Simulation of Modelica Models on the CUDA Architecture

by

Per Östlund

LIU-IDA/LITH-EX-A--09/062--SE

2009-11-23
Final Thesis

Simulation of Modelica Models on the CUDA Architecture

by

Per Östlund

LIU-IDA/LITH-EX-A--09/062--SE

2009-11-23

Supervisor: Kristian Staväker
Department of Computer and Information Science, Linköpings Universitet

Examiner: Peter Fritzon
Department of Computer and Information Science, Linköpings Universitet
Abstract

Simulations are very important for many reasons, and finding ways of accelerating simulations are therefore interesting. In this thesis the feasibility of automatically generating simulation code for a limited set of Modelica models that can be executed on NVIDIA’s CUDA architecture is studied. The OpenModelica compiler, an open-source Modelica compiler, was for this purpose extended to generate CUDA code.

This thesis presents an overview of the CUDA architecture, and looks at the problems that need to be solved to generate efficient simulation code for this architecture. Methods of finding parallelism in models that can be used on the highly parallel CUDA architecture are shown, and methods of efficiently using the available memory spaces on the architecture are also presented.

This thesis shows that it is possible to generate CUDA simulation code for the set of Modelica models that were chosen. It also shows that for models with a large amount of parallelism it is possible to get significant speedups compared with simulation on a normal processor, and a speedup of 4.6 was reached for one of the models used in the thesis. Several suggestions on how the CUDA architecture can be used even more efficiently for Modelica simulations are also given.
Contents

1 Introduction
  1.1 Motivation ............................................. 7
  1.2 Problem Formulation ................................. 8
  1.3 Choice of Technology ................................ 8
  1.4 Limitations ......................................... 8
  1.5 Previous Work ...................................... 9
  1.6 Goal ................................................. 9

2 Background
  2.1 Modeling and Simulation ............................ 10
    2.1.1 The Process and Purpose of Simulation ......... 10
    2.1.2 Generating Simulation Code ..................... 11
  2.2 OpenModelica ..................................... 11
  2.3 Numerical Integration .............................. 12
    2.3.1 Euler Integration ................................ 13
    2.3.2 Runge-Kutta Integration ....................... 14
  2.4 Compute Unified Device Architecture ............... 15
    2.4.1 The CUDA Hardware Architecture .............. 15
    2.4.2 Memory Hierarchy ................................ 16
    2.4.3 Efficiently Using Off-Chip Memory ............ 17
    2.4.4 Programming model .............................. 17
  2.5 Previous work ..................................... 20
    2.5.1 Task Graph ..................................... 20
    2.5.2 Task Merging ................................... 21
    2.5.3 Software Pipelining and Inline Solver ....... 21
    2.5.4 Technique Evaluation ........................... 21

3 Implementation
  3.1 Extracting a Task Graph from OMC ................. 23
    3.1.1 The CudaCodegenExt Interface ................. 23
    3.1.2 The CudaCodegen package ....................... 23
    3.1.3 The Task Graph ................................ 25
  3.2 Task merging ....................................... 27
    3.2.1 The TaskMergerNode Class ..................... 27
    3.2.2 The TaskMerger Class ........................... 27
    3.2.3 Merging Algorithms ............................. 28
    3.2.4 Approximating Costs ............................ 29
  3.3 Scheduling .......................................... 30
### 3.3 Critical Path Algorithm
- 3.3.1 Critical Path Algorithm ........................................ 30
- 3.3.2 Scheduling tasks ........................................ 30
- 3.3.3 Finding Equivalent Nodes ................................. 31
- 3.3.4 The Schedule Class ........................................ 32
- 3.3.5 Scheduling Algorithm ....................................... 33
- 3.3.6 Communication Scheduling ................................. 33

### 3.4 Code generation .............................................. 33
- 3.4.1 The CodeWriter Class ..................................... 33
- 3.4.2 The SharedMemoryAllocator Class ..................... 34
- 3.4.3 The ThreadIdIndexer Class .............................. 35
- 3.4.4 The CodeGenerator Class ................................. 35

### 3.5 RK4 Solver for OMC ........................................ 43

### 4 Results .......................................................... 45
- 4.1 Models .......................................................... 45
  - 4.1.1 TestModel ................................................ 45
  - 4.1.2 WaveEquationSample ................................... 46
  - 4.1.3 HeatedPlate2D ........................................... 47
- 4.2 Hardware ........................................................ 47
  - 4.2.1 CPU Configuration ...................................... 47
  - 4.2.2 GPU Configuration ...................................... 47
  - 4.2.3 Floating Point Precision ............................... 48
- 4.3 Measurements .................................................. 49
  - 4.3.1 TestModel ................................................ 49
  - 4.3.2 WaveEquationSample ................................... 50
  - 4.3.3 HeatedPlate2D ........................................... 51

### 5 Discussion ....................................................... 53
- 5.1 Result Interpretation ......................................... 53
- 5.2 Future work .................................................... 54
- 5.3 Conclusions .................................................... 55

### A Simulation Time Measurements ................................ 56
- A.1 TestModel .................................................... 56
- A.2 WaveEquationSample ....................................... 57
- A.3 HeatedPlate2D .............................................. 59
## List of Figures

2.1 A connection diagram and corresponding Modelica code for an electrical circuit. ................................................. 12
2.2 Numerical integration using the Forward Euler method. ........... 14
2.3 Simplified schematic of a GPU with 96 SPs. .......................... 15
2.4 A task graph representation of the assignment \( x = 3 \cdot y + \frac{5}{6} \). ... 20

3.1 The process of compiling a Modelica model to CUDA code. ....... 22
3.2 Control flow from front-end to CudaCodegen. .......................... 23
3.3 The ConstantDuplicate algorithm. ..................................... 28
3.4 The SingleChildMerge algorithm. ...................................... 28
3.5 The ChildMerge algorithm. ........................................... 29
3.6 The DuplicateChild algorithm. ....................................... 29
3.7 An example of the critical path algorithm. ............................. 30
3.8 An example of the task scheduling algorithm. ......................... 31
3.9 An example of a schedule for two processors. ........................ 32
3.10 Copying variables from the global to the shared memory. ......... 34

4.1 Circuit of resistors and capacitors. .................................... 45
4.2 Simulation time plot for the TestModel model. ........................ 50
4.3 Simulation time plot for the WaveEquationSample model. ........... 51
4.4 Simulation time plot for the HeatedPlate2D model. .................. 52
### List of Tables

4.1 GPU Specifications. .................................................. 48
4.2 Seconds spent in the different parts of the simulation. .......... 51

A.1 TestModel simulation times for the GeForce 8800 GTS. ....... 56
A.2 TestModel simulation times for the Tesla C1060, single precision. 56
A.3 TestModel simulation times for the Core 2 Duo E6600. ......... 57
A.4 WaveEquationSample simulation times for the GeForce 8800 GTS. 57
A.5 WaveEquationSample simulation times for the Tesla C1060, single precision. 57
A.6 WaveEquationSample simulation times for the Tesla C1060, double precision. 57
A.7 WaveEquationSample simulation times for the Core 2 Duo E6600. 58
A.8 Simulation times for different configurations of WaveEquationSample on the GeForce 8800 GTS. ......................... 58
A.9 Simulation times for different configurations of WaveEquationSample on the Tesla C1060, with single precision. ............ 58
A.10 Simulation times for different configurations of WaveEquationSample on the Tesla C1060, with double precision. .......... 58
A.11 HeatedPlate2D simulation times for the GeForce 8800 GTS. . . 59
A.12 HeatedPlate2D simulation times for the Tesla C1060, single precision. .................................................. 59
A.13 HeatedPlate2D simulation times for the Core 2 Duo E6600. .... 59
List of Listings

3.1 Parts of the CudaCodeGen::buildExpression function ........ 24
3.2 The TaskGraph::CreateBinopTask method .................... 26
3.3 The CodeGenerator::EmitBinop method ......................... 36
3.4 The CodeGenerator::EmitTask method ......................... 37
3.5 Part of the code generated from the TestModel model .... 38
3.6 An example of a simulation loop, generated from the TestModel model. ........................................ 42
3.7 The simulation loop for the RK4 solver implemented for OMC. 43
4.1 Modelica code for the TestModel model ......................... 46
4.2 Modelica code for the WaveEquationSample model ........... 46
4.3 Modelica code for the HeatedPlate2D model ................... 47
Acronyms

**API** Application Programming Interface. 17, 18, 20

**ATMM** Aronsson’s Task Merging Method. 21, 28

**CPU** Central Processing Unit. 7, 9, 16, 18, 19, 43, 47–49, 53–55

**CUDA** Compute Unified Device Architecture. 7–10, 12, 15–22, 32, 35, 38, 40, 41, 43–45, 48–51, 54, 55

**DAE** Differential Algebraic Equation. 12, 23, 24

**DAG** Directed Acyclic Graph. 20

**DRAM** Dynamic Random Access Memory. 16, 17

**FIFO** First In, First Out. 31

**FPU** Floating Point Unit. 49

**GPGPU** General-Purpose computing on Graphics Processing Units. 7, 8, 48, 55

**GPU** Graphics Processing Unit. 7–9, 15–19, 31, 32, 38, 41, 47–50, 53–55

**MIMD** Multiple Instructions, Multiple Data. 16, 40

**ODE** Ordinary Differential Equation. 8, 12

**OMC** OpenModelica Compiler. 8, 9, 12, 20–23, 25, 26, 43, 44, 46–50

**OpenCL** Open Computing Language. 8, 55

**QSS** Quantized State System. 9

**RK4** Fourth-Order Runge-Kutta. 9, 13, 15, 35, 41–44

**SFU** Special Function Unit. 15, 30, 55

**SIMD** Single Instruction, Multiple Data. 16, 21, 28, 30–36, 40, 55

**SIMT** Single Instruction, Multiple Thread. 16

**SM** Streaming Multiprocessor. 15, 16, 30, 32, 34, 38, 40, 41, 49, 53, 55

**SP** Scalar Processor. 15, 16, 30, 32, 55
Chapter 1

Introduction

Graphics Processing Units (GPUs) have in recent years started to become increasingly programmable, and due to their highly parallel structure they are well suited for a wide range of data-parallel algorithms. GPUs can for some problems be several orders of magnitude faster than a general-purpose Central Processing Unit (CPU), and it is therefore interesting to look at ways of exploiting the power of a GPU. This thesis will study the feasibility of automatically generating simulation code from Modelica models that can be run on a GPU.

Motivation for why this is an interesting problem to look at will be given in this chapter, and a closer look at the problem will also be given. Some motivation behind the choice of technologies that will be used to solve the problem will also be presented, as well as some limitations and previous work that will influence this thesis. Finally the main goal of this thesis will be presented.

1.1 Motivation

Although GPUs have traditionally been used only for graphics acceleration, and as such have had relatively fixed functionality compared with CPUs, they have in recent years become increasingly programmable. The release of technologies such as the Compute Unified Device Architecture (CUDA) by NVIDIA [5] and the Stream SDK by AMD [3] has given rise to a new field of research: General-Purpose computing on Graphics Processing Units (GPGPU). The theoretical processing power and memory bandwidth of GPUs have far surpassed that of CPUs in recent years due to the highly parallel structure of GPUs. At the time of writing the fastest mainstream single-GPU graphics card, AMD’s ATI Radeon HD 5870, is capable of 2720 GFLOPS [2], while the fastest mainstream CPU, Intel’s Core i7 965, is capable of only 51.2 GFLOPS [8]. GPUs need massively data-parallel programs to approach their peak performance though.

Simulations, as will be discussed in chapter 2.1, are very important for many reasons, and they can be very computationally heavy. As is also discussed in chapter 2.1 it is often preferable to use an equation-based language such as Modelica when constructing models for simulation purposes, instead of writing simulation code by hand. There may thus be much to gain if code that can harness even a part of the power of GPUs can be automatically generated from Modelica models.
1.2 Problem Formulation

The task of automatically generating simulation code from Modelica models that can be run on a GPU contains two major problems that will need to be solved. The first problem is how to extract the necessary information from a Modelica model that is needed to generate simulation code. The second problem is how to generate efficient simulation code that can take advantage of a GPU’s parallel structure.

1.3 Choice of Technology

OpenModelica [18] is a project with the goal of providing an open-source Modelica environment for research purposes, and has been chosen as the basis for this thesis. The open-source nature of OpenModelica makes it very suitable for this thesis, because the compiler can easily be customized by directly changing the compiler’s source code. This makes the problem of extracting information from a Modelica model easier, because the OpenModelica Compiler (OMC) can perform the necessary parsing of models and manipulation of equations that is needed. A module that generates simulation code for GPUs can then be inserted into OMC where the normal code generator is usually executed. OpenModelica will be discussed further in chapter 2.2.

For GPGPU purposes there exists several ways of programming GPUs, of which CUDA and the Stream SDK have already been mentioned. CUDA and the Stream SDK are vendor specific however, for NVIDIA and AMD respectively, which limits their usefulness somewhat. Other alternatives are the Open Computing Language (OpenCL) [17], a cross-platform open standard, and DirectCompute, a part of Microsoft’s DirectX [11]. Both NVIDIA and AMD support OpenCL and DirectCompute, but unfortunately neither had been released for public use when work on this thesis began. CUDA has therefore been chosen for this thesis, the reasons for this choice being the relative maturity of CUDA and also because the necessary hardware was already available to the author of this thesis. Understanding the CUDA architecture and programming model will be essential to understanding the work done in this thesis, and CUDA will therefore be covered in detail from the point of view of this thesis in chapter 2.4.

1.4 Limitations

Modelica is a large language, and implementing code generation for the whole language is outside the scope of this thesis. The set of models that will be possible to simulate will therefore be restricted to:

- pure continuous-time systems.
- systems that can be reduced to Ordinary Differential Equation (ODE) systems without algebraic loops.
- systems where the initial values of all variables and parameters are known at compile time.
The first two limitations are related to the choice of numerical integration method. OMC uses DASSL, which is deemed too complex for this thesis. In this thesis the Fourth-Order Runge-Kutta (RK4) method will be used instead, which is much simpler while still accurate enough to give meaningful results in most cases. The RK4 method will be covered in chapter 2.3.

The last limitation only means that initial values will not be taken as input to the simulation program, but directly inserted into the simulation code. This reduces the complexity of the simulation code somewhat, but has no impact on the simulation itself.

1.5 Previous Work

Some work has previously been done in the field of automatic parallelization of equation-based Modelica models, which this thesis is partly based on. Peter Aronsson studied in his Ph.D. thesis [1] how equation-based simulation programs could be automatically parallelized with the help of task merging, which resulted in the ModPar module for OMC. Håkan Lundvall later improved on Aronssons work in [9], by inlining the numerical solver and introducing software pipelining. While these techniques work well on CPUs it remains to be seen if they are suitable for GPUs, something that will be considered in this thesis.

Some work has also been done in [10] on an implementation that generates simulation code for CUDA, which uses the event-based integration method Quantized State System (QSS). That implementation mostly focuses on the QSS algorithm itself though, while this thesis will focus on how the GPU can be used efficiently for simulation purposes, and it will not have any major influence on the work done in this thesis.

1.6 Goal

The goal of this thesis is to evaluate the feasibility of simulating Modelica models on GPUs. This goal will be achieved by implementing a module for OMC that automatically generates simulation code for CUDA from the limited set of models outlined in 1.4. The simulation code will use the numerical integration method RK4. Since OMC only supports the DASSL and Euler method it will be necessary to also implement support for RK4, so that the performance of the generated code for CUDA can be evaluated relative to the performance of the generated code for CPUs.
Chapter 2

Background

This chapter will give the theoretical background needed to understand the implementation. The first section will define the processes of modeling and simulation, and motivate why equation-based languages such as Modelica are important. The second section will then take a look at the OpenModelica project, which will provide the tools needed to extract information from Modelica models. The third section describes numerical integration, which will be needed to simulate models, in particular the Runge-Kutta method that has been chosen for this thesis. CUDA will then be covered thoroughly, since a good understanding of the architecture will be necessary to generate efficient simulation code. The last section will then evaluate techniques used in previous works, to see if they are suitable for CUDA.

2.1 Modeling and Simulation

This section will explain what the processes of modeling and simulation are, and motivate why there is a need for them. It will also look at what options there are for generating simulation code.

2.1.1 The Process and Purpose of Simulation

A model can be described as a simplified, often mathematical, representation of a complex system. An electrical circuit can for example be represented by mathematical equations that describe the circuit’s behaviour. The behaviour of a thrown ball can with the help of mechanics also be described in a mathematical way. The purpose of modeling is to create models that contain the relevant properties and that describes the real system in an accurate way.

When a model has been acquired it can be used for experiments, and this is the concept of simulation. Simulation of a model means that an instance of the model is created, and the behaviour of that instance is studied over time. For an electrical circuit this would mean setting up the equations that describe the circuit, and then studying how the variables in the equations change over time. A thrown ball would in the same way be described by physical equations, and the balls trajectory can then be followed by evaluating the equations at different points in time.
What is then the purpose of simulation? A circuit can be built so that measurements can be taken directly from it, and a thrown ball’s trajectory can be recorded and measured. Building real systems can be costly though, and it is not always possible to acquire the measurements needed. Consider the process of determining the best design for an electrical circuit. Instead of building several different circuits and taking measurements on them it will likely be both faster and cheaper to simulate them instead. Or consider that there is a risk that a spacecraft might malfunction and explode when trying to land on the moon. Surely it would be better to simulate the spacecraft than actually building it and sending it to the moon only to see if it explodes or not.

2.1.2 Generating Simulation Code

As explored in the previous section there is obviously a great need for simulations. To simulate even simple models requires a large amount of computations though, and computers are therefore typically used for simulations. Generating simulation code that can be executed on a computer can be done in several different ways, and in this section three alternatives will be described.

The first alternative is to write a computer program by hand in an ordinary programming language that simulates the desired model. This is a time-consuming and error-prone process that often leads to ad-hoc implementations that are difficult to maintain.

Another alternative is to use a block-based tool, such as Simulink [19], where the model is constructed from graphical blocks with fixed inputs and outputs. This avoids the need for the manual transformation and simplification of equations that are needed with hand-written implementations. The blocks are still causal though, i.e. the inputs and outputs are fixed, so if the user wishes to redefine the inputs and outputs of the model a complete reconstruction of the model is often needed.

The third alternative is to use an equation-based language, for example Modelica, where the model is described using equations. Such a language leaves the causality unspecified to the user, so that the same models can be reused with different inputs and outputs. Consider for example a simple electrical resistor modeled with Ohm’s law, \( V = I \times R \). If the resistance \( R \) is fixed for the resistor it’s possible to use the resistor to study how the voltage \( V \) changes when the current \( I \) changes, or how \( I \) changes when \( V \) changes. This allows more complex models to be constructed from simpler models in much the same way as with block-based tools, but with the advantage of being able to redefine the inputs and outputs of a model.

2.2 OpenModelica

The Modelica language is an equation-based, object-oriented modeling language developed by the Modelica Association [12]. It can be used to model the dynamic behaviour of technical systems using differential, algebraic and discrete equations. The meaning of object-oriented in the Modelica case is that models are represented by classes, which allows larger models to be built from class instances. An example of a simple electrical circuit and its corresponding Modelica code can be seen in figure 2.1. The electrical circuit is thus constructed
A connection diagram and corresponding Modelica code for an electrical circuit.

```
model SimpleCircuit
    Resistor R1(R=10);
    Capacitor C(C=0.01);
    Resistor R2(R=100);
    Inductor L(L=0.1);
    SineVoltage 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 SimpleCircuit;
```

Figure 2.1: A connection diagram and corresponding Modelica code for an electrical circuit.

from instances of electrical component classes from the Modelica library. The `model` keyword marks the class as a restricted class that can not be used in connections. For more information about the Modelica language, see [6].

As was mentioned in section 1.3, OpenModelica is a project that aims to provide an open-source Modelica environment for research purposes. It consists of a Modelica compiler, OMC, as well as other tools that forms an environment for creating and simulating Modelica models. In this thesis the focus will be on OMC, since it will provide the front-end of the Modelica-to-CUDA compiler. OMC is written in MetaModelica, an extended subset of Modelica that makes it suitable for compiler construction. Among these extensions are pattern matching and support for several additional data structures such as tuples and lists.

OMC is internally loosely divided into a front-end and a back-end. The front-end parses a Modelica model and produces an equation system. This system is a Differential Algebraic Equation (DAE) system, but note that only DAEs that can be reduced to Ordinary Differential Equation (ODE) systems are considered in this thesis. The back-end then sorts these equations and also tries to optimize them, and finally it generates C-code from the equations. The C-code is then linked with a C runtime library, which includes numerical solvers that can be chosen at runtime. See [18] and [6] for more detail on how OMC works. In this thesis both the front-end and back-end of OMC will be used, but the final code generation step will be replaced with a code generator that produces CUDA code.

### 2.3 Numerical Integration

One of the inputs to the final code generation step, with the limited set of models used in this thesis, is a set of explicit ODEs on the form:

\[
\dot{x} = f(x(t), u(t), p, t) \tag{2.1}
\]
and a set of initial conditions:
\[ x(t = t_0) = x_0 \] (2.2)
where \( x \) is the state vector, \( u \) is the input vector, \( p \) is a vector of parameters and/or constants, and \( t \) represents time. However, one of the limitations listed in section 1.4 was that all initial values are known at compile time, so the \( p \) vector can be ignored. The form then becomes:
\[ \dot{x} = f(x(t), u(t), t) \] (2.3)
The task is thus to calculate the values of the state vector \( x \) in a given time interval. This could be done by analytically finding the anti derivative to each element in \( \dot{x} \), but doing so may be difficult or even impossible in some cases. The state variables will instead be approximated by using a numerical integration method. Numerical integration is a vast subject, and the rest of this section will only give a short introduction from a practical point of view. For a more thorough and theoretical view on numerical integration as related to simulation, see [4].

One way of approximating the state variables is to do a Taylor-Series expansion about any given point in time for every element \( x_i \) in the state vector:
\[ x_i(t + h) = x_i(t) + \frac{dx_i(t)}{dt} \cdot h + \frac{d^2x_i(t)}{dt^2} \cdot \frac{h^2}{2!} + \ldots \] (2.4)
Plugging equation 2.3 in then yields:
\[ x_i(t + h) = x_i(t) + f_i(t) \cdot h + \frac{df_i(t)}{dt} \cdot \frac{h^2}{2!} + \ldots \] (2.5)
Using different number of terms of the Taylor-Series expansion and approximations of higher state derivatives then yields different integration methods. While the RK4 method is the only integration method used in this thesis it is instructive to first look at Euler integration, a much simpler method.

### 2.3.1 Euler Integration

Euler integration is the result of simply truncating equation 2.5 after the second term, which gives:
\[ x(t + h) \approx x(t) + h \cdot f(x(t), u(t), t) \] (2.6)
or equivalently
\[ x(t + h) \approx x(t) + h \cdot \dot{x}(t) \] (2.7)
Euler integration has the advantage of not requiring any approximation of higher-order derivatives. This particular method is called the Forward Euler method, and it can be represented graphically as in figure 2.2. The method thus works by taking small steps in the direction of the functions derivative, or in other words the curves slope. Since this is only an approximation of the real curve it will introduce a small error for each step, corresponding to the truncated terms. Each step will therefore introduce an error of the order \( h^2 \), with a total accumulated error of the order \( h \). Smaller steps will thus give a more accurate solution, and infinitely small steps will give the exact solution. Taking infinitely small steps would mean taking infinitely many steps though, which of course is not feasible on a computer.
Using this method for simulation simply means calculating each state variable’s derivative, and then calculating the next value for each state variable based on its derivative and current value:

Step 1: \[ \dot{x}(t_0) = f(x(t_0), u(t_0), t_0) \]
\[ x(t_0 + h) = x(t_0) + h \cdot \dot{x}(t_0) \]

Step 2: \[ \dot{x}(t_0 + h) = f(x(t_0 + h), u(t_0 + h), t_0 + h) \]
\[ x(t_0 + 2h) = x(t_0 + h) + h \cdot \dot{x}(t_0 + h) \]

Step 3: \[ \dot{x}(t_0 + 2h) = f(x(t_0 + 2h), u(t_0 + 2h), t_0 + 2h) \]
\[ x(t_0 + 3h) = x(t_0 + 2h) + h \cdot \dot{x}(t_0 + 2h) \]

etc.

2.3.2 Runge-Kutta Integration

There exists a whole family of Runge-Kutta methods of different types and orders. In this thesis one of the most commonly used versions will be used, the explicit fourth order Runge-Kutta method (RK4). This method is more complex than the Euler method, but still somewhat similar, and looks like this:

\[ k_1 = f(x(t), u(t), t) \]
\[ k_2 = f(x(t + \frac{h}{2} \cdot k_1), u(t + \frac{h}{2}), t + \frac{h}{2}) \]
\[ k_3 = f(x(t + \frac{h}{2} \cdot k_2), u(t + \frac{h}{2}), t + \frac{h}{2}) \]
\[ k_4 = f(x(t + h \cdot k_3), u(t + h), t + h) \]

\[ x(t + h) \approx x(t) + h \cdot \frac{1}{6} \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \]

The first four steps in the method evaluates \( f \) at different points in time, and in the last step a weighted sum of these values is calculated. The next value of
\( x \) is then calculated in a similar way to what the Euler method uses, but using the weighted sum instead of \( \dot{x} \).

While RK4 involves more computations compared with the Euler method it is also more precise, with a per step error on the order of \( h^5 \) and a total accumulated error on the order of \( h^4 \). This makes RK4 more suitable than the Euler method for scientific calculations that often require good precision.

### 2.4 Compute Unified Device Architecture

Understanding the CUDA architecture will be crucial to generating efficient simulation code that can take advantage of the GPU. This section will therefore first introduce the CUDA hardware architecture, and it will then present the programming model that will be used to write CUDA code. Most of the information in this section has been taken from the official documentation [13], which is recommended for a more thorough look at CUDA.

#### 2.4.1 The CUDA Hardware Architecture

The building block of the CUDA hardware architecture is the Streaming Multi-processor (SM). In the current architectures, NVIDIA G80 to GT200, each SM consists of eight Scalar Processors (SPs), two Special Function Units (SFUs) for transcendental functions\(^1\), one instruction unit and some on-chip memory. The entire GPU is then made up of a number of SMs, as well as some off-chip memory, see figure 2.3. This gives a scalable architecture where the performance of the GPU can be varied by having more or less SMs.

![Simplified schematic of a GPU with 96 SPs.](image)

Figure 2.3: Simplified schematic of a GPU with 96 SPs.

To be able to take advantage of this architecture a program meant to run on the GPU, known as a *kernel*, needs to be massively multi threaded. When a kernel is executed on the GPU it is divided into *thread blocks*, where each thread block contains an equal amount of threads. These thread blocks are automatically distributed among the SMs, so a programmer needs not consider

\(^1\)Such as trigonometric and exponential functions.
the amount of SMs a certain GPU has. When a SM is given one or more thread blocks to execute it divides the blocks further into warps, where each warp is a group of 32 threads. The instruction unit then selects a warp that is ready for execution, and issues the next instruction of that warp to the threads in the warp. All threads in a warp thus executes one common instruction at a time. If any threads in a warp take divergent execution paths, then each of these paths will be executed separately, and the threads will then converge again when all paths have been executed. This means that some SPs will be idle if threads in a warp diverge. It is thus important that all threads of a warp agree on an execution path for optimal performance.

This architecture is akin to the Single Instruction, Multiple Data (SIMD) architecture that vector processors use, and that most modern general-purpose CPUs have limited capabilities for too. NVIDIA call this architecture Single Instruction, Multiple Thread (SIMT) instead, the difference being that each thread can execute independently, albeit at the cost of reduced performance. It is also possible to regard each SM as a separate processor, which enables Multiple Instructions, Multiple Data (MIMD) parallelism. Using only MIMD parallelism will not make it possible to take full advantage of a GPU’s power though, since each SM is a SIMD processor.

### 2.4.2 Memory Hierarchy

As can be seen in figure 2.3 there are several different types of memory in the CUDA hardware architecture. At the lowest level each SP has a set of 32-bit registers, either 8192 or 16384 registers per SM depending on the GPU’s capabilities. These registers are shared between all threads allocated to a SM, so the number of thread blocks that a SM can have active at the same time is limited by the register usage of each thread. Accessing a register typically requires no extra clock cycles per instruction, except for some special cases where delays may occur.

Besides the registers there is also the shared memory, which is shared by all SPs in a SM. The shared memory is implemented as fast on-chip memory, and accessing the shared memory is generally as fast as accessing a register. Since the shared memory is accessible by all threads in a block it allows the threads to cooperate efficiently by giving them fast access to the same data. The amount of shared memory is quite limited, on current architectures only 16 kB per SM.

The last bit of on-chip memory is the memory labeled data cache in figure 2.3. This memory is in reality two different caches, the constant cache and the texture cache. Both of these are read-only memories that caches read-only data from the off-chip memory. [13] describes these caches in greater detail, but because they will not be used in this thesis they will not be discussed further.

The largest amount of memory is available in the form of off-chip Dynamic Random Access Memory (DRAM). The amount of off-chip memory on modern graphics cards range from several hundred megabytes to as much as four gigabytes. The DRAM memory is much slower than the on-chip memories though, with latencies of between 400 to 600 clock cycles for a memory access. This memory is also the only memory that is accessible to external devices such as the CPU. This is accomplished by allowing memory transactions over the PCIe bus that most, if not all, CUDA enabled graphics cards use to communicate with the rest of the computer system.
2.4.3 Efficiently Using Off-Chip Memory

Using the off-chip DRAM memory, as discussed in the previous section, introduces relatively large latencies. It is in almost all cases necessary to use this memory though, since it makes up the major part of all available memory and is the only memory accessible to external devices. Several techniques are therefore used to reduce, and in some cases completely eliminate, the penalties of using this memory.

The most important technique is memory coalescing. While the DRAM memory have large latencies it can access whole chunks of memory at a time. The GPU uses this to coalesce memory accesses by a half-warp, the first or second half of a warp, into one memory transaction. There are several conditions that must be fulfilled for this to be possible though. For the purpose of this thesis it is enough to know that it is possible to coalesce memory accesses if each thread in a half-warp accesses a word, where the word size can be 4, 8 or 16 bytes, and those words lie in a contiguous sequence in memory. By coalescing memory accesses and having several warps active at the time it is thus possible to mask the latencies associated with the off-chip memory somewhat.

An often used programming technique that exploits memory coalescing is to read data from the off-chip memory to the shared memory in such a way that memory coalescing can be used. The calculations are then performed using the much faster shared memory, and the result is then transferred back to the off-chip memory afterwards. This allows non-symmetric memory access patterns in the calculations, while using the benefit of memory coalescing.

2.4.4 Programming model

There currently exists two different interfaces that can be used to write CUDA programs: C for CUDA and the CUDA driver Application Programming Interface (API). The most low-level of these is the CUDA driver API, which is a lower-level API for the C programming language. It provides functions for loading and launching kernels in the form of binary or assembly code, which is usually obtained by compiling kernels written in C.

The other interface is C for CUDA, which is an extension of C. These extensions allow the programmer to specify and call kernels directly in a program’s code. The program can then be compiled with the nvcc compiler, which is a compiler front-end that simplifies the process of compiling C for CUDA programs. It supports both C and C++, but kernels are currently restricted to the C subset of C++. A mix of C, C++ and C for CUDA code can therefore be used, and nvcc then takes care to call the appropriate tools to compile the code to an executable.

C for CUDA comes with a runtime API which can be used to access CUDA specific functionality. It is an abstraction of the driver API, and is generally easier to use than the driver API. C for CUDA also supports device emulation which makes it possible to debug CUDA code that will run on a GPU, something not possible with the CUDA driver API.

The CUDA driver API offers some advantages though, in the form of better level of control and language independence, since it handles kernels in binary or assembly form. C for CUDA will be used in this thesis however, since it offers the easiest interface to generate stand-alone code for.
The rest of this section will therefore describe the C for CUDA interface, as well as some important CUDA concepts.

### 2.4.4.1 CUDA Concepts

An important concept in CUDA is the distinction between *host* and *device*. The host is what executes normal programs, and the device works as a coprocessor to the host which runs CUDA threads by instruction from the host. This typically means that a CPU is the host and a GPU is the device, but it’s also possible to debug CUDA programs by using the CPU as both host and device. The host and the device are assumed to have their own separate address spaces, the *host memory* and the *device memory*. The host can use the CUDA runtime API to control the device, for example to allocate memory on the device and to transfer memory to and from the device.

Another concept is that of a *kernel*, which has already been discussed earlier in section 2.4.1. A kernel is simply a C function that can be executed on the device. When a kernel is executed by CUDA it will be executed in parallel by multiple threads, as opposed to normal C functions which are only executed once per invocation. A kernel is defined as a normal C function, but with the specifier `__global__`, and it is executed by using a new `<<<...>>>` syntax that will be discussed further in later sections.

A call to a kernel function is also asynchronous, which means that control will immediately be returned to the host, and the host and the device will then execute independently. In the current architecture a device can only execute one kernel at a time in though, so kernels will have to wait until all previous kernels are finished. Some tasks, such as synchronous memory transfers or the `cudaThreadSynchronize` function, will wait until all other tasks are completed, and as such provide a way to synchronize the host and the device.

### 2.4.4.2 Thread Hierarchy

When a kernel is executed it is executed by a number of threads in parallel. These threads are, as discussed previously in section 2.4.1, grouped into equally shaped thread blocks. These blocks form a grid that can have either one or two dimensions, and the blocks themselves can have up to three dimensions. The threads can in this way be organised in a way that is natural for the problem to be solved. The dimensions of the grid and blocks are given as the first and second argument of the `<<<...>>>` syntax when launching a kernel. Launching a kernel named foo with 16 blocks, each of which have 4x4 threads for a total of 256 threads, would thus look like this:

```c
// dim3 is three-dimensional, unused dimensions are set to 1
dim3 block_dim(4, 4);
foo<<<16, block_dim>>>(/*kernel arguments*/);
```

When a kernel is executed by multiple threads each thread will run the same code. This means that there must be some way of distinguishing between threads to be able to do any meaningful work, otherwise each thread would do exactly the same work. For this purpose there are a couple of variables that are accessible within the kernel, that uniquely identifies the thread. The first of these is `threadIdx`, which is a 3-component vector that identifies a threads index in a block. The components can be accessed as `threadIdx.x, threadIdx.y`
and threadIdx.z. A block’s index in a grid is similarly available from the blockIdx vector, and the dimensions of a block and the grid are also available in the form of the blockDim and gridDim vectors.

For an example of these concepts, consider the process of brightening an image. This would involve increasing the value of each pixel in the image. For the purpose of this example, let the image be 256 pixels wide and 256 pixels high. The image is then divided into a grid of 16 by 16 blocks, where each block consists of 16 by 16 pixels. This may not be the most efficient way of dividing the image, but it demonstrates the discussed concepts well. The code would then look like this:

```c
// The kernel
__global__ void brighten_pixel(int image[256][256])
{
    // Compute the pixel coordinates for this thread
    int pixel_x = blockIdx.x * blockDim.x + threadIdx.x;
    int pixel_y = blockIdx.y * blockDim.y + threadIdx.y;

    image[pixel_x][pixel_y] += /* some color value */
}

int main()
{
    // Invoking the kernel
    dim3 grid_dim(16, 16); // 16x16 blocks in the grid
    dim3 block_dim(16, 16); // 16x16 threads in each block

    brighten_pixel<<<grid_dim, block_dim>>>(image);
}
```

A large number of threads will thus be launched, were each thread takes care of a single pixel. The blockIdx and blockDim variables are multiplied to compute the beginning of a block of pixels in both dimensions, and the threadIdx variable is then added to find a specific pixel in that block. Each thread thus computes the coordinates for its pixel, and then increases the pixel’s value by some amount. The number of threads spawned in this example is therefore the same as the amount of pixels in the image, 256 · 256 = 65536 threads, which would be an enormous amount of threads on a normal CPU. On a GPU it is necessary to use a large amount of threads to achieve good performance though.

### Memory

As discussed in section 2.4.2 there are several different types of memory available to a CUDA kernel, and it is important to understand how they are used. Allocation of registers is handled automatically by the CUDA compiler, and registers are thus not directly accessible to a kernel. Kernel parameters and local variables usually end up in registers though.

The shared memory is handled explicitly by the programmer, and the number of bytes of shared memory to allocate for each block is the third argument of the `<<<..>>>` syntax. The shared memory is accessible to all threads in a block, and its lifetime is also only that of a block.

There are also two memory spaces residing in the off-chip memory, the local and global memory spaces. Local kernel variables that could not fit into registers are placed in the local memory, something that is best avoided due to the
performance penalty of the off-chip memory. Accesses to the local memory are always coalesced though, because local kernel variables are by definition per-thread.

The global memory is, as the name implies, accessible by all threads in a kernel. It persists between kernel executions, and is also accessible to both the host and the device. The CUDA runtime API includes several functions that can be used by the host to manipulate the device’s global memory. `cudaMalloc` and `cudaFree` are the equivalents of the C functions `malloc` and `free`, and are used to allocate and deallocate global memory on the device. There is also the `cudaMemcpy` function for transferring data between the host memory and the device’s global memory, and `cudaMemset` which is used to fill a piece of global memory with a constant value. There are also a couple of other functions available for allocating multidimensional and pitched arrays, as well as for asynchronous memory transfers. They are described in [13], but because they are not used in this thesis they will not be covered here.

2.5 Previous work

As was discussed in section 1.5 there has been work done to automatically parallelize simulation of Modelica models, which resulted in the ModPar module for OMC. With an understanding of the CUDA architecture it is now possible to look at how the techniques used in the ModPar module may be used in this thesis.

2.5.1 Task Graph

A task graph is a Directed Acyclic Graph (DAG) where every vertex represents a task and the edges of the graph represents precedence constraints on the tasks. Each vertex and edge is also associated with a cost, where the cost of a vertex is the cost of performing the associated task, and the cost of an edge is the cost of sending data between two tasks. A task graph thus forms an internal representation that can be analysed and manipulated to detect parts of a program that can be parallelized. An example of a task graph, with costs omitted, can be seen in figure 2.4, where the assignment $x = 3\times y + \frac{z}{5}$ has been described as a task graph.

![Figure 2.4: A task graph representation of the assignment $x = 3\times y + \frac{z}{5}$.](image)
2.5.2 Task Merging

A major contribution of Aronsson’s work in [1] was the Aronsson’s Task Merging Method (ATMM), a method of merging tasks using a graph rewrite system. This method merges tasks in a task graph in such a way as to reduce the granularity of the graph while retaining as much parallelism as possible. Reducing the granularity of the task graph is desirable, since fewer vertices makes it easier for a scheduling algorithm to schedule the tasks. Merging tasks might remove some parallelism though, for example by merging all tasks into a single vertex, thereby eliminating all parallelism. A successful task merging method must thus balance these considerations well, which the ATMM method does.

2.5.3 Software Pipelining and Inline Solver

While the ModPar module developed by Aronsson was somewhat successful, it was bottlenecked by using a central solver. This meant that the calculation of the state derivatives where parallelized, but the numerical integration was done sequentially on only one processor. Lundvall [9] later improved the ModPar module by inlining the numerical integration stage into the task graph, thereby allowing that stage to also be parallelized. He also introduced software pipelining by making sure that communication only occurs between two neighbouring processors and in only one direction, which means that the sending processor can continue working even if the receiving processor falls behind.

2.5.4 Technique Evaluation

When evaluating the techniques used in ModPar it was found that using task graphs and task merging might be a good way of easily finding SIMD parallelism in models. A task graph will therefore be built from the equation system produced by OMC, and the tasks will then be merged. The MetaModelica language does not allow functions to contain state though, so the same approach as the ModPar module use will be taken. This means that a small MetaModelica module will be used to export the equations to an external C++ module, where the task merging, scheduling and code generation will take place.

The software pipelining used by Lundvall will not be used though, since it was designed to avoid communication bottlenecks and to minimize waiting in a distributed memory system. The CUDA architecture is a shared memory system though, and these problems does not really exist in CUDA since all processors have access to a global memory. Inlining the numerical solver into the task graph is not done either, since that would require barrier synchronization of all processors to work, which is not really supported by CUDA. The task execution and numerical integration steps will therefore be placed in separate kernels, since kernel calls work as implicit barrier synchronizations.
Chapter 3

Implementation

There are several different ways that OMC can be extended, and in this thesis the same way as that of the ModPar [1] module was chosen. The module was thus 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. An overview of the whole process can be seen in figure 3.1.

Modelica Model

OMC Front-end

DAELow

CudaCodegen

Equations

TaskGraph

Task graph

TaskMerger

Merged task graph

Scheduler

Scheduled tasks

CodeGen

CUDA Code

Figure 3.1: The process of compiling a Modelica model to CUDA code.
3.1 Extracting a Task Graph from OMC

Extracting a task graph from OMC involves several steps. The OMC front-end first parses a Modelica model, and the resulting intermediate representation is then passed to the OMC back-end if the compile flag to OMC is given. The relevant information is then extracted from the intermediate representation and sent to the CudaCodegen package, which is a package that was developed for this thesis. The CudaCodegen package processes this information further, and then uses the CudaCodegenExt interface to pass the information to the external C++ module that builds a task graph from this information.

3.1.1 The CudaCodegenExt Interface

The CudaCodegenExt package works as an interface between the CudaCodegen package and the C++ TaskGraph class. It uses MetaModelica’s support for external functions to export an interface to the C++ module. CudaCodegenExt contains functions for adding variables and tasks to the task graph, a function to assign variables to tasks and a function to initiate the code generation.

3.1.2 The CudaCodegen package

One of the outputs from the OMC front-end is a DAE system. In the back-end, the Main.optimizeDae function, it is translated into the DAELow form by the DAELow.lower function, and the equations of the DAE system are then sorted by the DAELow.matchingAlgorithm function. Among the outputs from the sorting algorithm are two vectors of indices, one for equations and one for variables. The equation indices represents which equation a certain variable should be solved in, while the variable indices represents which variable should be solved in which equation.

The DAELow.strongComponents function is also called here, which identifies subsystems of equations that must be solved together, i.e. algebraic loops. Algebraic loops are not handled in this thesis, so the output from this function can simply be seen as a list of equations that are sorted in the order they need to be solved in.

The DAELow, along with the index vectors and sorted equation list, is then passed to the code generation stage. This normally means the Main.simcodegen function, or the Main.modpar function if ModPar is used. In this thesis a call to the CudaCodegen.generateCUDA function is inserted into the back-end at the same place where the simcodegen and modpar functions are called. The control flow from the front-end to CudaCodegen is modelled in figure 3.2.
The `generateCUDA` function then uses the `DAELow.translateDAE` to index the variables of the DAE into different arrays depending on their type, and the `DAELow.calculateValues` function to calculate the initial values of all variables.

The `CudaCodegen.buildInits` function is then called, which adds all variables and parameters to the task graph, along with their initial values. Each equation is then added to the task graph with the `CudaCodegen.buildBlocks` function, which calls the `CudaCodegen.buildEquation` function for each equation. The `buildEquation` function uses `Exp.solve` to solve the equation for the variable indicated by the vector of variable indices, and the solved expression is then sent to the `buildExpression` function that recursively adds each component of the expression to the task graph via the `CudaCodegenExt` interface. Parts of the `buildExpression` function can be seen in code listing 3.1, where some cases have been omitted.

```plaintext
public function buildExpression
    input Exp.Exp expr;  // Input is an expression.
    output Integer taskId;  // Output is a task identifier.

algorithm
    taskId := matchcontinue(expr)  // Check the type of the expression.
    local
        Integer id1, id2, op_id;
        Integer i;
        Real r;
        Exp.ComponentRef cr;
        Exp.Exp e1, e2;
        Exp.Operator op;
        case (Exp.ICONST(i))  // An integer constant.
            equation
                // Create a new constant task.
                id1 = CudaCodegenExt.createConstTask(intString(i));
                then id1;
        case (Exp.RCONST(r))  // A floating-point constant.
            // Analogous to integer constant.
        case (Exp.CREF(componentRef = cr))  // A variable.
            equation
                // Try and look the task identifier associated to it up.
                id1 = CudaCodegenExt.getTaskId(Exp.crefStr(cr));
                then id1;
        case (Exp.CREF(componentRef = cr))  // A variable again.
            equation
                // The previous case failed, so the variable is not
                // associated to a task yet. Create a new lookup task.
                id1 = CudaCodegenExt.createLookupTask(Exp.crefStr(cr));
                then id1;
        case (Exp.BINARY(e1, op, e2))  // A binary operation.
            equation
                // Call buildExpression recursively to add the operand
                // expressions to the task graph.
                id1 = buildExpression(e1);
                id2 = buildExpression(e2);
                // Create a new binary operation task.
                op_id = CudaCodegenExt.createBinopTask(
                    Exp.binopSymbol(op),  // Convert operator to a string.
                    id1,  // The task identifier for the first operand.
                    id2);  // The task identifier for the second operand.
                then op_id;
            end matchcontinue;
    end buildExpression;
```

Listing 3.1: Parts of the `CudaCodegen.buildExpression` function.
3.1.3 The Task Graph

The building block of the task graph is the Task class that represents a single vertex in the graph. The TaskGraph class itself then maintains a list of Tasks, and has its own functions for adding new vertices and edges. The TaskGraph class also needs to keep track of information regarding variables, such as which task a variable is associated with. This is handled by an instance of the VariableTable class, which maintains a table of Variable instances.

3.1.3.1 The Task Class

The Task class contains four things: an operation type, an operator type, a cost and a list of arguments. The operation type represents the class that the task belongs to, such as variable lookup or binary function calls. The operator type then further specifies the type of the task. A binary function call might for example be an addition or a subtraction. This distinction between operation and operator is made to easier handle a large amount of different task types and to simplify the code generation.

The cost of each task represents the execution cost of the task, and is in this thesis a simple function of its operation type, see 3.2.4. The list of arguments of the task is then the tasks that the task depends on, i.e. the task is seen as a function with the arguments being the arguments to the function. The argument list is implemented as a list of task identifiers.

3.1.3.2 The Variable Class

The Variable class represents a variable in the model, and as such it contains both the internal name of the variable as used by OMC as well as the variable’s original name as specified in the Modelica model. It also keeps track of the variable’s initial value, and also which task is associated with it, if any. The associated task of a variable is the task responsible for calculating the variable’s value. Variables thus have no vertices of their own in the task graph, and associating a task with a variable means that the variable will be assigned the result of the task. Finally, each Variable has an index that determines in which array and in which position in that array the variable may be found. For example, all state variables are placed into a $x$ array in the generated code, while all algebraic variables are placed into a $y$ array. This index is implemented as its own class, the VariableIndex class, to allow it to be used as a light-weight index for the C++ map container class.

3.1.3.3 The VariableTable Class

The VariableTable class is simply a collection of Variables, and has a method Insert that can be used to insert new variables into the table. It then has Find methods to look up variables by either the name assigned to it by OMC or by the task identifier assigned to it, and an Assign methods to assign a task identifier to a variable. Finally it also has a method List that can be used to obtain a list of all variables of a certain type.
### 3.1.3.4 The TaskGraph Class

The TaskGraph class is used by the CudaCodegenExt interface to collect information about variables and to build a task graph from the equations that the OMC back-end produces. For the purpose of collecting variable information it has the method `AddVariable` to add a new variable, the `AssignVariable` method to assign a variable to a task and the `LookupVariable` method to look up the task identifier associated with a variable. Internally it uses an instance of the VariableTable class to keep track of this information.

The class also contains methods for adding new tasks to the task graph, such as the `CreateBinopTask` method for creating a new binary operation task. These methods take the type of operation along with any arguments to the operation as input. Each task is also associated with a unique identifier of the type `Task::Id`, which is simply the tasks position in the task list maintained by the TaskGraph class. This identifier is returned by the task creating methods, and used by the CudaCodegen::buildExpression to identify parts of an expression. The `CreateBinopTask` method, see listing 3.2, is for example called by `buildExpression` via the CudaCodegenExt interface, see listing 3.1.

```cpp
Task::Id TaskGraph::CreateBinopTask(const std::string &op,
    Task::Id id1, Task::Id id2)
{
    // Extract the type of the operator.
    char op_type = extract_op_type1(op);
    BinaryOperator::type type;

    // Check which type the operator is.
    switch(op_type)
    {
        case '+': type = BinaryOperator::ADD; break;
        case '-': type = BinaryOperator::SUB; break;
        case '*': type = BinaryOperator::MUL; break;
        case '/': type = BinaryOperator::DIV; break;
        case '^': type = BinaryOperator::POW; break;
        default:
            std::cerr << "Unknown binary operation: "
                << op_type << std::endl;
            error = true;
            return -1;
            break;
    };

    // Add a new binary operation task to the task graph.
    int task_id = AddVertex(Operation::BINARY, type);
    // Connect the operators to the newly created task.
    AddEdge(task_id, id1);
    AddEdge(task_id, id2);

    // Return the task identifier for the new binary operation task.
    return task_id;
}
```

Listing 3.2: The TaskGraph::CreateBinopTask method

Finally it also contains the ExitCuda method that is called as the last step in the CudaCodegen package. This method instantiate the TaskMerger, Scheduler and CodeGenerator classes, and uses those instances to generate the target code from the task graph.

26
3.2 Task merging

The task merging phase begins by instantiating the TaskMerger class, giving the instance a reference to the task list maintained by TaskGraph. The TaskMerger instance builds a new graph out of TaskMergerNode instances, which is more suitable for task merging. The TaskMerger instance then applies graph merging algorithms on the graph until it is no longer possible to merge any more tasks.

3.2.1 The TaskMergerNode Class

This class represents a node in a task graph that can contain several tasks. When first instantiated it only contains one task, so the graph built by TaskMerger is initially identical in layout to the one built by TaskGraph. The TaskMergerNode class keeps track of both child nodes and parent nodes though, where the Task class only keeps track of the identifiers of the tasks it depends on. This means that the task graph built by TaskMerger is bidirectional and more suitable for task merging compared with the task graph built by TaskGraph, which is only a list of loosely associated tasks.

The TaskMergerNode class contains several important methods, such as the Clone, DetachFromGraph and Merge methods. The Clone method creates a clone of a TaskMergerNode instance, while DetachFromGraph does the opposite by removing a TaskMergerNode instance from the graph. The Merge method merges two nodes, which is done by copying the tasks, child nodes and parent nodes to one of the nodes, taking special care if the two nodes are directly related to each other. Besides these methods there are also methods for adding and removing children and parents from a node.

The TaskMergerNode also contains the methods Cost and Level. The Cost method returns the total cost of all tasks in a node, while the Level method returns the level of the node. The level of a node is simply defined as the level of the parent with the largest level, in addition to the cost of communication between the node and that parent. Both the cost and level of a node are calculated only when needed, to avoid unnecessary work. This is done by caching the cost and level of a node, and setting a flag that they need to be recalculated whenever the graph is changed.

3.2.2 The TaskMerger Class

The TaskMerger class is the class that does the real task merging work. As previously mentioned it begins by building a new task graph out of TaskMergerNode instances, and the Merge method then merges the tasks and returns a list of nodes. The Merge method works by saving the number of nodes in the graph, and then applying the merge algorithms. If the number of nodes in the graph is unchanged it stops, otherwise it continues to apply the merge algorithms in this way until the graph changes no more. When that happens it simply returns the merged graph, which consists of a list of TaskMergerNodes where each node may contain one or more tasks.
3.2.3 Merging Algorithms

In his work, Aronsson [1] defines the ATMM, a method for merging tasks. While this method has been proven to work well it was deemed too complex to implement in this thesis, so a somewhat simpler set of algorithms were used instead. These algorithms focus on merging tasks in such a way that any eventual SIMD parallelism in the models can be easily exploited.

3.2.3.1 ConstantDuplicate

If a node has no children and only one task in the initial graph, then it must represent either a constant or a state variable, which means that the task’s value is constant from the point of view of the task graph. If this was not true, then it would not be possible to calculate the value of the node’s parent, since the node’s value would be unknown. Cloning such a node so that each of its parents get its own copy of the node would then separate parts of the task graph without imposing any further cost. This is therefore done in the ConstantDuplicate algorithm, which is illustrated in figure 3.3.

![Figure 3.3: The ConstantDuplicate algorithm.](image)

3.2.3.2 SingleChildMerge

The SingleChildMerge algorithm is one of the algorithms presented by Aronsson [1], and is illustrated in figure 3.4. It checks if any node has a single parent, and if that parent has a single child. If that is the case, then no parallelism will be lost by merging the nodes. Merging such nodes will also reduce the overall number of nodes in the graph, which reduces the work needed to be done by the other algorithms.

![Figure 3.4: The SingleChildMerge algorithm.](image)
3.2.3.3 ChildMerge

This algorithm is almost the same as the SingleChildMerge algorithm, except that the parent of the node can have several children, see figure 3.5. The algorithm checks that a node only has a single parent, and that the execution cost of the node is less than the communication cost between the node and its parent. If that is the case, then it probably is best to schedule both nodes on the same processor, so the nodes are merged. This might not always be the case though, but it works well enough for the purpose of this thesis.

![Figure 3.5: The ChildMerge algorithm.](image)

3.2.3.4 DuplicateChild

This algorithm is similar to the DuplicateConstant algorithm. It checks if a node is childless and if it has more than one parent. If that is the case, and the node’s execution cost is less than the cost of communicating the result of the node to each of its parents, then the node is cloned so that each parent get its own copy of the node. This means that work is duplicated whenever it is cheaper to do the work multiple times than to send the result to everyone that depends on it. DuplicateChild is illustrated in figure 3.6, which is the same figure as figure 3.3 for DuplicateConstant. The reason for this is that they do the same operation, but for different preconditions.

![Figure 3.6: The DuplicateChild algorithm.](image)

3.2.4 Approximating Costs

Some of the task merging algorithms use the cost of executing tasks and for communication between nodes. For an optimal result these should be as exact as possible, but the only way to determine the exact costs is to actually run tests
on the hardware that is used. However, in this thesis the actual task merging is not the focus, but finding SIMD parallelism that can be exploited. It is therefore enough to use crude approximations to the real costs.

The cost of unary and binary operations is therefore approximated as 1 in this thesis, while special functions, such as trigonometric functions, are approximated with the cost 4. This reflects the fact that a SM has eight SPs but only two SFUs. The cost of communication is set to 100, which reflects the latencies introduced by the global memory.

### 3.3 Scheduling

When the task graph has been merged it is necessary to determine in which order the tasks should be executed, and whether they should be executed in parallel on different SMs or not. This is done in two steps. First the nodes in the merged task graph is scheduled with the critical path algorithm, and then the tasks in each node are scheduled. The scheduler also tries to find nodes that are operationally equivalent to other nodes, and schedule those nodes to be executed in parallel on the same SMs.

#### 3.3.1 Critical Path Algorithm

The critical path algorithm was chosen because it is well known and relatively simple to implement. It works by selecting the critical path in a graph, which is the longest execution path with respect to total execution time, and scheduling all nodes on that path on the same processor. To find this path it therefore starts by finding the node with the highest level, since that is the task that will be executed last of all nodes. It then continues by examining that node’s parents, and selects the parent with the highest level. In this way it goes through the graph until it finds a node with no parents, and the path that it has taken is then the critical path.

An example of this algorithm can be seen in figure 3.7, where the bold path is the critical path, the numbers in the nodes the execution cost for the nodes and the numbers on the edges the communication costs. The critical path thus has the cost $4 + 3 + 2 + 6 + 6 = 21$.

![Figure 3.7: An example of the critical path algorithm.](image)

#### 3.3.2 Scheduling tasks

When the nodes have been scheduled it is necessary to also schedule the tasks inside the nodes. This is done by the `Scheduler::ScheduleTasks` method, which
takes a TaskMergerNode instance and returns a scheduled list of Task::Id. The scheduling algorithm first adds all tasks that has no dependencies that belong to the same node to a First In, First Out (FIFO) queue. As long as this queue is not empty the algorithm then removes the first task from the queue and adds it to the schedule. It then inspects all nodes that depend on that task and adds those to the queue, given that they have not already been scheduled and that all their dependencies have been scheduled. This makes sure that all tasks are scheduled, and that they are scheduled so that their dependencies are executed first. See figure 3.8 for an example of the task scheduling algorithm.

![Figure 3.8: An example of the task scheduling algorithm.](image)

### 3.3.3 Finding Equivalent Nodes

To be able to use the GPU in an optimal way it is necessary to use SIMD parallelism, as discussed in section 2.4.1. Since SIMD works by executing the same instructions on different data it is necessary to find collections of tasks that execute the same instructions. Finding subgraphs of the task graph that are equal in that way would be a computationally expensive problem, since it would require comparing all possible subgraphs of the task graph to each other. By observing that tasks that can be executed in parallel most likely end up in different nodes it is possible to construct an algorithm that finds tasks that can be executed in a SIMD fashion, although not all such cases.

The Scheduler::FindEquivalentNodes method takes a TaskMergerNode instance, and returns a list of nodes that can be executed with SIMD. It compares the node to all other nodes, and immediately discards those nodes that do not contain the same amount of tasks as the regarded node. If two nodes do not contain the same amount of tasks it is not possible for them to be operationally equal. If the method finds a task with the same number of tasks it compares the tasks of the two nodes. This is done by comparing tasks at the same position in both nodes, i.e. first the first tasks in each node is compared, then the second tasks, etc. If any pair of tasks does not have the same operation and operator type it is deemed not equal, otherwise they are equal.

It might not be obvious why this is a sufficient test of operational equality, but it can be seen by considering that nodes represent subgraphs of the task graph. If all tasks in the subgraph have the same type, then they must also have an equal set of dependencies. If any task in a subgraph would have a different set of dependencies compared with the equivalent task in another subgraph, either
by having a different number of dependencies or dependencies of different types, then those dependencies would differ in type between the subgraphs. If all pairs of equivalent tasks in two subgraphs have the same type, they must thus have an equal set of dependencies, and the two subgraphs must then have the same layout. It is therefore both necessary and sufficient to compare the type of all tasks in two nodes to determine whether they are operationally equivalent or not.

This is not enough though, because it is also necessary to check that all nodes that are to be executed by SIMD use disjoint variable sets. This is not a limitation of SIMD itself, but a consequence of how the shared memory is used in the generated code, see section 3.4.3. Whenever an operationally equivalent node is found its variables are therefore compared with any other nodes already found, and if any variables used by the node is already used by another node then the node is deemed not equivalent.

The FindEquivalentNodes method thus finds nodes that can be executed together in a SIMD fashion, which allows several nodes to be executed for the same cost as one node. However, because each SM in the GPU has only eight SPs in the current architectures, this means that only eight nodes can be executed simultaneously on a SM. It is therefore best to not execute all equivalent nodes on the same processor if there are many equivalent nodes, so the algorithm terminates when it has found a certain number of nodes. A limit of 16 or 32 nodes have been found to produce good results in most cases, but the optimal limit depends on the model. It is also a rather hard problem to find an optimal limit due to the complexity of the CUDA architecture, so in the current implementation the upper limit is set manually.

3.3.4 The Schedule Class

The Schedule class represents a complete schedule. It contains execution paths, where an execution path is a list of tasks executed in order. Several execution paths are collected into an execution path list, which represents SIMD execution. A list of execution path lists then form the schedule for a single processor, and the complete schedule is then a list of processor schedules. An example of a schedule can be seen in figure 3.9.

![Figure 3.9: An example of a schedule for two processors.](image)

The Schedule class then has the method NewPathList that inserts a new
execution path list into a processor schedule, and InsertPath that inserts an execution path into the last created execution path list. It also keeps track of how many processors that are in use, and the maximum number of threads used, which the code generator needs to know.

3.3.5 Scheduling Algorithm

The complete scheduling algorithm begins with a list of TaskMergerNode instances and a list of processors. As long as the node list is not empty it then uses the critical path algorithm to find the critical path. It calculates the execution cost of the critical path, and schedules those nodes to the processor with the least amount of work already allocated to it. In addition to this it also uses the FindEquivalentNodes method to find any equivalent nodes to the nodes on the critical path, and schedules those nodes to be executed on the same processor in a SIMD fashion. The nodes on the critical path, and those that have been found to be operationally equivalent, are then removed from the graph and scheduled with the ScheduleTasks method. The execution paths that ScheduleTasks returns are inserted into an instance of the Schedule class. The algorithm then continues by finding the new critical path in the graph and scheduling it, until there are no more nodes left to schedule.

3.3.6 Communication Scheduling

If all models were embarrassingly parallel, then the scheduling would be complete now. This is obviously not the case though, so communication between processors is unfortunately necessary in many cases. The scheduling algorithm must therefore inspect each node and determine if any of the tasks have a dependency that is scheduled on another processor. If any such dependencies are found, then it will be necessary for the simulation to wait until that processor has calculated the dependency and sent it to the processor that needs it. The scheduler therefore inserts signals and locks into the schedule, and also determines what data needs to be sent where. This information is inserted into the schedule as special execution paths. Such an execution path can either contain signals, locks, variables to send or variables to receive, and the first task identifier in the path marks its type. How the communication is implemented is then for the code generator to decide, which is covered in section 3.4.4.2.

3.4 Code generation

When the tasks have been scheduled by the scheduler it is time to generate the final code. This is done by the CodeGenerator class, with help from the CodeWriter, SharedMemoryAllocator and ThreadIdIndexer classes.

3.4.1 The CodeWriter Class

The CodeWriter class is used by the CodeGenerator class to automatically format and output code. It encapsulates an std::ostream, which makes it possible to easily redirect the output, for example to the standard output for debugging purposes. It also contains numerous methods for outputting different
types of code segments, such as variable assignments, function calls and much more. CodeWriter also keeps track of the indentation level of the generated code automatically. This makes the CodeGenerator implementation cleaner, since it can focus on generating the structure of the code without concerning itself with how the code should be formatted.

3.4.2 The SharedMemoryAllocator Class

As was previously discussed in section 2.4.3 it is very important to use the off-chip memory efficiently. All variables that are used by a SM are therefore copied to the shared memory before they are used, and the results are then copied back to the global memory. When the code generator is processing a processor schedule it then uses an instance of the SharedMemoryAllocator to allocate each variable it comes across with the SharedMemoryAllocator::Allocate method. This method takes a Variable and maps it to an element of the shared memory, thus returning an index to a shared memory array that the code generator can use.

The class is also responsible for outputting the allocation and copy-back functions that copies the variables to and from the shared memory. The copy-back function will simply be a list of assignments, where the derivatives and algebraic variables are copied back to the global memory. The allocation function can be a bit more efficient though. It starts with one variable, and then collects all variables that has an index no more than 16 larger than that variable. It can then issue a coalesced memory read that reads 16 variables at a time, the size of a coalesced read of 32-bit variables, and then move those variables to were they should be in the shared memory. It does not matter if not all 16 variables are used, since it costs as much to read one variable as it does to read 16 variables. Figure 3.10 shows an example of this, where a number of state variables from the global array \( x \) are copied to a shared memory array \( s \). \( s_{\text{tmp}} \) is then the temporary shared memory array that is used to store the variables before they are copied to their respective places in \( s \).

![Figure 3.10: Copying variables from the global to the shared memory.](image)

This makes it possible to take advantage of coalesced memory reads even if the model itself does not access memory in such a pattern. This also makes it possible to arrange the memory in such a way that SIMD parallelism can be implemented. Unfortunately it is not possible for the copy-back function to use the same technique, since it would risk overwriting other variables. The amount of data copied by the copy-back function is generally much smaller than the
allocation function though, so the impact of not using memory coalescing in the copy-back function has been found to be mostly negligible.

3.4.3 The ThreadIdIndexer Class

The ThreadIdIndexer is the class that is responsible for making SIMD work. Because the execution paths in an execution path list are operationally equivalent they will contain the same number of variables, and the variables will be accessed in the same way. If the first path’s first variable get an index of 5 in the shared memory array, and each path has 4 variables, then the second path’s first variable will have index $5 + 4 = 9$, and so on. So if $id$ represents the thread id of the thread executing one of the execution paths, then a variable can be accessed as $s[base + offset * id]$.

ThreadIdIndexer therefore takes an execution path list, a table of variables and a SharedMemoryAllocator, and creates a mapping from the variables of the first execution path to a base-offset pair. It begins by adding all variables in the first execution path to the shared memory allocator. This ensures that all those variables get adjacent indices in the shared memory array. It then iterates through the two first execution paths simultaneously, and for each pair of tasks that represents a variable it looks up the indices in the shared memory allocator. The index of the task belonging to the first execution path is saved as the base index for the variable associated with the task, and the difference between the two indices is saved as the offset. When the code generator generates code for an execution path list with more than one execution path it can then use the ThreadIdIndexer to look up the base and offset for the variables in the first execution path. By using these indices it is then possible for the code generator to generate code that can be executed by the multiple threads. This means that the execution paths must have disjoint variable sets though, since the algorithm will not work if several variables have the same shared memory index.

3.4.4 The CodeGenerator Class

The CodeGenerator class is the class that is responsible for generating the CUDA code. It takes a task schedule, a list of tasks and a list of variables, and outputs the final code. It begins by instantiating a CodeWriter that is used for all code output, and then uses that instance to output a header containing some necessary includes and some timing functions used to measure the simulation time. Next it outputs the tasks in the task schedule, which makes up the largest part of the generated simulation code, and the initial values and names for all variables. It then outputs the RK4 integration functions along with a function that outputs the result to a data file, and finally the main function of the simulation is generated.

3.4.4.1 Generating Task Code

The CodeGenerator::EmitTasks method is responsible for the largest amount of work in the code generator, which is to generate the code for all tasks in the schedule. This is done by iterating through the schedule one processor at a time, and for each new processor a new SharedMemoryAllocator is instantiated. The shared memory allocator then outputs the definitions for the allocation and
copy-back functions, and the code generator then starts constructing a function that will execute the tasks for the current processor. The first thing generated for the task function is a definition of the shared memory array that will be used, and then a call to the allocation function that the shared memory allocator will generate is inserted.

The code generator then starts generating code for the tasks in the processor schedule by iterating through the execution path lists in that schedule. If an execution path list only contain one execution path, then the first task is inspected to see if it is a special execution path for communication. If the first task is one of the special task identifiers that signals a lock or signal list, then the rest of the list is a list of locks or signals that need to be generated. If it instead indicates that it is a receive or send list, then the rest of the list is a list of variables that need to be either sent or received. How the communication is implemented is covered in section 3.4.4.2.

If the execution path list instead is a normal execution path list, then the code generator checks how many execution paths it contains. More than one execution path means that they should be executed in a SIMD fashion, so a ThreadIdIndexer is instanced in that case, and all variables in the execution path list are added to the shared memory allocator to make sure that memory is reserved for them in the shared memory array. Since SIMD means that the same code is executed by multiple threads it is then only necessary to generate code for the first execution path, regardless of how many execution paths there are in the execution path list. It is also necessary to make sure that the code is only executed by the exact amount of threads that it should be executed by, so it is wrapped in an if-statement that check that the thread id is less than the number of threads that should be used.

The actual code generation for a single task is done by the EmitTask method. It begins by checking the operation type of the task, and calling other methods such as EmitBinop method that generates code for a binary operator. These help methods then check the operation type of the task, and returns the generated code for the operation. The EmitBinop method can be seen in listing 3.3.

```cpp
std::string CodeGenerator::EmitBinop(int op, const arg_list &args) {
    // Make sure that the binary operation only has two arguments.
    assert(args.size() == 2 &&
        "Invalid number of arguments to binary operation");

    std::string op_code;
    switch(op) // Generate code for the operator.
    {
    case BinaryOperator::ADD: op_code = " + "; break;
    case BinaryOperator::SUB: op_code = " - "; break;
    case BinaryOperator::MUL: op_code = " * "; break;
    case BinaryOperator::DIV: op_code = " / "; break;
    default: assert(false && "Unknown binary operator"); break;
    }

    // Look up the names of the variables that store the operands,
    // and concatenate them with the operator.
    return LookupVariableCode(args[0]) + op_code
        + LookupVariableCode(args[1]);
}
```

Listing 3.3: The CodeGenerator::EmitBinop method.
The `LookupVariableCode` is used by these methods to look up task arguments, which begins by checking if the argument happens to be the special variable `time`, in which case it simply returns `t`. Otherwise it checks if a thread indexer has been instantiated, and if so it uses the thread indexer to look up the correct code for the argument. If the code for the argument still has not been found it checks if there is a temporary variable associated with the argument, and if so it returns the temporary variables name. The final option is to use the shared memory allocator to look up the argument, which should always work if all other options have failed.

When the code for executing a task has been created it is necessary to store the result somewhere, and `EmitTask` therefore continues by checking if a variable has been assigned to the task. If so, it generates code to assign the variable to the result, again using `LookupVariableCode` to find the code for the variable. If no variable has been assigned, then the task is part of a larger expression and a temporary variable is allocated with `AllocateTemporaryVariable` and assigned the result. The temporary variables are simply named `tmp#`, where `#` is a unique number. A somewhat simplified version of the `EmitTask` method can be seen in listing 3.4.

```cpp
void CodeGenerator::EmitTask(Writer &o, Task::Id task_id)
{
  const Task &task = tasks[task_id];
  int task_op = task.OperatorType();
  const arg_list &task_args = task.Arguments();

  std::string result;
  switch(task.OperationType())
  {
    case Operation::LOOKUP:
      return; // There is nothing to do with a lookup task.
      break;
    case Operation::UNARY:
      result = EmitUnop(task_op, task_args);
      break;
    case Operation::BINARY:
      result = EmitBinop(task_op, task_args);
      break;
    case Operation::FUNCALL:
      result = EmitFunction(task_op, task_args);
      break;
    default:
      assert(false && "Unknown operator type");
      break;
  }

  // If this is not a lookup task we need to assign the result
  // to some variable.
  if(task.OperationType() != Operation::LOOKUP)
  {
    // Assign the result to an existing variable if an existing
    // variable is bound to this task, otherwise create a new
    // temporary variable.
    Variable var;
    if(variables.Find(task_id, var))
    {
      // Use the code writer to generate an assignment.
      o.Assign(LookupVariableCode(var), result);
    }
  }
```
else
{
   // Use the code writer to define a new variable.
   o.Define("real", AllocateTemporaryVariable(task_id), result);
}
}

Listing 3.4: The CodeGenerator::EmitTask method

A call to the copy-back function is then inserted when code for all tasks has
been generated. Since this function does not use memory coalescing it is only
executed by one thread, so it is wrapped in an if-statement that check that the
thread id is 0. This is the last thing done in a task execution function, and
the shared memory allocator is then used to generate the allocation and copy-
back functions. These functions can only be generated after the task execution
functions, since the shared memory is allocated while generating code for the
tasks. The copy-back function is as was previously said very simple, being only a
list of assignments. The allocation function is slightly more complex, and begins
by allocating a small shared memory array that will be used to temporarily hold
the values read from the global memory. It then issues a memory read to that
array that all threads will execute, and then uses one thread to move the memory
from the temporary array to the larger shared memory array that will be used
when executing the tasks. It continues reading variables in this fashion until all
variables used by the processor has been read. Any variables that could not be
coalesced with other variables are read separately at the end of the function.

Because CUDA executes a single kernel at a time it is necessary to have a
kernel that distributes the work. This is done by the generated execute_tasks
kernel, that executes the different task execution functions depending on the
block id. This means that each task execution function is executed as a single
block of threads, and they will thus be distributed to the SMs of the GPU
automatically.

There is thus quite a lot going on in the task code generation, so a look at
how the generated code looks might be instructional. In listing 3.5 below is a
code segment taken from the code generated from the TestModel model, which
will be described in section 4.1.1. Something to note about the code is that it
does not use the C types float or double, but the type real. real is simply
a type declared with typedef, making it easy to change between float and
double.

// The task execution function.
_device_ void execute_tasks_3(real *dx, real *x, real *y, real *c,
   bool *l, real t)
{
   int id = threadIdx.x;
   extern _shared_ real s[]; // Allocate the shared memory
   allocate_3(s, dx, x, y); // Call the allocation function

   // This is the code that does actual work, and it will be
   // executed by 20 threads.
   if(threadIdx.x < 20)
   {
      // The thread id is here used to index into the
      // shared memory array.
      real tmp0 = s[1 + 4 * id] + s[0 + 4 * id];
      real tmp1 = -2 * s[2 + 4 * id];
real tmp2 = tmp1 + tmp0;
real tmp3 = powf(64, 2);
s[3 + 4 * id] = tmp3 * tmp2;
}

// Call the copy-back function, and make sure that it is only
// executed by a single thread.
if(threadIdx.x == 0)
{
    copy_back_3(s, dx, x, y);
}

_device_ void allocate_3(real *s, real *dx, real *x, real *y)
{
    // Allocate a temporary shared memory array.
    real *s_temp = &s[80];
    // Copy static variables into the temporary array.
    s_temp[threadIdx.x] = x[2 + threadIdx.x];

    // Use one thread to move the data from the temporary array
    // to the correct places in the shared memory array.
    if(threadIdx.x == 0)
    {
        s[0] = s_temp[0];
        s[2] = s_temp[1];
        s[4] = s_temp[2];
        ...
        s[21] = s_temp[17];
        s[24] = s_temp[18];
        s[26] = s_temp[19];
    }

    // Continue reading more variables.
    s_temp[threadIdx.x] = x[22 + threadIdx.x];
    if(threadIdx.x == 0)
    {
        ...
    }

    // Only one algebraic variable is used, so it is read directly.
    if(threadIdx.x == 0)
    {
        s[78] = y[1];
    }

_device_ void copy_back_3(real *s, real *dx, real *x, real *y)
{
    // Copy back all derivatives and algebraic variables.
    dx[3] = s[3];
    dx[6] = s[7];
    dx[9] = s[11];
    ...
    dx[57] = s[75];
    dx[60] = s[79];
    y[1] = s[78];
}

// The kernel that calls the task execution functions.
_global_ void execute_tasks(real *dx, real *x, real *y, real *c,
bool *l, real t)
// Determine which function call based on the block id.
switch(blockIdx.x)
{
    case 0: execute_tasks_0(dx, x, y, c, l, t); break;
    case 1: execute_tasks_1(dx, x, y, c, l, t); break;
    ...
}

Listing 3.5: Part of the code generated from the TestModel model.

3.4.4.2 Generating Communication Code

C for CUDA has primitives for synchronizing threads in a block, but no support for synchronizing thread blocks. This is because the CUDA architecture is built for SIMD parallelism, and all threads in a kernel should optimally execute the same instructions. This is however not always the case when Modelica models are simulated, and MIMD parallelism is sometimes necessary. As was discussed in section 2.4.1 it is possible to use MIMD parallelism by regarding each SM as a separate processor, which is used by placing each task execution function in its own thread block. However, in some cases it becomes necessary to communicate between task execution functions if any parallelism is to be used. The lack of synchronization primitives for blocks then becomes a problem.

This problem is solved in this thesis by using spin locks that operate on global memory, since that is the only writable memory accessible to all blocks. A vector of booleans, called \( l \), is therefore allocated in the global memory, along with a vector of floating point numbers, called \( c \). If one task execution function needs values from another task execution function, then a lock is placed in the code, which might look like the following:

```c
while (!l[5])
{
    c[0]++;
}
```

The lock consists of a while-statement that loops until an element of the \( l \) array becomes true. Each lock gets an element allocated in the \( l \) array, so \( l[5] \) in the code above means that it is the sixth lock. For each iteration it increments the first element in the \( c \) array with one. The incrementation has no practical purpose other than keeping the CUDA compiler from removing the otherwise empty loop when optimizing the code. It was found that the only way to avoid this optimization was to do something that produces effects outside the function, in this case changing an element of the global memory. The first element of the \( c \) vector is therefore allocated for this purpose.

When a task execution function has calculated values that another task execution function needs, it then moves the values to their allocated places in the \( c \) array. It then signals the other task execution function that the values are ready to be used by setting the corresponding lock to true. The other task execution function can then fetch the values from the \( c \) array and use them. After each call to the `execute_tasks` kernel it is then necessary to reset all variables in the \( l \) array to false with a call to `cudaMemset`. This call is omitted if no communication is used in the simulation though.

40
This is a rather inefficient way of doing communication, since the global memory introduces relatively large latencies, but the CUDA architecture is not really constructed for block-to-block communication. There is also no way of knowing which block is executed by which SM, or in which order they are executed. This means that this kind of communication limits the number of blocks to the number of SMs available, since there is a risk for deadlocks otherwise. Block-to-block communication is therefore best avoided, and rarely needed for models that are interesting to simulate on GPUs anyway.

3.4.4.3 Integration Functions

The RK4 numerical integration is implemented as three kernels that accumulate the \( k \) values used by RK4, see section 2.3. They also updates the state variables in the \( x \) vector according to the RK4 method. The first of these kernels is implemented as follows:

```c
__global__ void step_and_increment1(real *x, real *old_x, real *d_x, real *k, real h) {
    const int id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id < STATES)
    {
        real dx = d_x[id];
        k[id] = dx;
        x[id] += h * dx;
    }
}
```

It begins by calculating the thread id, and checks that the id is smaller than the number of states. These kernels are executed by several blocks with multiple threads, to efficiently use the GPU, so if the number of states is not an exact multiple of the block size there might be more threads than states executing. The derivative that will be used is then copied to a local variable, to avoid reading from the global memory more than necessary. The \( k \) value is then set to the derivative, since it is the first step, and the state variable is updated according to RK4. Both the second and third kernel then does the same operations:

```c
real dx = d_x[id];
k[id] += 2.0 * dx;
x[id] = old_x[id] + h * dx;
```

They thus accumulate the \( k \) values, and then use the old values of the state variables to calculate the new values. The fourth and final kernel then uses the \( k \) values and the old values of the state variables to calculate the new values of the states variables. It looks like the following:

```c
real new_x = old_x[id] + h * (d_x[id] + k[id]);
x[id] = new_x;
old_x[id] = new_x;
```

\( k[id] \) will contain the first three \( k \) values, and \( d_x[id] \) the final \( k \) value. So \( k[id] \) and \( d_x[id] \) are added together to form the addition of the \( k \) values in the final step of RK4, and the new value of the state value can then be calculated. The state variable is then updated, and the value is also saved in the \( old_x \) array that is used to preserve the old state variable values during the calculations.
3.4.4.4 Result Output

The result from the simulation, the values of all variables in each time step, is output to a data file that can be plotted. A function that does this is therefore also generated, simply called `output`. It loops through a list of variables, and outputs the name of each variable followed by two columns. The first column is the time for each time step, while the second column is the variables value in each time step.

3.4.4.5 The Main Function

The main function is the last function that is generated, and the first thing that is generated is the allocation of some variables. These include `t`, the current time, `h`, the size of each time step, and `steps`, the number of time steps simulated. `half_h` and `h_div_6` are also defined, representing a half time step and one sixth of a time step.

A number of device variables are then defined. `d_x`, `d_dx` and `d_y` are arrays used for the state, derivative and algebraic variables. `d_old_x` is used to save old state variable values and `d_k` to accumulate the `k` values, both used in RK4. `d_l` and `d_c` are then used for communication, see 3.4.4.2. Memory for these arrays is allocated on the device by the host with `cudaMalloc`, and the initial variable values are then copied to `d_x`, `d_dx`, `d_y` and also `d_old_x`. Memory for saving the results from the simulation is also allocated on the host.

When all host and device variables have been allocated it is time to generate the simulation loop, that will call the kernels and perform the whole simulation. Calls to the task execution kernel and the integration kernels are interleaved, and the time is incremented according to RK4. The results from each time step are then copied from the device to the host and saved. An example of a generated simulation loop can be seen in listing 3.6.

```c
// Determine the size of the shared memory needed.
int shmemb_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 the first integration kernel.
    execute_tasks<<<7, 20, shmemb_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, shmemb_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, shmemb_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, shmemb_size>>>(d_dx, d_x, d_y, d_c, d_l, t);
```
Listing 3.6: An example of a simulation loop, generated from the TestModel model.

Three kernel arguments are given to `execute_tasks`, which is the number of blocks, the number of threads per block and the amount of shared memory to allocate. These numbers are determined by the scheduler and shared memory allocator. The integration kernels do not use any shared memory, so only the number of blocks and threads are given to them. These values are determined by always using 32 threads per block, the number of threads in a warp, and dividing the number of state variables by 32 and rounding up to get the number of blocks.

The final code that is generated is the end of the main function. Code to open a file and output the results from the simulation with the `output` function is generated, and `cudaFree` and `free` is finally used to free all memory allocated during the simulation.

3.5 RK4 Solver for OMC

Support for the RK4 method was also implemented in OMC, so that the performance of the CUDA simulation code can be compared to the normal simulation code for the CPU. This was done by implementing a RK4 solver in the runtime library used by the simulation code normally generated by OMC. This solver was based on the already existing Euler solver, using the same structure as the CUDA code. The simulation loop can be seen in listing 3.7.

```c
    double sim_time = start;
    for (double sim_time = start; sim_time <= stop; sim_time += step) {
        functionODE();
        step_and_increment1(globalData, k, half_step);
        globalData->timeValue += half_step;
        functionODE();
        step_and_increment2(globalData, k, half_step);
        functionODE();
        step_and_increment3(globalData, k, step);
        globalData->timeValue += half_step;
        functionODE();
        step_and_integrate(globalData, k, step / 6.0);
        functionDAE_output();
        emit();
    }
```

Listing 3.7: The simulation loop for the RK4 solver implemented for OMC.

The `functionODE` corresponds to the `execute_tasks` kernel used in listing 3.6, and the `functionDAE_output` calculates output variables. Output variables are
not really handled in the CUDA code though, and they are simply added to the task graph and calculated in `execute_tasks`. This means that they are calculated unnecessarily often by the CUDA code, but the amount of output variables is in general quite small. The `emit` function then saves the calculated values, and corresponds to the `cudaMemcpy` calls in listing 3.6.

The RK4 step functions are then implemented in much the same way as for the CUDA simulation code. The `step_and_increment1` function is for example implemented as:

```c
void step_and_increment1(DATA *data, double *k, double step)
{
    for(int i = 0; i < data->nStates; ++i)
        
        k[i] = data->statesDerivatives[i];
        data->states[i] = data->oldStates[i] +
            step * data->statesDerivatives[i];
}
```

The `DATA` structure contains all variables in the simulation, as well as the state variable values from the previous step. The difference between the step kernels in the CUDA code and these step functions are that the kernels operate on only one element, but are executed in parallel by multiple threads. The step functions instead iterates through all state variables. In the end they both do the same operations though.

The `main` function, in the `c_runtime/simulation_runtime.cpp` file, was then extended so that it is possible to specify that the simulation should use the RK4 method instead of DASSL or Euler. It is then possible to change the integration method by editing the `ModelName_init.txt` file created by OMC.
Chapter 4

Results

To be able to evaluate the performance of the generated CUDA code some models will be needed, and the models that were chosen for this thesis will be presented in the first section of this chapter. The second section then contains some information about the hardware that was used to simulate the models. Measurements taken during the simulations are then presented in the last section.

4.1 Models

To be able to evaluate the performance of the generated simulation code for CUDA it is necessary to have some models that can be simulated and measured. Three models have been chosen for this purpose. They were chosen because of their parallel nature and for their ability to be easily scaled in size. They also only use basic features of Modelica, since implementing more advanced Modelica features would have been outside the scope of this thesis.

4.1.1 TestModel

This model is taken from [10], and it models an electrical circuit. The circuit consists of a series of resistors and capacitors connected to each other in the way shown in figure 4.1. The code for this model can be seen in listing 4.1.

![Circuit of resistors and capacitors.](image)
model TestModel
  parameter Integer N = 4;
  input Real inputVars[1](start = 1.0);
  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 TestModel;

Listing 4.1: Modelica code for the TestModel model.

By changing the $N$ parameter in the model is is possible to change the size of the circuit, by adding more resistors and capacitors.

4.1.2 WaveEquationSample

The WaveEquationSample model models pressure dynamics in 1D ducts, and the original version can be found in [6]. It is also included in the OpenModelica test suite that accompanies the OMC source code. The version of the model used in this thesis has been simplified somewhat due to the use of functions in the original version, a Modelica feature not implemented in this thesis. As with the TestModel model it is possible to change the size of the model by setting a parameter $n$ that determines how many sections the duct should be divided in. The code for this model can be found in listing 4.2.

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 4.2: Modelica code for the WaveEquationSample model.
### 4.1.3 HeatedPlate2D

The last model used in this thesis, HeatedPlate2D, is taken from [1]. It models the temperature on a two-dimensional thermal plate, and the code for this model can be found in listing 4.3. This model was abandoned by Aronsson because it did not involve enough computations per state variable to give any speedup with his parallelization scheme. This had to do with high communication costs, which should be less of a problem in this implementation. As with the other models it is possible to change the size of the model by changing a parameter, in this case \( n \). A change in \( n \) will increase the size of the model quadratically though, since the model is two-dimensional.

```modelica
model HeatedPlate2D
  parameter Integer n = 4;
  parameter Real L = 1;
  parameter Real k = 0.001;
  Real u[n,n](start=fill(20,n,n));
  Real h=L*L/(n*n);
  equation
    for y in 1:n loop
      der(u[n,y]) = -0.167;
      u[1,y] = 80;
    end for;
    for x in 2:n-1 loop
      der(u[x,n]) = -0.001*u[x,n];
      u[x,1] = 40 + 20*cos(2*3.1415/n*x);
    end for;
    for x in 2:n-1 loop
      for y in 2:n-1 loop
        der(u[x,y]) = (k*(u[x,y-1]-(2*u[x,y])+u[x,y+1]))/h +
                       (k*(u[x-1,y]-(2*u[x,y])+u[x+1,y]))/h;
      end for;
      end for;
end HeatedPlate2D;
```

**Listing 4.3:** Modelica code for the HeatedPlate2D model.

### 4.2 Hardware

The models described in 4.1 were all simulated on three different hardware configurations, one CPU and two GPUs.

#### 4.2.1 CPU Configuration

The CPU used to execute the code generated by OMC was an Intel Core 2 Duo E6600. It has two cores, each with a clock frequency of 2.4 GHz, and represents a fairly normal mainstream CPU. Only one of its cores were used for the simulations though, since OMC normally does not generate parallel code.

#### 4.2.2 GPU Configuration

The GPUs that were used was a NVIDIA GeForce 8800 GTS 320MB [15] and a NVIDIA Tesla C1060 [16].
The GeForce 8800 GTS, based on the G80 architecture, was one of the first GPUs using the CUDA architecture, and it was one of the most powerful GPUs at its release in early 2007. Many new GPUs have since been released though, and GPUs with the same level of performance can now be found in the lower to middle segment of the GPU market. The same E6600 that was used to execute the code generated by OMC was also used as host for the GeForce 8800 GTS.

The NVIDIA Tesla C1060 is on the other hand based on the newer GT200 architecture, which adds new CUDA features such as support for double-precision floating-point numbers. The Tesla C1060 belongs to NVIDIA's Tesla brand, which contains GPUs dedicated to GPGPU. The Tesla GPUs therefore lack the ability to output images to a display, and typically have more memory than their GeForce counterparts. The Tesla lineup also contains server racks with multiple Tesla GPUs, but the C1060 used in this thesis is a single card similar to the GeForce cards. An Intel Xeon E5345 with a clock frequency of 2.0 GHz was used as host for the Tesla C1060.

The specifications of both these GPUs can be found in table 4.1. One thing to note is that the Tesla C1060 is compatible with PCIe 2.0, but the host motherboard seems to only be capable of using PCIe 1.0 according to the `lspci` Linux tool. The Tesla C1060 is therefore not operating at its full capacity, since memory transfers between the host and the C1060 will be limited by the motherboard.

<table>
<thead>
<tr>
<th></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 4.1: GPU Specifications.

4.2.3 Floating Point Precision

CPUs and GPUs use a fixed amount of bits to represent floating-point numbers, which can become a problem when doing scientific computations that require high precision. Most mainstream CPUs and GPUs follow the IEEE 754 standard for floating-point arithmetic, which includes what is usually called the single and double precision floating-point formats. The single precision format, `float` in C, is a 32-bit format, while the double precision format, `double` in C, is a 64-bit format. Double precision floating-point is generally preferred for scientific computations, as discussed in [4].
However, for GPUs it has usually been enough with single precision, because of the visual nature of their computations. As such it is only the most recent GPUs that even have support for double precision floating-point, and the performance penalty for using double precision instead of single precision is high. The support for double precision floating-point is yet new though, and better support can be expected in future GPUs.

The code generated by OMC use double precision floating-point though, so most of the comparisons will be made between double precision on CPUs and single precision on GPUs. This is not quite as unfair as it might sound, since the difference between single and double precision on CPUs is not particularly large. Most mainstream CPUs actually use 80-bit Floating Point Units (FPUs), and the floating-point calculations are done with 80-bit precision regardless of the original precision of the operands, unless SSE extensions are used [7]. Double precision floating-point means twice the amount of data compared with single precision though, so there is usually some decrease of performance with double precision.

Comparing single precision on GPUs against double precision on CPUs is thus not particularly unfair if the intent is to measure single precision performance. Because many models does not require double precision it is therefore still interesting to look at single precision performance, and since future GPUs will surely become better at double precision it also gives an indication to what future GPUs might be capable of.

4.3 Measurements

In this section measurements from simulating the models in section 4.1 will be presented. All models where simulated five times on each hardware configuration, and the simulation time was measured. The simulation time was measured with the `clock_gettime` function from the standard C library in both the OMC generated code and the CUDA code, and only the simulation itself was measured to avoid differences in things such as how the results are output. The time is therefore recorded just before the simulation loop is started, and the elapsed time is then output when the simulation loop is finished. All simulations were run for 10 seconds with 0.001 second time steps, which means 10000 time steps. The upper limit of the number of thread blocks used by the generated CUDA code was set to 60 and the number of threads per block to 32, which in practice means that as many blocks as possible are used. Since none of the models require block-to-block communication this does not cause any problems, and it allows several blocks to be executed on the same SM which helps to hide some of the memory latency.

4.3.1 TestModel

A plot of the average simulation times for the TestModel model can be seen in figure 4.2, and the raw data can be found in tables A.1 to A.3. The size of the model was varied by changing the $N$ parameter. The smallest value of $N$ was chosen to be 64, and $N$ was then doubled until it became 4096. At that point the model produced simulation code that the CUDA compiler had problem compiling due to the size of the generated code.
4.3.2 WaveEquationSample

The WaveEquationSample was simulated with the same method as the TestModel model, by using a value of 30 as the smallest value of $n$ and then doubling it until the generated code could no longer be compiled by the CUDA compiler. However, in this case it was the simulation program generated by the OMC compiler that failed first, because it was unable to allocate enough memory to simulate the model with $n = 3840$ for 10 seconds. It was therefore simulated for 5 seconds instead, and the resulting simulation time was then doubled to approximate the simulation time for 10 seconds. This should be a very good approximation, since the amount of work will be doubled if the simulation time is doubled. It is possible that the extra amount of memory involved with a doubling of the simulation time might influence the performance though, so the simulation time for $n = 3840$ can be seen as a lower bound.

This model was also simulated with both single and double precision on the Tesla C1060 GPU, to see how much double precision impacts the performance. A plot of the average simulation time can be seen in figure 4.3, and the raw data can be found in tables A.4 to A.7.

Other tests were also run on the WaveEquationSample model, with the intention of finding out how much time is spent in different parts of the simulation. In the first test the task execution was disabled, by commenting out the code between the allocation and copy-back functions. Next the calls to the `execute_tasks` kernels were disabled, so that only the integration kernels and memory transfers between device and host were executed. The final step was to disable the integration kernels too, leaving only the memory transfers. The simulation times obtained from these tests can be found in tables A.8 to A.10. By taking the differences between these tests it is thus possible to see how much time is spent in different parts of the simulation, and these data are presented in table 4.2 below.
Figure 4.3: Simulation time plot for the WaveEquationSample model.

<table>
<thead>
<tr>
<th></th>
<th>8800 GTS</th>
<th>C1060 single precision</th>
<th>C1060 double precision</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 Memory 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>

Table 4.2: Seconds spent in the different parts of the simulation.

4.3.3 HeatedPlate2D

The HeatedPlate2D model is two-dimensional in nature, unlike the TestModel and WaveEquationSample models which are one-dimensional. The parameter $n$ was therefore chosen to be smaller than in the other models. The lower bound was chosen to be 4 and an upper bound 48, since larger values of $n$ than 48 produced too much code for the CUDA compiler to compile. The average simulation time can be seen in figure 4.4, and the raw data can be found in tables A.11 to A.13.
Figure 4.4: Simulation time plot for the HeatedPlate2D model.
Chapter 5

Discussion

5.1 Result Interpretation

By looking at the plots in chapter 4.3 it is immediately evident that it is not only feasible to run simulations on GPUs, but for sufficiently large models it is also possible to achieve better performance than on a CPU. While the simulation times on small models are much worse on the GPUs compared with the CPU, the GPUs retain a relatively flat curve, while the simulation times on the CPU rises much faster. The reason for this is likely that the GPUs need many thread blocks with many threads to fully utilize their power, and smaller models therefore do not use all SMs available. The simulation times on the CPUs instead behaves as could be expected by approximately doubling when the model size is doubled. Good performance is achieved on the WaveEquationSample in particular, where the GeForce 8800 GTS is approximately 4.6 times faster than the E6600. With the TestModel a speedup of 2.5 was achieved, while a more modest speedup of 1.5 was achieved for the HeatedPlate2D model which is somewhat harder to parallelize. It is important to remember that only one core on the dual-core E6600 was used though.

While the relative performance between the GPUs and CPU is as could be expected, there are some things that are somewhat unexpected. The first of these is that the GeForce 8800 GTS is consistently faster than the Tesla C1060. Considering that the C1060 has better specification compared with the 8800 GTS, see table 4.1, this is certainly curious. Looking at table 4.2 it can be seen that the C1060 is much slower on the task execution itself, and slightly slower on the device to host memory transfers. The small difference in device to host memory transfers might be attributed to differences in the host hardware and software, and especially the fact that the C1060 seems to be operating on a PCIe 1.0 bus while actually being PCIe 2.0 compatible. The much slower task execution on the C1060 is harder to explain though, and further investigation would be needed to determine the exact cause of this.

The second unexpected thing is that the double precision performance of the C1060 actually is not that far from its single precision performance, even though the C1060 is capable of almost 12 times as many FLOPS in single precision compared to double precision. By again looking at table 4.2 it can be seen that it is actually memory operations that make up the majority of the
simulation time though. Double precision means twice the amount of data compared to single precision, which is a performance difference much smaller than the factor of 12 in FLOPS. The simulation time when using double precision is therefore bounded mostly by the memory performance, and not the floating point operation performance. Something that is very curious is that the task execution is significantly faster with double precision than with single precision though. This is in fact so curious that the tests were run several times, always with the same results, and the cause of this anomaly is still unknown. However, the result is that the C1060 is approximately 2.7 times faster than the E6600 on the largest WaveEquationSample model even when using double precision.

As might have been suspected it is therefore memory transactions that take most of the time, and the actual computations are almost for free. The models must also be quite large so that enough threads can be spawned to fully utilize the GPU. While the models used in this thesis are parallel enough to easily spawn many threads, the computations per variable ratio is quite low. Models with more computations per variable should see an even larger performance increase when using a GPU.

5.2 Future work

In the current implementation almost all work is done on the GPU, but this is probably not the best option since the CPU will be idle while the GPU is working. The GPU is very suitable for highly parallel models with large number of variables, but the CPU is better suited for general simulations. Something that could be worth researching in the future is therefore to combine the strengths of both GPU and CPU, by finding parts of the model that can be parallelized and computed on the GPU while the rest of the model is simulated on the CPU. If completely independent parts of a model can be found it might also be possible to simulate those parts with a single kernel, instead of calling multiple kernels per time step as is done in the current implementation. This would mean that the variables could be kept in the shared memory during the whole simulation, although their values would still have to be saved for each time step and transferred back to the host in the end.

If only parallel models are considered though, then it might be worth investigating the possibility of implementing barrier synchronization using the global memory. In the current implementation the problem of synchronization was solved by using different kernels for task execution and the integration steps, since only one kernel can be executed at a time. This would also allow the variables to remain in the shared memory, and the performance gain might be larger than the performance loss of extra global memory transactions.

A problem that needs to solved for any of these options to be viable for larger models is the size of the generated code. The CUDA compiler starts having problems when the generated code becomes larger than around 30000 lines. A large majority of code that the current implementation generates consists of shared memory allocations, so finding a way of reducing this code will make it possible to simulate larger models on the GPU.

It is also worth considering future generations of GPUs. NVIDIA has recently released information about their next architecture, codenamed Fermi [14], which will have several improvements that might benefit applications such as
those explored in this thesis. One of the improvements are the addition of a
cache hierarchy, where each SM will have a 64 KB large memory that will be
used as both shared memory and L1 cache. A 768 KB large L2 cache will also
be placed between the SMs and the off-chip memory. This would probably
help in the current implementation where the off-chip memory is used quite a
lot. Another improvement is that the double precision performance has been
greatly improved, with a four time increase in double precision performance
per SP being reported. The number of SPs per SM has also increased from 8
to 32, which means that SIMD parallelism will be even more important. The
SFU per SP ratio has decreased though, since each SM will only have 4 SFUs
compared with 2 SFUs per SM in the current architectures. The SFUs can be
used simultaneously with the SPs though, which is not possible in current ar-
chitectures, so for code that uses a mix of transcendental functions and normal
operations this should be an improvement. It will also be possible to execute
multiple kernels concurrently, so smaller kernels that does not use the whole
GPU can be executed at the same time to increase the GPU utilization. This
might make it easier to generate efficient code that only uses CUDA for parts
of the simulation, where the generated kernels do not need to be large enough
to fill the GPU by themselves. Fermi will also feature more accurate floating
point numbers, because it implements the IEEE 754-2008 floating point stan-
dard instead of the currently used IEEE 754-1985. These improvements, along
with several other, means that the Fermi architecture may prove to be much
more suitable for simulating Modelica models than the current architectures.

Both NVIDIA and AMD have also released OpenCL compatible drivers,
which means that it is now possible to develop GPGPU applications that can
run on both NVIDIA and AMD GPUs. It might therefore be interesting to use
OpenCL instead of CUDA to achieve better hardware support, although CUDA
may be capable of better performance on NVIDIA GPUs due to being under
NVIDIAs direct control.

5.3 Conclusions

The goal of this thesis was to evaluate the feasibility of simulating Modelica
models on GPUs, and it has been shown in this thesis that it is possible to
simulate at least a subset of all models. For certain models it has been shown
that it is even possible to get significant speedups when simulating them on a
GPU compared with a CPU, but this requires large and highly parallel models.
Several possible ways in which the GPU utilization for Modelica simulation can
be improved has been suggested though, and considering that improved GPGPU
architectures will soon be released it seems like this area of research is worthy
of further study.
Appendix A

Simulation Time Measurements

In the following section the simulation times that were measured for each of the models will be presented. All times are measured in seconds.

A.1 TestModel

<table>
<thead>
<tr>
<th>N</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.997</td>
<td>0.981</td>
<td>0.981</td>
<td>0.980</td>
<td>0.995</td>
<td>0.987</td>
</tr>
<tr>
<td>128</td>
<td>1.111</td>
<td>1.071</td>
<td>1.070</td>
<td>1.117</td>
<td>1.066</td>
<td>1.087</td>
</tr>
<tr>
<td>256</td>
<td>1.117</td>
<td>1.111</td>
<td>1.119</td>
<td>1.097</td>
<td>1.097</td>
<td>1.109</td>
</tr>
<tr>
<td>512</td>
<td>1.198</td>
<td>1.189</td>
<td>1.187</td>
<td>1.218</td>
<td>1.181</td>
<td>1.195</td>
</tr>
<tr>
<td>1024</td>
<td>1.331</td>
<td>1.317</td>
<td>1.311</td>
<td>1.332</td>
<td>1.348</td>
<td>1.328</td>
</tr>
<tr>
<td>2048</td>
<td>1.966</td>
<td>1.992</td>
<td>1.959</td>
<td>1.960</td>
<td>2.001</td>
<td>1.976</td>
</tr>
<tr>
<td>4096</td>
<td>2.942</td>
<td>2.941</td>
<td>2.927</td>
<td>2.930</td>
<td>2.915</td>
<td>2.931</td>
</tr>
</tbody>
</table>

Table A.1: TestModel simulation times for the GeForce 8800 GTS.

<table>
<thead>
<tr>
<th>N</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.282</td>
<td>1.297</td>
<td>1.298</td>
<td>1.296</td>
<td>1.297</td>
<td>1.294</td>
</tr>
<tr>
<td>128</td>
<td>1.401</td>
<td>1.402</td>
<td>1.402</td>
<td>1.391</td>
<td>1.393</td>
<td>1.398</td>
</tr>
<tr>
<td>256</td>
<td>1.430</td>
<td>1.429</td>
<td>1.429</td>
<td>1.429</td>
<td>1.429</td>
<td>1.430</td>
</tr>
<tr>
<td>512</td>
<td>1.500</td>
<td>1.504</td>
<td>1.500</td>
<td>1.501</td>
<td>1.500</td>
<td>1.501</td>
</tr>
<tr>
<td>1024</td>
<td>1.615</td>
<td>1.615</td>
<td>1.614</td>
<td>1.617</td>
<td>1.615</td>
<td>1.615</td>
</tr>
<tr>
<td>2048</td>
<td>2.383</td>
<td>2.388</td>
<td>2.384</td>
<td>2.387</td>
<td>2.384</td>
<td>2.385</td>
</tr>
</tbody>
</table>

Table A.2: TestModel simulation times for the Tesla C1060, single precision.
### Table A.3: TestModel simulation times for the Core 2 Duo E6600.

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.092</td>
<td>0.086</td>
<td>0.091</td>
<td>0.089</td>
<td>0.087</td>
<td>0.089</td>
</tr>
<tr>
<td>128</td>
<td>0.175</td>
<td>0.187</td>
<td>0.174</td>
<td>0.189</td>
<td>0.186</td>
<td>0.182</td>
</tr>
<tr>
<td>256</td>
<td>0.442</td>
<td>0.434</td>
<td>0.450</td>
<td>0.433</td>
<td>0.445</td>
<td>0.441</td>
</tr>
<tr>
<td>512</td>
<td>0.880</td>
<td>0.896</td>
<td>0.883</td>
<td>0.895</td>
<td>0.914</td>
<td>0.894</td>
</tr>
<tr>
<td>1024</td>
<td>1.747</td>
<td>1.739</td>
<td>1.736</td>
<td>1.786</td>
<td>1.736</td>
<td>1.749</td>
</tr>
</tbody>
</table>

### A.2 WaveEquationSample

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.839</td>
<td>0.844</td>
<td>0.831</td>
<td>0.840</td>
<td>0.841</td>
<td>0.839</td>
</tr>
<tr>
<td>60</td>
<td>0.963</td>
<td>0.944</td>
<td>0.946</td>
<td>0.955</td>
<td>0.960</td>
<td>0.954</td>
</tr>
<tr>
<td>120</td>
<td>1.130</td>
<td>1.109</td>
<td>1.108</td>
<td>1.105</td>
<td>1.114</td>
<td>1.113</td>
</tr>
<tr>
<td>240</td>
<td>1.138</td>
<td>1.132</td>
<td>1.146</td>
<td>1.144</td>
<td>1.136</td>
<td>1.140</td>
</tr>
<tr>
<td>480</td>
<td>1.247</td>
<td>1.234</td>
<td>1.256</td>
<td>1.241</td>
<td>1.261</td>
<td>1.248</td>
</tr>
<tr>
<td>960</td>
<td>1.441</td>
<td>1.470</td>
<td>1.526</td>
<td>1.432</td>
<td>1.427</td>
<td>1.460</td>
</tr>
<tr>
<td>1920</td>
<td>2.034</td>
<td>2.011</td>
<td>1.981</td>
<td>1.982</td>
<td>1.965</td>
<td>1.995</td>
</tr>
<tr>
<td>3840</td>
<td>3.185</td>
<td>3.099</td>
<td>3.089</td>
<td>3.096</td>
<td>3.154</td>
<td>3.125</td>
</tr>
</tbody>
</table>

### Table A.4: WaveEquationSample simulation times for the GeForce 8800 GTS.

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>1.122</td>
<td>1.118</td>
<td>1.119</td>
<td>1.122</td>
<td>1.120</td>
<td>1.120</td>
</tr>
<tr>
<td>60</td>
<td>1.242</td>
<td>1.235</td>
<td>1.235</td>
<td>1.241</td>
<td>1.235</td>
<td>1.238</td>
</tr>
<tr>
<td>120</td>
<td>1.418</td>
<td>1.419</td>
<td>1.419</td>
<td>1.418</td>
<td>1.419</td>
<td>1.419</td>
</tr>
<tr>
<td>240</td>
<td>1.464</td>
<td>1.464</td>
<td>1.464</td>
<td>1.467</td>
<td>1.465</td>
<td>1.465</td>
</tr>
<tr>
<td>480</td>
<td>1.547</td>
<td>1.547</td>
<td>1.545</td>
<td>1.546</td>
<td>1.546</td>
<td>1.546</td>
</tr>
<tr>
<td>960</td>
<td>1.725</td>
<td>1.723</td>
<td>1.725</td>
<td>1.725</td>
<td>1.724</td>
<td>1.724</td>
</tr>
<tr>
<td>1920</td>
<td>2.308</td>
<td>2.305</td>
<td>2.301</td>
<td>2.306</td>
<td>2.305</td>
<td>2.305</td>
</tr>
</tbody>
</table>

### Table A.5: WaveEquationSample simulation times for the Tesla C1060, single precision.

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>1.244</td>
<td>1.243</td>
<td>1.239</td>
<td>1.250</td>
<td>1.251</td>
<td>1.245</td>
</tr>
<tr>
<td>60</td>
<td>1.453</td>
<td>1.453</td>
<td>1.452</td>
<td>1.451</td>
<td>1.453</td>
<td>1.452</td>
</tr>
<tr>
<td>120</td>
<td>1.706</td>
<td>1.706</td>
<td>1.706</td>
<td>1.707</td>
<td>1.706</td>
<td>1.706</td>
</tr>
<tr>
<td>240</td>
<td>1.801</td>
<td>1.798</td>
<td>1.796</td>
<td>1.798</td>
<td>1.797</td>
<td>1.798</td>
</tr>
<tr>
<td>480</td>
<td>1.956</td>
<td>1.953</td>
<td>1.951</td>
<td>1.954</td>
<td>1.952</td>
<td>1.953</td>
</tr>
<tr>
<td>960</td>
<td>2.266</td>
<td>2.265</td>
<td>2.267</td>
<td>2.264</td>
<td>2.267</td>
<td>2.266</td>
</tr>
<tr>
<td>3840</td>
<td>5.397</td>
<td>5.403</td>
<td>5.401</td>
<td>5.397</td>
<td>5.396</td>
<td>5.399</td>
</tr>
</tbody>
</table>

### Table A.6: WaveEquationSample simulation times for the Tesla C1060, double precision.
### Table A.7: WaveEquationSample simulation times for the Core 2 Duo E6600.

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.095</td>
<td>0.084</td>
<td>0.086</td>
<td>0.086</td>
<td>0.084</td>
<td>0.087</td>
</tr>
<tr>
<td>60</td>
<td>0.168</td>
<td>0.164</td>
<td>0.165</td>
<td>0.163</td>
<td>0.169</td>
<td>0.166</td>
</tr>
<tr>
<td>120</td>
<td>0.354</td>
<td>0.362</td>
<td>0.348</td>
<td>0.347</td>
<td>0.350</td>
<td>0.352</td>
</tr>
<tr>
<td>240</td>
<td>0.815</td>
<td>0.826</td>
<td>0.827</td>
<td>0.800</td>
<td>0.786</td>
<td>0.811</td>
</tr>
<tr>
<td>480</td>
<td>1.643</td>
<td>1.702</td>
<td>1.622</td>
<td>1.670</td>
<td>1.646</td>
<td>1.657</td>
</tr>
</tbody>
</table>

### Table A.8: Simulation times for different configurations of WaveEquationSample on the GeForce 8800 GTS.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>3.185</td>
<td>3.099</td>
<td>3.089</td>
<td>3.096</td>
<td>3.154</td>
<td>3.125</td>
</tr>
<tr>
<td>Task execution disabled</td>
<td>2.957</td>
<td>2.956</td>
<td>2.982</td>
<td>2.955</td>
<td>2.953</td>
<td>2.961</td>
</tr>
<tr>
<td>Shared memory disabled</td>
<td>1.516</td>
<td>1.516</td>
<td>1.516</td>
<td>1.524</td>
<td>1.535</td>
<td>1.521</td>
</tr>
<tr>
<td>Only memory transfers</td>
<td>1.093</td>
<td>1.109</td>
<td>1.109</td>
<td>1.096</td>
<td>1.112</td>
<td>1.104</td>
</tr>
</tbody>
</table>

### Table A.9: Simulation times for different configurations of WaveEquationSample on the Tesla C1060, with single precision.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>5.397</td>
<td>5.403</td>
<td>5.401</td>
<td>5.397</td>
<td>5.396</td>
<td>5.399</td>
</tr>
<tr>
<td>Task execution disabled</td>
<td>5.015</td>
<td>5.009</td>
<td>5.012</td>
<td>5.010</td>
<td>5.006</td>
<td>5.010</td>
</tr>
<tr>
<td>Shared memory disabled</td>
<td>2.725</td>
<td>2.719</td>
<td>2.725</td>
<td>2.724</td>
<td>2.722</td>
<td>2.723</td>
</tr>
<tr>
<td>Only memory transfers</td>
<td>2.277</td>
<td>2.274</td>
<td>2.285</td>
<td>2.274</td>
<td>2.278</td>
<td>2.278</td>
</tr>
</tbody>
</table>

### Table A.10: Simulation times for different configurations of WaveEquationSample on the Tesla C1060, with double precision.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>5.397</td>
<td>5.403</td>
<td>5.401</td>
<td>5.397</td>
<td>5.396</td>
<td>5.399</td>
</tr>
<tr>
<td>Task execution disabled</td>
<td>5.015</td>
<td>5.009</td>
<td>5.012</td>
<td>5.010</td>
<td>5.006</td>
<td>5.010</td>
</tr>
<tr>
<td>Shared memory disabled</td>
<td>2.725</td>
<td>2.719</td>
<td>2.725</td>
<td>2.724</td>
<td>2.722</td>
<td>2.723</td>
</tr>
<tr>
<td>Only memory transfers</td>
<td>2.277</td>
<td>2.274</td>
<td>2.285</td>
<td>2.274</td>
<td>2.278</td>
<td>2.278</td>
</tr>
</tbody>
</table>
A.3 HeatedPlate2D

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.802</td>
<td>0.823</td>
<td>0.786</td>
<td>0.786</td>
<td>0.787</td>
<td>0.797</td>
</tr>
<tr>
<td>8</td>
<td>0.944</td>
<td>0.938</td>
<td>0.928</td>
<td>0.937</td>
<td>0.936</td>
<td>0.937</td>
</tr>
<tr>
<td>16</td>
<td>1.408</td>
<td>1.391</td>
<td>1.410</td>
<td>1.386</td>
<td>1.389</td>
<td>1.397</td>
</tr>
<tr>
<td>24</td>
<td>1.601</td>
<td>1.588</td>
<td>1.584</td>
<td>1.613</td>
<td>1.641</td>
<td>1.605</td>
</tr>
<tr>
<td>32</td>
<td>1.813</td>
<td>1.756</td>
<td>1.763</td>
<td>1.822</td>
<td>1.798</td>
<td>1.790</td>
</tr>
<tr>
<td>48</td>
<td>2.868</td>
<td>2.869</td>
<td>2.819</td>
<td>2.835</td>
<td>2.806</td>
<td>2.840</td>
</tr>
</tbody>
</table>

Table A.11: HeatedPlate2D simulation times for the GeForce 8800 GTS.

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.068</td>
<td>1.070</td>
<td>1.071</td>
<td>1.071</td>
<td>1.070</td>
<td>1.070</td>
</tr>
<tr>
<td>8</td>
<td>1.201</td>
<td>1.198</td>
<td>1.198</td>
<td>1.201</td>
<td>1.201</td>
<td>1.200</td>
</tr>
<tr>
<td>16</td>
<td>1.693</td>
<td>1.695</td>
<td>1.694</td>
<td>1.693</td>
<td>1.694</td>
<td>1.694</td>
</tr>
<tr>
<td>24</td>
<td>1.868</td>
<td>1.871</td>
<td>1.868</td>
<td>1.868</td>
<td>1.868</td>
<td>1.869</td>
</tr>
<tr>
<td>32</td>
<td>1.978</td>
<td>1.976</td>
<td>1.977</td>
<td>1.977</td>
<td>1.975</td>
<td>1.977</td>
</tr>
<tr>
<td>48</td>
<td>3.033</td>
<td>3.030</td>
<td>3.031</td>
<td>3.031</td>
<td>3.031</td>
<td>3.031</td>
</tr>
</tbody>
</table>

Table A.12: HeatedPlate2D simulation times for the Tesla C1060, single precision.

<table>
<thead>
<tr>
<th>n</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.015</td>
<td>0.016</td>
<td>0.015</td>
<td>0.016</td>
<td>0.016</td>
<td>0.016</td>
</tr>
<tr>
<td>8</td>
<td>0.054</td>
<td>0.053</td>
<td>0.055</td>
<td>0.053</td>
<td>0.055</td>
<td>0.054</td>
</tr>
<tr>
<td>16</td>
<td>0.358</td>
<td>0.362</td>
<td>0.359</td>
<td>0.365</td>
<td>0.357</td>
<td>0.360</td>
</tr>
<tr>
<td>24</td>
<td>1.029</td>
<td>1.048</td>
<td>1.094</td>
<td>1.021</td>
<td>1.059</td>
<td>1.050</td>
</tr>
<tr>
<td>32</td>
<td>1.555</td>
<td>1.545</td>
<td>1.553</td>
<td>1.535</td>
<td>1.554</td>
<td>1.548</td>
</tr>
</tbody>
</table>

Table A.13: HeatedPlate2D simulation times for the Core 2 Duo E6600.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Per Östlund