1 Introduction

This chapter is based on paper 6. More or less all the previous work described in this thesis has focused on automatic parallelization of equation-based models. That is, it is entirely up to the compiler of finding and analyzing parallelism. In this chapter however, a different approach is investigated. In this chapter we introduce several extensions to the algorithmic part of the Modelica language. In other words it is up to the end user modeler to express which parts of a model that should be simulated in parallel and corresponding OpenCL code is generated. The new constructs include parallel variables, parallel functions, parallel for-loops, etc. It is very important to note that these new language constructs are not part of the official language specification, they have rather been added to the OpenModelica compiler for experimental reasons. A similar approach was taken with the NestStepModelica implementation [27].

8.2 Implementation

In this section we shall briefly introduce some of the new language constructs.
8.2. Implementation

8.2.1 Parallel Variables

Recall that all data to be used on the GPU (the device) must be copied explicitly by the programmer. Therefore a special keyword has been added that specifically tells the compiler that a variable should be allocated on the device. An example of this is given in Listing 8.1.

```modelica
function parvar
    Integer m = 1000;
    Integer A[m, n];
    Integer B[m, n];
    parallel Integer pA[m, m];
    parallel Integer pB[m, m];
end parvar;
```

Listing 8.1: Example of parallel variables in Modelica.

The first three variables are located in the normal host memory while the two last matrices will be allocated on the device. We can then do the copying of data between the host memory and the device memory in a normal fashion. The assignments $A := B$, $pA := A$, $B := pB$ and $pA := pB$ would all be valid in function parvar.

8.2.2 Parallel Functions

With the help of the keyword `parallel`, parallel functions can be defined in Modelica. They correspond to OpenCL functions defined in kernel files or CUDA device functions and these functions are for distributed independent parallel execution. A `parallel` function must be called from an other `parallel` function, from a `kernel` function (see below) or from inside a `parfor` loop. Parallel functions can neither have `parfor` loops inside of them nor is it possible to declare explicit `parallel` variables inside of them (since a `parallel` function is already executing on the GPU device).

8.2.3 Kernel Functions

`Kernel` functions correspond to CUDA global functions and to OpenCL kernel functions. They can be called from serial host code and are entry functions for parallel execution. They can not, however, be called from the body of a `parfor` loop or from other `kernel` functions. They can not have `parfor` loops in their bodies nor can they have any explicit parallel variables. They are defined by using the kernel keyword, see Listing 8.2.
Chapter 8. Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-core Simulation

Listing 8.2: Example of kernel function in Modelica.

In the function above a parallel function multiply is called. Note the kernel keyword before the function.

8.2.4 Parallel For-Loops

A parallel for-loop is written using the `parfor` keyword and it is a loop meant to be executed in a parallel fashion using a device. There are some constraints in order to make this work. First of all, all variables referenced inside a loop must be `parallel` variables. Secondly, one iteration can not have a loop-carried dependency to another iteration. An example of a function with a parallel loop is given in Listing 8.3.

Listing 8.3: Example of parallel for-loops in Modelica.
8.3 Measurements

In the code above the function multiply is a parallel function. Note that the variables referenced inside the loop all are parallel variables. It is enough to specify the `parfor` keyword for the outermost loop, the inner loops will all be considered as parallel loops. The iterations of the loop specified with the `parfor` keyword are equally distributed among available processors. If there are more iterations than threads then some threads might perform more than one iteration.

8.2.5 OpenCL Functionalities

Some additional features have been added for management and execution of parallel operations.

- `oclbuild(String)` Builds an OpenCL source file and returns an OpenCL program object.
- `oclkernel(oclprogram, string)` The first argument is a previously built OpenCL program and the second argument is a kernel. The function creates an OpenCL kernel object.
- `oclsetargs(oclkernel,...)` This function takes a previously created kernel object and a variable number of arguments. It sets each argument to one in the kernel definition.
- `oclexecute(oclkernel)` Executes the specified kernel.

8.2.6 Synchronization and Thread Management

There are also some features for managing threads and synchronizations, they are briefly described below.

- `globalThreadId()` This function can only be called from a parallel or kernel function and it returns the global id of the current thread.
- `localThreadId()` This function can only be called from a parallel or kernel function and it returns the local id of the current thread (not finalized).
- `globalBarrier()` A global barrier that makes sure that all threads reach this point before any thread is allowed to continue.
- `localBarrier()` Used to synchronize all local threads (not finalized).

8.3 Measurements

Measurements from simulating two models from the implemented benchmark suite is presented in this section. All simulations where run with time step 0.2, with the DASSL solver, start time 0.0 and with a stop time of 0.2 seconds.
Chapter 8. Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-core Simulation

(it makes sense with the same time step and duration since we are dealing with purely algorithmic models). The time measurement is taken as the difference from when the simulation loop start and simulation loop finishes.

- **Matrix Multiplication.** A $M \times K$ matrix $C$ is produced from multiplying an $M \times N$ matrix $A$ by an $N \times K$ matrix $B$. A considerable speedup has been achieved as a result of parallel simulation of this model on parallel platforms since this model presents a very great level of data-parallelism.

- **LU Decomposition.** The Gaussian Elimination method is used to decompose a matrix to lower and upper triangular forms, which can be used for solving a system of linear equations $Ax = B$. The size of the problem were successively increased the values of the parameters $M$, $N$, and $K$ of matrices $A$ and $B$ (both matrices had the same size).

- **Stationary Heat Conduction.** This model models the transformation of energy in stationary surfaces. Thermal energy transfers to surfaces with lower temperatures from surfaces with higher temperatures occur. A parameter $N$ determines the size of the models which refers to the size of the surface and an equidistant grid.

For executing sequential code (generated by the old OpenModelica compiler): the Intel Xeon E5520 CPU, with 16 cores, each with 2.27 GHz clock frequency. For executing parallel code by our new code generator the same CPU was used and the NVIDIA Fermi-Tesla M2050 GPU. The simulation execution times are used as results to give us information regarding to the following considerations. The measurements were performed to validate that the code generator generates efficient parallel code and to ensure that the Modelica language extensions are successfully targeted toward the OpenCL architecture. The time plots can be seen in Figure 8.1, Figure 8.2 and Figure 8.3 respectively.

### 8.4 Discussion

In this section some novel language constructs for the algorithmic part of the Modelica language have been discussed. Several measurements were provided from a benchmark test suite, MPAR, containing models using these new language constructs. The models contains for instance heavy computations over large matrices and considerable speedups compared to normal CPU execution were obtained. It is important to once more note that these language constructs are not part of the official Modelica language specification, rather they have been implemented in the OpenModelica compiler for experimental purposes. The results obtained are not surprising considering that we are studying the imperative parts of the Modelica language.
Figure 8.1: Simulation Time Plot for Matrix Multiplication Model as a Function of Model Parameter $M, N, \text{and } K$.

Figure 8.2: Simulation Time Plot for LU Decomposition Model as a Function of Model Parameters $M, N, \text{and } K$. 
Chapter 8. Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-core Simulation

Figure 8.3: Simulation Time Plot for Heat Conduction Model as a Function of Model Parameter $N$. 

Simulation Time (seconds)

CPU E5520 (Serial)
CPU E5520 (Parallel)
GPU M2050 (Parallel)

$N$

32 64 128 256 512 1024 2048

0 1,000 2,000 3,000 4,000 5,000 6,000
Chapter 9

Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

Traditionally most Modelica compilers have expanded arrays into scalars and array equations into scalar equations, one equations for each array element. This has advantages in providing specific symbolic manipulation for each array element equations. However, this approach also has serious disadvantages when trying to exploit data parallelism from arrays. In this chapter an approach is investigated to keep arrays unexpanded throughout the Modelica compilation process in order to facilitate exploiting data parallelism, e.g. for efficient execution on GPUs. The main approach is to avoid expanding array operations and to combine many references of array elements into references of whole arrays or array slices.

9.1 Introduction

The present OpenModelica Compiler (OMC) handles array-related constructs such as arrays of state variables, array equations and for-equations by expanding them into scalar variables and scalar equations. Work has been started resulting in a preliminary prototype to provide functionality to keep arrays of state variables and array equations unexpanded throughout the compilation process. This functionality is turned on by a compilation flag. Several changes in the OMC compilation process are needed. This involves
Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

changes in more or less all parts of the compilation process (in the front-end, in the middle parts, and in the code generation part). The most difficult task involves changes in the equation sorting and equation processing part of the compilation process. The focus in this chapter is on the equation sections of Modelica; array constructs in algorithmic sections of Modelica are discussed in Chapter 8. Keeping array equations and arrays of state variables unexpanded opens up the possibility of generating more efficient code for execution on parallel architectures, for instance GPUs. The equation handling parts in the OpenModelica compilation process were briefly described in Chapter 2. By keeping array equations and array variables unexpanded the equation sorting should become faster since the number of equations will be lower. Keeping array equations unexpanded results in smaller matrices and data structures thus leading to less memory consumption and less compilation time. Keeping array equations unexpanded is also beneficial for normal serial code since the number of statements that are generated in the final executable code is reduced. One could for instance generate a for-loop instead of many assignments statements. Finally, note that in the traditional approach when array equations are expanded, at the code generation phase the information about which equations belongs together has been lost, thus loosing opportunities for generation of data-parallel code. Modelica models containing operations over large arrays of state variables often originate from models derived from a discretization of a partial differential equation, one such model was given in Listing 6.1.

In this chapter we begin by describing the problem with the current approach in Section 9.6. We then continue in Section 9.3 and Section 9.4 with describing some initial algorithms that should be applied to array equations and array for-equations so that they can be handled easier in the equation sorting phase which is described in Section 9.5. We then continue with a section on the actual implementation in OMC, Section 9.6. Finally, we end with a discussion section in Section 9.7.

9.2 Problems with Expanding Array Equations

The following are three problems with the current approach of expanding array equations. The mentioned loss of array-related information makes it more difficult e.g. to generate data-parallel code

- The current equation matching algorithm in the OMC back-end assumes that array equations and array variables have all been expanded into scalar equations and scalar variables, thus leading to large data structures, large memory consumption and long compilation times as a result.

- At the code generation phase all the original explicit array operations in the model have been lost.
• The dimension sizes of arrays and the number of equations related to specific array equations have been lost.

9.3 Splitting For-Equations with Multiple Equations in their Bodies

The following can be concluded from the Modelica Language Specification [25] regarding for-equations: since the solution order of the equations in a Modelica model is not specified it does not matter if we split a for-equation into multiple for-equations with one equation each (with the current approach for-equations are expanded, merged into the larger model equation system, and the ordering might change). For example, in Listing 9.2 and Listing 9.3 the for-equation in the model is transformed into several for-equations.

9.3.1 Algorithm

An algorithm in pseudo code is given in Listing 9.1.

```plaintext
for each equation in the for-equation body do
    create a separate for-equation containing the equation,
    use the same head
```

Listing 9.1: Splitting array for-equations with multiple equations in the body.

9.3.2 Examples

An example of using this splitting approach is given below in Listing 9.2 and 9.3.

```plaintext
model TestModel
    parameter Integer n = 4;
    Real u[n](start = 1.0);
    Real x[n](start = 1.0);
    equation
        for y in 1:n loop
            der(u[y]) = -0.167;
            der(x[y]) = 80;
        end for;
end TestModel;
```

Listing 9.2: Modelica model containing a for-equation with multiple equations in the body.

The model in Listing 9.2 can be transformed into the model in Listing 9.3.

```plaintext
model TestModel
    parameter Integer n = 4;
    Real u[n](start = 1.0);
    Real x[n](start = 1.0);
```
Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

Listing 9.3: Modelica model containing for-equations with one equation each in their bodies.

9.4 Transforming For-Equations and Array Equations into Slice Equations

In order to have all equations on a standard uniform format it is suitable to transform all for-equations\(^1\) and array equations into slice equations. Such equations have a simple form where all array references have the shape of array slices which are indexed.

9.4.1 Algorithm

An algorithm in pseudo code for conversion of for-equations and array equations to slice equations is given in Listing 9.4. The reason is to simplify index reduction, BLT sorting, etc. Basically there are a few forms: equations containing whole arrays (without any indexing), or array slices, or array elements, or for-equations. Single scalar array elements can be eliminated by combining them into slices. For-equations can be transformed into array equations with slices. Whole arrays can be trivially converted to a slices that are the same as the array (e.g. from 1 to end). Overlapping array slices is not handled in the current algorithm but the algorithm can be extended with the following approach: partition the slices into smaller slices so that the overlapping part becomes its own slice.

\[
\begin{align*}
\text{for each equation do} \\
\quad \text{case equation is for-equation} \\
\quad \quad \text{for each for-equation iterator do} \\
\quad \quad \quad \text{case iterator used as an array index} \\
\quad \quad \quad \quad \text{replace by computing an array slice of the array indexing using the dimension data from the for-equation head} \\
\quad \quad \quad \text{remove for-equation head, use only body equation} \\
\quad \quad \text{otherwise} \\
\quad \quad \quad \text{replace by expanding into an array constructor with iterator(s) using the dimension data, from the for-equation head}
\end{align*}
\]

\(^1\)Another approach could be not to transform for-equations. For instance algorithmic for-loops in algorithm sections are currently handled as they are.
9.4. Transforming For-Equations and Array Equations into Slice Equations

remove for-equation head, use only body equation
otherwise
  for each variable reference (if not already slice reference)
  create a slice reference using the dimension information
  for that variable.

Listing 9.4: Splitting array for-equations with multiple equations in the body.

9.4.2 Examples

Several examples of using this approach is given below. The model in Listing 9.5 can be transformed into the model in Listing 9.6.

```model TestModel
parameter Integer n = 4;
Real u[n] (start = 1.0);
Real x[n] (start = 1.0);
equation
  for y in 1:n loop
    der(u[y])=-0.167;
  end for;

  for y in 1:n loop
    der(x[y])=80;
  end for;
end TestModel;
```

Listing 9.5: Modelica TestModel model containing for-equations.

```model TestModel
parameter Integer n = 4;
Real u[n] (start = 1.0);
Real x[n] (start = 1.0);
equation
  der(u[1:n])=-0.167;
  der(x[1:n])=80;
end TestModel;
```

Listing 9.6: Modelica TestModel model containing slice-equations.

The model in Listing 9.7 can be transformed into the model in Listing 9.8.

```model WaveEquationSample
import Modelica.SIunits;
parameter SIunits.Length L = 10 "Length of duct";
parameter Integer n = 30 "Number of sections";
parameter SIunits.Length dl = L/n "Section length";
parameter SIunits.Velocity c = 1;
SIunits.Pressure[p] p(each start = 1.0);
Real[n] dp(start = fill(0,n));
equation
  p[1] = exp(-(L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp = der(p); // Array equation
```

Listing 9.7: Modelica WaveEquationSample model containing for-equations.

Listing 9.8: Modelica WaveEquationSample model containing slice-equations.
Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

9.5 Matching and Sorting of Unexpanded Array Equations

Here an outline of matching and sorting (see Chapter 2) of unexpanded array equations is given, for at least a subset of possible models. The algorithm assumes that the model is balanced (same number of equations and variables) and there are no overlapping array reference slices (array slices are non-overlapping if each array element belongs to at most one slice). Handling overlapping array reference slices needs some modifications of the algorithm: partition the slices into smaller slices so that the overlapping part becomes its own slice.

9.5.1 Algorithm

The algorithm is divided into several steps, shown in Listing 9.9, 9.10, 9.11, 9.12, 9.13 and 9.14.
9.5. Matching and Sorting of Unexpanded Array Equations

for each equation do
  for each variable reference do
    case not array slice reference
      insert the index of variable in the set for the equation.
      add a minus (−) sign if state,
      no minus sign if derivative of state.
    otherwise array slice reference
      insert the index of variable in the set for the equation
      store the array slice reference.
      add a minus (−) sign if state,
      no minus sign if derivative of state.

Listing 9.9: Step 1: Generate a list with one set for each equation, containing all variable references in the equation.

create an array of lists of booleans, one list of booleans for each variable (empty lists to begin with)
for each set in the list from step 1 do
  for each variable reference in the set do
    if the variable reference index is negative then skip
    else
      if variable reference has a slice that overlaps
      with another slice in the list from the array
      (with the same variable name)
      then return
      else insert a variable reference with slice
      information into the list for the correct array entry
  if overlapping slices we can not continue with the remaining steps


Check if the number of equations equals the number of variables
if not the algorithm can not continue with the remaining steps

Listing 9.11: Step 3: Check to make sure that the model is balanced.

for each set in the list from step 1 do
  create a new row in the matrix
  for each variable reference in the set do
    if the variable reference is not negative insert into row
    else do nothing

Listing 9.12: Step 4: Building the incidence matrix (see Chapter 2).

Do the matching as usual but now we also need to check
that the dimension on the left side equals the
dimension on the right side.

Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

As before, no major changes needed.


9.5.2 Examples

Several examples of the algorithm in the previous section are given in this section.

Example 1

The Modelica model for the first example is given in Listing 9.15.

```
model NonExpandedArray1
  parameter Integer p = 500;
  Real x[p];
  Real y[p];
  Real z[p];
  Real q[p];
  Real r[p];
equation
  2.3232*y + 2.3232*z + 2.3232*q + 2.3232*r = der(x);
  der(y) = 2.3232*x + 2.3232*z + 2.3232*q + 2.3232*r;
  2.3232*x + 2.3232*y + 2.3232*q + 2.3232*r = der(z);
  der(r) = 2.3232*x + 2.3232*y + 2.3232*z + 2.3232*q;
  2.3232*x + 2.3232*y + 2.3232*z + 2.3232*r = der(q);
end NonExpandedArray1;
```

Listing 9.15: Example 1: Modelica Model NonExpandedArray1.

```
eq 1  x[1:n], -y[1:n], -z[1:n], -q[1:n], -r[1:n]
eq 2  -x[1:n], y[1:n], -z[1:n], -q[1:n], -r[1:n]
eq 3  -x[1:n], -y[1:n], z[1:n], -q[1:n], -r[1:n]
eq 4  -x[1:n], -y[1:n], -z[1:n], q[1:n], r[1:n]
eq 5  -x[1:n], -y[1:n], -z[1:n], q[1:n], -r[1:n]
```

The above model has no overlapping slices and the model is balanced. The above model will result in the following matrix.

```
\begin{pmatrix}
  eq & \text{der}(x[1:n]) & \text{der}(y[1:n]) & \text{der}(z[1:n]) & \text{der}(q[1:n]) & \text{der}(r[1:n]) \\
  1 & 1 & 0 & 0 & 0 & 0 \\
  2 & 0 & 1 & 0 & 0 & 0 \\
  3 & 0 & 0 & 1 & 0 & 0 \\
  4 & 0 & 0 & 0 & 1 & 0 \\
  5 & 0 & 0 & 0 & 1 & 0 \\
\end{pmatrix}
```

=> Sorting =>
9.5. Matching and Sorting of Unexpanded Array Equations

\[
\begin{pmatrix}
    \text{eq} & \text{der}(x[1:n]) & \text{der}(y[1:n]) & \text{der}(z[1:n]) & \text{der}(q[1:n]) & \text{der}(r[1:n]) \\
    1 & 1 & 0 & 0 & 0 & 0 \\
    2 & 0 & 1 & 0 & 0 & 0 \\
    3 & 0 & 0 & 1 & 0 & 0 \\
    5 & 0 & 0 & 0 & 1 & 0 \\
    4 & 0 & 0 & 0 & 0 & 1 
\end{pmatrix}
\]

**Example 2**

The Modelica model for the second example is given in Listing 9.16.

```modelica
model WaveEquationSample
    import Modelica.SIunits;
    parameter SIunits.Length L = 10 "Length of duct";
    parameter Integer n = 30 "Number of sections";
    parameter SIunits.Length dl = L/n "Section length";
    parameter SIunits.Velocity c = 1;
    SIunits.Pressure [n] p(each start = 1.0);
    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;
```

Listing 9.16: Example 2: Modelica model WaveEquationSample.

=> Transform for-equation =>

```modelica
model WaveEquationSample
    import Modelica.SIunits;
    parameter SIunits.Length L = 10 "Length of duct";
    parameter Integer n = 30 "Number of sections";
    parameter SIunits.Length dl = L/n "Section length";
    parameter SIunits.Velocity c = 1;
    SIunits.Pressure [n] p(each start = 1.0);
    Real [n] dp(start = fill(0,n));
    equation
        p[1] = exp(-(-L/2)^2);
        p[n] = exp(-(-L/2)^2);
        dp = der(p);
end WaveEquationSample;
```

Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

In this model there are no overlapping slices and the model is balanced. The above model will result in the following incidence matrix. The columns \(dp[1]\) and \(dp[n]\) result from the check in step 2, which is an extended version.

\[
\begin{pmatrix}
\, eq \, & \text{der}(dp[2 : n - 1]) \, & \text{der}(p[1 : n]) \, & dp[1] \, & dp[n] \\
1 & 0 & 0 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{pmatrix}
\]

=> This causes Pantelides algorithm\cite{18} to detect that equation 1 and 2 must be derived to get an equation for \(dp[1]\) and \(dp[n]\), where \(p[1]\) and \(p[n]\) are dummy states =>

\[
\begin{pmatrix}
4 & 1 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{pmatrix}
\]

=> Sorting =>

Example 3

The Modelica model for the third example is given in Listing 9.18.

```
class ArraySlice1
  Real a [4];
equation
  a[{1,3}] = a[{2,4}];
  a[1]=time;
  a[4]=1;
```
end ArraySlice1;

Listing 9.18: Example 3: Modelica model ArraySlice1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The above model has no overlapping slices and the model is balanced. The above model will result in the following matrix.

\[
\begin{pmatrix}
1 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 1 & 1 \\
3 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

=> Sorting =>

\[
\begin{pmatrix}
3 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 \\
2 & 0 & 0 & 1 & 1 \\
\end{pmatrix}
\]

9.6 Implementation

In Listing 9.19 some of the code of the main execution flow of the OpenModelica compiler is presented. See also Figure 9.1. The main point of entry is the function translateFile. We shall describe the main changes to the different phases of the compilation process in the different sections below.

- **Instantiation** The instantiation is started with a call to function instantiate from translateFile in Listing 9.19. See Section 9.6.1.
- **Lowering** The lowering is started with a call to function backend-DAE.lower from optimizeDae in Listing 9.19. See Section 9.6.2.
- **Equation Sorting and Matching** The equation sorting and matching is started with a call to function BackendDAEUtil.getSolvedSystem from optimizeDAE in Listing 9.19. See Section 9.6.3.
- **Code Generation** The entry function to the code generation is function simcodegen which is called from optimizeDae in Listing 9.19. See Section 9.6.5.
Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

Figure 9.1: Main execution flow (function calls) of the OpenModelica compiler.

```
protected function translateFile
  "function: translateFile
   This function invokes the translator on a source file.
   The argument should be a list with a single file name,
   with the rest of the list being an optional
   list of libraries and .mo-files if the file is a .mo-file"
  input list<String> inStringLst;
algorithm
  := matchcontinue (inStringLst)
  local
    Absyn.Program p, pLibs;
    list<SCode.Element> scode;
    DAE.DAElist d_1, d;
    String s, str, f;
    list<String> libs;
    Absyn.Path cname;
    ...
    Env.Cache cache;
    Env.Env env;
    DAE.FunctionTree funcs;
    list<Absyn.Class> cls;

    // A .mo-file, followed by an optional list of
    // extra .mo-files and libraries. The last class
    // in the first file will be instantiated.
    case (f :: libs)
      equation
        isModelicaFile(f);

        // Parse the first file.
        (p as Absyn.PROGRAM(classes = cls)) = Parser.parse(f);
        ...
```

9.6. Implementation

// Instantiate the program.
(cache, env, d_1, scode, cname) = instantiate(p);
...

// Run the backend.
opimizeDae(cache, env, scode, p, d, d, cname);
then ()
end matchcontinue;
end translateFile;

protected function optimizeDae
"function: optimizeDae
Run the backend.
Used for both parallel and for normal execution."
input Env.Cache inCache;
input Env.Env inEnv;
input SCode.Program inProgram1;
input Absyn.Program inProgram2;
input DAE.DAElist inDAElist3;
input DAE.DAElist inDAElist4;
input Absyn.Path inPath5;
algorithm
match continue (inCache, inEnv, inProgram1, inProgram2,
inDAElist3, inDAElist4, inPath5)
local BackendDAE.BackendDAE dlow, dlow_1;
array <list<Integer>> m, mT;
array <Integer> v1, v2;
 BackendDAE.StrongComponents comps;
list <SCode.Element> p;
Absyn.Program ap;
DAE.DAElist dae, daeimpl;
Absyn.Path classname;
Env.Cache cache;
Env.Env env;
DAE.FunctionTree funcs;
Strings str;
case (cache, env, p, ap, dae, daeimpl, classname)
equation
true = runBackendQ();
funcs = Env.getFunctionTree(cache);
dlow = BackendDAECreate.lower(dae, funcs, true);
dlow_1 = BackendDAEUtil.getSolvedSystem(cache, env, dlow, funcs, NONE(), NONE(), NONE());
modpar(dlow_1);
Debug.execStat("Lowering Done", CevalScript.RT_CLOCK_EXECSTAT_MAIN);
simcodegen(dlow_1, funcs, classname, ap, daeimpl);
then
() ;
...matchcontinue;
end
Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

end optimizeDae;

protected function simcodegen
"function simcodegen
  Generates simulation code using the SimCode module"
input BackendDAE.BackendDAE inBackendDAE5;
input DAE.FunctionTree inFunctionTree;
input Absyn.Path inPath;
input Absyn.Program inProgram3;
input DAE.DAElist inDAElist4;
algorithm :=
  matchcontinue (inBackendDAE5, inFunctionTree, inPath, inProgram3, inDAElist4)
  local BackendDAE.BackendDAE dlow;
  DAE.FunctionTree functionTree;
  String cname_str, file_dir;
  Absyn.ComponentRef a_cref;
  Absyn.Path className;
  list <SCode.Element> p;
  Absyn.Program ap;
  DAE.DAElist dae;
array <Integer> ass1, ass2;
array <list <Integer>> m, mt;
BackendDAE.StrongComponents comps;
SimCode.SimulationSettings simSettings;
String methodbyflag;
Boolean methodflag;
  case (dlow, functionTree, className, ap, dae)
    equation
      true = RTOpts.simulationCg();
      Print.clearErrorBuf();
      Print.clearBuf();
      cname_str = Absyn.pathString(className);
      simSettings = SimCode.createSimulationSettings (0.0, 1.0, 500, 1e-6,"dassl", "", "mat", ".", false, "");
      (. . .) = SimCode.generateModelCode(dlow, functionTree, ap, dae, className, cname_str, SOME(simSettings));
      Debug.execStat("Codegen Done", CevalScript.RTCLOCK_EXECCONV_MAIN);
    then
      () ;
      ...
    end matchcontinue;
  end simcodegen;

Listing 9.19: Main control flow of the OpenModelica compiler.
9.6. Implementation

9.6.1 Instantiation - Symbolic Elaboration

Handling non-expanded arrays in the instantiation (symbolic elaboration) phase involves changes in the InstSection package as well as the Static package, which is called by functions in the InstSection package for static evaluation of expressions. The InstSection package is called from the Inst package.

- In InstSection.InstEquationCommonWork and the case /* equality equations e1 = e2 */ the +a flag is retrieved from RTopts.splitArrays and sent as a Boolean flag to Static.elabExp. RTopts.splitArrays is false if no expansion should be done.

- In InstSection.condenseArrayEquation a condition, true=RTOpts.splitArrays has been inserted in order to stop the code from being executed if no expansion should be done.

- In InstSection.instArrayEquation some minor changes have been done for not expanding array equations and the code is guarded with the RTOpts.splitArrays flag.

- In the Static package a performVectorization flag has been added to many functions in order to make sure that expressions are not expanded if the flag is set, it was missing in many places.

- In Static.crefVectorize it has been arranged so that no vectorization takes place if the vectorization flag is false.

- In Lookup.makeDimensionSubscript a new case has been added so that the type dimension is not expanded.

9.6.2 Lowering

The main changes to handle non-expanded arrays in the lowering process are described below.

- A new data structure that contains array equations has been added to the lowered form. This data structure stores a lowered array equation. In BackendDAE.Equation a new record MULTIDIMEQUATION2 has been added. This record simply represents an unexpanded array equation.

- In BackendDAECreatelower guarded code has been inserted that places the array equations from the array equation array in BackendDAE.DAE into the normal ordered equation array in BackendDAE.DAE. The actual function for performing this transformation can be found in BackendDAEUtil (from BackendDAE.MULTIDIMEQUATION to BackendDAE.MULTIDIMEQUATION2).
Chapter 9. Compilation of Unexpanded Modelica Array Equations for Execution on Graphics Processing Units

- BackendDAECreate.lowerMultidimeqns and BackendDAECreate.lowerMultidimeqns2 have both been modified.

9.6.3 Equation Sorting and Matching

- In BackendDAEUtil.traversingincidenceRowExpFinder new cases for handling component references and derivative functions call with references to unexpanded arrays have been added. The problem was that the lowering phase unexpanded arrays are inserted into the environment with variables with the addVar function but with a subscript list. Then when encountering a variable reference and trying to look it up with getVar the algorithm does not have a subscript list so the look up fails. In BackendDAEUtil.traversingincidenceRowExpFinder a change has been made so that the algorithm looks up the variable with getVar but first attaches a subscript list to the component reference (by using the size in the type information).

- In BackendDAEUtil.incidenceRow a new case has been added for BackendDAE.MULTIDIMEQUATION2.

9.6.4 Finding Strongly Connected Components

- The strongly connected components phase that detects if equation nodes belong to a strongly connected component in the data dependency graph can more or less be left unmodified.

9.6.5 CUDA Code Generation

- New data structures for array equations have been added to SimCode.SimEqSystem.

- In the function SimCode.callTargetTemplates a new case have been added for generating CUDA code.

- A new Susan2 OpenModelica text template, SimCodeCUDA.tpl has been created which generates a CUDA-based fourth-order Runge-Kutta solver as well as the compiled code for the array equations. The CUDA Runge-Kutta solver is based on the one implemented in [41].

9.7 Discussion

In this chapter we have discussed how Modelica array equations can be kept unexpanded throughout the compilation process. Previously such equations were expanded into scalar equations and handled as normal scalar equations.

2Susan is a text template language that is used in the OpenModelica project for creating code generators [43]
9.7. Discussion

However, if array equations can be kept unexpanded there are several benefits. We get a faster compilation process, especially the equation sorting phase becomes faster since we have smaller and fewer data structures to process. We can also more easily compile more efficient and faster code. Keeping array equations unexpanded makes it much easier to detect data parallelism when compiling code for parallel architectures such as GPUs.
Chapter 10

Discussion

In this chapter we summarize the work described in the previous chapters. In Section 10.1 we first discuss GPU architectures in general followed by Section 10.2 where we try to answer the research question from Chapter 1 whether GPU architectures are suitable for parallel simulation of equation-based models. Finally in Section 10.3 we end with a summary and future work.

10.1 What Kind of Problems are Graphics Processing Units Suitable for?

It is important to note that GPUs have traditionally been used for handling computer graphics computations. They are suitable for algorithms where processing of large blocks of data is done in a data-parallel fashion. Initially GPUs were used for texture-mapping and polygon rendering, geometric calculations (such as the rotation and translation of vertices), oversampling and interpolation techniques to reduce aliasing. Other example of applications that can be accelerated in video decoding include: motion compensation, inverse discrete cosine transform, intra-frame prediction, bit stream processing, inverse quantization (IQ), etc. Applications that are ideally suitable for GPUs have large data sets, high parallelism, and minimal dependency between data elements. [9]

The advent of General Purpose GPU (GPGPU) computing has expanded the usage of GPUs to not only include computer graphics calculations. This move into other fields of computations has come from the observation that most computer graphics calculations include computations on matrices and vectors. GPGPUs have been made possible by the addition of stream processing on non-graphics data features, which in turn have been made possible by the addition of programmable stages and higher precision arithmetic to the rendering pipelines. Although GPGPU and architectures/software platforms
10.2. Are Graphics Processing Units Suitable for Simulating Equation-Based Modelica Models?

The problem of simulating equation-based Modelica models is essentially one of solving a system of differential equations, either in ODE or DAE form, given initial values of the state variables, values of constants and parameters, start time, stop time, time step, etc. (there is also the algorithmic part of Modelica, see below). The front-end and middle part of a Modelica compiler removes all high-level structure and we essentially end up with an equation system that is to be solved at runtime. When solving such an equation system on a computer a numerical method for solution of ODEs is applied (solving it analytically is not practically possible with a computer in the general case). There are two categories of such numerical methods that have been described in this thesis: time-stepping based methods (e.g., Euler, Runge-Kutta, DASSL, etc.) and quantization of state methods (QSS).

With the classical time-stepping approach there is essentially a central loop that usually is run on one computational node\(^1\). When using a system containing GPUs this central loop is most suitable to run on the host CPU. In each time step data must be gathered and distributed from the host memory to the device memory to perform the time step computations, which is time consuming. Another problem is simply that we need to distribute the equation system over the GPU device architecture for computation in each time step. However this is not an easy task since some parts of the equation system might potentially depend on other parts of the equation system, in fact this must in general be assumed. We must therefore communicate between different parts of the distributed equation system and it is often not clear how this can be done in a cost efficient manner. In the approach of Chapter 6 this was done (when data between different streaming multiprocessors needed to be sent) by synchronizing with the global memory which was very slow. The problem of solving a system of differential equations is not data-parallel in nature in the general case. Models that exhibits a large degree of data-parallelism are suitable though, which is discussed below.

Yet another problem is that the equation system may contain algebraic

\(^1\)Also an approach of inlining the solver and replicating code has been tried, which was described briefly in Chapter 2 and is described in more details in [21]
loops (simultaneous equation systems), in other words an additional solver step must be applied to each such simultaneous equation system in each time iteration. This could potentially be done on one streaming multiprocessor for each simultaneous equation system. Even with a distributed solver approach, such as the approach described in [21], we have the same problem of communication of data between different computations; the same notion still holds, we are trying to solve a problem that is linear in nature on a highly data-parallel architecture. The QSS-based method is somewhat more suitable for parallel computations, its suitability for GPUs is discussed below.

It is also important to note that in our work we have not involved some of the more complex features of Modelica. We have limited ourselves to models which result in purely continuous time equation systems. We have not yet dealt with hybrid models, in other words models that can contain both continuous-time and discrete-time variables and language constructs. For dealing with such models one approach could be to run the solver on the host CPU, calculate the events there and in each time step compute the equation system on the GPU device(s).

There are also purely algorithmic models which consists of imperative code and data-parallel models. These kind of models are different from the above and the previous discussion does not concern these models. Instead this is discussed below.

10.2.1 Discussion on The Various Approaches of Simulating Modelica Models on GPUs

In this section we again discuss the various approaches from the different chapters of the thesis. It is important to note that although the general problem of solving an ODE or DAE equation system may not be very data-parallel in nature, there are however important subsets of models that contain data-parallel features where the use of GPUs is suitable.

- **Simulation of Equation-Based Models with Task Graph Creation on Graphics Processing Units** This approach might work for equation systems where there is not much dependency between different parts of the equation system, where not much communication is needed between different streaming multiprocessors. However in the general case a task graph arising from an equation-based models is not necessarily data-parallel in nature, thus it is not easy to map a task graph for execution with a GPU, since communications between different parts of the equation system is needed. Also the amount of memory transfers taking place between the CPU and the GPU might be time consuming.

- **Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units** The approach presented in this thesis
10.3. Summary and Future Work

was an attempt to compile Modelica to QSS-based code on a GPU, with a small and simplified model. Simulation times were bad because of the amount of memory transfers taking place. Although QSS-based simulations might be suitable for parallel executions we still have the same problem with GPUs as with time-stepping methods: the computation of the equation system is not in general a data-parallel task, which GPUs are good at. Even with the QSS-based method, when updating the states, the same equation system must be solved, even though when to solve this equation system is different from a time-stepping method.

• **TLM Component-Based Partitioning** The approach with TLM component-based partitioning has not yet been tried with GPUs at our research group PELAB. TLM-based component partitioning was described briefly in Chapter 3. One approach could be to put each partitioned submodel on a streaming multiprocessor. The computations on the streaming multiprocessors could run fairly independent given that the partition of the original model results in submodels that are fairly independent.

• **Compilation of Unexpanded Modelica Array Equations for Efficient Simulation on Graphics Processing Units** In the work in Chapter 9 and Chapter 7 we look at a restricted subset of Modelica models. We mainly focus on models that are data-parallel in nature, or has features that are data-parallel in nature. For these kind of models GPUs are suitable.

• **Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-core Simulation** In Chapter 8 we looked at purely algorithmic models that were data-parallel in nature. For these kind of models GPUs are suitable.

10.3 Summary and Future Work

As discussed earlier, the general problem of solving an ODE or DAE equation system may not be very data-parallel in nature. There are however important subsets of models that contain data-parallel features where the use of GPUs are suitable. These are mainly models that contain operations over large arrays of state variables (or other variables). For such models it is suitable to compile the array operations directly into GPU-based code. A related approach is to search for data-parallelism in the resulting compiled equation system, i.e. to reconstruct array operations from sets of similar operations on scalar variables (array elements). However, it is more efficient for the compiler to directly compile array operations than to later reconstruct them from scalars.
Chapter 10. Discussion

One kind of Modelica models that could be interesting for simulation on GPUs are models containing Partial Differential Equations (PDEs). The Modelica language standard does not currently contain constructs for modeling PDEs. However, such constructs were proposed in [38], where PDEs in the context of Modelica were discussed extensively. Currently PDEs can be modeled in Modelica via a discretization approach using for-equations; the WaveEquationSample model in Listing 6.1 is an example of such a model. These kind of models are almost always highly data-parallel in nature.

An interesting future work could be to try to classify the Modelica models that are suitable for simulation with GPUs and those models where a different simulation architecture would be more suitable. A question then is whether this classification should be done at the front-end, before all the structure is removed, or later in the compilation process when the whole equation system is available as one system. Applying machine learning techniques could be one interesting feature work. With machine learning techniques computers use empirical data to learn various behaviors. The goal would be to run OpenModelica with many different models and the compiler could then learn what kind of architecture is most suitable for what kind of model. [24]

It is important to note that GPGPU is evolving rapidly. CUDA for instance now supports function pointers, recursion, C++ templates, virtual methods, etc., as noted in Chapter 2. However, as David Black-Schaffer, one of the developers of OpenCL notes [2], the underlying hardware architecture is still one optimized for data-parallel problems. He proposes the following check list for determining whether your application is suitable to be implemented on GPUs.

- Is your application data-parallel?
- Is your application computationally intensive?
- Do you wish to avoid global synchronization?
- Does your application need lots of bandwidth?
- Is use of small caches OK for your application?
- Does your application use single precision?\footnote{For GPGPUs with good double precision support, which is becoming more and more common, this check is not needed of course.}

Regarding the ongoing discussion whether to use CPUs or GPUs, in [11] it is claimed that the gap in performance between CPUs and GPUs is overestimated and it is suggested that the performance gap can be decreased, which is demonstrated for a set of example applications, provided the right optimization techniques are applied to the CPU implementation. Perhaps future generations of CPUs and GPUs will converge towards each other.
Appendix A

Quantized State Systems
Generated CUDA Code

The following code example contains the model dependent part of the code of the experiment presented in chapter 5 with 8 state variables, where three output variables are defined. The original Modelica model is given in Listing A.1.

```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
    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;
```

Listing A.1: Test Model

Two output files are produced: model.h and model.cu. The first one is a C-CUDA header file and contains the function prototypes of the routine contained in the second one.

```
#define _MODEL_H

JECTED

# define _MODEL_H

```
Chapter A. Quantized State Systems Generated CUDA Code

```c
#define NUMBER_STATES 8
#define NUMBER_INPUTS 1
#define NUMBER_OUTPUTS 3
#define NUMBER_EVENTS 10
#define SIMULATION_STEP 0.001

#include "model.h"
#include "inclusion.h"

/* Initialization */
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;
}

/* 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 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 "inclu ded.h"
#include "mod e l.h"

/* Initialization */
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;
```
```c
int i = threadIdx.x;

void derivative
{
    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;
    }
}

void output
{
    int i = threadIdx.x;
}
```

```c
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;
}
```

```c
device void y2 (float* y, float* x, float* u, float* t, unsigned* c) {
    y[2] = x[7];
}
```

```c
device void y1 (float* y, float* x, float* u, float* t, unsigned* c) {
    y[1] = x[3];
}
```

```c
device void y0 (float* y, float* x, float* u, float* t, unsigned* c) {
    y[0] = x[0];
}
```

Listing A.2: Generated CUDA QSS Code
Bibliography


BIBLIOGRAPHY


In this thesis we investigate techniques and methods for parallel simulation of equation-based, object-oriented (EOO) Modelica models on graphics processing units (GPUs). Modelica is being developed through an international effort via the Modelica Association. With Modelica it is possible to build computationally heavy models; simulating such models however might take a considerable amount of time. Therefor techniques of utilizing parallel multi-core architectures for simulation are desirable. The goal in this work is mainly automatic parallelization of equation-based models, that is, it is up to the compiler and not the end-user modeler to make sure that code is generated that can efficiently utilize parallel multi-core architectures. Not only the code generation process has to be altered but the accompanying run-time system has to be modified as well. Adding explicit parallel language constructs to Modelica is also discussed to some extent. GPUs can be used to do general purpose scientific and engineering computing. The theoretical processing power of GPUs has surpassed that of CPUs due to the highly parallel structure of GPUs. GPUs are, however, only good at solving certain problems of data-parallel nature. In this thesis we relate several contributions, by the author and co-workers, to each other. We conclude that the massively parallel GPU architectures are currently only suitable for a limited set of Modelica models. This might change with future GPU generations. CUDA for instance, the main software platform used in the thesis for general purpose computing on graphics processing units (GPGPU), is changing rapidly and more features are being added such as recursion, function pointers, C++ templates, etc.; however the underlying hardware architecture is still optimized for data-parallelism.

Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units

Kristian Stavåker

Modelica, GPU, CUDA, OpenCL, Modeling, Simulation
Department of Computer and Information Science  
Linköpings universitet  
Licentiate Theses  
Linköpings Studies in Science and Technology  
Faculty of Arts and Sciences

No 17 Vojin Plavsic: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)


No 73 Ola Strömberg: A Structure Editor for Documents and Programs, 1986.


No 181 Henrik Eriksson: A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.


Johan Ringström: Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
Eva Toller: Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
Dan Frisöld: Metoder i användning - mot förbättring av systemutveckling genom situationell metodkonstruktion och metodanalys, 1995.
Stefan Flodin: Efficient Management of Object-Oriented Queries with Late Binding, 1996.
Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.
Mikael Lind: Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
Kristina Larsen: Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag, 1996.
Jörgen Lindström: Överföringsanvändning av kommunikationsteknik, 1996.
Fredrik Eklund: Declarative Error Diagnosis of GAPLog Programs, 1997.
Mats Gustafsson: Bringing Role-Based Access Control to Distributed Systems, 1997.
Bengt E W Andersson: Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörssystem i samverkan om utbyte av information, 1998.
No 766 Martin V. Howard: Designing dynamic visualizations of temporal data, 1999.
No 787 Charlotte Bjerkegren: Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 807 Svein Bergum: Managerial communication in telework, 2000.
FiF-a 32 Karin Hedström: Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 820 Jean Paul Meynard: Control of industrial robots through high-level task programming, 2000.
FiF-a 34 Göran Hultgren: Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 Magnus Kåld: The role of management control systems in strategic business units, 2000.
FiF-a 49 Stefan Holgersson: IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
FiF-a 51 Per Oscarsson: Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 931 Fredrik Elg: Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidenst in kontroll av mikrovärlden Moro, 2002.
No 938 Bourhane Kadmiry: Fuzzy Control of Unmanned Helicopter, 2002.
No 964 Peter Bunu: Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 991 Almut Herzog: Secure Execution Environment for Java Electronic Services, 2002.
Andreas Hansson: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.


Andreas Käll: Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.

He Tao: Aligning and Merging Biomedical Ontologies, 2006.


FiF-a 90 Amra Hallilovic: Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.

FiF-a 91 Hanna Broberg: Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.


Jiri Tranka: Prerequisites for data sharing in emergency management, 2007.


Erik Kuiper: Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.


Martin Karresaud: Completing the Picture - Fragments and Back Again, 2008.


Mirko Thorstenson: Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.


Mohammad Saifullah: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011.
Anna Vapen: Contributions to Web Authentication for Untrusted Computers, 2011.
Kristian Stavåker: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.