Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on CUDA-enabled GPUs

Kristian Stavåker[2] Daniel Rolls[1] Jing Guo[1] Peter Fritzson[2] Sven-Bodo Scholz[1]

[1]School of Computer Science, University of Hertfordshire, United Kingdom,
{d.s.rolls,j.guo,s.scholz}@herts.ac.uk
[2]Programming Environment Laboratory, Department of Computer Science, Linköping University, Sweden
{peter.fritzson,kristian.stavaker}@liu.se

Abstract

Mathematical models, derived for example from discretisation of partial differential equations, often contain operations over large arrays. In this work we investigate the possibility of compiling array operations from models in the equation-based language Modelica into Single Assignment C (SAC). The SAC2C SAC compiler can generate highly efficient code that, for instance, can be executed on CUDA-enabled GPUs. We plan to enhance the open-source Modelica compiler OpenModelica, with capabilities to detect and compile data parallel Modelica for-equations/array-equations into SAC WITH-loops. As a first step we demonstrate the feasibility of this approach by manually inserting calls to SAC array operations in the code generated from OpenModelica and show how capabilities and runtimes can be extended. As a second step we demostrate the feasibility of rewriting parts of the OpenModelica simulation runtime system in SAC. Finally, we discuss SAC2C's switchable target architectures and demonstrate one by harnessing a CUDA-enabled GPU to improve runtimes. To the best of our knowledge, compilation of Modelica array operations for execution on CUDA-enabled GPUs is a new research area.

Keywords Single Assignment C, Modelica, data parallel programming, OpenModelica, CUDA, GPU, SAC

1. Introduction

Mathematical models, derived for example from discretisation of partial differential equations, can contain computationally heavy operations over large arrays. When simulating such models, using some simulation tool, it might be beneficial to be able to compute data parallel array operations on SIMD-enabled multicore architectures.

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

One opportunity for data parallel execution is making use of graphics processing units (GPUs) which have in recent years become increasingly programmable. The theoretical processing power of GPUs has far surpassed that of CPUs due to the highly parallel structure of GPUs. GPUs are, however, only good at solving certain problems of data parallel nature. Compute Unified Device Architecture (CUDA) [12] is a software platform for Nvidia GPUs that simplifies the programming of their GPUs.

This paper is about unifying three technologies which will be briefly introduced. These are OpenModelica, SAC2C and CUDA. OpenModelica [14] is a compiler for the object-oriented, equation-based mathematical modeling language Modelica [11, 3]. SAC2C [20] is a compiler for the Single Assignment C [19] functional array programming language for efficient multi-threaded execution. We are interested in using SAC2C's CUDA backend [7] that will enable Modelica models to benefit from NVidia graphics cards for faster simulation. Even without this backend SAC2C can generate highly efficient code for array computations, see for instance [17]. We want to investigate the potential of producing SAC code with OpenModelica where opportunities for data parallelism exist.

Work has been planned to enhance the OpenModelica compiler with capabilities to detect and compile arrays of equations defined in Modelica using for-loops into SAC code. From now on these for-loops will be referred to as for-equations. The overall goal of this investigation is to get a clear overview of the feasibility of this technique before any further work. In this paper we investigate how the OpenModelica runtime system and generated code can be amended to call SAC compiled libraries. This is achieved by manually inserting calls to SAC in the code generated from OpenModelica for array based operations. We also examine the feasibility of rewriting parts of the OpenModelica simulation runtime system in SAC. We perform measurements of this new integrated runtime system with and without CUDA and perform stand-alone measurements of CUDA code generated with SAC2C.

Prior work exists on the generation of parallel executable code from equation-based (Modelica) models [1, 10]. In these publications a task graph of the entire equation

system was first generated and then distributed and scheduled for execution. Ways to inline the solver and pipeline computations were also investigated in [10]. However, no handling of data parallel array operations for the purpose of parallel execution in the context of Modelica was done in any of these publications. For work on parallel differential equation solver implementations in a broader context than Modelica see [15, 9, 16].

The remaining sections of the paper are organized as follows. Section 2 introduces the model we wish to simulate thus giving a clear overview of the case study we will use throughout the rest of the paper. In Section 3 we discuss the OpenModelica compiler and the compilation and simulation of Modelica code and also briefly discuss the proposed changes of the OpenModelica compiler needed for the Modelica to SAC compilation. Section 4 contains a description of SAC, gives SAC code that OpenModelica could eventually produce and gives results and analysis from the first experiments of integrating SAC code with OpenModelica. In Section 5 we give an overview of CUDA, how the SAC2C compiler generates CUDA code, results from experiments and an analysis of how this fits in to the overall goals of this paper. Finally, in Section 6, we draw some conclusions and discuss future work.

2. Case Study

In this section we introduce the Modelica model we wish to simulate. The model has a parameter that can be altered to increase or decrease the default number of state variables. The model introduced here is compiled by the OpenModelica compiler into C++ code and linked with a runtime system. The runtime system will simulate the model in several time steps and each time step involves some heavy array computations. Simulation involves, among other things, computing the values of the time-dependent state variables for each time step from a specified start to stop time. Time-independent algorithm sections and functions are also allowed in Modelica.

2.1 One-dimensional Wave Equation PDE Model

The wave equation is an important second-order linear partial differential equation for waves, such as sound waves, light waves and water waves. Here we study a model of a duct whose pressure dynamics is given by the wave equation. This model is taken from [3] (page 584). The present version of Modelica cannot handle partial differential equations directly since there is only the notion of differentiation with respect to time built into the language. Here we instead use a simple discretisation scheme represented using the array capabilities in Modelica. Research has been carried out on introducing partial differential equations into Modelica, see for instance [18].

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

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

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

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

where $p_i = p(x_i + (i-1) \cdot \Delta x, t)$ on an equidistant grid and Δx is a small change in distance. We assume an initial pressure of 1. We get the following Modelica model where the pressure to be computed is represented as a one-dimensional array p of size n, where the array index is the discretized space coordinate along the x-coordinate, and the time dependence is implicit as is common for a continuous-time Modelica variable.

```
1  model WaveEquationSample
2    import Modelica.SIunits;
3    parameter SIunits.Length L = 10 "Length of duct";
4    parameter Integer n = 30 "Number of sections";
5    parameter SIunits.Length dl = L/n "Section length";
6    parameter SIunits.Velocity c = 1;
7    SIunits.Pressure[n] p(each start = 1.0);
8    Real[n] dp(start = fill(0,n));
9  equation
10   p[1] = exp(-(-L/2)^2);
11   p[n] = exp(-(L/2)^2);
12   dp = der(p);
13   for i in 2:n-1 loop
14     der(dp[i]) = c^2 * (p[i+1] - 2 * p[i] + p[i-1]) / dL^2;
15   end for;
16 end WaveEquationSample;
```

On line 1 we declare that our entity should be a model named 'WaveEquationSample'. This model basically consists of two sections: a section containing declarations of parameters and variables (lines 3 to 8) followed by an equation section (lines 9 to 15). A parameter is constant for each simulation run but can be changed between different simulation runs. On line 2 we import the package $SIunits$ from the Modelica standard library.

The two arrays p and dp declared on lines 7 and 8 are arrays of state variables. We can tell that they are arrays of state variables since they occur in derivative expressions in the equation section, thus their values will evolve over time during the simulation run.

The first two equations on lines 10 and 11 state that the first and last pressure value should have a constant value, given by exponent expressions. The third equation on line 12 states that an element in the dp array is equal to the derivative of the corresponding element in the p array. With the present OpenModelica version this equation will result in n scalar equations; we view this kind of equation as an implicit for-equation. The fourth equation on lines 13 to 15 is a for-equation that will result in $n-2$ scalar equations.

3. OpenModelica

OpenModelica is an open source implementation of a Modelica compiler, simulator and development environment for research as well as for educational and industrial purposes. OpenModelica is developed and supported by an international effort, the Open Source Modelica Consortium (OSMC) [14]. OpenModelica consists of a Modelica compiler, OMC, as well as other tools that form an environment for creating and simulating Modelica models.

3.1 The OpenModelca Compilation Process

Due to the special nature of Modelica, the compilation process of Modelica code differs quite a bit from programming languages such as C, C++ and Java. Here we give a brief overview of the compilation and simulation process for generating sequential code. For a more detailed description the interested reader is referred to [2] or [3].

The OpenModelica front-end will first instantiate the model, which includes among other things the removal of all object-oriented structure, and type checking of all equations, statements, and expressions. The output from the OpenModelica front-end is an internal data structure with separate lists for variables, equations, functions and algorithm sections.

For-equations are currently expanded into separate equations. This means that currently each is analysed independently. This is inefficient for large arrays. Thus, for our purpose it would be better if the structure of for-equations is kept throughout the compilation process instead of being expanded into scalar equations.

From the internal data structure executable simulation code is generated. The mapping of time-invariant parts (algorithms and functions) into executable code is performed in a relatively straightforward manner: Modelica assignments and functions are mapped into assignments and functions respectively in the target language of C++. The WaveEquationSample model does not contain any algorithm sections or functions and hence the result of instantiating the WaveEquationSample model in section 2 is one list of parameters, one list of state variables and one list of equations.

The handling of equations is more complex and involves, among other things, symbolic index reduction, topological sorting according to the causal dependencies between the equations and conversion into assignment form. In many cases, including ours, the result of the equation processing is an explicit ordinary differential equation (ODE) system in assignment form. Such a system can be described mathematically as follows.

$$\dot{x} = f(x(t), y(t), p, t) \quad x(t = t_0) = x_0 \; . \tag{3}$$

Here $x(t)$ is a vector of state variables, $\dot{x}$ is a vector of the derivatives of the state variables, $y(t)$ is a vector of input variables, p is a vector of time-invariant parameters and constants, x_0 is a vector of initial values, f denotes a system of statements, and t is the time variable. Simulation corresponds to solving this system with respect to time using a numerical integration method, such as Euler, DASSL or Runge-Kutta.

The output from the OpenModelica back-end consists of a source file containing the bulk of the model-specific code, for instance a function for calculating the right-hand side f in the equation system 3; a source file that contains code for compiled Modelica functions; and a file with initial values of the state variables and of constants/parameters along with other settings that can be changed at runtime.

The ODE equation system in sorted assignment form ends up in a C++ function named `functionODE`. This function will be called by the solver one or more times in each time step (depending on the solver). With the current OpenModelica version, `functionODE` will simply contain a long list of statements originating from the expanded for-equations but work is in progress to be able to keep for-equations throughout the compilation process.

3.1.1 Compilation of WaveEquationSample Model

In this section we illustrate the present OpenModelica compilation process, with the help of the WaveEquationSample model from section 2. In the next section we will discuss how OpenModelica has to be altered if we wish to compile Modelica for-equations into SAC WITH-loops. By instantiating WaveEquationSample we get the following system of equations. All high-order constructs have been expanded into scalar equations and array indices start at 0.

```
p[0] = exp(-(-L / 2.0) ^ 2.0);
p[n-1] = exp(-(L / 2.0) ^ 2.0);
der(p[0]) = p[0];
  .
  .
  .
der(p[n-1]) = p[n-1];
der(dp[0]) = 0;
der(dp[1]) = c^2.0 * ((p[2]+(-2.0*p[1]+p[0])) * dL^-2.0);
  .
  .
  .
der(dp[n-2]) = c^2.0 * ((p[n-1]+(-2.0*p[n-2]+p[n-3])) * dL^-2.0);
der(dp[n-1]) = 0;
```

The above equations corresponds to line 10 to 15 in the original WaveEquationSample model. The rotated ellipsis denotes lines of code that are not shown. The assignments to zero are later removed from the system since they are constant (time independent). From the instantiated code above we can define the following four expressions (where $0 \leq Y \leq n-1$ and $2 \leq X \leq n-3$):

EXPRESSION 3.1.
```
p[Y]
```

EXPRESSION 3.2.
```
c^2.0*((p[2] + (-2.0*p[1] + p[0]))*dL^-2.0)
```

EXPRESSION 3.3.
```
c^2.0*((p[X+1] + (-2.0*p[X] + p[X-1]))*dL^-2.0)
```

EXPRESSION 3.4.
```
c^2.0*((p[n-1]+ (-2.0*p[n-2] + p[n-3]))*dL^-2.0)
```

These expressions correspond roughly to the different types of expressions that occur in the right-hand side of the equation system. The generated code will have the following structure in pseudo code where ... denotes ranges.

```
void functionODE(...) {
  // Initial code
  tmp0 = exp((-pow((L / 2.0), 2.0)));
  tmp1 = exp((-pow(((-L) / 2.0), 2.0)));

  stateDers[0 ... (NX/2)-1] = Expression 3.1;

  stateDers[NX/2] = Expression 3.2;

  stateDers[(NX/2 + 1) ... (NX - 2)] = Expression 3.3;

  stateDers[NX-1] = Expression 3.4;
}
```

The state variable arrays p and dp in the original model have been merged into one array named $stateVars$. There is also a corresponding $stateDers$ array for the derivatives of the state variable arrays. The constant `NX` defines the total number of state variables. The actual generated code (in simplified form) for `functionODE` will look like this:

```
void functionODE(...) {
//--- Initial code ---//
//---
tmp0 = exp((-pow((L / 2.0), 2.0)));
tmp1 = exp((-pow(((-L) / 2.0), 2.0)));

stateDers[0]=stateVars[0 + (NX/2)];
.
.
.
stateDers[(NX/2)-1]=stateVars[((NX/2)-1) + (NX/2)];


stateDers[NX/2] = (c*c) * (stateVars[((NX/2)+1)-(NX/2)]+
                    ((-2.0 * stateVars[(NX/2)-(NX/2)])+
                                        tmp1))/(dL*dL);

stateDers[NX/2 + 1]=(c*c)*(stateVars[((NX/2)+2)-(NX/2)]+
                ((-2.0 * stateVars[((NX/2)+1)-(NX/2)])+
             stateVars[(NX/2)-(NX/2)]))/(dL*dL);
.
.
.
stateDers[NX - 2]= (c*c)*(stateVars[(NX - 1)-(NX/2)]+
                ((-2.0 * stateVars[(NX - 2)-(NX/2)])+
             stateVars[(NX - 3)-(NX/2)]))/(dL*dL);

stateDers[NX-1] = (c*c)*(tmp0 + ((-2.0 *
                stateVars[(NX-1)-(NX/2)])+
          stateVars[(NX-2)-(NX/2)]))/(dL*dL);
//---
//--- Exit code ---//
}
```

This function obviously grows large as the number of state variables increases. Our intention is to rewrite this code and since it is potentially data-parallel we can use the language SAC for this.

3.1.2 Proposed Compilation Process

Several changes to the compilation process have to be performed in order to compile for-equations into SAC WITH-loops. A Modelica for-equation should have the same semantic meaning as before. Right now a for-equation is first expanded into scalar equations. These scalar equations are merged with all other equations in the model and the total set of equations are sorted together. So equations inside the original loop body might end up in different places in the resulting code. This leads to restrictions on what kind of for-equations should be possible to compile into WITH-loops, at least for-equations containing only one equation inside the body should be safe.

In the front-end of the compiler expansion of for-equations into scalar equations should be disabled. We would then get the following code with the WaveEquationSample model.

```
1  p[0] = exp(-(-L / 2.0) ^ 2.0);
2  p[n-1] = exp(-(L / 2.0) ^ 2.0);
3  for i in 0:n-1 loop
4    der(p[i]) = dp[i];
5  end for;
6  der(dp[0]) = 0;
7  for i in 1:n-2 loop
8    der(dp[i]) = c^2.0 * ((p[i+1]+(-2.0*p[i]+p[i-1])) * dL^-2.0);
9  end for;
10 der(dp[n-1]) = 0;
```

New internal data structures that represents a for-equation should be added; one for each internal intermediate form. In the equation sorting phase it might be possible to handle a for-equation as one equation. The equations inside the loop body have to be studied for possible dependencies with other equations outside the loop. The main rule imposed on Modelica models is that there are as many equations as there are unknown variables; a model should be balanced. Checking whether a model is balanced or not can be done by counting the number of equations and unknown variables inside the loop body and adding these numbers with the count from the rest of the model. In the final code generation phase of the compilation process a Modelica for-equation should be mapped into a SAC WITH-loop. This mapping, as long as all checks have proved successful, is relatively straightforward.

4. Single Assignment C

SAC combines a C-like syntax with Matlab-style programming on n-dimensional arrays. The functional underpinnings of SAC enable a highly optimising compiler such as SAC2C to generate high performance code from such generic specifications. Over the last few years several auto-parallelising backends have been researched demonstrating the strengths of the overall approach. These backends include POSIX-thread based code for shared memory multicores [6], CUDA based code for GPGPUs [7] as well as backends for novel many core architectures such as the Microgrid architecture from the University of Amsterdam [8]. All these backends demonstrate the strength of the SAC approach when it comes to auto-parallelisation (see [5, 4, 17] for performance studies).

4.1 Data Parallelism and SAC

Almost all syntactical constructs from SAC are inherited from C. The overall policy in the design of SAC is to enforce that whatever construct looks like C should behave in the same way as it does in C [6].

The only major difference between SAC and C is the support of non-scalar data structures: In C all data structures are explicitly managed by the programmer. It is the programmers responsibility to allocate and deallocate memory as needed. Sharing of data structures is explicit through the existence of pointers which are typically passed around as arguments or results of functions.

In contrast, SAC provides n-dimensional arrays as stateless data structures: there is no notion of pointers whatsoever. Arrays can be passed to and returned from functions in the same way as scalar values can. All memory related issues such as allocations, reuse and deallocations are handled by the compiler and the runtime system. Jointly the compiler and the runtime system ensure that memory is being reused as soon as possible and that array updates are performed in place whenever possible.

The interesting aspect here is that the notion of arrays in SAC actually matches that of Modelica perfectly. Both languages are based on the idea of homogeneously nested arrays, i.e., the shape of any n-dimensional array can always

be described in terms of an n-element shape vector which denotes the extents of the array with respect to the individual axes. All array elements are expected to be of the same element type. Both languages do consider 0-dimensional arrays scalars. The idea of expressing array operations in a combinator style is promoted by both languages.

To support such a combinator style, SAC comes with a very versatile data-parallel programming construct, the WITH-loop. In the context of this paper, we will concentrate our presentation on one variant of the WITH-loop, the `modarray` WITH-loop. A more thorough discussion of SAC is given in [19]. A `modarray` WITH-looptake the general form

with {
($lower_1$ <= *idx_vec* < $upper_1$) : $expr_1$;
⋮
($lower_n$ <= *idx_vec* < $upper_n$) : $expr_n$;
} : **modarray** (*array*)

where *idx_vec* is an identifier and $lower_i$ and $upper_i$, denote expressions for which for any i $lower_i$ and $upper_i$ should evaluate to vectors of identical length. $expr_i$ denote arbitrary expressions that should evaluate to arrays of the same shape and the same element type. Such a WITH-loop defines an array of the same shape as *array* is, whose elements are either computed by one of the expressions $expr_i$ or copied from the corresponding position of the array *array*. Which of these values is chosen for an individual element depends on its location, i.e., it depends on its index position. If the index is within at least one of the ranges specified by the lower and upper bounds $lower_i$ and $upper_i$, the expression $expr_i$ for the highest such i is chosen, otherwise the corresponding value from *array* is taken.

As a simple example, consider the WITH-loop

```
1  with {
2    ([1] <= iv < [4]) : a[iv] + 10;
3    ([2] <= iv < [3]) : 0 * a[iv];
4  } : modarray( a)
```

It increments all elements from index `[1]` to `[3]` by `10`. The only exception is the element at index position `[2]`. As the index `[2]` lies in both ranges the expression associated with the second range is being taken, i.e., it is replaced by `0`. Assuming that `a` has been defined as `[ 0, 1, 2, 3, 4]`, we obtain `[0, 11, 0, 13, 4]` as a result.

Note here, that selections into arrays as well as the WITH-loops themselves are shape-generic, i.e., they can be applied to arrays of arbitrary rank. Assuming that the same WITH-loop is computed with `a` being defined as

```
1  a = [ [  0,  1,  2,  3,  4],
2        [  5,  6,  7,  8,  9],
3        [ 10, 11, 12, 13, 14],
4        [ 15, 16, 17, 18, 19],
5        [ 20, 21, 22, 23, 24]];
```

this would result in an array of the form

```
1      [ [  0,  1,  2,  3,  4],
2        [ 15, 16, 17, 18, 19],
3        [  0,  0,  0,  0,  0],
4        [ 25, 26, 27, 28, 29],
5        [ 20, 21, 22, 23, 24]]
```

Note also, that it was crucial to use `0 * a[iv]` in the second range to make a shape-generic application possible. If we had used `0` instead, the shapes of the expressions would have been detected as incompatible and the application to the array `a` of rank 2 would have been rendered impossible.

4.2 SAC2C, a highly optimising compiler for SAC

SAC2C (see [20] for details) is a compiler for SAC which compiles SAC programs into concurrently executable code for a wide range of platforms. It radically transforms high level programs into efficiently executable C code. The transformations applied do not only frequently eliminate the need to materialise arrays in memory that hold intermediate values but they also attempt to get rid of redundant computations and small memory allocated values as well. Its primary source of concurrency for auto-parallelisation are the WITH-loops. They are inherently data-parallel and, thus, constitute a formidable basis for utilising multi- and many-core architectures. Details of the compilation process can be found in various papers [19, 6].

In order to hook up compiled SAC code into an existing C or C++ application, the SAC2C toolkit also supplies an interface generator named SAC4C. It enables the creation of a dynamically linked library which contains C functions that can be called from C directly.

4.3 Writing OpenModelica Generated Code in SAC

Section 3.1 defined four index expressions for defining the state derivatives array. Their placement into the generated array, `stateDers`, can be represented in SAC as WITH-loop partitions in the following way

```
1  with {
2   ([0] <= iv < [NX/2]) :  Expression 3.1;
3
4   ([NX/2] <= iv <= [NX/2]) :  Expression 3.2;
5
6   ([NX/2] < iv < [NX-1]) :  Expression 3.3;
7
8   ([NX-1] <= iv <= [NX-1]) :  Expression 3.4;
9  } : modarray(stateVars)
```

In legal SAC syntax this can be written as the following.

```
1  with {
2   ([0] <= iv < [NX/2]) :
3       stateVars[iv + (NX/2)];
4
5   ([NX/2] <= iv <= [NX/2]) :
6       (c * c) * (stateVars[(iv+1) - (NX/2)] +
7       ((-2d * stateVars[iv - (NX/2)]) + tmp1))
8       / (dL * dL);
9
10  ([NX/2] < iv < [NX-1]) :
11      (c * c) * (stateVars[(iv+1) - (NX/2)] +
12      ((-2d * stateVars[iv - (NX/2)]) +
13      stateVars[iv-1 - (NX/2)]))
14      / (dL * dL);
15
16  ([NX-1] <= iv <= [NX-1]) :
17      (c * c) *
18      (tmp0 + ((-2d * stateVars[iv - (NX/2)]) +
19      stateVars[iv-1 - (NX/2)])) / (dL * dL);
20 } : modarray(stateVars)
```

The above code defines two single elements within the result array and two large sub-arrays. The equivalent code in OpenModelica-generated C++ and the array the code populates grows linearly with number of state variables.

We modified the OpenModelica-generated C++ code so that instead of computing the ODE system in OpenModelica generated C++, a function call is made in `functionODE` to a SAC function containing the above WITH-loop. Both pieces of code are semantically equivalent. The code was patched so that using a pre-processor macro either the original OpenModelica produced code is invoked or a call is made to a dynamically linked library implemented in SAC that produces the same result.

In the above strategy we make at least one call to SAC in each time step. One alternative to the above strategy of writing this piece of code in SAC is to write the whole solver, or at least the main solver loop, in SAC. We did this for the Euler solver where the main loop looks as follows.

```
1  while (time < stop)
2  {
3    states = states + timestep * derivatives;
4    derivatives = functionODE(states, c, 1, dL);
5    time = time + timestep;
6  }
```

This simple SAC codeblock moves the Euler solver forward in time in steps of `timestep`. Note here that `states` and `derivatives` are arrays and not scalars. The arithmetic operations applied to these arrays are applied to each element with each array. In each time step state variables and state derivatives are calculated. Within each step the SAC version of `functionODE` is invoked.

In the following section we outline experiments where firstly the WITH-loop and secondly the complete SAC Euler solver including the WITH-loop are integrated with OpenModelica.

4.4 Experiments Using SAC with OpenModelica Generated Code

All experiments in this paper were run on CentOS Linux with Intel Xeon 2.27GHz processors and 24Gb of RAM, 32kb of L1 cache, 256Kb of L2 cache per core and 8Mb of processor level 3 cache. SAC2C measurements were run with version 16874 and svn revision number 5625 of OpenModelica was used. C and C++ compilations were performed with Gcc 4.5. The model we used was the WaveEquationSample model introduced in Section 2. The experiments in this section all run sequential code.

Since OpenModelica does not yet have awareness of data parallel constructs inherent in for-equations in the equation section of models it was only feasible to run the compiler for relatively small problem sizes. As mentioned earlier, equations over arrays are expanded into one equation for each element. Even when making some modifications to the OpenModelica code base for the sake of the experiment we were only able to increase the n defined in the original Modelica model to numbers in the low thousands. Anything above this size becomes increasingly infeasible in compile time and resource limits are met at runtime. These problem sizes are big enough still to demonstrate the feasibility of linking SAC modules with C++ code. Computational Fluid Dynamics simulations for instance may however operate on very large arrays.

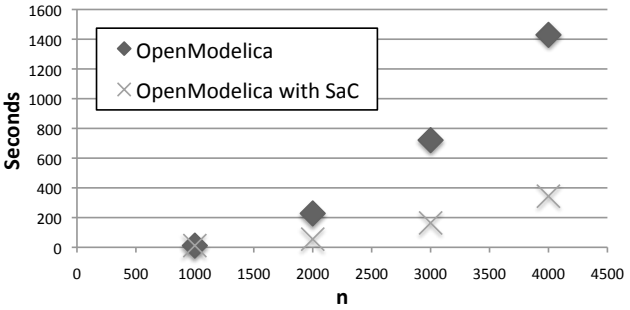

Figure 1. The WaveEquationSample run for different number of sections (n) with functionODE implemented as pure OpenModelica-generated C++ code and as OpenModelica-generated C++ code with functionODE implemented in SAC. Start time 0.0, stop time 1.0, step size 0.002 and without CUDA.

4.4.1 Invoking a SAC WITH-loop from OpenModelica

For our first experiment we altered `functionODE` from the code produced by the OpenModelica compiler so that instead of computing the state derivatives in normal sequential C++ code, a call to SAC code is made. Our SAC code consists primarily of the WITH-loop from Section 4.3. Since in this first experiment only `functionODE` is modified the loop from Section 4.3 is inside the OpenModelica-generated C++ code in this example. The new C++ code that calls the SAC code includes copying of the states array before it is passed to SAC and copying of the array returned by SAC to the state derivatives array in the C++ code. Some copying is required currently because SAC allocates an array with the result. This creates a penalty for the SAC implementation. In a future OpenModelica compiler it is hoped this allocation can be delegated to SAC so that the copying can be removed.

Whilst OpenModelica does an efficient job of taking models and writing code that can make use of different runtime solvers to solve these models, no provisions exist yet for creating highly data parallel code from obviously data parallel models. Our first result shows that if the compiler were to produce SAC code it would be possible to produce code that can feasibly operate on the large arrays that are inevitably required. This in itself can broaden the range of models that OpenModelica could be used to handle.

Figure 1 shows the time taken to simulate the models by running the OpenModelica programs with the two above-described setups for increasing values of n. The experiments were run with the default OpenModelica solver which is the DASSL solver. The simulation was run with timestep 0.002, start time 0 and stop time 1. The results show significant improvements in speed of execution of the SAC implementation already as n raises to values above 1000. For many desired simulations these are relatively small numbers.

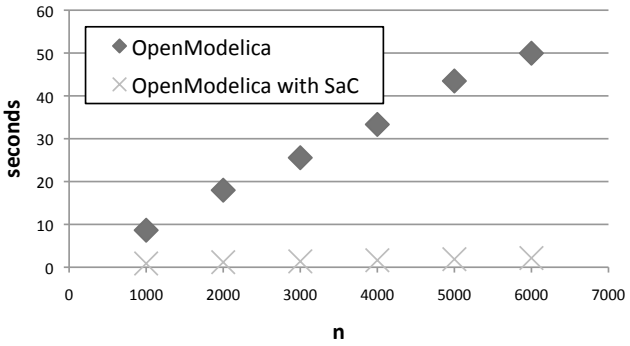

Figure 2. The WaveEquationSample run for different number of sections (n) with functionODE and Euler loop implemented as pure OpenModelica-generated C++ code and as OpenModelica-generated C++ code with functionODE and Euler loop implemented in SAC. Start time 0.0, stop time 100.0, step size 0.002 and without CUDA.

4.4.2 Linking Euler SAC2C Generated Libraries with OpenModelica Generated Code

For a second experiment we used the SAC-implemented Euler solver code consisting primarily of the for-loop in Section 4.3. This code makes calls to the WITH-loop from our previous experiment. For this experiment the OpenModelica Euler solver was used instead of DASSL since the code is simpler but the algorithm is well known and performs a (simple) time step integration, and is hence appropriate for a feasibility study.

This time it was the OpenModelica solver that was patched rather than OpenModelica-generated code. The simulation was run from 0 to 100 seconds with steps of 0.002 seconds. We patched the OpenModelica-solver code so as not to allocate large blocks of memory for the request. This allowed us to run for larger values of n and more time steps. In addition the OpenModelica compiler was patched to not output intermediate time-steps and the SAC code behaves in the same way. As before the SAC version of the code includes additional memory allocation and copying of the data structures passed to and received from the module that will be removed in the future.

Figure 2 shows the time taken to calculate derivatives by running two patched versions of the WaveEquationSample model generated with OpenModelica for increasing values of n.

When using the SAC libraries the performance benefits are significant. We attribute this to OpenModelica's current implementation. Currently globally defined pointers into globally defined arrays are referred to in the code. An array is calculated that is dependent on values in another array and each element of each array is referenced using the globally defined pointers. We believe that the C compiler was unable to efficiently optimise this code where array sizes were large. Improving this will require some changes to the OpenModelica compiler and runtime system which will in themselves certainly have performance benefits.

The model for this experiment operates on a vector of state values. Some computational fluid dynamics applications operate on three-dimensional state spaces. In terms of

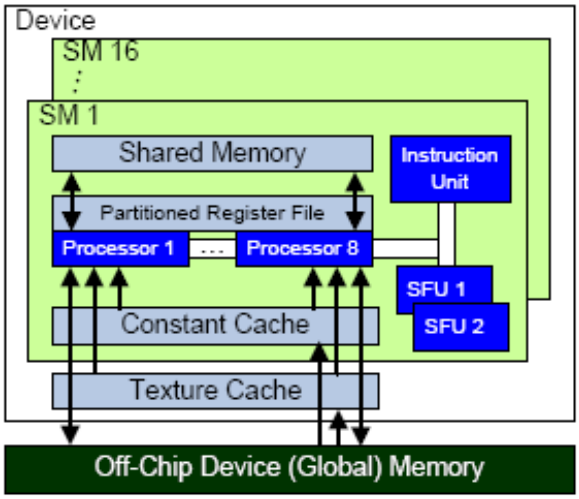

Figure 3. CUDA-enabled GPU hardware architecture.

OpenModelica models these may manifest as three-level nested for-equations. These could map perfectly into SAC where a lot of work [19] has already gone into optimisation for the efficient execution of multi-dimensional arrays taking into account memory access patterns and potential vectorisations. Any future integration of SAC2C into OpenModelica would inevitably make use of these optimisations.

The patches and command line calls used in the experiments in this section can be found in [20].

5. Compute Unified Device Architecture

In recent years, the processing capability of graphics processing units (GPUs) has improved significantly so that they are used to accelerate both scientific kernels and real-world computational problems. Two main features of these architectures render them attractive: large numbers of cores available and their low cost per MFLOP compared to large-scale super-computing systems. Their peak performance figures have already exceeded that of multi core CPUs while being available at a fraction of the cost. The appearance of programming frameworks such as CUDA (Compute Unified Device Architecture) from Nvidia minimises the programming effort required to develop high performance applications on these platforms. To harness the processing power of modern GPUs, the SAC compiler has a CUDA backend which can automatically parallelise WITH-loops and generate CUDA executables. We will briefly introduce the CUDA architecture and programming model before demonstrating the process of compiling the computational kernel of the case study example into a CUDA program.

5.1 Hardware Architecture

Figure 3 shows a high-level block diagram of the architecture of a typical CUDA-enabled GPU. The card consists of an array of Streaming Multiprocessors (SM). Each of these SMs typically contains 8 Streaming Processors (SP).

The organisation of the memory system within CUDA is hierarchical. Each card has a device memory which is common to all streaming multiprocessors, connected through a shared bus, as well as externally. To minimise the contention at that bus, each streaming multiprocessor has a relatively small local memory referred to as Shared Memory shared across all streaming processors. In addition to this, each streaming processor has a set of registers.

5.2 The CUDA Programming Model

The CUDA programming model assumes that the system consists of a host, which is a traditional CPU, and one or more CUDA-enabled GPUs. Programs start running sequentially at the host and call CUDA thread functions to execute parallelisable workloads. The host needs to transfer all the data that is required for computation by the CUDA hardware to the device memory via the system bus. The code that is to be executed by the cores is specified in a function-like unit referred-to as a *kernel*. A large number of threads can be launched to perform the same kernel operation on all available cores at the same time, each operating on different data. Threads in CUDA are conceptually organised as a 1D or 2D grid of blocks. Each block within a grid can itself be arranged as a 1D, 2D or 3D array of cells with each cell representing a thread. Each thread is given a unique ID at runtime which can be used to locate the data upon which they should perform the computation. After each kernel invocation, blocks are dynamically created and scheduled onto multiprocessors efficiently by the hardware.

5.3 Compiling SAC into CUDA

Most of the high level array operations in SAC are a composition of the fundamental language construct - the data parallel WITH-loop. The CUDA backend of the SAC compiler identifies and transforms parallelizable WITH-loops into code that can be executed on CUDA-enabled graphic GPUs. Here we demonstrate the process of compiling the computational kernel of the wave equation PDE model, expressed as a WITH-loop, into equivalent CUDA program (See Figure 4). The compilation is a two-staged process:

- **Phase I**: This phase introduces host-to-device and device-to-host transfers for data arrays referenced in and produced from the WITH-loop. In the example shown, array `stateVars` introduces host-to-device transfers. The final result computed within the GPU, the array $\texttt{stateDers}^{\texttt{D}}$, introduces a device-to-host transfer.
- **Phase II**: This phase lifts computations performed inside each generator as a separate CUDA kernel. In this example, four kernels (i.e. `k1`, `k2`, `k3` and `k4`) are created, each corresponds to one WITH-loop generator.

CUDA kernels are invoked with a special syntax specifying the CUDA grid/block configuration. In the example, each kernel invocation creates a thread hierarchy composed of a one-dimensional grid with one-dimensional blocks in it. Device array variables $\texttt{stateDers}^{\texttt{D}}$ and $\texttt{stateVars}^{\texttt{D}}$, along with scalars(`c`, `dL`, `tmp0`, `tmp1`), are passed as pa-

```
stateDers = with {
       ( [0] <= iv < [256]) :
          stateVars[iv+256];
       ( [256] <= iv < [257]) :
          (c*c)*(stateVars[(iv+1)-256]+
          ((-2f*stateVars[iv-256])+tmp1))/(dL*dL);
       ( [257] <= iv < [511]) :
          (c*c)*(stateVars[(iv+1)-256]+
          ((-2f*stateVars[iv-256])+
          stateVars[(iv-1)-256]))/(dL*dL);
       ( [511] <= iv <= [512]) :
          (c*c)*
          (tmp0*((-2f*stateVars[iv-256])+
          stateVars[(iv-1)-256]))/(dL*dL);
   } : modarray(stateVars);
```

⇓ ⇓ ⇓

```
stateVars^D = host2device( stateVars);
stateDers^D = with {
       ( [0] <= iv < [256]) :
          stateVars^D[iv+256];
       ( [256] <= iv < [257]) :
          (c*c)*(stateVars^D[(iv+1)-256]+
          ((-2f*stateVars^D[iv-256])+tmp1))/(dL*dL);
       ( [257] <= iv < [511]) :
          (c*c)*(stateVars^D[(iv+1)-256]+
          ((-2f*stateVars^D[iv-256])+
          stateVars^D[(iv-1)-256]))/(dL*dL);
       ( [511] <= iv < [512]) :
          (c*c)*
          (tmp0*((-2f*stateVars^D[iv-256])+
          stateVars^D[(iv-1)-256]))/(dL*dL);
   } : modarray(stateVars^D);
stateDers = device2host( stateDers^D);
```

⇓ ⇓ ⇓

```
stateVars^D = host2device( stateVars);
dim3 grid1(16), block1(16);
k1<<<grid1, block1>>>
   ( stateDers^D, stateVars^D);
dim3 grid2(1), block2(16);
k2<<<grid2, block2>>>
   ( stateDers^D, stateVars^D, c, dL, tmp1);
dim3 grid3(16), block3(16);
k3<<<grid3, block3>>>
   ( stateDers^D, stateVars^D, c, dL);
dim3 grid4(1), block4(16);
k4<<<grid4, block4>>>
   ( stateDers^D, stateVars^D, c, dL, tmp0);
stateDers = device2host( stateDers^D);

__global__ void k1( float *stateDers^D,
                    float *stateVars^D) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x;
   stateDers^D[ivs] = stateVars^D[ivs+256];
}

__global__ void k2( float *stateDers^D,
                    float *stateVars^D) {
                    float c, float dL, float tmp1) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x+256;
   stateDers^D[ivs] = (c*c)*(stateVars^D[(ivs+1)-256]+
                ((-2f*stateVars^D[ivs-256])+tmp1))
                /(dL*dL);
}

__global__ void k3( float *stateDers^D,
                    float *stateVars^D) {
                    float c, float dL) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x+257;
   stateDers^D[ivs] = (c*c)*(stateVars^D[(ivs+1)-256]+
                ((-2f*stateVars^D[ivs-256])+
                stateVars^D[(ivs-1)-256]))/(dL*dL);
}

__global__ void k4( float *stateDers^D,
                    float *stateVars^D) {
                    float c, float dL, float tmp0) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x+511;
   stateDers^D[ivs] = (c*c)*
                (tmp0*((-2f*stateVars^D[ivs-256])+
                stateVars^D[(ivs-1)-256]))/(dL*dL);
}
```

Figure 4. Compiling an example WITH-loop to CUDA.

rameters to the kernels. Each thread running the kernel calculates the linear memory offset of the data being accessed using a set of built-in variables `blockIdx`, `blockDim` and `threadIdx`. This offset is then used to access array `stateVars`^{D}. The final result is written into `stateDers`^{D}.

5.4 Running SAC modules with CUDA

For experiments with CUDA we used CUDA 3.0 and a Tesla C1060. C and C++ compilations were performed with GCC 4.5 except for those invoked by CUDA which used GCC 4.3 since CUDA currently does not support the latest GCC compilers.

In the following experiment more SAC code was written to call the SAC code from Section 4.4.2. When doing this we were able to increase n to much higher numbers to simulate future potential of models with very large numbers of states. Our SAC program simply calls the SAC Euler code from Section 4.4.2 in the same way that OpenModelica called it in our previous experiment. The values used to call the function are explicitly hidden from the compiler so that SAC2C cannot make use of this knowledge when optimising the function. Due to current limitations in older Nvidia cards like the Tesla in our experiment we have used floats not doubles for this experiment. When running on newer cards this modification is not necessary. All parameters used for this experiment match the previous experiment except that we were able to raise n by a factor of one-thousand. It is currently not feasible to raise n this high with code generated with the current OpenModelica compiler since it tries to generate instructions for each extra computation required rather than using a loop and because it allocates large blocks of memory that depend on the size of n. With the exception of these two changes the SAC code matches that of the previous experiment.

Two versions of SAC programs were compared. In one SAC2C was invoked with an option specifying that the code should be highly optimised by the C compiler. In the other an option was added to invoke the SAC2C CUDA backend. The code was linked against both the CUDA libraries and libraries for our SAC module. The SAC code used for the CUDA and non-cuda library versions is identical. As before the patches and command line calls used in the experiment can be found at [20].

The results from the experiment are shown in Figure 5. In both cases time increases linearly as n increases. SAC with CUDA performs significantly better than SAC without CUDA. This is because in each iteration of derivative calculation the input array, i.e. the state variables, is a function of only the `timestep` and the derivative array computed in the previous iterations. This means both arrays can be retained on the GPU main memory without the need of transferring back to the host. The CUDA backend is capable of recognising this pattern and lifting both transfers before and after the WITH-loop (see Figure 4) out of the main stepping loop. With application of this optimisation, each iteration contains pure computation without the overhead of host-device data transfers. Moreover, the large problem sizes provide the CUDA architecture with abundance of data parallelism to exploit to fully utilise the available re-

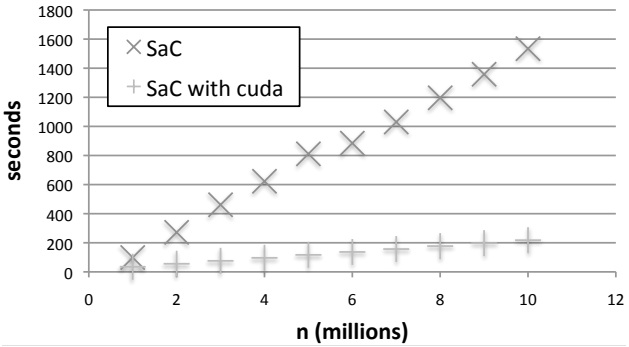

Figure 5. WaveEquationSample state derivative calculations embedded with an Euler loop, both written entirely in SAC, run sequentially and with CUDA for varying number of sections (n). Start time 0.0, stop time 100.0 and step size 0.002.

sources. Given the stencil-like access pattern in the computational kernel, potential data reuse can be exploited by utilizing CUDA's on-chip shared memory. This continued work will further improve the performance.

All experiments with SAC in this paper produced code to either run sequentially or with the CUDA target. For future work we'd like to try the experiments with SAC's already mature pthread target which has already shown positive results [17]. Work is underway to produce a target of C code with OpenMP directives to make use of parallelisation work in C compilers. All these projects and future projects provide interesting potential for future work.

Note that the experiments have demonstrated a benefit when using SAC that materialises for each group of time steps for which intermediate steps are stored. The storing of both states and state derivatives between intermediate time-steps can be a requirement for users of these models and the effect on performance as the number of save points is increased is the next obvious study. When interfacing with SAC there are two ways of doing this. One is to call a SAC function for every group of steps for which a save point is desired. If OpenModelica were to no longer allocate memory for the entire result and instead write the result to file periodically then this method would be the most scalable but it would give SAC2C little opportunity for optimisation. Alternatively one call to the SAC module could be made and SAC could return all desired save points. This would give SAC2C the best chance for optimisation but would have the constraint that the result from the function call would need to be small enough to fit into memory. Ideally a hybrid approach might be desired.

6. Conclusions

Modelica code often contains large arrays of variables and operations on these arrays. In particular it is common to have large arrays of state variables. As of today the OpenModelica compiler has limited support for executing array operations efficiently or for exploiting parallel architectures by, for instance using CUDA-enabled GPU-cards. This is something we hope will be improved in future versions of the compiler and runtime system.

In this work we have investigated ways to make use of the efficient execution of array computations that SAC and SAC2C offer, in the context of Modelica and OpenModelica. We have shown the potential of generating C++ code from OpenModelica that can call compiled SAC binaries for execution of heavy array computations. We have also shown that it is possible to rewrite the main simulation loop of the runtime solver in SAC, thus avoiding expensive calls to compiled SAC binaries in each iteration.

In doing this we have shown the potential for the use of SAC as a backend language to manage the efficient execution of code fragments that the OpenModelica compiler can identify as potentially data parallel. To the best of our knowledge this has not been done with a Modelica compiler before. The integration with SAC allowed experiments to be run with a larger number of state variables than was previously feasible. Moreover, we have shown that the SAC2C compiler can both produce efficient sequential code and produce code targeted for an underlying architecture supporting parallel execution. In this case we exploited the potential of a GPGPU. SAC2C can do this without any changes to the SAC code itself.

Nvidia has recently released the new Fermi architecture [13] which has several improvements which are important in the area of mathematical simulation, a cache hierarchy, more shared memory on the multiprocessors and support for running several kernels at a time.

The next planned stage in this on-going project is to enhance the OpenModelica compiler to pass for-equations through the compiler and to generate SAC source code and compile it automatically.

Acknowledgments

Partially funded by the European FP-7 Integrated Project Apple-core (FP7-215216 — Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs).

Partially funded by the European ITEA2 OPENPROD Project (Open Model-Driven Whole-Product Development and Simulation Environment) and CUGS Graduate School in Computer Science.

References

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

[2] Francois Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, 2006.

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

[4] Clemens Grelck. Implementing the NAS Benchmark MG in SAC. In Viktor K. Prasanna and George Westrom, editors, *16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.

[5] Clemens Grelck and Sven-Bodo Scholz. HPF vs. SAC — A Case Study. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference (Euro-Par'00), Munich, Germany*, volume 1900 of *Lecture Notes in Computer Science*, pages 620–624. Springer-Verlag, Berlin, Germany, 2000.

[6] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[7] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Towards Compiling SaC to CUDA. In Zoltan Horváth and Viktória Zsók, editors, *10th Symposium on Trends in Functional Programming (TFP'09)*, pages 33–49. Intellect, 2009.

[8] Stephan Herhut, Carl Joslin, Sven-Bodo Scholz, and Clemens Grelck. Truly nested data-parallelism. compiling sac to the microgrid architecture. In: IFL'09: Draft Proceedings of the 21st Symposium on Implementation and Application of Functional Languages. SHU-TR-CS-2009-09-1, Seton Hall University, South Orange, NJ, USA., 2009.

[9] Matthias Korch and Thomas Rauber. Scalable parallel rk solvers for odes derived by the method of lines. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 830–839. Springer, 2003.

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

[11] Modelica and the Modelica Association, accessed August 30 2010. `http://www.modelica.org`.

[12] CUDA Zone, accessed August 30 2010. `http://www.nvidia.com/object/cuda_home_new.html`.

[13] Fermi, Next Generation CUDA Architecture, accessed August 30 2010. `http://www.nvidia.com/object/fermi_architecture.html`.

[14] The OpenModelica Project, accessed August 30 2010. `http://www.openmodelica.org`.

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

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

[17] Daniel Rolls, Carl Joslin, Alexei Kudryavtsev, Sven-Bodo Scholz, and Alexander V. Shafarenko. Numerical simulations of unsteady shock wave interactions using sac and fortran-90. In *PaCT*, pages 445–456, 2009.

[18] Levon Saldamli. *PDEModelica A High-Level Language for Modeling with Partial Differential Equations*. PhD thesis, Linköping University, 2006. Dissertation No. 1016.

[19] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[20] Single Assignment C Homepage, accessed august 30 2010. `http://www.sac-home.org`.