ParModelica: Extending the Algorithmic Subset of Modelica with Explicit Parallel Language Constructs for Multi-core Simulation

by

Mahder Gebremedhin

LIU-IDA/LITH-EX-A—11/043—SE

2011-10-15
Final Thesis

ParModelica: Extending the Algorithmic Subset of Modelica with Explicit Parallel Language Constructs for Multi-core Simulation

by

Mahder Gebremedhin

LIU-ID A/ LITH-EX-A—11/043—SE

2011-10-15

Supervisor: Kristian Stavåker
Examiner: Peter Fritzson
Acknowledgment

I would like to thank Peter Fritzson for giving me the chance to work on this interesting thesis work and for providing a working environment motivated by creativity and understanding rather than by strict supervision. I would also like to thank Kristian Stavåker for his technical assistance and supervising the thesis work, Per Östlund for his quick responses in OpenCL related issues, Mohsen Torabzadeh-Tari for his assistance with non-technical issues. I would also like to thank Afshin Hemmati Moghadam for his patience and help throughout the rather frustrating testing and debugging phase of this thesis work.
Abstract

In today’s world of high tech manufacturing and computer-aided design simulations of models is at the heart of the whole manufacturing process. Trying to represent and study the variables of real world models using simulation computer programs can turn out to be a very expensive and time consuming task. On the other hand advancements in modern multi-core CPUs and general purpose GPUs promise remarkable computational power.

Properly utilizing this computational power can provide reduced simulation time. To this end modern modeling environments provide different optimization and parallelization options to take advantage of the available computational power. Some of these parallelization approaches are based on automatically extracting parallelism with the help of a compiler. Another approach is to provide the model programmers with the necessary language constructs to express any potential parallelism in their models. This second approach is taken in this thesis work.

The OpenModelica modeling and simulation environment for the Modelica language has been extended with new language constructs for explicitly stating parallelism in algorithms. This slightly extended algorithmic subset of Modelica is called ParModelica. The new extensions allow models written in ParModelica to be translated to optimized OpenCL code which can take advantage of the computational power of available Multi-core CPUs and general purpose GPUs.
Contents

Contents ............................................................................................................................................ 1
List of Listings .................................................................................................................................. 4
List of Figures .................................................................................................................................. 5
List of Acronyms ............................................................................................................................... 6
1 Introduction ...................................................................................................................................... 7
  1.1 Modeling and Simulation .......................................................................................................... 7
  1.2 General Purpose Graphic Processing Unit (GPGPU) programming .................................. 8
  1.3 Thesis Overview ...................................................................................................................... 8
  1.4 Intended Audience .................................................................................................................... 8
2 Background ...................................................................................................................................... 9
  2.1 Modelica ................................................................................................................................... 9
  2.2 OpenModelica ........................................................................................................................ 10
  2.3 MetaModelica and the OpenModelica Compiler (OMC) .................................................... 10
  2.4 The OpenCL Architecture ....................................................................................................... 11
    2.4.1 Platform Model .................................................................................................................. 11
    2.4.2 Execution Model .............................................................................................................. 12
    2.4.3 Memory Model .................................................................................................................. 13
    2.4.4 Programming Model ........................................................................................................ 14
  2.5 The Susan Template Language ............................................................................................... 15
  2.6 Previous Work ......................................................................................................................... 15
3 Extending the Algorithmic Subset: Extensions ............................................................................. 16
  3.1 Overview .................................................................................................................................. 16
  3.2 Parallel Variables ..................................................................................................................... 16
  3.3 Parallel Functions ..................................................................................................................... 18
  3.4 Kernel Functions ...................................................................................................................... 18
  3.5 Parallel For Loop – parfor ....................................................................................................... 19
  3.6 Built-in Functions .................................................................................................................... 20
  3.7 Synchronization and Thread Management ............................................................................ 21
  3.8 OpenCL Functionalities .......................................................................................................... 22
4 Extending the Algorithmic Subset: Implementation .................................................................... 24
4.1 Front-end .............................................................................................................. 25
  4.1.1 Keywords: Lexical and Syntax Analysis ....................................................... 26
  4.1.2 Module Absyn: The Abstract Syntax Tree .................................................. 28
  4.1.3 Module Algorithm ....................................................................................... 29
  4.1.4 Module Ceval: Constant Evaluation ........................................................... 29
  4.1.5 Module ClassInf ......................................................................................... 30
  4.1.6 Module DAE: DAE Equation Management and Output ....................... 30
  4.1.7 Module Inst: Code Instantiation/Elaboration ........................................... 31
  4.1.8 Module SCode ............................................................................................ 31
  4.1.9 Module Static ............................................................................................. 32
  4.2 Back-end ........................................................................................................... 32
    4.2.1 Module SimCode ....................................................................................... 33
  4.3 Susan Code Generation ..................................................................................... 37
    4.3.1 SimCodeC ................................................................................................. 38
  4.4 OpenCL Runtime ............................................................................................... 46
    4.4.1 OpenCL-C Runtime Library ..................................................................... 46
    4.4.2 OpenCL Helper ....................................................................................... 47
    4.4.3 OpenCL Offline Compiler ...................................................................... 48
    4.4.4 Memory Management ............................................................................ 49
  5 Discussion ............................................................................................................. 60
    5.1 Accomplishments ......................................................................................... 60
    5.2 Performance .................................................................................................. 60
    5.3 MPAR Benchmark Suite ............................................................................. 61
    5.4 Future Work .................................................................................................. 64
    5.5 Conclusion .................................................................................................... 65
  Appendix A Sample Source Code ................................................................ .......... 66
    A.1 MatMult.mo ................................................................................................. 66
    A.2 MatMult_one_function.h ........................................................................... 67
    A.3 MatMult_one_functions.cpp ..................................................................... 68
    A.4 MatMult_one_parallelFunctions.cl .......................................................... 71
    A.5 MatMult_one_kernels.cl ............................................................................. 74
Appendix B  OpenCL-C Runtime Library Functions .................................................. 76
  B.1  Utility Functions .......................................................................................... 76
  B.2  Interfacing Functions .................................................................................. 78
  B.3  Scalar Assignment Functions ....................................................................... 79
  B.4  Miscellaneous Functions ............................................................................ 80
Bibliography ............................................................................................................. 82
List of Listings

Listing 3.2-1 Modelica parallel variables ................................................................. 16
Listing 3.3-1 Modelica parallel function ................................................................. 18
Listing 3.4-1 Modelica kernel function ................................................................... 18
Listing 3.5-1 Modelica parallel for loop ................................................................. 20
Listing 3.8-1 Executing user written OpenCL files ................................................. 23
Listing 4.1-1 ANTLR parsing rules for parallel variables ...................................... 27
Listing 4.1-2 ANTLR parsing rules for parallel and kernel functions ....................... 27
Listing 4.1-3 ANTLR parsing rules for parfor ......................................................... 28
Listing 4.1-4 New additions to Absyn datatypes ..................................................... 29
Listing 4.2-1 SimCode data structures with the parallel extensions ......................... 33
Listing 4.2-2 SimCode.elaborateFunctions, parallel variables in serial functions .... 34
Listing 4.2-3 SimCode.elaborateFunctions, kernel functions .................................. 36
Listing 4.3-1 Susan Code generation top level ......................................................... 39
Listing 4.3-2 Template function parVarInit ............................................................ 40
Listing 4.3-3 Modelica parfor .................................................................................. 42
Listing 4.3-4 C code for a parfor loop .................................................................... 42
Listing 4.3-5 OpenCL kernel for a parfor loop ........................................................ 43
Listing 4.4-1 integer array structure ....................................................................... 48
Listing 4.4-2 device_buffer struct .......................................................................... 51
Listing 4.4-3 Function ocl_create_execution_memory_buffer() ............................. 52
Listing 4.4-4 Structures used for memory management in Kernels ....................... 53
Listing 4.4-5 Memory initialization in kernels ......................................................... 54
Listing 4.4-6 Function initialize_buffer() ............................................................. 55
Listing 4.4-7 Function alloc_integer_array_c99_2 .............................................. 58
List of Figures

Figure 2.1-1 OpenModelica Environment ................................................................. 9
Figure 2.3-1 OMC compilation phases ................................................................. 10
Figure 2.4-1 OpenCL Platform Model ................................................................. 11
Figure 2.4-2 OpenCL Execution Model ................................................................. 12
Figure 2.4-3 Conceptual OpenCL device architecture with processing elements (PE) .... 13
Figure 2.4-4 Memory Region - Allocation and Memory Access Capabilities ................. 14
Figure 3.8-1 OMC Extended compilation phases ..................................................... 24
Figure 4.1-1 Main modules of the OpenModelica Compiler ...................................... 25
Figure 4.3-1 Code generation traversal ................................................................. 38
Figure 4.4-1 OpenCL-C runtime system ............................................................... 46
Figure 4.4-2 Division of execution memory between threads and data types ................. 56
Figure 5.2-1 Serial Matrix Multiplication on Intel Q6600 @ 2.4 GHz core utilizations .... 62
Figure 5.2-2 Parallel Matrix Multiplication on Intel Q6600 @ 2.4 GHz core utilizations ... 63
Figure 5.2-3 Parallel Matrix Multiplication on Intel E5520 @ 2.27GHz core utilizations ... 63
List of Acronyms

AMD Advanced Micro Devices.
ANTLR ANother Tool for Language Recognition.
API Application Programming Interface.
APP Application.
AST Abstract Syntax Tree.
CPU Central Processing Unit.
CUDA Compute Unified Device Architecture.
DAE Differential Algebraic Equation
GPGPU General-Purpose Graphics Processing Unit.
GPU Graphics Processing Unit.
IDA Institutionen för datavetenskap.
MDT Modelica Development Tooling.
MPAR Modelica Parallel.
OCL OpenCL.
OMC OpenModelica Compiler.
OpenCL Open Computing Language.
OSMC Open Source Modelica Consortium.
PARFOR PARallel FOR
PELAB Programming Environments LABoratory.
SDK Software Development Kit.
SPMD single program, multiple data.
1 Introduction

In this work the OpenModelica compiler [1] is extended with additional parallel language constructs to enable explicit parallel algorithms in addition to the currently available serial constructs. We use the name ParModelica for this slightly extended Modelica to emphasize PARAllel MODELICA.

This thesis work is focused on simulating large and complex Modelica [2] models on parallel architectures; especially on highly data parallel Graphical Processing Units (GPU). Harnessing the computation powers of current data parallel General Purpose Computing on Graphics Processing Unit (GPGPU) architectures promises a reduced simulation time for some models.

The implementation is primarily focusing on generating optimized OpenCL [3] code for models while at the same time providing the necessary mechanism for generating CUDA [4] code.

Motivations behind the choice of target language will be given in detail in subsequent chapters.

1.1 Modeling and Simulation

A System is defined as an organized structure composed from a set of interconnected and correlated objects. A System exists and operates in time and space. In order to study the properties of a System it is necessary to observe its behaviors and outputs for different environments or input parameters. This process is what is called and Experiment. However it is not always feasible to study the properties of a System with actual Experiments. Usually a physical system will turn out to be very complicated and difficult to directly exercise or experiment on it. For this reason Models of the System are used instead. A Model is a simplified representation of a system. They are built to study the behaviors of a System. This is called Modeling. Models usually include only the details of a System which are relevant to the behavior which is to be studied. Experiments can be performed on a Model of a System to study its behaviors. These results in what is called Simulation. A Simulation in its simplest form is: An experiment performed in a Model [5].

In the context of this work Models are computerized representations of a System described using Mathematical Modeling languages, specifically Modelica. A Simulation is the computerized experiment on these Models with the help of a computer. Simulations are computer executions of a Model over time to study its behavior.
1.2 General Purpose Graphic Processing Unit (GPGPU) programming.

A GPGPU is a general purpose Graphics Processing Units (GPUs) designed for use in data-parallel graphic as well as non-graphic computations. Traditionally the use of most GPUs was limited to processing of only graphics data. However, in recent years it has become more common to use them for processing of non-graphic scientific and engineering computations as well.

GPGPU programming is based on the concept of using the CPU and GPU as heterogeneous computing units. The CPU is used to execute serial parts of the computation and manage the GPU while the GPU is used, as another highly parallel processing unit, to perform parallel parts of the computation.

Different frameworks of programming for GPUs are available now. OpenCL, CUDA, DirectX [6], OpenGL [7] and DirectCompute [8] are some examples. The last three frameworks are more focused on the traditional use of using GPUs for processing of graphic data. However CUDA and OpenCL provide a rather complete implementation for proper GPGPU programming. These two are used widely to implement non-graphic heavy computations.

1.3 Thesis Overview

The remainder of this thesis report is organized as follows:

- Chapter 2 provides some background information on the technologies used for the thesis work.
- Chapter 3 explains the new parallel programming constructs added to the OpenModelica compiler and shows their proper usage.
- Chapter 4 explains in detail how the thesis work is implemented.
- Chapter 5 discusses the achievements of the thesis work as well as some performance discussions. It also provides suggestions for future work.

1.4 Intended Audience

This thesis work is intended for readers familiar with the Modelica modeling language, the OpenModelica simulation environment and OpenCL parallel programming framework. However, most users with basic knowledge of compiler construction and parallel programming can understand it too.
2 Background

2.1 Modelica

Modelica [2] is a non-proprietary, object-oriented, equation based, multi-domain modeling language for component-oriented modeling of complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. Its development is overseen by the non-profit Modelica Association [9]. The Modelica Association also develops the open source Modelica Standard Library which contains more than 900 model components and 600 functions from many domains (at the moment of this writing).

Modelica is an object-oriented language with a general class concept. Modelica classes can contain equations. Equations do not describe assignment but equality and have no predefined causality. Unlike assignment statements equations can have expressions on both right and left sides of the assignment. These mathematical equations are manipulated symbolically by the compiler to determine their order of execution. Modelica is suited for component based model development: It provides the constructs for creating and connecting components allowing construction of complex multi-domain models from small reusable components.

Figure 2.1-1 OpenModelica Environment
2.2 OpenModelica

OpenModelica [1] is an open-source Modelica-based modeling and simulation environment intended for industrial and academic usage. Its long-term development is supported by a non-profit organization – the Open Source Modelica Consortium (OSMC) [10].

The Programming Environments Laboratory (PELAB) [11] at Linköping University, together with OSMC, is developing the OpenModelica modeling and simulation environment including the OpenModelica Compiler (OMC) for the Modelica language (including the MetaModelica extensions). There is also an Eclipse plug-in in Modelica Development Tooling (MDT) which includes a debugger. A Template Code Generation language called Susan [12] [13] is also available. Figure 2.1-1 shows the components of the OpenModelica environment.

2.3 MetaModelica and the OpenModelica Compiler (OMC)

MetaModelica [14] is an extended subset of the Modelica language designed mainly for the purpose of implementing the OpenModelica Compiler. These extensions include pattern equations, match expressions, list, tuple, option and uniontype. The extensions provide MetaModelica with the necessary mechanisms needed for language specification and design. Since MetaModelica is a direct extension of Modelica it is the ideal language to implement the OpenModelica Compiler. Most parts of OMC are currently implemented with MetaModelica. At the time of this thesis work another thesis work [15] was being done targeted on implementing a MetaModelica parser code generator for OMC, thereby taking OMC one step closer to complete MetaModelica based implementation.

Figure 2.3-1 OMC compilation phases.

The OpenModelica Compiler is composed of a number of compilation stages. Figure 2.3-1 shows these compilation phases.
2.4 The OpenCL Architecture

OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. The OpenCL programming language is based on C99 with some extensions for parallel execution management. By using OpenCL it is possible to write parallel algorithms that can be easily ported between multiple devices with minimal changes to the source code.

The OpenCL framework is composed of the OpenCL; programming language, API, libraries and a runtime system to support software development. The framework can be divided in to a hierarchy of models: Platform Model, Memory model, Execution model and Programming model. A brief description of these models is given in the following sections. However, for a complete understanding of the OpenCL framework it is recommended that the reader refers to [16].

2.4.1 Platform Model

The OpenCL platform model is defined as a Host connected to one or more OpenCL devices. The OpenCL devices are divided into one or more Computing Units (CU) which in turn are divided into one or more Processing Elements (PE). The host is responsible for managing the executions on OpenCL devices. This management includes: identifying and initializing OpenCL devices, data copy operations and submitting parallel jobs to the OpenCL device.

Figure 2.4-1 OpenCL Platform Model\(^1\)

\(^1\) Picture taken from [16].
2.4.2 Execution Model

The execution of an OpenCL program consists of two parts. The Host program which executes on the host and the OpenCL program which executes on the OpenCL device. The host program manages the execution of the OpenCL program. An OpenCL program is a collection of kernels which execute as a separate and independent program. Kernels are executed simultaneously by all threads specified for the kernel execution. The number and mapping of threads to Computing Units of the OpenCL device is handled by the host program. Each thread executing an instance of a kernel is called a work-item. Each thread or work item has unique id to help identify it. Work item can have additional id fields depending on the arrangement specified by the host program. Work-items can be arranged into work-groups. Each work group will have a unique id. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. This arrangement of work-items is shown in Figure 2.4-2.

A wide range of programming models can be mapped onto this execution model. OpenCL explicitly supports two of these models; the data parallel programming model and the task parallel programming model.

---

Figure 2.4-2 OpenCL Execution Model²

² Picture taken from [16].
2.4.3 Memory Model

The OpenCL memory space is divided into four parts:

- **Global Memory**: This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.
- **Constant Memory**: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.
- **Local Memory**: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.
- **Private Memory**: A region of memory private to a work-item. Variables defined in one work-item’s private memory are not visible to another work-item.

This division of memory spaces is shown in Figure 2.4-3.

Figure 2.4-3 Conceptual OpenCL device architecture with processing elements (PE)

The access and allocation rights of the host and kernels to these memory spaces are shown in Figure 2.4-4.

3 Picture taken from [16].
2.4.4 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models, as well as supporting hybrids of these two models. The primary programming model driving the design of OpenCL is data parallel.

2.4.4.1 Data Parallel Programming Model

Data parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. In a strictly data parallel model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. OpenCL implements a relaxed version of the data parallel programming model where a strict one-to-one mapping is not a requirement.

<table>
<thead>
<tr>
<th>Global</th>
<th>Constant</th>
<th>Local</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Dynamic allocation</td>
<td>Dynamic allocation</td>
<td>Dynamic allocation</td>
</tr>
<tr>
<td>Read / Write access</td>
<td>Read / Write access</td>
<td>No access</td>
<td>No access</td>
</tr>
<tr>
<td>Kernel</td>
<td>No allocation</td>
<td>No allocation</td>
<td>Static allocation</td>
</tr>
<tr>
<td>Read / Write access</td>
<td>Read-only access</td>
<td>Read / Write access</td>
<td>Read / Write access</td>
</tr>
</tbody>
</table>

Figure 2.4-4 Memory Region - Allocation and Memory Access Capabilities

OpenCL provides a hierarchical data parallel programming model. There are two ways to specify the hierarchical subdivision. In the explicit model a programmer defines the total number of work-items to execute in parallel and also how the work-items are divided among work-groups. In the implicit model, a programmer specifies only the total number of work-items to execute in parallel, and the division into work-groups is managed by the OpenCL implementation.

2.4.4.2 Task Parallel Programming Model

The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space. It is logically equivalent to executing a kernel on a Compute Unit with a work-group containing a single work-item. Under this model, users express parallelism by:

- Using vector data types implemented by the device,
- Enqueuing multiple tasks, and/or
• Enqueing native kernels developed using a programming model orthogonal to OpenCL.

2.5 The Susan Template Language

Code generation in OMC is implemented using the Susan Template language [12], [13], [17]. A template language is a language for specifying the transformation of structured data into a textual target data representation. Using a dedicated template code generator makes the target code generation process a lot easier and convenient. The previous method of embedding the text code to be generated directly into MetaModelica functions which are used to generate it makes debugging and modification very difficult. In contrast, code generators written using template languages are more convenient to modify and extend; exactly what is done in this thesis work.

The Susan template language is a template based code generation language developed for and used in the OpenModelica compiler. Susan templates are compiled to MetaModelica code and used for code generation in the desired target language.

2.6 Previous Work

Different approaches of parallelizing Modelica models have been studied in the past. Most of these works are based on automatic parallelization where the compiler analyzes the models to find parallelism. The ModPar module for the OpenModelica compiler studied the feasibilities of automatic parallelization with the help of task merging. This was done by Peter Aronsson for his Ph.D. thesis work [18]. Håkan Lundvall improved this work by in-lining the numerical solver and introducing software pipelining [19]. These works are targeted on parallelizing simulations on multi-core CPUs.

Some work has also been on automatic parallelization of models for execution using modern GPUs. This has been studied in Per Östlund’s Master thesis work of automatically generating parallel CUDA code for execution on NVIDIA GPUs [20].

Yet another approach to parallelization is using explicit language constructs to explicitly state parallelism in the model code. In this approach extracting parallelism is the programmer’s job instead of the compiler. This approach has been studied to some extent with NestStepModelica [21].
3 Extending the Algorithmic Subset: Extensions

3.1 Overview

As mentioned earlier most previous work regarding parallel execution support in the OpenModelica compiler has been focused on automatic parallelization where the burden of finding and analyzing parallelism has been put on the compiler. In this work, however, this responsibility is left to the end user programmer. The compiler provides additional high level language constructs needed for explicitly stating parallelism in the algorithmic part of the modeling language. This among others includes parallel variables, parallel functions, kernel functions and parallel for loops indicated by the parfor keyword. There are also some target language specific constructs and functions (in this case for OpenCL). All these extensions are collectively called ParModelica Extensions. These will all be presented in this chapter.

The focus of the current work is on parallelizing executions for highly data parallel SPMD architectures. The current implementation generates OpenCL code for parallel algorithms. OpenCL was given priority over CUDA because of its portability. Generating OpenCL code ensures that simulations can be run with parallel support on OpenCL enabled Graphics and Central Processor Units (GPU and CPU). This includes many multi-core CPUs from Intel [22] and Advanced Micro Devices (AMD) [23] as well as a range of GPUs from NVIDIA [24] and AMD [23] (for a complete list of supported devices see [25]). However explicit CUDA code generation is also planned to be supported and the current implementation provides most, if not all, constructs needed for CUDA code generation and execution as well.

3.2 Parallel Variables

Listing 3.2-1 Modelica parallel variables

```
function parvar
  Integer m = 1000;
  Integer A[m];
  Integer B[m];
  parallel Integer pm;
  parallel Integer pn;
  parallel Integer pA[m];
  parallel Integer pB[m];
end parvar;
```

Parallel variables are variables allocated in the memory space of the device used for parallel computation. OpenCL code can be executed on host CPU as well as on GPUs and CUDA code executes only on GPU. Since the OpenCL and CUDA enabled GPUs use their own local (different from CPU) memory for execution all necessary data should be available on the specific device’s memory. Even when running OpenCL computations on CPU the variables
used for parallel execution need to be explicitly stated so that the OpenCL drivers and APIs can handle them properly.

Modelica parallel variables are declared simply by preceding the variable declaration with the `parallel` keyword as shown in Listing 3.2-1.

The first three variables are allocated in the host memory. The last four variables are allocated in the memory space of the device used for parallel execution. In OpenCL case this can be the host CPU itself or any available GPU.

Parallel variables can be passed between functions as arguments. Copying data between host and parallel device memory is as simple as assigning the variables to each other. The compiler and the runtime system handle the details of the operation. The assignments shown below would all be valid in the function shown above.

<table>
<thead>
<tr>
<th>A := B</th>
<th>Serial assignment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pA := A</td>
<td>Copy from host memory (A) to parallel or OpenCL execution memory (pA), write operation</td>
</tr>
<tr>
<td>B := pB</td>
<td>Copy from parallel or OpenCL execution memory (pB) to host memory (B). read operation</td>
</tr>
<tr>
<td>pA := pB</td>
<td>Copy from one device memory (pB) to other memory space on the same device (pA).</td>
</tr>
<tr>
<td>pm := m, n := pm, pn := pm</td>
<td>Scalar versions of the above three assignments.</td>
</tr>
</tbody>
</table>

Parallel variables can only be declared inside a serial function. Variables in kernel and parallel functions (discussed below) are parallel by default and do not need to be explicitly specified.

The current implementation has some restrictions on parallel variables

- Any computational algorithmic statements involving parallel variables should be in parallel for loops. These include arithmetic operations on scalar parallel variables and indexing of parallel arrays. Assignments are allowed anywhere in the algorithmic section of modelica. This constraint is due to the target languages (OpenCL and generally most GPGPU paradigms) and would not probably change in the future.
- Parallel variables cannot be initialized with default values. The first declaration in Listing 3.2-1 shows a default value initialization. Some initialization options for arrays currently work. However it is not properly tested and is not supported with this implementation. Default initialization will be supported soon.

In this implementation all parallel variables are declared in the global memory space. See 4.4.4 and 4.3.1.1 for more information.
3.3 Parallel Functions

Modelica parallel functions in this implementation correspond to OpenCL functions defined in kernel files or CUDA’s `__device__` functions. These are functions available independently to every thread executing on a device. Parallel functions in Modelica are defined in the same way as normal functions except that they are preceded by the `parallel` keyword as shown in Listing 3.3-1.

The code for parallel functions is generated in the target language for parallel execution. In the current implementation OpenCL code is generated.

Parallel functions have some constraints

- They cannot have parallel for loops in their algorithm.
- They cannot have any explicitly declared parallel variables. Parallel functions execute on the parallel device’s memory. Therefore every variable in parallel functions is already parallel and is allocated in the device memory.
- They can only call other parallel functions or supported built-in functions.
- Recursion is not allowed.
- They can only be called from a body of a parfor loop or from kernel functions i.e. they are not directly accessible to serial parts of the algorithm.

3.4 Kernel Functions

The code for kernel functions is generated in the target language for parallel execution. In the current implementation OpenCL code is generated.

Kernel functions have some constraints

- They cannot have parallel for loops in their algorithm.
- They cannot have any explicitly declared parallel variables. Parallel functions execute on the parallel device’s memory. Therefore every variable in parallel functions is already parallel and is allocated in the device memory.
- They can only call other parallel functions or supported built-in functions.
- Recursion is not allowed.
- They can only be called from a body of a parfor loop or from kernel functions i.e. they are not directly accessible to serial parts of the algorithm.
Kernels functions correspond to OpenCL and CUDA __kernel functions and __global__ functions respectively. These are entry functions to execution on a device. They can be called from serial parts of Modelica code to start parallel execution on a parallel device. Kernel functions are independently executed by every thread in the launch.

Modelica kernel functions are defined in the same way as normal functions except that they are preceded by the kernel keyword. A possible implementation example is shown in Listing 3.4-1 below. multiply is the parallel function listed in Listing 3.3-1. The special built-in utility function ocl_get_global_id is discussed in section 3.7.

The number of threads to be used for the kernel execution can be set by using the function ocl_set_num_threads discussed in 3.7. This function should be called before any kernel call since Modelica kernel functions will set the number of kernels to the default before returning. Otherwise the default number of threads will be used to execute the kernel function. The default is the maximum number of threads of the parallel execution device. The current implementation supports only one dimensional arrangement of one dimensional work groups.

There are some constraints on kernel functions:

- They cannot have parfor loops in their algorithm body.
- They cannot have any parallel variables. Kernel functions execute on the parallel device’s memory. Therefore every variable in kernel functions is already parallel and is allocated in the device memory.
- They can only call parallel functions or supported built-in functions. They cannot call other kernel functions.
- They cannot be called from a body of parfor loop or from other kernel functions.

### 3.5 Parallel For Loop – parfor

Modelica parallel for loops are basically normal for loops with some additional constraints on the body of the loop. These constraints are needed to make sure the iterations can be run simultaneously and independently without any specific order while giving the desired result, i.e., no loop-carried dependencies from one iteration to the next. A Modelica parallel for loop is identified by the keyword parfor as shown in Listing 3.5-1 below. multiply is the parallel function listed in Listing 3.3-1.

The iterations of a parfor loop are equally distributed among available processors. If the range of the iteration is smaller than or equal to the number of threads the parallel device supports, each iteration will be done by a separate thread. If for example our device supports 1024 threads and the loop has 512 iterations then 512 threads will be launched and will each execute a separate iteration. If the number of iterations is larger than the number of threads available then some threads might perform more than one iteration. If for example we have a loop with 768 iterations and a device with a 512 thread limit then 512 threads will be
launched which will execute iterations 1 to 512. The remaining 256 iterations will be done by
the first 256 threads out of the 512 as a second step. In future enhancements parfor will be
given the extra feature for specifying the desired number of threads explicitly instead of
automatically launching threads as described above.

Listing 3.5-1 Modelica parallel for loop

```modelica
parfor i in 1:m loop
  for j in 1:pm loop
    ptemp := 0;
    for h in 1:pm loop
      ptemp := multiply(pA[i,h], pB[h,j]) + ptemp;
    end for;
    pC[i,j] := ptemp;
  end for;
end parfor;
```

The choice of target architecture and language has put some constraints on parfor loops.

- All variable references in the loop body must be to parallel variables.
- Iterations should not be dependent on other iterations – no loop-carried dependencies.
- All function calls in the body should be to parallel functions or supported built-in
  functions only.
- The iterator of a parallel for loop must be of integral type.
- The start, step and end values of a parallel for loop iterator should be integral types.

The first constraint is needed since OpenCL executions can be run on another device than the
host CPU where the rest of the simulation code is being executed. To make sure that desired
data is made available in the device memory before start of parallel execution this rule must
be obeyed. If for example OpenMP was used for the parallel execution then we would not
need this constraint since OpenMP code always runs on the CPU with threads accessing CPU
shared memory. There is a reason why the compiler does not automatically detect and copy
all variables used or referenced in the loop body. Even if it would be reasonable to
automatically copy all needed variables to the device memory, which variables should we
copy back? Copying all variables back after the execution of the parfor loop means that we
have to perform a lot of unnecessary expensive copies. In addition this gives the programmer
a better control over the rather expensive memory operations.

### 3.6 Built-in Functions

Some built-in functions have been extended to accept parallel variables as arguments. Accepting
parallel arguments means that the computations of the function will be performed
on the parallel execution device instead of a single thread on the host CPU. The return values
from these extended parallel built-in functions are currently only parallel variables. For
example consider the built-in function `transpose` which is used to compute the transpose of a
matrix. If a serial matrix is passed to this function as argument the computation will be done on the host CPU and a serial matrix is returned. However if a parallel matrix is given as argument then the computation will be done in parallel on the available device. The return variable will be a parallel variable.

The serial/parallel combination of arguments/return values should be diversified in the future to give more options for the programmer. The compiler should detect the types assigned to return variables and handle any necessary copying automatically.

The rules set above on serial/parallel arguments/return-values combination are not hard rules than more of choice of implementation and might change in the future. However according to the current implementation any built-in function call involving parallel arguments will return parallel variables.

### 3.7 Synchronization and Thread Management

A number of functions related to Synchronization and thread management are also available. These functions are very similar to the OpenCL work-item function (see [26]). These functions are:

- **ocl_set_num_threads(Integer, Integer)**: is used to specify the number of threads to be used for a kernel function execution. The current implementation supports dimensional arrangement of one dimensional work groups. For example calling this function with 1024 and 64 will create a one dimension 1024/64=16 one dimensional work-groups of 64 work-items or threads each. This function should only be called from inside a serial function. It should be called prior to any kernel function call if the number of threads is to be specified for the kernel. Otherwise the kernels will execute with the default number of threads which is the supported maximum. This function is also overloaded to take just one Integer argument. In this case the given integer value will specify the number of threads or work-items to be launched. The actual arrangement of these threads into work-groups will be decided automatically by the OpenCL runtime system. This usage is shown in Listing 3.8-1.

- **ocl_get_global_id(Integer)**: returns the global id of the thread currently executing the function or kernel. This function should only be called from inside a parallel function or kernel function. The input argument should currently be 0 i.e. only one dimension supported.

- **ocl_get_local_id(Integer)**: returns the local id of the thread currently executing the function or kernel. This function should only be called from inside a parallel function or kernel function. The input argument should currently be 0 i.e. only one dimension supported.

- **ocl_get_global_size(Integer)**: returns the total number of threads currently executing the function or kernel. This function should only be called from inside a parallel
function or kernel function. The input argument should currently be 0 i.e. only one dimension supported.

- `ocl_get_local_size(Integer)` : returns the total number of threads in the work-group of the calling thread. This function should only be called from inside a parallel function or kernel function. The input argument should currently be 0 i.e. only one dimension supported.

- `ocl_global_barrier()` : is used to synchronize all threads currently launched. All threads must reach this point before any thread is allowed to continue. This function should only be called from inside a parallel function, kernel function or body of parfor loop.

### 3.8 OpenCL Functionalities

Automatically generated code might not always be as efficient as a manually written code. If the need arises for a finer control over operations like data distribution and synchronization built-in functions are available for compiling and executing user-written OpenCL code directly from another source file.

- `oclbuild(String)` : This function takes only one String argument. The argument is the name of the OpenCL source file to be built. It returns an integer (type defined as `cl_program` for clarity) which is used as an id of the built program. This id is used in consequent calls to refer to this OpenCL program. Users can have as many as 10 files built in the same Modelica code (10 within scope) at a time. This limit can be increased anytime. It is just assumed to be enough.

- `oclkernel(oclprogram, String)` : This function takes two arguments. The first one is the id (Integer) of the OpenCL program built by a previous call to `oclbuild()`. The second argument is the name the kernel or function in that specific program which the user wants to create a kernel for. Users can create as many as 10 kernels at any time. This function also returns an Integer (type defined as `cl_kernel`) for the same reason as `oclbuild()`.

- `oclsetargs(oclkernel,...)` : This function is used to set arguments to an OpenCL kernel. It takes a variable number of arguments. However the first argument should be the id of the kernel to be executed (an Integer or `cl_kernel`). After the first argument a variable number of parallel variables follow. These are the actual arguments to the OpenCL kernel. This function does not return anything.

- `oclexecute(oclkernel)` : This function is used to execute a kernel. It takes the id of the kernel as an argument. After executing the kernel the user can copy back any of the arguments attached to the kernel earlier to obtain just the desired results.

Users can declare OpenCL programs as `cl_program` and kernels as `cl_kernel`. These types are just normal type definitions of Integer made just for clarity purposes. They are included with built-in functions so they can be used readily any time. A simple usage of
theses utility functions is shown in Listing 3.8-1 below. The OpenCL kernel function can perform any operation as long as the arguments match.

All the above operations are synchronous in OpenCL jargon. They will return only when the specified operation is completed. Further functionality is planned to be added to these functions to provide better control over execution.

Listing 3.8-1 Executing user written OpenCL files.

```plaintext
function userFile
  input Integer m;
  parallel input Integer pA[m,m];
  parallel input Integer pB[m,m];
  parallel output Integer pC[m,m];

  cl_program pro;
  cl_kernel ker;

algorithm
  //build the opencl program from the file
  pro := oclbuild("testmat.cl");

  //create the desired kernel from
  //the available kernels in the built program
  ker := oclkernel(pro, "user_func");

  //set the arguments to the kernel created
  oclsetargs(ker,pA,pB,pC,m);

  //set m threads to run.
  ocl_set_num_threads(m);

  //run the kernel
  oclexecute(ker);

end userFile;
```
4 Extending the Algorithmic Subset: Implementation

This thesis work has introduced new language constructs to the Modelica algorithm subset. Introducing a new language construct requires adding new keywords to the language subset, recognizing and representing these keywords with proper data structures, manipulating and propagating the data structures and finally generating code in the desired target language. These operations require modifications of multiple phases or modules starting from the parser down to the code generator. Consequently this thesis work is not a separate module added to compilation flow. Instead it can be thought as a collection of small extensions to each relevant module. A simplified structure of the new compilation process is shown in Figure 3.8-1.

The OpenModelica compiler is internally divided in to two main parts; the Front-end and the Back-end. These two main parts are composed of a number of modules. A simplified overall structure showing the most important modules is depicted in Figure 4.1-1 below. The discussions in the next sections are organized according to this structure.
4.1 Front-end

The OMC front-end is comprised of the modules responsible for lexical and syntax analysis\(^4\), type checking, handling object-oriented operations like inheritance and modification, and fixing package inclusion and look up as well as import statements.

The new extensions are added in a way that parallel variables and functions as well as kernel functions can be processed with the same syntax and semantic rules as their serial counterparts. This is achieved by expressing parallelism as a separate attribute which is processed only when needed. This will be discussed in detail in section 4.1.1 below. Handling parallel attributes this way, as per need basis, results in a simplified implementation in which most of the work in the Front-end of the compiler is related to properly propagating the additional information so that it is available for the Back-end.

![Main modules of the OpenModelica Compiler](image)

Figure 4.1-1 Main modules of the OpenModelica Compiler\(^5\)

Front-end of OMC is composed of a number of modules and files. Modification has been done to most of the files and to almost all modules. The discussion in the following sections will provide the theory behind the implementation with a few code examples. The discussions are limited to extensions done to the main modules. Interested reader can get the whole source code from the OpenModelica repository [27].

\(^4\) The lexer+parser is written in C and is actually implemented as an external module outside of the front-end. However technically it belongs to the front-end of the Compiler.

\(^5\) Picture taken from [29].
4.1.1 Keywords: Lexical and Syntax Analysis

The OpenModelica Complier (OMC) uses ANTLR \[28\] for lexical and syntax analysis (lexer and parser)\(^6\). This section, however, will not include a detailed description of ANTLR. The main focus here will be the motivation and the consequences of the preferred implementation with brief examples from the parser generator.

In most programming languages, including Modelica, variables and functions have different attributes attached to them. For example in C a variable can be a `const, static, public, private` and so on.

Modelica variables attributes include those describing direction, variability, flow and stream. Direction describes whether a variable is an input, output or neither. This attribute is described by the keywords `input` and `output`. This can be seen on Appendix A.1. Variability is described by the keywords `discrete, parameter and constant` (having none of these makes the variable `variable`). Flow variables should have the `flow` keyword preceding them otherwise they are considered as non-flow variables. The same goes for `stream`. The divisions: direction, variability, flow and stream can be considered mutually exclusive. That means a variable can be preceded by at most one keyword from each group, for example a variable can be `input, output or none`. It cannot be both input and output. However as far as the parser is concerned an `input/output` variable can also be `constant, parameter or discrete`.

In much the same way as variables, functions also have attributes.

4.1.1.1 parallel and kernel

Parallelism is a new attribute added to Modelica variables and functions. Variables and functions can now be `parallel`. The parallelism is more like the stream and flow keywords and stands in its own division i.e. variables and functions are either parallel or not. For example it is possible to have a `parallel input` variable as shown Listing 3.8-1. The ANTLR generated parser recognizes the parallel keyword and constructs the corresponding data types in the Abstract Syntax Tree of OMC called Absyn.

Listing 4.1-1 shows part of the implementation for handling the parallel keyword (parallel variables).

\[ \text{Listing 4.1-1} \]

\(^6\) This thesis work is based on OpenModelica 1.6 which uses ANTLR.
In addition to parallel, Modelica functions can also be kernel functions. The parser identifies these by the `kernel` keyword. Listing 4.1-2 shows part of the implementation for handling parallel and/or kernel functions. Note that according to the parsing rules a function can be either a `parallel` or `kernel` but not both. It can also be neither i.e. serial function.

Listing 4.1 ANTLR parsing rules for parallel variables
```
type_prefix returns [void* flow, void* stream, void* parallel, void* variability, void* direction] : 
  (fl=FLOW|st=STREAM|prl=T_PARALLEL)? 
  (di=DISCRETE|pa=PARAMETER|co=CONSTANT)? (in=T_INPUT|out=T_OUTPUT)? 
  
  { 
    $flow = mk_bcon(fl); 
    $stream = mk_bcon(st); 
    $parallel = mk_bcon(prl); 
    $direction = in ? Absyn__INPUT : out ? Absyn__OUTPUT : Absyn__BIDIR; 
  };
```

Listing 4.1-2 ANTLR parsing rules for parallel and kernel functions.
```
class_type returns [void* ast] : 
  ( CLASS { ast = Absyn__R_5fCLASS; } 
  | OPTIMIZATION { ast = Absyn__R_5fOPTIMIZATION; } 
  | MODEL { ast = Absyn__R_5fMODEL; } 
  | RECORD { ast = Absyn__R_5fRECORD; } 
  | BLOCK { ast = Absyn__R_5fBLOCK; } 
  | ( e=EXPANDABLE )? CONNECTOR { ast = e ? Absyn__R_5fEXP_5fCONNECTOR : Absyn__R_5fCONNECTOR; } 
  | TYPE { ast = Absyn__R_5fTYPE; } 
  | T_PACKAGE { ast = Absyn__R_5fPACKAGE; } 
  | (prl=T_PARALLEL|ker=KERNEL)? FUNCTION { ast = Absyn__R_5fFUNCTION(mk_bcon(prl),mk_bcon(ker)); } 
  | UNIONTYPE { ast = Absyn__R_5fUNIONTYPE; } 
  | OPERATOR { 
      (f=FUNCTION | r=RECORD)? 
      { 
        ast = f ? Absyn__R_5fOPERATOR_5fFUNCTION :
          r ? Absyn__R_5fOPERATOR_5fRECORD :
              Absyn__R_5fOPERATOR; 
      } 
  });
```

4.1.1.2 `parfor`: parallel for loop

In the front-end of OMC parallel for loops are handled in a very similar way as normal for loops. The only difference in the parser regarding parfor loops is that it will construct the `parfor` record in the AST instead of the one for normal `for` loops. Right now the parser (and the rest of OMC) is capable of recognizing and handling parallel for loops in both equation
and algorithm parts of Modelica code. However the current implementation is intended and tested for the algorithm part only. The parser rules for parfor are shown in Listing 4.1-3.

Listing 4.1-3 ANTLR parsing rules for parfor

```plaintext
parfor_clause_e returns [void* ast] :
  PARFOR is=for_indices LOOP es=equation_list T_END PARFOR {ast =
    Absyn__EQ_5fPARFOR(is,es);} 

parfor_clause_a returns [void* ast] :
  PARFOR is=for_indices LOOP as=algorithm_list T_END PARFOR {ast =
    Absyn__ALG_5fPARFOR(is,as);} 
```

4.1.2 Module Absyn: The Abstract Syntax Tree

Absyn is the abstract syntax representation of Modelica source code. The Absyn, like most of OMC, is written in MetaModelica. It is implemented in the file Absyn.mo in the Compiler/Frontend directory. The Absyn mainly contains the data types and functions used for constructing the AST as well as some function for printing the AST. Only parts related to the extensions done in this thesis work will be discussed here. For a more detailed description of the Absyn module please refer to [29].

The record ATTR in the ElementAttributes uniontype has been extended with a new Boolean variable called parallelPrefix which is set to true or false by the parser depending on the parallelism of the variable related to it. This field is used by subsequent modules to identify whether a variable is parallel or not. The R_FUNCTION record, used to indicate that a CLASS is restricted to FUNCTION, has been extended with two Boolean variables indicating whether it is a parallel or kernel function.

New record datatypes have been added to represent parallel for loops in Equation and Algorithm uniontypes. The functions traverseEquation and traverseAlgorithm, which are used to traverse the equation and algorithm parts of Modelica code respectively, are also extended to support parallel for loops. Parallel for loops are traversed in almost similar ways with the same semantic rules as normal for loops.

There are also some additions related to properly printing the extensions with the AST. A summary of the main additions is shown in Listing 4.1-4.
4.1.3 Module Algorithm

Algorithm module handles algorithm sections of a model code. It is used by the Inst Module (see 4.1.7) to represent algorithms. This module only builds the data structure. A new function called `makeParFor` is added to it. This function type checks the iterator and body parts in the parallel for loop and creates the `DAE.STMT_PARFOR` construct. This function is called by the `instParForStatement_dispatch` function in `InstSection.mo` to instantiate a parfor loop (see 4.1.7).

4.1.4 Module Ceval: Constant Evaluation

This module handles constant propagation and expression evaluation, as well as interpretation and execution of user commands. In `Ceval.mo` there is a function called `cevalCallFunction`. This function is used to evaluate call expressions. A small modification is done on `cevalCallFunction` to properly propagate parallel and kernel functions calls by analyzing the corresponding prefixes.
Cevalfunction.mo contains the implementation for constant evaluating DAE.Function objects. A new function called evaluateParForStatement is added to this file. It is used to constant evaluate elements of parfor statement like the start and step values.

### 4.1.5 Module ClassInf

This module is related to class inferences and restrictions. The only addition to this module is support for properly printing the new extensions attributes.

### 4.1.6 Module DAE: DAE Equation Management and Output

The DAE is another intermediate representation containing only flat Modelica. These include equations, algorithms, variables and functions. This module provides the data structures for the DAE intermediate representation as well as functions for handling the DAE structure. It also provides some functions to other modules.

The major additions or modifications to this module are explained here. First the data type definitions which are added in DAE.mo are discussed.

- The record FUNCTION in the Function uniontype represents a Modelica function in the DAE representation. This record has been extended with two Boolean variables to identify kernel and parallel functions. This are set based on the corresponding prefixes from SCode (see 4.1.8) when the DAE is constructed.
- New record, STMT_PARFOR, is added to Statement uniontype which represents an Algorithm statement. The new record represents a parfor loop.
- The uniontype Attributes, which represents the attributes of elements, now contains information about parallelism too.
- A new uniontype Parallel is defined. This record is used to identify parallelism in variables. Based on this uniontype variables can be identified as either PARALLEL or NON_PARALLEL.
- Record VAR in the uniontype Element represents a variable in the DAE. This record now contains the Parallel record, defined above, as an additional attribute.

The file DAEUtil.mo provides the utility functions for handling the DAE. The main additions to this file are discussed below.

- Four new functions for extracting variables depending on parallelism are added. These are used by the backend SimCode module (see 4.2.1.1) to extract and handle parallel variables before code generation. The functions are shown below.
  - getParallelVars: This function is used to extract parallel variables from a list of input variables.
  - getParallelArgs: This function extracts all parallel variables which are also arguments to functions i.e. parallel input variables.
- **isParallelVar**: this function succeeds if a given variable is parallel. It is used by
  `getParallelVars` to extract parallel variables.
- **isParallelArg**: succeeds if a given variable is a parallel argument. It is used by
  `getParallelArgs`.

- The function `traverseDAEEquationsStmts` has been extended to handle parallel for loops or `STMT_PARFOR`.
- The `FuncArg` tuple used to represent function arguments now also contains information about parallelism of arguments. This is needed to extract parallel arguments in the SimCode module.
- In addition to the extensions mentioned above a considerable amount of modification is done to make sure that attributes and other information is propagated properly through the DAE.

The file DAEDump.mo is used to print or output the DAE. This file has been modified to support the corresponding changes or extensions.

### 4.1.7 Module Inst: Code Instantiation/Elaboration

The Inst module is one of the largest and complicated parts of the OMC. It is responsible for instantiating Modelica models. It takes the SCode representation of a model and generates a DAE representation after elaborating the components, flattening inheritance and generating equations. For a more in depth and detailed description of this module please see [29].

The parts of instantiation related to this thesis work are the ones involving variables and functions. A considerable amount of change has been done to this module to properly process and transfer the new extension attributes from SCode to DAE. A new function called `instParForStatement` is added to instantiate parfor loops. This function uses the `instParForStatement_dispatch` function. The later function does the actual job of instantiating a parallel for loop. It elaborates the iterators, checks for constants and dispatches the instantiation of the body. Then it uses the `makeParFor` function from the Algorithm module (see 4.1.3) to make a parallel for loop in the DAE.

One thing to mention here is overloaded parallel functions. Currently parallel overloaded functions are not allowed and will not be instantiated. However overloading is a very good feature to have. Especially if we have a function that we want to call from inside kernel functions or parfor loops as well as from normal serial parts of the algorithm. Overloading should be supported in the future.

### 4.1.8 Module SCode

The abstract syntax tree or Absyn is translated to another intermediate representation called SCode. This is done in order to make translation of programs easier. The Absyn is a very similar to the parsing tree and keeps the structure of the source file. The SCode provides a
better representation for subsequent operations on the intermediate form. According to the OpenModelica System Documentation [29] the SCode structure differs from the Absyn in the following respects

- All variables are described separately. In the source and in the AST several variables in a class definition can be declared at once, as in `Real x, y[17];`. In the SCode this is represented as two unrelated declarations, as if it had been written `Real x; Real y[17];`.
- Class declaration sections. In a Modelica class declaration the public, protected, equation and algorithm sections may be included in any number and in any order, with an implicit public section first. In the SCode these sections are collected so that all public and protected sections are combined into one section, while keeping the order of the elements. The information about which elements were in a protected section is stored with the element itself.

The data type extensions made to the SCode are almost the same as to those made for the Absyn (see 4.1.2). The methods needed for properly translating the Absyn representations of the new extensions to the SCode representation are added to the file `SCodeUtil.mo`. These include methods for translating parallel for loops, attributes of parallel variables and handling restrictions of Class corresponding to parallel and kernel functions.

4.1.9 Module Static

This module is used for static semantic analysis of expressions. The module is also responsible for elaborating function arguments by type checking positional and named arguments. The functions related to this later operation have been modified to support parallel variables since parallel variables can also be function arguments. Note that there is no error reporting mechanism implemented right now to detect if a given function argument matches the parallelism of the actual defined argument. This error, right now, will only be caught by the C compiler when compiling the generated C code to an executable.

The module also contains elaboration handler for built-in variables. The function `elabBuiltinHandler` is responsible for this. All new built-in functions are handled here. These functions include the special functions used for management of OpenCL kernels like `oclBuild` which are explained in section 3.6. In addition to the special function the additional parallel versions of normal functions like `partranspose` are also elaborated here.

4.2 Back-end

The resulting flat Modelica DAE form the Front-end is passed to the Back-end. The Back-end sorts equations, performs optimizations and index reduction operations and prepares the model for code generation. Since this thesis work and implementation involves support for
Algorithm parts only, the most relevant parts of the Back-end to this thesis work are code generation related operations, specifically the SimCode module.

The SimCode module is extended to properly represent and handle the new language constructs. These extensions are discussed in the next sections.

### 4.2.1 Module SimCode

The SimCode module takes the backend DAE and prepares it for code generation using Susan templates. It generates the SimCode data structure which is convenient for Susan code generation. The SimCode data structure contains all necessary information for proper code generation in the desired target language. In addition to defining the SimCode data structure the module also provides many functions for manipulation of the DAE as well as the SimCode representations.

Since this module is highly related to actual code generation there are a considerable amount of additions and extensions made to it. For a clear view of these extensions it is more appropriate to divide them based on the target extension and discuss them separately. First
any modification done regarding parallel variables will be discussed. Then parallel function
and kernel function related modifications are discussed in turn. The data structures
representing parallel variables, serial functions, parallel functions and kernel functions are
shown in Listing 4.2-1 above.

4.2.1.1 Parallel variables and Serial Functions

The VARIABLE record in the variable uniontype of the SimCode data structure has been
extended with a Boolean field to identify parallelism. This is the same as what is done in
Absyn, DAE and SCode.

The function typesSimFunctionArg is used to translate a function argument from
Types.FuncArg (type definition of DAE.FuncArg tuple discussed in 4.1.6 third bullet of second
series) to SimCode’s VARIABLE data structure. The function now preserves the information
about parallelism of each argument during the conversion. This function is used by
elaborateFunctions to get a list of function arguments.

Listing 4.2-2 SimCode.elaborateFunctions, parallel variables in serial functions.

```java

case (DAE.FUNCTION(path = fpath, 
    functions = DAE.FUNCTION_DEF(body = daeElts)::_,type_ = tp as 
    (DAE.T_FUNCTION(funcArg=Args, funcResultType=restype), _), 
    partialPrefix=false,parallelPrefix=false,kernelPrefix=false), 
    rt, recordDecls, includes, libs)
equation
    outVars = Util.listMap(DAUEUtil.getOutputVars(daeElts),daeInOutSimVar);
    prlVars = Util.listMap(DAUEUtil.getParallelVars(daeElts),daeInOutSimVar);
    funArgs = Util.listMap(args, typesSimFunctionArg);

    (recordDecls,rt_1) =elaborateRecordDeclarations(daeElts,recordDecls, rt);
    vars = Util.listFilter(daeElts, isVarQ);
    varDecls = Util.listMap(vars, daeInOutSimVar);
    algs = Util.listFilter(daeElts, DAEUtil.isAlgorithm);
    bodyStmts = Util.listMap(algs, elaborateStatement);

then
    (FUNCTION(fpath,outVars,funArgs,varDecls,bodyStmts, 
      prlVars),rt_1,recordDecls,includes,libs);

```

All parallel variables in a serial Modelica function which are not function arguments are
gathered in to a separate list called prlVars (part of the FUNCTION record) as shown in
Listing 4.2-2. This is done by using the DAEUtil.getParallelVars function from the DAE
module. Note that the parallel variables are not removed from the list of all variables of the
function. A new copy of them is created and kept. They are left in the general list of variables
in order to allow the proper declaration and initialization.
Creating a new list containing copies of parallel variables is needed for proper operations of parallel for loops. Every parallel for loop is converted to a separate kernel function. These kernel functions take the parallel variables in the function as arguments. Please refer to 4.3.1.2 for a detailed description why this separation is needed and how the code parfor loops is generated.

In the current implementation, each of these kernels generated for a parallel for loop takes all the parallel variables defined in the function scope of the for loop. For example in a function containing two parallel for loops and a couple of parallel variables, all parallel variables are given as arguments to both kernels generated for each parfor loop. This is done since there is no way of telling which subset of the whole parallel variables in the function is needed by each parfor loop. In future work a mechanism should be devised to identify exactly which parallel variables are needed by which for loop. One possible way to achieve this would be to extend the syntax of the parfor loop to specify the variables it is going to use in its body. Another solution would be to try and identify the variables by doing a proper scanning of the parfor loop body. This will help reduce the number arguments that have to be set to each kernel of a parfor loop. Remember that the current implementation does not introduce any unnecessary copy or transfer of data. Data is copied between device and host based on assignments only. The situation discussed above only increases the number of arguments which makes the code a little harder to understand. A couple of warning messages about variables not being used is also generated by the OpenCL complier. These messages are currently suppressed by using a compiler flag.

4.2.1.2 Parallel functions

Parallel functions are declared using the parallel keyword. After passing through the different intermediate representations and operations, the backend DAE_FUNCTION construct reaches the elaborateFunctions function in SimCode. In previous modules parallel functions were identified from other types of functions by the Boolean variable parallelPrefix. In the SimCode however there is a new record representing just parallel functions. This record is called PARALLEL_FUNCTION. This clear separation makes the Susan code generation easier. This new record is similar to the normal FUNCTION record representing normal function except that the new record does not contain the list prlVars. This is not surprising since all variables in a parallel function are expected to be parallel. Remember that parallel functions are executed on the OpenCL or CUDA device which means all data should already be on device memory before they are called.

Parallel functions, in SimCode, are elaborated by the second case statement of elaborateFunctions. This is almost the same operation as what is shown in Listing 4.2-2, except now there is no separation in to the prlVars list and instead of creating a FUNCTION record at the end a PARALLEL_FUNCTION record is created in the SimCode.
Kernel functions need a bit different approach since they act different from both parallel and serial functions. For the same reasons as parallel functions, Kernel functions are allowed to contain only parallel variables. Therefore, unlike serial functions, we don’t need any explicit separation of parallel and serial variables.

A significant difference of kernel functions from parallel and serial functions is the lack of output or return variables. In OpenCL and CUDA Kernel functions are always void. However in the Modelica (this thesis work) Kernel functions are allowed to have output variables. As shown in Listing 4.2-2 output variables are extracted in to a separate list for serial and parallel functions. However this extraction means only creating a copy of the variables in a new list. The output variables will also be present in the list varDecls. The list varDecls is used to generate the declarations of variables. The extracted list is used to generate code specific to only output vars. Regarding kernel function the output variables should not be present in the declarations since there cannot be any return statement in the actual kernel. Instead calling kernel functions and retuning variables from them is handled by the kernel interface which is discussed in detail in section 4.3.1.4. For now it is enough to assume that there is a need to not include output variables in the declaration list. This is achieved by using a function called isVarNotInputNotOutput which selects only non-input and non-output variables. When creating the varDecls list this function is passed to the Util.listMap function as argument instead of the daeInOutSimVar function. In addition the lists containing output variables and function arguments are generated with slight modifications.

Two new functions, daeInOutSimVarKernelInterface and typesSimFunctionArgKernelInterface, are used to generate the lists outVars and funArgs respectively. These functions, in
addition to extracting the desired variables, mark all variables as parallel. This is done in order to be able to use the same Susan template functions for all declarations instead of defining a new template function just for variables in kernels and kernel interfaces.

Parts of the function `elaborateFunctions` responsible for handling kernel functions is shown in Listing 4.2-3.

The SimCode including the above parallel execution related information as well as a large amount of other information is passed to the template of the desired target language. In this case it will be passed to SimCodeC for C code generation. SimCodeC is automatically generated from the SimCodeC.tpl template file. SimCodeC.tpl is discussed in detail in the next sections.

### 4.3 Susan Code Generation

Code generation in OMC is implemented using the Susan Template language. Using a dedicated template code generator makes the target code generation process a lot easier and convenient. The traditional methods of embedding the text code to be generated directly in to functions which are used generate them make debugging and modification very difficult, time consuming and error prone. In contrast code generators written using template languages are more convenient to modify and extend; exactly what is done in this thesis work. For more detailed information about Susan the reader is referred to [12], [17].

The OpenCL programming language is based on C99 standard plus some extensions. Therefore OpenCL code generation is very similar to C code generation. Support for C code generation is already available for the OpenModelica Compiler. This C code generation has been extended to generate OpenCL code for the new parallel extensions of OMC. C code generation is now C+OpenCL code generation and is used when models are pure serial as well as parallel.

A significant amount of modifications and extensions have been done to the code generation, more specifically to the file `SimCodeC.tpl`. In order to make the discussions clear and less confusing the changes are discussed according to the target extension. A general overview of the structure of `SimCodeC.tpl` and code generation process is given first. Then modifications involving parallel variables are discussed. Parallel for loops, parallel functions and kernel functions are explained in turn. A simple model of matrix multiplication is also provided to illustrate the code generation process. This is shown in Appendix A.1. The model demonstrates two approaches of implementing matrix multiplication. First approach uses a parallel for loop and a parallel function. The second one uses kernel function instead of the parallel for loop. Relevant parts of the code generated for these two models are shown in Appendix A.2 - A.5. They are referenced intensively in the discussions.
4.3.1 SimCodeC

Due to the tree like nature of most algorithms, template code generation is also a tree like process. Generation starts at the root or the highest structure and traverses all branches one after the other. Since template code generation is based on function calls and returns traversal of a code branch is performed in one direction at time as shown in Figure 4.3-1. The steps down and up the tree correspond to template function calls and returns respectively. Necessary data structures are passed down the tree as function arguments. Buffers are used to collect information from lower level steps and provide it to higher level steps. These buffers provide a way to generate specific code fragment in a desired location.

Figure 4.3-1 Code generation traversal

Note that in the simplified arrangement shown in Figure 4.3-1 nodes correspond to each function call not to the function itself. Multiple nodes in this arrangement can be different calls to the same template function. Calls can also be recursive. In fact once the Susan C code generation gets to the algorithmic parts of the Model it becomes very recursive and net like procedure. Recursion can be direct as well as indirect. For example consider the following arrangement. An If clause containing a for loop which contains a number of if clauses. The template function responsible for If clauses will be called first. This will call the template function for for loop. The for loop function calls the function for if and so on. This example case shows an indirect recursion which is very common in the code generation. In addition there is no limit or specification on the number of branches or function calls. Figure 4.3-1 is just a simplification.

Susan Code generation starts with a creation of a text file as shown in Listing 4.3-1. A number of files are created for a single model. Each of these files is used for distinct portions of the simulation process. These files are explained in the subsequent sections. After creating the
text files the function responsible for each text file is called. This functions output their results directly into the text files.

Two additional files are generated for parallel operations. These files will be named as `ModelName_parallelFunctions.cl` and `ModelName_kernels.cl`. The first file contains code for parallel functions and kernel functions plus the header definitions and includes the `helper.cl` file discussed in 4.4.2. The second file contains kernels created for parallel for loops and includes the `ModelName_parallelFunctions.cl` file. The `ModelName_kernels.cl` file is built by the OpenCL compiler. As a result code in both the `ModelName_parallelFunctions.cl` file and `helper.cl` file will be built and compiled. This compiling operation is done only once per simulation. The binary form of the OpenCL program is made available to functions by the global variable `ocl_kernels_program` defined in the OpenCL runtime library.

Listing 4.3-1 Susan Code generation top level

```cpp
template translateModel(SimCode simCode)
    "Generates C code and Makefile for compiling and running a simulation of a Modelica model."
    ::= match simCode
        case SIMCODE(modelInfo=modelInfo as MODELINFO(_)) then
            let &kernels = buffer "" /*BUFK*/
            let()= textFile(simulationFile(simCode, &kernels /*BUFK*/),
                '<%fileNamePrefix%>.cpp')
            let()= textFile(simulationFunctionsHeaderFile(fileNamePrefix,
                modelInfo.functions, externalFunctionIncludes, recordDecls),
                '<%fileNamePrefix%>_functions.h')
            let()= textFile(simulationParallelFunctionsFile(fileNamePrefix,
                modelInfo.functions, literals, &kernels /*BUFK*/),
                '<%fileNamePrefix%>_parallelFunctions.cl')
            let()= textFile(simulationFunctionsFile(fileNamePrefix,
                modelInfo.functions, literals, &kernels /*BUFK*/),
                '<%fileNamePrefix%>_functions.cpp')
            let()= textFile(recordsFile(fileNamePrefix, recordDecls),
                '<%fileNamePrefix%>_records.c')
            let()= textFile(simulationMakefile(simCode),
                '<%fileNamePrefix%>.makefile')
            let()= textFile(simulationKernelsFile(fileNamePrefix, &kernels),
                '<%fileNamePrefix%>_kernels.cl')
            if simulationSettingsOpt then //tests the Option<> for SOME()
                let()= textFile(simulationInitFile(simCode),
                    '<%fileNamePrefix%>_init.txt')
            end translateModel;
```

### 4.3.1.1 Parallel variables and Serial functions

The Code for Modelica serial functions is generated in the `ModelName_functions.cpp` file. The template function responsible for this portion of code generation is
**SimulationFunctionsFile.** This function generates the code for each serial function in the source Modelica code one by one.

Variables in the SimCode representation contain a Boolean field identifying their parallelism. This field is used by the template functions to identify and dispatch parallel variables.

The code generation for a serial Modelica function starts with variable declarations and initializations. This is handled by the template function `varInit`. This function will defer all parallel variables to a new function called `parVarInit`. The Susan code for this template function is shown in Listing 4.3-2 to give a general idea of how the code generation works.

---

**Listing 4.3-2 Template function parVarInit**

```plaintext
template parVarInit(Variable var, String outStruct, Integer i, Text &varDecls /*BUFP*/, Text &varInits /*BUFP2*/, Text varFrees /*BUFPF*/)
"Generates code to initialize parallel variables. Does not return anything: just appends declarations to buffers."
:=
match var
case var as VARIABLE(____) then
    let varName = '<%contextCref(var.name,contextFunction)%>'
    let instDimsInit = (instDims |> exp =>
        daeExp(exp, contextFunction, &varInits /*BUFC*/, &varDecls /*BUFD*/)
    ;separator=', ')
    if instDims then
        let &varDecls += 'device_<%expTypeShort(var.ty)%>_array<%varName%>;<%n%>
        let &varInits += 'alloc_<%expTypeShort(var.ty)%>_array(&<%varName%>,<%listLength(instDims)%>,<%instDimsInit%>);<%n%>
        let &varInits += defaultValue
        let &varFrees += 'clReleaseMemObject(<%varName%>.data);<%n%>
        let &varFrees += 'clReleaseMemObject(<%varName%>.info_dev);<%n%>
        let &varFrees += 'free(<%varName%>.info);<%n%>
    ""
    else
        let &varDecls += 'device_<%expTypeShort(var.ty)%> <varName>%;<%n%>
        let &varInits += 'ocl_device_alloc(sizeof(modelica_<%expTypeShort(var.ty)%>));<%n%>
        let &varFrees += 'clReleaseMemObject(<%varName%>);<%n%>
    ""
else let &varDecls += '#error Unknown parallel variable type<%n%>' ""
end parVarInit;
```

A step by step description of the function’s operations is given here. The first two operations identify the name and number of dimensions of the variable. If `instDims` is not zero then the variable is an array. Otherwise it is a scalar variable. If the variable is an array the following operations are done.
• A device array (defined in the OpenCL-C runtime library) of the variable's data type will be declared i.e. declaration code added to the buffer varDecls. Refer to Appendix A.3:024.
• The array will be allocated. Refer to A.3:053 and B.2.
• Code for properly de-allocating any memory allocated for the array will be added to the varFrees buffer. Refer to A.3:097-099.

If the variables is a scalar

• A device scalar (defined in the OpenCL-C runtime library) will be declared. Refer to A.3:028.
• Memory is allocated depending on the data type of the scalar. Refer to A.3:057.
• The memory allocated will be freed. Refer to A.3:109

The buffers will be used by the calling function to properly place code fragments. Code generation for output variables is handled in a similar manner.

In the C runtime system copying of array variables is handled by the overloaded function `copy_datatype_array_data`. This function has been overloaded to support copy operations involving parallel arrays (see Appendix B.2). Assignments involving parallel arrays in Modelica code are translated to copy in the code generation and will be handled by one of these overloaded functions.

Assignment of scalars in Modelica code used to be translated to a simple assignment statement. This of course is not suitable for assignments between parallel and serial scalar variables. To this end assignments of scalar variables are now handled by overloaded functions instead of simple assignment operations. If assignments involve parallel variables the proper copy operations between device memory and host memory are done. If a scalar assignment does not involve parallel variables then normal assignments will be done.

It should be mentioned here that at the time of this writing a new version of the implementation was finalized. The new implementation represents all parallel scalar variables as normal serial variables up until kernel execution. This way Modelica scalar assignment can be translated to normal C assignment regardless of parallelism, solving the above problem and a couple of others involving kernel functions. This new implementation is not used as a base for this thesis since it was delayed by an unfortunate bug that delayed tests on 64 bit testing systems used for the thesis work.

### 4.3.1.2 Parallel for loop: parfor

Modelica parallel for loops are translated into OpenCL kernels. These kernels contain some iteration management operations, a main for loop and the corresponding body of the parfor loop. Part of the matrix multiplication Modelica code given in A.1 is shown in Listing 4.3-3
for a quick reference. The relevant parts of C and OpenCL code generated for the parfor loop are shown in Listing 4.3-4 and Listing 4.3-5, respectively.

The Modelica parfor loop shown has an iterator named i, a starting value of 1, step value of 1 and stop value of m. This is the only information needed to generate the C part of the code.

The execution memory buffer is allocated before starting any kernel execution. For a detailed explanation of this buffer see 4.4.4.1. The iterator information about start, step and end value is set to temporary variables. Then the corresponding kernel is created. Next the kernel arguments are set with the help of function `ocl_set_kernel_args`. The first argument to this function is the kernel created in the previous step. The second argument is the number of arguments to be set to the kernel. The next two arguments are a pointer to a variable containing the size of the execution memory buffer and a pointer to the buffer itself. This information is used by kernels to initialize their memory states as explained in section 4.4.4.3.

The rest arguments are all the parallel variables declared in the current function. The structure representing parallel arrays in the OpenCL runtime has three fields. The first field is the memory pointer to the data in device memory. The second field is a pointer to device memory where information about the array is stored. This information is used for OpenCL execution. The third field is a pointer to host memory where the information about the array is stored. This field is used for host C code execution. Every parallel array is resolved in to two arguments, device memory pointers for the data and information about the array i.e. the first two fields of the device array structure.

After setting the arguments the kernel is executed and finally any extra memory is deallocated. The functions ocl_* are all defined in the OpenCL-C runtime library. They provide
a compact and clear code generation option by hiding the rather long and complicated code in the background. These functions are discussed in Section 4.4.1.

Listing 4.3-5 OpenCL kernel for a parfor loop

```c
__kernel void kernel_0(modelica_integer loop_start,
                       modelica_integer loop_end,
                       modelica_integer loop_step,
                       modelica_integer buffer_size,
                       __global void* exec_buffer,
                       __global modelica_integer* _pA_data,
                       __global modelica_integer* info_pA,
                       __global modelica_integer* _pB_data,
                       __global modelica_integer* info_pB,
                       __global modelica_integer* _pC_data,
                       __global modelica_integer* info_pC,
                       __global modelica_integer* _pm_ptr,
                       __global modelica_integer* _ptemp_ptr)
{

    //thread managing for 'for loops'.
    modelica_integer inner_start = (get_global_id(0)*loop_step) + loop_start;
    modelica_integer stride = get_global_size(0) * loop_step;
    for(modelica_integer _i = (modelica_integer) inner_start;
        in_range_integer(_i, loop_start, loop_end); _i += stride)
    {

        //reconstruct global arrays back into global array structs
        //ALL scalars are treated as PRIVATE!!!!
        integer_array _pA;
        _pA.data = _pA_data;
        _pA.ndims = info_pA[0];
        _pA.dim_size = info_pA + 1;
        integer_array _pB;
        _pB.data = _pB_data;
        _pB.ndims = info_pB[0];
        _pB.dim_size = info_pB + 1;
        integer_array _pC;
        _pC.data = _pC_data;
        _pC.ndims = info_pC[0];
        _pC.dim_size = info_pC + 1;
        modelica_integer _pm = *_pm_ptr;
        modelica_integer _ptemp = *_ptemp_ptr;

    }

    Loop Body

    }
```

The above parfor loop code generation is handled by template function `algStmtParForRange_impl` and a number of other template functions. The helper functions are used to properly format the generated code e.g. generating the argument list.
The OpenCL kernel code generated for the above parfor loop is shown Listing 4.3-5. The kernels generated for each parfor loop have names which are globally unique. The naming is handled by a new counter called `tmpTickGlobal()`. Unlike other counters which are set upon entry to a new function this counter is never reset throughout the code generation. This is done in order to avoid possible name collisions among kernels generated for parfor loops in different functions. The first three arguments to the kernel function are the start, step and end values of the iterator for the original Modelica parfor loop. They are used to dictate the iterations of the main for loop of the kernel.

The main for loop is needed for cases where the number of threads launched is not the same as the number of iterations in the parfor loop. It also handles cases where the loop step is not 1. Every thread will get the start of its iteration by using its global id. Assume a for loop with start, step and end values of 10, 2 and 100 respectively with 30 threads available. Thread 0 will have a start value of 10. Thread 1 will have a start value of 12. Similarly thread 29 will have a start value of 68. All threads will have a step or stride of 60 and a stop value of 100. This way thread 0 will execute iterations for \( i \) value of 10 and 70. Thread 1 will execute 12 and 72. Thread 15 will execute iterations for 40 and 100. Threads 16 to 29 will only execute one iteration each i.e. iterations 42, 44 …to 68.

Each thread will do a reconstruction of arrays and creates a copy of each scalar before executing the body of the loop. Lines 048 to 061 in Listing 4.3-5 show these operations. Arrays are reconstructed in order to be able to properly carry out different array operations like indexing and dereferencing. The structures representing device arrays are discussed in detail in section 4.4.2. The scalar variables passed as shared global memory pointers will be assigned to local copies of the corresponding types (line 60 and 61). This makes sure that there will be no access conflicts between threads when they use these scalar variables.

Reconstruction of arrays and scalars is performed inside the body of the parfor loop to enforce complete isolation of each iteration. This method makes sure that no scalar value will be carried over between possible multiple iterations made by a single thread. This multiple iteration happens when the number of threads is less than the number of iterations of the parfor loop as in the example given above.

The for loop body code generated in OpenCL is almost the same code as the C code for normal for loops. One major difference is that all function call statements in the OpenCL will have the extra argument, a pointer to `memory_state`, which is used to keep track of memory usage. In addition some utility function calls will be modified to address the lack of function overloading mechanisms in OpenCL.

The template function `newKernel` handles the code generation of kernels for parfor loops. This function is used by `algStmtParForRange_impl` for each parfor loop. Code for parfor loops is deferred to and collected in a single buffer throughout the C code generation. This buffer is later printed to the file `ModelName_kernels.cl`.
4.3.1.3 Parallel functions

OpenCL code for all Modelica parallel functions is generated to a separate file called `ModelName_parallelFunctions.c`. In C code generation the definition of a function is printed to one file and the declarations of the file are generated to a different header file. In the OpenCL case, however, both the header declarations and definition of a parallel function are printed to a single file in order to reduce the number of additional files generated.

The OpenCL code of parallel function declaration and definition adds a new extra argument, `memory_state` pointer, to each parallel function. The need for extra argument is discussed in detail in section 4.4.4. The body of a parallel function is generated the same way as normal function except for some minor differences. A typical OpenCL code for a parallel function resembles A.4 lines 001-035.

The Susan template code generation for parallel functions is handled at the top level by the functions `parallelFunctionHeaders` and `parallelFunctionBodies`. These functions use a number of other functions to properly format and generate the desired code.

4.3.1.4 Kernel functions

Code generation for kernel functions needs more special approach for two main reasons.

- Calls to kernel functions in Modelica are as simple as calls to normal functions. In actual C and OpenCL code, however, executing a kernel involves a number of complicated steps. These steps include: creating a kernel from an OpenCL program, setting the arguments to the kernel with the correct datatypes and order, queuing the kernel execution and finally releasing the kernel object.
- Modelica does not allow passing variables by reference which means any result from a function will have to be in one of the output variables. For this reason Modelica kernel functions are allowed to have output variables. However OpenCL kernels are required to be void i.e. they do not return any variable directly.

These two problems are solved by creating an interface C function for each OpenCL kernel. This function, being a C function, can be called by other serial Modelica functions in a normal manner. The C interface function will allocate the memory for output variables. Remember that input variables are allocated in the calling function and local variables are allocated by the OpenCL kernel in the dedicated execution memory buffer (see 4.4.4). Then the interface will handle all operations needed to create and execute the corresponding kernel in a similar way as the parfor loop kernels. After the kernel has executed and returned the interface function will copy the output variable to the return variable making it available to the rest of the C code. The sample source code in A.3 demonstrates a simple kernel interface function. See lines 120-176.
The actual algorithmic body code of a kernel function is generated to the file `ModelName_parallelFunctions.c`. This portion of Modelica kernel function code is similar to that of a parfor loop except that there are no iteration specific arguments and, of course, no main for loop. A kernel function can be thought of as a parallel for loop where the iteration step is 1 and the number of threads is equal to the number of iterations i.e. no iteration related thread management is needed. The OpenCL part of a kernel function code can be seen in the sample code in A.4.039-141.

Since Modelica kernel functions translate to both OpenCL and C code they are handled by both branches of code generation. One branch involves the serial C code generation with a number of dedicated functions for the kernel interface. The other branch is with parallel Modelica functions with dedicated functions for the OpenCL part of the kernel code.

4.4 OpenCL Runtime

![OpenCL-C runtime system](image)

The OpenCL runtime is a collection of utilities and functionalities necessary for the proper execution of the generated OpenCL code. These include; the OpenCL-C runtime library which provides the connection between the host execution and the device execution, the OpenCL Helper file which is analogous to the c_runtime library of serial execution for parallel execution and the OpenCL offline compiler which is used for early error detection.

The corresponding arrangement of the relevant runtime components is shown in Figure 4.4-1.

4.4.1 OpenCL-C Runtime Library

The OpenCL-C runtime library provides the mechanisms for connecting the OpenCL device execution and the host serial C execution. It acts as a bridge between the parallel device and the host. The library provides a number of functionalities.

The Library defines the data structures used to represent parallel variables. These data structures are used to characterize parallel variables in the serial C code. The device integer array structure representing an integer array on a parallel device is shown in Listing 4.4-1.
The library is also responsible for all OpenCL related initialization operations like device selection, creating contexts on devices, building OpenCL source code from a source file or a string buffer, setting arguments to and launching kernels and so on. It provides clear and concise functions for OpenCL operations by hiding the rather long and complicated OpenCL operations in the background. The library also provides all the mechanisms necessary for data transfer operations between the host and the OpenCL device. These include copy operations of: host to device, device to host and device to device.

Parallel implementations of built-in functions like transpose are also available in this library. The built-in functions are arranged according to the type and number of input arguments as well as the type of return variable.

The library provides the necessary mechanism for proper error reporting related to OpenCL. Functions for debugging operations are also available in the library.

The main functions available in the OpenCL-C runtime library are shown in Appendix B together with a brief description of their operations.

4.4.2 OpenCL Helper

Simulating a model using the OpenModelica compiler and runtime system involves C code generation. The generated C code is then compiled and linked with the libraries which provide operations for the simulation. One of these libraries is the c_runtime library. This library provides, among many other things, the structures and operations necessary to represent arrays.

When using C it is possible to implement operations in one library and link them later if they are needed. OpenCL, on the other hand, has no linking mechanisms. This, in my opinion, is due to the fact that OpenCL code is compiled differently depending on the target device. The target device is identified at runtime only.

The lack of linking mechanisms in OpenCL means that any helper methods needed by the implementation have to be in source code format and have to be compiled with the rest of the code. In this thesis work this is implemented in the file helper.cl which can be found in the opencl_runtime folder in the source code. The file is included as a header by every OpenCL code generated for a model. This section gives an overview of the key functionalities provided by this file.

The structures used to represent a device array are defined in this file. These structures contain the information about the array they represent. The structure representing an integer array is shown in Listing 4.4-1. The first member is the number of dimensions of the array. The second member holds a pointer to an array (normal array) which contains the sizes of each dimension. The last member points to the actual data of the array.
There are also structures and functions defined to be used in memory management. These structures and functions are discussed in detail in section 4.4.4-Memory Management.

This file also contains a number of pre-processor directives necessary for an efficient use of the current code generator. Some functionality in the code generated by the C code generator is not supported in OpenCL or need to be modified in some way to ensure proper execution.

Listing 4.4-1 integer array structure

```c
struct gi_array
{
    int ndims;
    __global modelica_integer* dim_size;
    __global modelica_integer* data;
};
typedef struct gi_array integer_array;
```

This usage of pre-processor directives is a kind of quick fix used to minimize the change needed to the current Susan C code generator. This should be phased out step by step. There are only a few of these left right now.

A good example of this usage is shown by the `get_memory_state()` and `restore_memory_state(state)` functions. These functions are used in the host C code to keep track of the memory state of simulation. The structures for memory management in the host C code are global variables that can be accessed readily by the single thread running on CPU. However in the OpenCL case the structures for memory management are owned individually by each thread and are not global (see 4.4.4 paragraph 8). Therefore the memory management structures have to be passed as an extra argument to every function involved in the execution. This means that the functions mentioned need to be changed in some way to support this behavior. This is done by translating calls to these functions to normal body code. For example the `get_memory_state()` function is lined to `(*(memory_state->current_state))` which will give access to the current state. The memory state structure is passed as an extra argument to every function in the OpenCL code.

The `helper.cl` file also contains definitions for a number of utility functions. Functions for common operations like finding range, modulo, remainder, max and min of array and so on are declared and defined in this file.

### 4.4.3 OpenCL Offline Compiler

OpenCL code is compiled into OpenCL program by the serial host code. This compilation process happens when the host code is actually executing i.e. after host code is compiled. Hence any errors in the source OpenCL code will be reported when the host code is being executed. This is acceptable for small programs where cancelling or interrupting the host execution and fixing the OpenCL code. However if the source code is very large, which is
very common in Modelica and other simulation environments, this can cause waste of time and execution resource only to find out the OpenCL code contains an error. The OpenCL offline “compiler” is provided to solve this problem.

The OpenCL offline compiler of this implementation is a small program which uses the actual OpenCL compiler of the selected platform or device to test compile the generated OpenCL source code. The OpenCL test compilation is done prior to the C code compilation. If there is an error in the OpenCL code then the C code will not be compiled. This way programmers can locate and fix OpenCL errors without the need to start the actual simulation. The compiler, in addition to compiling the source code, also provides the binary resulting from the compilation in a file with .bin extension according to the model name. This binary code can be reused for other purposes.

4.4.4 Memory Management

Simulations usually consume a lot of memory. This is more visible when simulating models with a fairly large number of variables. Specifically models involving large arrays consume a lot of memory. Obviously there should be a noble way handling memory operations like allocating, indexing, referencing and de-allocating memory properly.

The OpenModelica compiler’s C runtime library handles memory allocation and de-allocation operations. Prior to starting any simulation the runtime environment allocates the buffers needed for the simulations. A state of memory is also initialized. Consequent memory allocation requests are given memory space from these buffers. This method of allocating buffers before start of simulations provides a very convenient way of managing memory. Memory is requested once from the Operating System and is released at the end of simulation. Functions allocate any necessary memory from the buffer. At the end of each function the memory state is restored to the state before the function entry i.e. all memory is returned to the buffer. If simulations run in to any memory problems like over-allocation users will be notified of the errors properly.

The runtime memory is allocated as global arrays for each type of variable. Global arrays are created for Integer, Real, String, Boolean, Size_, Index_ and Char type variables. The source code for these arrays and many more memory related operations are located in the files memory_pool.h/c in the c_runtime folder in the source code.

Now let’s consider OpenCL related cases. Considering only for loops we can see that there will be no need to allocate memory inside a kernel in the OpenCL execution. Note that this is different from the allocation of memory buffers from the Host C code on the OpenCL device. What we are interested here is any array related memory allocation inside a given kernel. Since Modelica codes declare (and allocate) all variables before the start of the algorithm section we are insured that parfor loops will not allocate any new memory in their bodies. Of course there might be some scalar allocations of integers or real variables but we are not
interested in those since the memory requirement for them is very small and are allocated and de-allocated automatically. The fact that there are no new array declarations in a parfor loop ensures that the generated kernel for any parfor loop will not allocate any new arrays. It will only use the arrays or buffers created by the host code on the device and passed to the kernel as arguments (parallel variables in the original Modelica code).

If we consider parallel functions and kernels we can immediately see that we will need some kind of memory management since they are allowed to create arrays in their body code.

Naturally the same structure of memory management as the C runtime is desirable for the OpenCL runtime environment which needs to allocate, use and de-allocate memory when running kernels on the OpenCL device. However this task is complicated by two things in OpenCL.

The first problem is that, unlike the normal C execution on the host CPU, OpenCL execution involves multiple threads. Each thread needs its own isolated memory fraction to perform its part of the computation. A fairly simple solution to this problem is to allocate buffers in the same manner as the C runtime described above. Each thread will create its own buffers and use them to service any allocation requests from its algorithm. These buffers are allocated in OpenCL private memory space and are only accessible to the owner thread. Functions will allocate and release memory in the same manner as the C runtime. This was the original solution implemented in the OpenCL runtime. However it soon became clear that most devices do not provide enough local memory to support the memory requirements of the Modelica models involving array computations. The available local memory is usually in tens of Kilo Bytes. For example the NVIDIA Tesla M2050 available on the Fermi machine in IDA provides only 48KB of local memory. If we hundreds of threads running they will have to divide this memory space equally which takes the available memory per thread to hundreds of Bytes only.

In order to provide threads with a much larger memory we would have to allocate a fairly large buffer in the OpenCL global memory address space and divide this between the threads. A direct solution to this would be to allocate arrays globally just like the C runtime system. We can declare the arrays in the kernel’s file but outside of any specific kernel (globally) just like global variables are declared in normal C code. This way all kernels and all threads will have access to this memory. Access to this memory will be slower than those to the local address space but we will have the necessary memory space to run array computations. However this is where the second problem comes in to the picture. OpenCL specifications allow globally allocated memory only in the __constant memory space. That means the allocated memory will be available to all threads and kernels but it will be read only memory. Clearly we need a read-write memory space so this option also cannot be used in the current implementation.
The other option, which is implemented right now, is to allocate the buffer from the Host C code just like any other array arguments. The pointer to this memory space is passed to all kernels as the first argument. Prior to any computation each kernel initializes this memory (kind of formatting the bulk memory) any divides it between threads and variable types. Every function in the OpenCL implementation which involves any memory related operations takes an extra structure argument providing information about the memory state. Any memory changes the function makes will be reflected in this structure. This solution is much more complicated than the previous two solutions described above. However it is the only solution that meets all our requirements for the Model generated code. This process is divided into four steps and described below in detail.

4.4.4.1 Creating the Buffer

There is a structure, defined in ocl_util.h, which represents an OpenCL device memory buffer. This structure is shown in Listing 4.4-2

This simple structure contains the pointer to the memory buffer in the OpenCL device and the size in bytes of that memory. In the generated code there will be one copy of this structure defined in the ModelName_functions.c file for a given model. This structure will be used throughout the generated code to hold information about the execution buffer available for kernels

```
Listing 4.4-2 device_buffer struct

struct dev_buff{
    cl_mem buffer;
    modelica_integer size;
};
```

The function `ocl_create_execution_memory_buffer(device_buffer*)` defined in the ocl_util.c/h file and shown in Listing 4.4-3 is used to actually allocate the execution buffer. The function takes the device buffer structure, allocates the necessary memory and assigns the proper values to the structure’s members.

The size of memory allocated for the buffer is described as a fraction of the `CL_DEVICE_GLOBAL_MEM_SIZE` [16] which is the global memory size available in the OpenCL device. If this value is greater than `CL_DEVICE_MAX_MEM_ALLOC_SIZE`, which is the maximum size of memory that can be allocated at once, then `CL_DEVICE_MAX_MEM_ALLOC_SIZE` bytes will be allocated instead. The desired fraction to be allocated is defined in `ocl_util.h` as `OCL_BUFFER_SIZE_FRACTION` and by default set to 4. This means that a quarter of the `CL_DEVICE_GLOBAL_MEM_SIZE` will be allocated by default. This is in accordance with the OpenCL specifications [30] stating:

`CL_DEVICE_MAX_MEM_ALLOC_SIZE` - “Max size of memory allocation in bytes. The minimum value is max (1/4th of `CL_DEVICE_GLOBAL_MEM_SIZE`, 128*1024*1024)”
This function is called once per simulation. After the function completes its operation the device_buffer structure is ready for use.

Listing 4.4-3 Function ocl_create_execution_memory_buffer()

```c
void ocl_create_execution_memory_buffer(device_buffer* d_buff){
    cl_int err;
    cl_ulong mem;
    cl_ulong mem2;
    cl_mem tmp;
    cl_ulong size;
    clGetDeviceInfo(ocl_device, CL_DEVICE_GLOBAL_MEM_SIZE,
        sizeof(cl_ulong), &mem, NULL);
    clGetDeviceInfo(ocl_device, CL_DEVICE_MAX_MEM_ALLOC_SIZE,
        sizeof(cl_ulong), &mem2, NULL);
    size = mem/OCL_BUFFER_SIZE_FRACTION;
    size = (size > mem2 ? mem2 : size);
    tmp = clCreateBuffer(device_context, CL_MEM_READ_WRITE,
        size, NULL, &err);
    ocl_error_check(OCL_CREATE_BUFFER, err);
    if(err){
        printf("Error: Allocating execution memory buffer");
        exit(1);
    }  
    d_buff->buffer = tmp;
    d_buff->size = size;
}
```

4.4.4.2 Attaching the Buffer to Kernels

Every kernel execution takes the device buffer’s members as arguments. The two members of the structure are set as arguments to the ocl_set_kernel_args() function described in Appendix B.1. More specifically the members of the structure are set as the sixth and seventh arguments to ocl_set_kernel_args() for kernels generated for parfor loops. This means they will be set as the fourth and fifth arguments to the actual kernel. Keep in mind that the first three arguments for a kernel generated for a parfor loop are the loop start, step and end variables. In contrast in kernels generated for kernel functions (see 4.3.1.4) the members of the device buffer kernel are set as the second and third arguments to ocl_set_kernel_args() which will set them as the first two arguments to the actual kernel. For a complete description of this it is recommended to understand sections 3.4, 3.5 and Appendix B.1.

4.4.4.3 Initializing (Formatting) the Buffer

Every kernel in the generated code initializes the device_buffer before starting any execution. This includes dividing the bulk execution memory between threads and data types. Before discussing how memory is divided and used between kernels it is helpful to discuss the data structures helping these operations.
There are some structures defined in the helper.cl file (see 4.4.2) which are used to keep track of memory states for each thread. These are shown below in Listing 4.4-4.

The first structure is the state_s structure. This structure contains pointers which point to the next non-allocated memory address for each type. Every thread has its own copy of this structure and uses it to keep track of its memory state. The count elements in the structure are the numbers of a data type that is already allocated. These are kept therefore debugging purposes only and are not a requirement.

Listing 4.4-4 Structures used for memory management in Kernels

```
struct state_s {
  _index_t real_count;
  _index_t integer_count;
  _index_t boolean_count;
  _index_t size_count;
  __global modelica_real* real_buffer_ptr;
  __global modelica_integer* integer_buffer_ptr;
  __global modelica_boolean* boolean_buffer_ptr;
  __global modelica_integer* size_buffer_ptr;
};
typedef struct state_s state;

struct buffer_s{
  __global modelica_real* real_buffer;
  __global modelica_integer* integer_buffer;
  __global modelica_boolean* boolean_buffer;
  __global modelica_integer* size_buffer;
};
typedef struct buffer_s buffer;

struct memory_s{
  state* current_state;
  buffer* local_buffer;
};
typedef struct memory_s memory;
```

The second structure is the buffer_s structure. Again every thread has its own copy of this structure. The members of this structure point to the beginning of the buffer portion for each data type. Note that the first structure holds the current state while the second structure contains, in a sense, the initial state. The need for this arrangement becomes clear as we discuss how the memory is divided and used. The third structure is used to simplify passing the state information as argument between kernel-parallel function or parallel function-parallel function. It holds pointers to the state_s and buffer_s structures.
At the beginning of every kernel there is the code fragment shown in Listing 4.4-5. Since a kernel is executed independently by each thread in the launch, every thread will create its own copies of these structures. The `initialize_buffer()` function is called next.

The `initialize_buffer()` function shown in Listing 4.4-6 is where the bulk execution memory is divided between threads and data types. Each buffer executes this function and initializes its own share of the memory. This operation involves a fair amount of pointer arithmetic and is very error prone. It is implemented because it was the only solution that was available. A step by step description of the operation is necessary to understand the whole process. Keep in mind that this function is executed by all threads independently.

Listing 4.4-5 Memory initialization in kernels

```c
////////////////////////////////////////////
//Initialize local memory state
buffer local_buffer;
state current_state;
memory memory_s;

memory* memory_state = &memory_s;
memory_state->current_state = &current_state;
memory_state->local_buffer = &local_buffer;
initialize_buffer(exec_buffer, buffer_size, memory_state);

////////////////////////////////////////////
```

The function takes the pointer to the bulk execution memory, the size of the memory space and the memory state structure as input arguments. Then it finds how many threads are running at the moment. Then follow the size variable declarations for each data type that is currently implemented. Then it will compute the size of buffer that each thread should get from the size of the total memory and the total number of threads.

After these operations each thread will find out where its portion of the buffer starts by using its own id and the size of buffer per thread. For example the buffer portion for thread id 0 starts at the beginning of the whole execution memory, the buffer portion for thread id 1 starts at buffer_per_thread plus the beginning of the whole execution memory and the buffer for thread id 2 starts at 2*buffer_per_thread after the beginning of the whole execution memory.
void initialize_buffer(__global void* exec_buffer, modelica_integer
  buffer_size, memory* current_memory)
{
  size_t num_threads;
  num_threads = get_global_size(0);

  size_t thread_buff_start;
  size_t buffer_per_thread;
  size_t real_buffer_per_thread;
  size_t integer_buffer_per_thread;
  size_t boolean_buffer_per_thread;
  size_t size_buffer_per_thread;

  buffer_per_thread = (buffer_size/num_threads);
  thread_buff_start = exec_buffer + get_global_id(0)*buffer_per_thread;

  current_memory->local_buffer->real_buffer = (__global
    modelica_real*)real_start;
  real_buffer_per_thread = (buffer_per_thread*PRECENTAGE_REAL_BUFFER)/100;

  size_t integer_start = real_start + real_buffer_per_thread;
  current_memory->local_buffer->integer_buffer = (__global
    modelica_integer*)integer_start;
  integer_buffer_per_thread = (buffer_per_thread*PRECENTAGE_INTEGER_BUFFER)
    /100;

  size_t size_start = integer_start + integer_buffer_per_thread;
  current_memory->local_buffer->size_buffer = (__global
    modelica_integer*)size_start;
  size_buffer_per_thread = (buffer_per_thread*PRECENTAGE_SIZE_BUFFER)/100;

  size_t boolean_start = size_start + size_buffer_per_thread;
  current_memory->local_buffer->boolean_buffer = (__global
    modelica_boolean*)boolean_start;
  boolean_buffer_per_thread = (buffer_per_thread*PRECENTAGE_BOOLEAN_BUFFER)
    /100;

  current_memory->current_state->real_count = 0;
  current_memory->current_state->integer_count = 0;
  current_memory->current_state->size_count = 0;
  current_memory->current_state->boolean_count = 0;

  current_memory->current_state->real_buffer_ptr =
    current_memory->local_buffer->real_buffer;
  current_memory->current_state->integer_buffer_ptr =
    current_memory->local_buffer->integer_buffer;
  current_memory->current_state->size_buffer_ptr =
    current_memory->local_buffer->size_buffer;
  current_memory->current_state->boolean_buffer_ptr =
    current_memory->local_buffer->boolean_buffer;
}

Listing 4.4.6 Function initialize_buffer()
The threads, after figuring out where their respective buffers begin, will start to divide the buffer between data types and initialize their memory states. The buffer for Real data types comes first i.e. the real buffer starts at the beginning of the buffer for the current thread. The
pointer to this location is cast accordingly and used to initialize the local buffer of the current thread. The size that is to be allocated for Real data types is determined from the constant PRECENTAGE_REAL_BUFFER defined at the beginning of the helper.cl (see 4.4.2) file. The buffer portion for the next data type, Integer type, starts right after the end of the Real data type buffer. This starting location is used to initialize the local buffer of the current thread in a similar manner as the Real data type buffer. The same operations follow for Size data type and Boolean data type buffers.

Now that the local buffer is initialized and is pointing to the right location for each thread and data type the next step is to initialize the current state structure. Remember that the local_buffer structure that was initialized earlier would not change throughout the kernel execution. It is used only used as a reference. It is the current_state structure that is used to keep track of how much memory is allocated and where the next allocation should be. Since the computation is still at the beginning of the kernel and no actual allocation has been done by the computation code yet the current state is the initial state.

The initial state for each thread is, of course, the state of the local_buffer structure. This is used to initialize the current_state structure.

The execution buffer is now ready for use. A pictorial description of these operations is shown in Figure 4.4-2.

4.4.4.4 Using the Memory Buffer

The operations of allocating memory space are explained here with an example of a one dimensional array allocation. In the OpenCL implementation in this thesis arrays are allocated by using functions. These functions have names that are self-explanatory. For example the function used to allocate a two dimensional integer array is called alloc_integer_array_c99_2 and is shown in Listing 4.4-7. The c_99 part is there to emphasis that this is written for the C99 standard which is used by OpenCL C.

In the original C runtime system for the Open Modelica Compiler these allocation operations are done by using function with variable number of arguments. Therefore all integer array allocations are done by a function called alloc_integer_array which determines the number of dimensions and sizes of dimensions from the arguments passed to it. However OpenCL 1.1 supports neither the stdarg standard library (the stdarg.h header which provides the features of using functions with variable number of arguments) nor overloaded functions. As a result there is a need to define a separate function for each dimension. At the moment the necessary functions for allocating (also indexing) are defined for one dimensional and two dimensional arrays only. However this can easily be extended to support larger dimensions.
The function *alloc_integer_array_c99_2* takes the pointer to the structure representing an array (see Listing 4.4-1), the number of dimensions of the array (kept only to have uniform interfaces with the C runtime, otherwise not needed since we already know the number of dimensions is 2 because of the function called), the sizes of the dimensions and the pointer to the memory state structure. The first operation is to set the value of *ndims* in the destination array to 2. Then it will check if we have enough space in the size buffer to store the dimension sizes of the array. This is done by checking if we would cross in to the memory area of the Boolean buffer which comes exactly after the end of the size buffer (see Figure 4.4-2). If we would run out of buffer i.e. cross in to the next data type’s buffer, an error message will be printed. This only works for CPU’s with AMD OpenCL drivers since NVIDIA and INTEL are not supporting the *printf* function in their implementations. When running on these two implementations the *printf* function is suppressed by using preprocessor macros.

If there is enough memory to hold the information about the size, the function will calculate the total number of elements in the array. Then it will check if the buffer limit will be passed
when allocating the array. If the limit is passed an error message is printed. Otherwise it will set the data pointer in the array structure to the location where the current real_buffer_pointer in the current_state structure points to. Then it will advance the pointer in the current state by the corresponding amount. The count of elements is also updated to help debugging.

As mentioned earlier this method of memory management involves a lot of pointer arithmetic operations and is very error prone. This approach is taken only because it is the only technique that fulfills the requirements for handling large array computations which is the main target in this thesis work.
5 Discussion

This chapter provides a short summary of what has been accomplished in this thesis work. A brief performance discussion regarding core utilization on Multi-core architectures is given. Then the new Modelica Parallel Benchmark suite is discussed. The discussion is finished with some suggestions for future work and a short conclusion.

5.1 Accomplishments

This thesis work has introduced a number of new explicit parallel programming constructs in to OpenModelica. The main goal of the thesis work is not to provide a complete parallel programming set. Rather the goal was to find out if explicit parallel programming is an area worth pursuing for the Modelica language.

The high level language constructs provided are not tied to any specific target language. Not even OpenCL. They can be used for any kind of parallel programming paradigm. They can be interpreted differently depending on what target language is to be used for parallel execution. For example parallel functions can be interpreted as OpenCL parallel functions or CUDA __device__ functions depending on the target language.

In this thesis work these constructs are translated to OpenCL code. Even though OpenCL is a relatively new language it has a promising future due to the fact that it is supported by multiple vendors. OpenCL provides a common parallel programming platform for CPUs, GPUs and ACCELERATORs. Using the new constructs with the proper OpenCL drivers, programmers can parallelize algorithms on a wide range of architectures.

5.2 Performance

This section does not provide performance results and speedups from parallelized test models. For these see the next section 5.3. Instead here the discussion will be on how well the generated code is taking advantage of the available computational resources.

Modelica Kernel functions are translated to OpenCL kernel functions. The programming and execution of a Modelica kernel function has a direct corresponding relation to the execution of an OpenCL kernel function. That means a properly written kernel function will of course provide the same performance as it OpenCL implementation (excluding the overhead of simulation related operation). However there is no direct corresponding element in OpenCL for a Modelica parfor loop. Modelica parfor loops are translated to OpenCL kernels and their execution is managed by the runtime system. Obviously the goal is to generate and execute optimized kernels which can take full advantage of the parallel device. How well would a parfor loop perform? A comparison of serial and parallel matrix multiplication is done to demonstrate the performance. The parallel test case is the matrix multiplication program given
in Appendix A.1 but without the kernel function call i.e. matrix multiplication using parfor loop. gDEBugger [31] was used to profile the execution of the matrix multiplication programs. The tests are done for matrices of size 1000x1000 on Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.4 GHz and Intel(R) Xeon(R) CPU E5520 @ 2.27GHz.

Figure 5.3-1 shows core utilizations for the serial matrix multiplication on Intel Q6600 (a 4 core CPU). As it can be seen clearly the cores are not running with full performance. The execution is not equally distributed among the available processors. This is not surprising since there is no explicitly stated parallelism in the code. The Operating System and the hardware will parallelize the execution to some extent; that is why all processors are executing to some extent. However there is still a lot of processing power that should be utilized. The serial multiplication on Intel E5520 (a 16 core CPU) also gives the same kind of performance graphs.

The core utilization graphs of the parallel matrix multiplication for both processors are shown in Figure 5.3-2 and Figure 5.3-3. This time the load is distributed evenly among the available processors through the parallel execution. The processors are utilized up to a 100 percent during parallel execution there by considerably reducing the total simulation time. The utilization dips seen on the graphs correspond to the serial part of the algorithm.

5.3 MPAR Benchmark Suite

The MPAR Benchmark suite [32] is a collection of parallel algorithms implemented in Modelica using the new parallel language constructs introduced in this thesis work. The suite is used as a means for evaluating the feasibility and performance of the generated OpenCL code when using the new Modelica language extension. The measurements are done for both sequential and parallel implementations of the benchmark suite using the generated code from the OpenModelica compiler on different hardware configurations including CPUs and GPUs.

The results and measurements obtained from the benchmark suite can be used to assess the performance gains resulting from the new extensions of this thesis work.
Figure 5.3-1 Serial Matrix Multiplication on Intel Q6600 @ 2.4 GHz core utilizations.
Figure 5.3-2 Parallel Matrix Multiplication on Intel Q6600 @ 2.4 GHz core utilizations.

Figure 5.3-3 Parallel Matrix Multiplication on Intel E5520 @ 2.27GHz core utilizations.
5.4 Future Work

There are a number of things that can be improved or added to the current implementation of explicit parallel programming of Modelica. Some of these are discussed here.

- CUDA code generation should be supported. This is relatively easy now since the high level constructs are available and reusable. The only parts of OMC that will require changes are the SimCode module and the Susan Code Generation.
- Semantic Error detection and reporting:
  - At the moment there is virtually no error reporting and handling mechanisms for semantic errors related to parallel algorithm constraints. For example as discussed in section 3.5 parallel for loops can only have parallel variables in their body part. However the compiler, right now, does not report any error if this is violated. Generally semantic error reporting mechanisms should be implemented to enforce the constraints imposed by the target language. For the OpenCL case error reporting for constraints stated in sections 3.2-3.8 should be implemented.
- The current parallel for loop implementation should be enhanced to support the following features:
  - Explicitly stating the desired number of threads to be used for parallel execution.
  - Automatically detecting the parallel variables used in the parfor loop and using only those as arguments to kernels to get a more concise target code. See 4.2.1.1.
  - Specifying the desired target language using annotations. This is important if other parallel programming paradigms are added to the extension e.g. if CUDA or OpenMP code generation is supported, the desired target language can be stated here.
- More built-in functions should be added or extended for parallel execution. The serial/parallel arguments/return-values combination should be extended. See section 3.6.
- For OpenCL (and also CUDA) code generation: mechanisms should be added for using local variables i.e. allocating and using variables in local or private memory space. See section 3.7.
- The OpenCL functionalities discussed in section 3.8 should be improved to provide a better control over thread management and execution.
- Overloading of parallel functions should be implemented.
5.5 Conclusion

The new parallel programming constructs provide a novel and convenient way of parallelizing data parallel algorithms. Of course these new constructs will provide performance gains for a certain field of algorithms; algorithms with data parallel algorithms suitable for SPMD execution. However these kinds of algorithms are common in many areas of modeling. Basic algorithms like matrix multiplication and LU decomposition are used directly or indirectly by many algorithms. This means any model involving these algorithms can benefit from the performance gains for the data parallel portions. Large array and matrix algorithms are usually very suitable for parallelizing. Using the new constructs with a well-designed parallel algorithm can provide huge performance gains.

It should be noted that the new parallel programming constructs introduced in this thesis work are not part of the official Modelica language specification. They are available only in the OpenModelica [1] compiler as optional extension.
A.1 MatMult.mo

001  //parallel function for multiplying two integers
002  //all variables in a parallel function are parallel variables.
003  parallel function multiply
004    input Integer a;
005    input Integer b;
006    output Integer c;
007    algorithm
008      c := a * b;
009    end multiply;
010
011  //kernel function for matrix multiplication.
012  //all variables in a kernel are parallel variables.
013  kernel function mmkernel
014    input Integer m;
015    input Integer pA[m,m];
016    input Integer pB[m,m];
017    output Integer pC[m,m];
018    Integer id;
019    Integer nr_threads;
020    Integer ptemp;
021    algorithm
022      //get the global thread id
023      id := ocl.get_global_id(0);
024      //get the global number of threads
025      nr_threads := ocl.get_global_size(0);
026      for i in id:nr_threads:m loop
027        for j in 1:m loop
028          ptemp := multiply(pA[i,h], pB[h,j]) + ptemp;
029        end for;
030        pC[i,j] := ptemp;
031      end for;
032    end mmkernel;
033    parallel Integer pm;
034    parallel input Integer pA[m,m];
035    parallel input Integer pB[m,m];
036    parallel output Integer pC[m,m];
037    parallel Integer temp;
038    algorithm
039      //m should be copied to device memory to
040      //be available for inner loops (which will
//be executed by OpenCL device)

pm := m;

parfor i in 1:m loop
  for j in 1:pm loop
    ptemp := 0;
    for h in 1:pm loop
      ptemp := multiply(pA[i,h],pB[h,j]) + ptemp;
    end for;
    pC[i,j] := ptemp;
  end for;
end parfor;

end parMatrixMult;

function start
  input Integer m;
  output Integer p;
  output Integer q;
  Integer A[m,m];
  Integer B[m,m];
  Integer C[m,m];
  Integer D[m,m];
  parallel Integer [m,m] pA;
  parallel Integer [m,m] pB;
  parallel Integer [m,m] pC;
  parallel Integer [m,m] pD;
  parallel Integer pm;

  //Initialize A and B
  for i in 1:m loop
    for j in 1:m loop
      A[i,j] := 1;
      B[i,j] := 2;
    end for;
  end for;

  pA := A;
pB := B;

  pC := parMatrixMult(m, pA, pB);

  //kernel function call is similar to normal function call
  pD := mmkernel(pm,pA,pB);

  C := pC;
  D := pD;
p := C[m,m];
q := D[m,m];

end start;

//In the model keep track of only one variable (p) to make it much faster.
model MatMult_one
  parameter Integer m = 100;
  Integer p;
  Integer q;

  algorithm
    (p,q):= start(m);
end MatMult_one;

A.2  MatMult_one_function.h

#ifndef MatMult_one__H

//...
A.3 MatMult_one_functions.cpp

```c
#include "MatMult_one_functions.h"

extern "C" {

// Buffer on OpenCL device for allocation of arrays
// in parallel functions and kernels.
device_buffer dev_buff;

// the OpenCL program. Made global to avoid repeated builds
cl_program simulation_ocl_programs[10];
modelica_integer nr_of_ocl_programs = 0;
cl_kernel simulation_ocl_kernels[10];
modelica_integer nr_of_ocl_kernels = 0;

// the OpenCL program. Made global to avoid repeated builds
// cl_program ocl_kernels_program = ocl_build_p_from_src("MatMult_one_kernels.cl", true);

#define start_rettype_1 targ1
#define start_rettype_2 targ2
typedef struct start_rettype_s {
    modelica_integer targ1; /* p */
    modelica_integer targ2; /* q */
} start_rettype;

start_rettype _start(modelica_integer _m);

typedef struct start Rettype_1_targ1
    modelica_integer targ1;
#define parMatrixMult_rettype_1 targ1
    typedef struct parMatrixMult_rettype_s {
        device_integer_array targ1; /* pC */
    } parMatrixMult_rettype;

parMatrixMult_rettype _parMatrixMult(modelica_integer _m, device_integer_array _pA, device_integer_array _pB);

#define mmkernel_rettype_1 targ1
    typedef struct mmkernel_rettype_s {
        device_integer_array targ1; /* pC */
    } mmkernel_rettype;

mmkernel_rettype _mmkernel(device_integer _m, device_integer_array _pA, device_integer_array _pB);

} #endif
```
014 | start_retype _start(modelica_integer _m)
015 | |
016 | {|
017 | start_retype tmp1;
018 | state tmp2;
019 | modelica_integer _p;
020 | modelica_integer _q;
021 | integer_array _A;
022 | integer_array _B;
023 | integer_array _C;
024 | integer_array _D;
025 | device_integer_array _pA;
026 | device_integer_array _pB;
027 | device_integer_array _pC;
028 | device_integer_array _pD;
029 | modelica_integer tmp3;
030 | modelica_integer tmp4;
031 | state tmp5;
032 | modelica_integer tmp6;
033 | modelica_integer tmp7;
034 | modelica_integer tmp8;
035 | state tmp9;
036 | modelica_integer tmp10;
037 | modelica_integer tmp11;
038 | modelica_integer tmp12;
039 | parMatrixMult_retype tmp13;
040 | mmkernel_retype tmp14;
041 | //save the number of kernels and ocl programs already created
042 | tmp3 = nr_of_ocl_kernels;
043 | tmp4 = nr_of_ocl_programs;
044 | /***************************************************************************/
045 |
046 | tmp2 = get_memory_state();
047 | |
048 | alloc_integer_array(_A, 2, _m, _m);
049 | alloc_integer_array(_B, 2, _m, _m);
050 | alloc_integer_array(_C, 2, _m, _m);
051 | alloc_integer_array(_D, 2, _m, _m);
052 | alloc_integer_array(_pA, 2, _m, _m);
053 | alloc_integer_array(_pB, 2, _m, _m);
054 | alloc_integer_array(_pC, 2, _m, _m);
055 | alloc_integer_array(_pD, 2, _m, _m);
056 | _pm = ocl_device_alloc(sizeof(modelica_integer));
057 | |
058 | tmp10 = (modelica_integer) 1; tmp11 = (1); tmp12 = _m;
059 | if (!tmp11) {
060 | MODELICA_ASSERT("assertion range step != 0 failed");
061 | } else if (((tmp11 > 0) && (tmp10 > tmp12)) || ((tmp11 < 0) && (tmp10 < tmp12))) {
062 | for(modelica_integer _i = (modelica_integer) 1; in_range_integer(_i, tmp10, tmp12);
063 | _i += tmp11) {
064 | |
065 | tmp9 = get_memory_state();
066 | tmp6 = (modelica_integer) 1; tmp7 = (1); tmp8 = _m;
067 | if (!tmp7) {
068 | MODELICA_ASSERT("assertion range step != 0 failed");
069 | } else if (((tmp7 > 0) && (tmp6 > tmp8)) || ((tmp7 < 0) && (tmp6 < tmp8))) {
070 | for(modelica_integer _j = (modelica_integer) 1; in_range_integer(_j, tmp6, tmp8);
071 | _j += tmp7) {
072 | |
073 | *
074 | |
075 | |
076 | |
077 | |
078 | |
079 | |
080 | |
081 | |
082 | |
083 | |
device_integer_array

//Free unwanted memory allocated
clReleaseMemObject(_pA.data);
c1ReleaseMemObject(_pA.info_dev);
free(_pA.info);
clReleaseMemObject(_pB.data);
c1ReleaseMemObject(_pB.info_dev);
free(_pB.info);
clReleaseMemObject(_pC.data);
c1ReleaseMemObject(_pC.info_dev);
free(_pC.info);
clReleaseMemObject(_pD.data);
c1ReleaseMemObject(_pD.info_dev);
free(_pD.info);
c1ReleaseMemObject(_pm);

//restore the number of kernels and ocl programs already created
nr_of_ocl_programs = tmp4;
r_of_ocl_kernels = tmp3;

return tmp1;
}

parMatrixMult_retype _parMatrixMult(modelica_integer _m, device_integer_array _pA, device_integer_array _pB)
{
    parMatrixMult_retype tmp1;
    state tmp2;
    device_integer_array _pC;
    device_integer _pm;
    device_integer _ptemp;
    modelica_integer tmp3;
    modelica_integer tmp4;
    state tmp14;
    modelica_integer tmp15;
    modelica_integer tmp16;
    modelica_integer tmp17;
    cl_kernel tmp18;
    alloc_integer_array(&tmp1.targ1, 2, _m, _m);
    //save the number of kernels and ocl programs already created
    tmp3 = nr_of_ocl_kernels;
    tmp4 = nr_of_ocl_programs;
    
    //**********parallel operations**********/
    ocl_create_execution_memory_buffer(&dev_buff);
    tmp15 = (modelica_integer) 1; tmp16 = (); tmp17 = _m;
    tmp18 = ocl_create_kernel(ocl_kernels_program, "kernel_0");
    ocl_set_kernel_args(tmp18, 15, tmp15, tmp17, tmp16, dev_buff.size, dev_buff.buffer, _pA.data, _pA.info_dev, _pB.data, _pB.info_dev, _pC.data, _pC.info_dev, _pm, _ptemp);
71 | ocl_execute_kernel(tmp18);
72 | clReleaseKernel(tmp18);
73 | clReleaseMemObject(dev_buff.buffer);
74 | /**************************************************************************/
75 | return:
76 | restore_memory_state(tmp2);
77 | copy_integer_array_data(&pC, &tmp1.targ1);
78 | //Free unwanted memory allocated.
79 | clReleaseMemObject(_pC.data);
80 | clReleaseMemObject(_pC.info_dev);
81 | free(_pC.info);
82 | clReleaseMemObject(_pm);
83 | clReleaseMemObject(_ptemp);
84 | //restore the number of kernels and ocl programs already created
85 | nr_of_ocl_kernels = tmp3;
86 | nr_of_ocl_programs = tmp4;
87 | return tmp1;
88 | }
89 | }
90 | }
91 | }
92 | }
93 | }
94 | //Interface function to _mmkernel defined in parallelFunctions.cl file.
95 | mmkernel_rettype _mmkernel(device_integer _m, device_integer_array _pA, device integer array _pB)
96 | {
97 | mmkernel_rettype tmp1;
98 | state tmp2;
99 | device integer_array _pC;
100 | cl kernel tmp3;
101 | alloc_integer_array(&pC, 2, _m, _m);
102 | alloc_integer_array(&tmp1.targ1, 2, _m, _m);
103 | tmp2 = get_memory_state();
104 | /**************************************************************************/
105 | ocl_create_execution_memory_buffer(&dev_buff);
106 | tmp3 = _ocl_create_kernel(ocl_kernels_program, "_mmkernel");
107 | _ocl_set_kernel_args(tmp3, 10, dev_buff.size, dev_buff.buffer, _m, _pA.data, _pA.info_dev, _pB.data, _pB.info_dev, _pC.data, _pC.info_dev);
108 | ocl_execute_kernel(tmp3);
109 | clReleaseKernel(tmp3);
110 | clReleaseMemObject(dev_buff.buffer);
111 | /**************************************************************************/
112 | return:
113 | restore_memory_state(tmp2);
114 | copy_integer_array_data(&pC, &tmp1.targ1);
115 | //Free unwanted memory allocated.
116 | clReleaseMemObject(_pC.data);
117 | clReleaseMemObject(_pC.info_dev);
118 | free(_pC.info);
119 | return tmp1;
120 | }
121 | }
122 | }
123 | }
124 | }
125 | }
126 | A.4 MatMult_one_parallelFunctions.cl
```c

#ifndef MatMult_one_H
#define MatMult_one_H

#include "helper.cl"

#define multiply_rettype_1 targ1

typedef struct multiply_rettype_s
{
    modelica_integer targ1;
} multiply_rettype;

multiply_rettype _multiply(
    modelica_integer _a,
    modelica_integer _b,
    memory* memory_state);

#endif

multiply_rettype _multiply(
    modelica_integer _a,
    modelica_integer _b,
    memory* memory_state)
{
    multiply_rettype tmp1;
    state tmp2;
    modelica_integer _c;
    tmp2 = get_memory_state();
    _c = (_a * _b);
    //_return:
    restore_memory_state(tmp2);
    tmpl.tar1 = _c;
    //mahge: Free unwanted meomory allocated
    return tmp1;
}

__kernel void _mmkernel(
    modelica_integer buffer_size,
    __global void* exec_buffer,
    __global modelica_integer* m_ptr,
    __global modelica_integer* pA_data,
    __global modelica_integer* info_pA,
    __global modelica_integer* pB_data,
    __global modelica_integer* info_pB,
    __global modelica_integer* pC_data,
    __global modelica_integer* info_pC)
{
    //Initialize local memory state
    buffer local_buffer;
    state current_state;
    memory memory_s;
    memory* memory_state = &memory_s;
    memory_state->current_state = &current_state;
    memory_state->local_buffer = &local_buffer;
    initialize_buffer(exec_buffer, buffer_size, memory_state);
    //Memory state initialized
    //reconstruct global arrays back into global array structs
    //ALL scalars are treated as PRIVATE!!!!
    modelica_integer _m = *m_ptr;
```

integer_array_pA;
_pA.data = _pA.data;
_pA.ndims = info_pA[0];
_pA.dim_size = info_pA + 1;
integer_array_pB;
_pB.data = _pB.data;
_pB.ndims = info_pB[0];
_pB.dim_size = info_pB + 1;
integer_array_pC;
_pc.data = _pc.data;
_pc.ndims = info_pC[0];
_pc.dim_size = info_pC + 1;

state tmpl1;
modelica_integer _id;
modelica_integer _nr_threads;
multiply_rettype tmp2;
state tmpl3;
modelica_integer tmp4;
modelica_integer tmp5;
state tmpl6;
modelica_integer tmp7;
modelica_integer tmp8;
modelica_integer tmp9;
modelica_integer tmp10;
modelica_integer tmpl11;
modelica_integer tmp12;
modelica_integer tmp13;
modelica_integer tmp14;

tmpl1 = get_memory_state();

_id = get_global_id();
_nr_threads = get_global_size();
tmpl2 = _id; tmpl3 = _nr_threads; tmpl4 = _m;

if (!tmpl13) {
  else if (!((tmpl13 > 0) && (tmpl12 > tmpl14)) || ((tmpl13 < 0) && (tmpl12 < tmpl14))) {
    for(modelica_integer _i = _id; in_range_integer(_i, tmpl12, tmpl14); _i += tmpl13) {
      tmpl1 = get_memory_state();
      tmp8 = (modelica_integer) _i; tmp9 = (j); tmp10 = _m;
      if (!tmp9) {
        else if (!((tmpl9 > 0) && (tmpl8 > tmpl10)) || ((tmpl9 < 0) && (tmpl8 < tmpl10))) {
          for(modelica_integer _j = (modelica_integer) _i; in_range_integer(_j, tmpl8, tmpl10); _j += tmpl9) {

            _temp = (modelica_integer) 0;
            tmp4 = (modelica_integer) 1; tmp5 = (i); tmp6 = _m;
            if (!tmp5) {
              else if (!((tmpl5 > 0) && (tmpl4 > tmpl6)) || ((tmpl5 < 0) && (tmpl4 < tmpl6))) {
                for(modelica_integer _h = (modelica_integer) 1; in_range_integer(_h, tmpl4, tmpl6); _h += tmpl5) {

                  tmp3 = get_memory_state();
                  tmp2 = multiply(*integer_array_element_addr_c99_2(_pA, 2, _i, _h)), (*integer_array_element_addr_c99_2(_pB, 2, _i, _j), memory_state);
                  _temp = (tmp2.multiply_rettype_1 + _p temp);
                  restore_memory_state(tmp3);
                }
              }
            }
          }
        }
      }
    }
  }

  restore_memory_state(tmpl1);
}
A.5 MatMult_one_kernels.cl

```c
#include "MatMult_one_parallelFunctions.cl"

_kernel void kernel_0(modelica_integer loop_start,
  modelica Integer loop_end,
  modelica_integer loop_step,
  modelica_integer buffer_size,
  __global void* exec_buffer,
  __global modelica_integer* _pA_data,
  __global modelica_integer* info__pA,
  __global modelica_integer* _pB_data,
  __global modelica_integer* info__pB,
  __global modelica_integer* _pC_data,
  __global modelica_integer* info__pC,
  __global modelica_integer* _pm_ptr,
  __global modelica_integer* _ptemp_ptr)
{

  ///////////////////////////////////////////////////////////////////////////
  //Initialize local memory state
  buffer local_buffer;
  state current_state;
  memory memory_s;
  memory* memory_state = &memory_s;
  memory_state->current_state = &current_state;
  memory_state->local_buffer = &local_buffer;
  initialize_buffer(exec_buffer, buffer_size, memory_state);

  ///////////////////////////////////////////////////////////////////////////
  //thread managing for 'for loops'.
  modelica_integer inner_start = (get_global_id(0) * loop_step) + (loop_start);
  modelica_integer stride = get_global_size(0) * loop_step;

  for(modelica_integer _i = (modelica_integer) inner_start; in_range_integer(_i, loop_start, loop_end); _i += stride)
  {
    ///////////////////////////////////////////////////////////////////////////
    //reconstruct global arrays back into global array structs
    //ALL scalars are treated as PRIVATE!!!!
    integer_array _pA;
    _pA.data = _pA_data;
    _pA.ndims = info__pA[0];
    _pA.dim_size = info__pA + 1;
    integer_array _pB;
    _pB.data = _pB_data;
    _pB.ndims = info__pB[0];
    _pB.dim_size = info__pB + 1;
    integer_array _pC;
    _pC.data = _pC_data;
    _pC.ndims = info__pC[0];
    _pC.dim_size = info__pC + 1;
    modelica_integer pm = *_pm_ptr;
    modelica_integer _ptemp = *_ptemp_ptr;
```

//temporary variables used in the kernel (originally loop)
multiply_rettype tmp5;
state tmp6;
modelica_integer tmp7;
modelica_integer tmp8;
modelica_integer tmp9;
state tmp10;
modelica_integer tmp11;
modelica_integer tmp12;
modelica_integer tmp13;

if (!tmp12) {
} else if (!((tmp12 > 0) && (tmp11 > tmp13)) || (tmp12 < 0) && (tmp11 < tmp13)) {
    for(modelica_integer _j = (modelica_integer) 1; in_range_integer(_j, tmp11, tmp13); _j += tmp12) {
        tmp10 = get_memory_state();
        _ptemp = (modelica_integer) 0;
        tmp7 = (modelica_integer) 1; tmp8 = (1); tmp9 = pm;
        if (!tmp8) {
            } else if (!((tmp8 > 0) && (tmp7 > tmp9)) || (tmp8 < 0) && (tmp7 < tmp9)) {
                for(modelica_integer _h = (modelica_integer) 1; in_range_integer(_h, tmp7, tmp9); _h += tmp8) {
                    tmp6 = get_memory_state();
                    tmp5 = multiply({*integer_array_element_addr_c99_2(_pA, 2, _i, _h)},
                                    {*integer_array_element_addr_c99_2(_pB, 2, _j, _h)}, memory_state);
                    _ptemp = (tmp5.multiply_rettype_1 + _ptemp);
                    restore_memory_state(tmp6);
                }
            }
        }
    }
}
## Appendix B  OpenCL-C Runtime Library Functions

### B.1 Utility Functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char* load_source_file(const char* fileName)</code></td>
<td>Reads kernels from a file.</td>
</tr>
<tr>
<td><code>void ocl_initialize()</code></td>
<td>Initializes OCL environment on first call.</td>
</tr>
<tr>
<td><code>void ocl_get_device()</code></td>
<td>Gets available OCL-enabled devices.</td>
</tr>
<tr>
<td><code>void ocl_create_context_and_comm_queue()</code></td>
<td>Creates context and command queues necessary for launching kernel.</td>
</tr>
<tr>
<td><code>cl_kernel ocl_create_kernel(cl_program program, const char* kernel_name)</code></td>
<td>Extracts and creates a kernel from a given program.</td>
</tr>
<tr>
<td><code>void ocl_set_kernel_args(cl_kernel kernel, int count, ...)</code></td>
<td>Sets Kernel arguments. count is the number of arguments being passed.</td>
</tr>
<tr>
<td><code>void ocl_execute_kernel(cl_kernel kernel)</code></td>
<td>Executes a kernel.</td>
</tr>
<tr>
<td><code>cl_program ocl_build_p_from_src(const char* source, int isfile)</code></td>
<td>Builds a program from a source file containing Kernels or from a text buffer if isfile = 0.</td>
</tr>
<tr>
<td><code>cl_mem ocl_alloc_init(void* src_data, size_t size)</code></td>
<td>Allocates memory space on device and returns the handle to the buffer object also initializes if from host memory IF src_data is not NULL. Size is the actual size in bytes.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>void ocl_create_execution_memory_buffer(device_buffer* d_buff)</code></td>
<td>ATTENTION: This function allocates a large amount of memory to be used for creating arrays inside parallel functions. If this fails try reducing the amount by increasing the <code>#define OCL_BUFFER_SIZE_FRACTION</code> which defines the fraction of memory from the available <code>GLOBAL_MEM_SIZE</code> to be used as buffer.</td>
</tr>
<tr>
<td><code>cl_mem ocl_alloc_init_real_arr(modelica_real* host_array, int a_size)</code></td>
<td>Allocates a double array on device and returns the handle to the buffer object also initializes it from host array IF host array is not NULL. Use size 1 to allocate a Scalar.</td>
</tr>
<tr>
<td><code>cl_mem ocl_alloc_init_integer_arr(modelica_integer* host_array, int a_size)</code></td>
<td>Allocates an integer array on device and returns the handle to the buffer object also initializes if from host array IF host array is not NULL. Use size 1 to allocate a Scalar.</td>
</tr>
<tr>
<td><code>void ocl_copy_device_to_device_real(cl_mem dev_src_array, cl_mem device_dest_array, int a_size)</code></td>
<td>Copies one real buffer to another on the device.</td>
</tr>
<tr>
<td><code>void ocl_copy_device_to_device_integer(cl_mem dev_src_array, cl_mem device_dest_array, int a_size)</code></td>
<td>Copies one integer buffer to another on the device.</td>
</tr>
<tr>
<td><code>void ocl_copy_back_to_host_real(cl_mem dev_output_array, modelica_real* dest_host_array, int a_size)</code></td>
<td>Copies a double array back to host.</td>
</tr>
<tr>
<td><code>void ocl_copy_back_to_host_integer(cl_mem dev_output_array, modelica_integer* dest_host_array, int a_size)</code></td>
<td>Copies an integer array back to host.</td>
</tr>
<tr>
<td><code>void ocl_copy_to_device_real(cl_mem dev_dest_array, modelica_real* src_host_array, int a_size)</code></td>
<td>Copies a double array to ALREADY allocated device buffer.</td>
</tr>
<tr>
<td><code>void ocl_copy_to_device_integer(cl_mem dev_dest_array, modelica_integer* src_host_array, int a_size)</code></td>
<td>Copies an integer array to ALREADY allocated device buffer.</td>
</tr>
<tr>
<td><code>void ocl_error_check(int operation, cl_int error_code)</code></td>
<td>checks error codes.</td>
</tr>
</tbody>
</table>
### B.2 Interfacing Functions

Overloaded functions from real/integer/boolean _array in the c_runtime library for allocating and copying arrays to OpenCL device. These functions are used to keep the interface with the serial C code generation. They are resolved to internal calls to the copy functions defined in the table above.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void ocl_real_matrix_matrix(const char* op, modelica_real* src_1, int M, modelica_real* src_2, int N, modelica_real* dest, int K)</td>
<td>Matrix-matrix operations returning a matrix. Used for easy arrangement of built-in functions.</td>
</tr>
<tr>
<td>void ocl_real_matrix_matrix(const char* op, modelica_real* src_1, int M, int N, modelica_real* dest)</td>
<td>Matrix operations returning a matrix. Used for easy arrangement of built-in functions.</td>
</tr>
<tr>
<td>void ocl_clean_up()</td>
<td>Cleaning up operations</td>
</tr>
<tr>
<td>size_t modelica_array_nr_of_elements(base_array_t *a)</td>
<td>Returns the number of elements in a given modelica base host array.</td>
</tr>
<tr>
<td>size_t device_array_nr_of_elements(device_array_t *a)</td>
<td>Returns the number of elements in a given modelica base device array.</td>
</tr>
<tr>
<td>cl_mem ocl_device_alloc_init(modelica_integer* host_array, size_t size)</td>
<td>Allocates memory on parallel device and initializes the memory from the given host Integer array.</td>
</tr>
<tr>
<td>cl_mem ocl_device_alloc_init(modelica_real* host_array, size_t size)</td>
<td>Allocates memory on parallel device and initializes the memory from the given host Real array.</td>
</tr>
<tr>
<td>cl_mem ocl_device_alloc(size_t size)</td>
<td>Allocates memory buffer on parallel device.</td>
</tr>
</tbody>
</table>

```c
void alloc_integer_array(device_integer_array *dest, int ndims, ...)
```
### B.3 Scalar Assignment Functions

Functions used for copying scalars. Scalars in the normal (serial C) code generation of Modelica are copied by assignment \((a = b)\). However, to be able to copy them between GPU and host CPU we need to change the assignments to copy functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void copy_assignment_helper_integer(modelica_integer* i1, modelica_integer* i2)</code></td>
<td>Resolves to normal integer assignment.</td>
</tr>
<tr>
<td><code>void copy_assignment_helper_integer(device_integer* i1, modelica_integer* i2)</code></td>
<td>Resolves to a copy operation for copying one device integer to another host integer.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>void copy_assignment_helper_integer(modelica_integer* i1, device_integer* i2)</code></td>
<td>Resolves to a copy operation for copying one host integer to another device integer.</td>
</tr>
<tr>
<td><code>void copy_assignment_helper_integer(device_integer* i1, device_integer* i2)</code></td>
<td>Resolves to a copy operation for copying one device integer to another device integer.</td>
</tr>
<tr>
<td><code>void copy_assignment_helper_real(modelica_real* i1, modelica_real* i2)</code></td>
<td>Resolves to normal real assignment.</td>
</tr>
<tr>
<td><code>void copy_assignment_helper_real(device_real* i1, modelica_real* i2)</code></td>
<td>Resolves to a copy operation for copying one device Real to another host Real.</td>
</tr>
<tr>
<td><code>void copy_assignment_helper_real(modelica_real* i1, device_real* i2)</code></td>
<td>Resolves to a copy operation for copying one host Real to another device Real.</td>
</tr>
<tr>
<td><code>void copy_assignment_helper_real(device_real* i1, device_real* i2)</code></td>
<td>Resolves to a copy operation for copying one device Real to another device Real.</td>
</tr>
</tbody>
</table>

### B.4 Miscellaneous Functions

These functions are added to solve a problem with a memory leak when returning arrays from functions. Arrays used to be assigned just like normal scalar variables. This causes the allocated memory on the lhs to be lost when the pointer is replaced with the new one. This fixes the problem for parallel arrays. For serial arrays the memory is restored when the function returns (not dynamic allocation), so the only lose in serial case is visible just until the function returns.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void swap_and_release(device_array* lhs, device_array* rhs)</code></td>
<td></td>
</tr>
<tr>
<td><code>void swap_and_release(base_array_t* lhs, base_array_t* rhs)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Debugging

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void print_array_info(device_real_array* arr)</code></td>
<td>Prints information about a device array. Useful for debugging.</td>
</tr>
<tr>
<td><code>void print_array(real_array_t* arr)</code></td>
<td>Prints a host array. Useful for debugging.</td>
</tr>
</tbody>
</table>
Bibliography


[18] Peter Aronsson, "Automatic Parallelization of Equation-Based Simulation Programs," Linköping University, Dissertation No. 1022,.


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära 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/

© Mahder Gebremedhin