Technical Report CSTN-087

Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations

K.A. Hawick*and D.P. Playne†

July 2010

Abstract

We show how compiler technology can generate fast and efficient yet human-readable data-parallel simulation code for solving certain partial differential equation (PDE) based problems. We present a code parser and generator based on an ANTLR grammar and tree walking approach that transforms a mathematical formulation of an equation such as the Cahn-Hilliard family into simulation software in C++ or in NVIDIA's Compute Unified Device Architecture (CUDA) language for programming Graphical Processing Units (GPUS). We present software architectural ideas, generated specimen code and detailed performance data on modern GPUs. We discuss how code generation techniques can be used to speed up code development and computational run time for related complex system simulation problems.

Keywords: automatic code generation; compiler; grammar; partial differential equation; finite difference, stencil; GPU.

1 Introduction

Many problems in computational science and engineering can be formulated in terms of partial differential equations (PDEs). A common simulation pattern involves the time-integration of an initial value model where the

*email: k.a.hawick@massey.ac.nz, Computer Science, Massey University Albany, North Shore 102-904, Auckland, New Zealand

†email: d.p.playne@massey.ac.nz

system is defined on a spatial mesh with spatial calculus operators in the equation. Although the mathematical and numerical methods for solving such problems are well known, it is still a tedious and error-prone task to write correct and efficient software for a new problem. The effort required to code software for a new hardware architecture such as multi-core CPUs and highly-data parallel accelerator devices such as Graphical Processing Units (GPUs) is even greater.

Although a great deal of techniques [1] are known for building optimising compilers [2, 3, 4] the goal of automatic parallelising compilation remains elusive. Some important progress was made for some data-parallel constructs[5, 6] and relatively recently for some regular data problems using GPUs[7]. However it seems likely that there are some general problems that compiler generators will probably never be able to do completely, without programmer assistance[8]. More optimistically however it **is** feasible to look at some specific classes of application domain problems and use compiler ideas to address them.

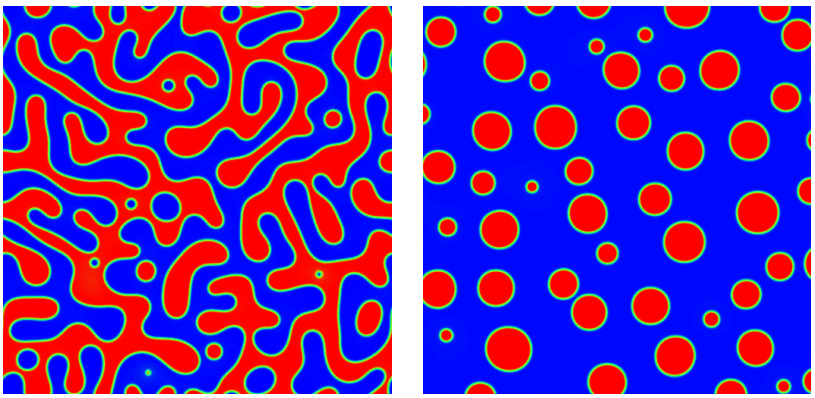

Figure 1: Two-dimensional segments of field solutions of the Cahn-Hilliard equation - time integrated for over 20,000 steps with a 50/50 concentration mix on the left and a 25/75 mix on the right.

In this paper we report on how modern compiler technology can be used to make a software generator tool that can create fast and readable data-parallel software for solving some PDE based problems. We focus on the Cahn-Hilliard equation for materials phase separation modelling which is first-order in the time derivative but fourth order in spatial calculus operators - an illustrative example of which is shown in Figure 1. We explain how a plain ASCII expression of the mathematical equation can be parsed and used to generate either CPU software in a language like C or C++, but also how we can generate GPU code written in a language like

NVIDIA's Compute Unified Device Architecture (CUDA)[9]. We discuss how different numerical techniques such as the time integration method or the finite differencing spatial stencil operators can be packaged in a library separate from the main code generator. We show how some extra configuration information can be specified by the user as part of the overall "simulation tree" to control the methods and parameters used in solving a particular PDE problem.

In addition to the considerable parallel performance our generated code can achieve compared to hand-generated software, we show also how a relatively minor change to the mathematical specification of the equation allows a whole new code to be generated relatively trivially. These approaches combined in saving on: programmer time; correctness testing effort; and production run time through data parallelism, support investigation of whole families of problems that would hitherto have taken a lot longer to tackle.

A number of software systems and algebraic problem solving environments allow users to automatically generate solver source code in standard programming languages such as Fortran[10, 11, 12]. A number of systems also address the problem of generating parallel code[13]. Research projects[14, 15] and commercial problem solving systems such as Matlab[16] or Mathematica[17] also support code generation from a mathematical formulation of equations. We are interested however in addressing both issues and therefore in generating highly performance optimised parallel programs that can be run on current generation processors and on accelerators such as Graphical Processing Units and other platforms. It is generally a hard problem to automatically parallelise serial programs written in traditional languages, but an advantage of starting from the underpinning mathematics is that more information on the structure of the calculation and its potential parallelisation is actually exposed and available for a code generation tool to exploit.

However, a more direct parallel code generation approach is now possible due to the advent of portable parallel languages such as OpenCL and the general revitalisation of data-parallel computing that has been stimulated by cheap GPUs and other accelerators. Generally exposure to the original mathematics of a PDE along with knowledge of the numerical discretisation scheme desired, gives a software generation tool more power to identify the parallelisation potential of the problem and specifically target a high performance implementation. The roles and emphasis of performance and portability are then reversed in this approach and portability may be achieved through implementations of the OpenCL target code.

In fact using our approach we believe it is possible to construct a number of inherent templates that will support generation of several target languages including CUDA, OpenMP or OpenCL. There appears to be

considerable scope for automatic generation of stencil source code that makes use of heuristics and other practical experience to achieve optimised implementations on present and emerging multicore processing devices[18].

Although there are a number of mathematical and numerical approaches such as finite-elements that can be expressed using this approach to code generation, we focus in this paper on regular mesh problems that can be solved using finite-difference methods. We do discuss different and employ numerical time integration techniques but we focus on stencil operators for the spatial calculus. Many problems in computational science and engineering can be formulated as stencils whereby a regular pattern of neighbouring data values are used to iteratively update a central value. A range of numerical methods for iteratively solving PDEs can be expressed in terms of stencil operators for spatial derivatives, the Laplacian operator, gradient, curl and other higher order operators.

The idea of generating PDE solver software is not new. As long ago as 1970, Cardenas and Karplus experimented with manually written programs that combine both translation and generation in a single *ad hoc* stage [19] partial differential equation language (PDEL) based on PL/1 syntax. Some important work is being done by Logg and collaborators on the semi-automatic generation of **Finite Element** algorithms. The FENICS[20] and DOLFIN[21] projects take a somewhat different approach to the one we do, making more heavy use of linear algebraic methods and the associated separately-optimised software for solving linear algebra and matrix-oriented problems such as BLAS[22], BLACS[23], LAPACK[24] and ScaLAPACK[25]. While it is also possible to formulate the Finite Difference methods that we employ using full matrix methods too, we focus (for the present at least) on direct methods and formulations for regular meshes that do not need full matrices and that make use of explicit sparse data storage methods. This allows us the luxury of worrying less about storage space for the spatial calculus and thus being able to experiment more readily with higher-order time-integration methods which themselves require multiple copies of the field data for intermediate fractional time steps. It is also of course a challenge to accommodate as large as possible model sizes within GPU memory for reasons discussed below in Section 4.

Stencil operators have been in use for parallel program optimisation for some years[26, 27]. In the case of image operators where a particular stencil might be well known with specific name it is straightforward to develop an optimised software library of optimised operator routines. For solving PDEs it is however harder to develop a general purpose library and automated code generation for a particular PDE with particular initial/boundary conditions and solver algorithm is more attractive[28]. Datta and collaborators discuss stencil generation using Lisp-parsing of Fortran-like expressions for the mathematics of the equation under consideration

and a system that generates the stencil in C or Fortran code[29]. It is then possible to apply the standard apparatus and systems of parallel programming such as message passing, parallel compiler macros or supercomputer vendor proprietary optimisation tools to obtain a working parallel implementation that can target modern multicore devices amongst other platforms[30].

In this article we discuss the general form of applicable partial differential field equation problems in Section 2. In Section 3 we focus in on the Cahn-Hilliard equation and discuss how it gives rise to a family of PDEs that would be tedious and error-prone to code for separately. Since GPUs form our principle platform target for the work reported in this paper, we give a brief summary of the salient architectural issues for GPUs in Section 4. The structure and operation of our parser and code generator is given in Section 5. We present some generated code examples and associated run-time performance data in Section 6 and discuss associated issues in Section 7 including numerical methods (Section 7.1) and floating point data types (Section 7.2). We offer some conclusions in and ideas for future work in Section 8.

2 Solving Partial Differential Field Equations

Many interesting problems in physics and other science areas can be formulated in terms of partial differential field equations that evolve in time. These problems fall into the general pattern:

$$\frac{du(\mathbf{r},t)}{dt} = \mathcal{F}(u,\mathbf{r},t) \tag{1}$$

where the time dependence is first order and the spatial dependence in the right hand side is often in terms of partial spatial derivatives such as $\nabla_x, \nabla_y, \nabla_z, \nabla^2, \nabla^2 \cdot \nabla^2, ...$

Some well known problems that fit this pattern are:

The **Cahn-Hilliard equation**[31, 32] which is expressed in terms of a scalar field u:

$$\frac{\partial u}{\partial t} = \mathrm{m}\nabla^2\left(-\mathrm{b}u + \mathrm{U}u^3 - \mathrm{K}\nabla^2 u\right) \tag{2}$$

where it is usual to truncate the series in the free energy[33] at the u^4 term, although some work has used up to the u^6 term [34]. Since we use the Cahn-Hilliard equation as our primary illustrative example for this paper, we give more details of its derivation in Section 3 below.

The **Time-Dependent Ginzburg Landau equation** [35] in terms of a complex scalar field u:

$$\frac{\partial u}{\partial t} = -\frac{p}{i}\frac{\partial^2 u}{\partial x^2} - \frac{q}{i}\left|u\right|^2 u + \gamma u \tag{3}$$

The field variable can also be a vector, such as the population variables in a spatially dependent **Lotka-Volterra system of equations**[36, 37].

$$\frac{d\mathbf{P}}{dt} = \mathcal{F}(\mathbf{P}) \tag{4}$$

where $\mathbf{P}$ might be vector of several population variables for predator and prey species and $\mathcal{F}$ might incorporate a matrix of cross-coupling terms and spatial calculus operators such as a Laplacian, in whatever dimensions (eg 2 or 3) the problem is posed.

In some cases the full details of the right-hand sides of these sort of equations are known and immutable parts of the field model. In other cases a family of equations can be generated by using different expansions or approximates. A good example is the Cahn-Hilliard equation where the free-energy term is usually approximated by a polynomial with second and fourth order terms, but alternatives such as including higher order terms make sense but are hard (tedious and error-prone) to implement.

A powerful idea to address implementation difficulties is therefore a software tool that can help generate lines of code in a standard programming language like C, C++, D, Java, Fortran, that implements one of the standard numerical approaches to solving the equation in question. There are some well known lines of approach to solving the numerical integration in time - storing the state of the entire model field that expresses the right hand side and applying second order methods such as the midpoint method (aka second-order Runge-Kutta) or higher-order methods such as the well-known Runge-Kutta Fourth order method as appropriate. We discuss how to tackle time integration aspects in Section 5.2. The spatial terms included in the right hand side can also be tackled with well known finite-difference stencil operators for operators such as the Laplacian and other spatial derivatives and we discuss how these can be incorporated into a code generator in Section 5.3.

Problem solving environments do allow some attempts at automatic numerical solution of equations such as these, but they do not necessarily produce speed-optimal or indeed maintainable lines of code. It is important that like terms be gathered together for efficient computation but there are also numerical stability considerations that affect the best safe way to combine numerical terms. We discuss these issues and how the code generator can combine stencil operators to produce good software solutions in Section 5.3

3 Cahn-Hilliard Equations

We provide a brief derivation of the Cahn-Hilliard example as a worked example. We show how some assumptions about the free energy that differ

from those normally made in the literature give rise to a family of equations. It is normally quite a lot of error-prone work to generate a new simulation program for each modified equation. Using our code generation tool however it is relatively trivial and we can rapidly experiment with additional terms and model assumptions.

The Cahn-Hilliard equation is usually derived from the notion that in a real material alloy the number of atoms of a given species is conserved and hence the concentration field must also be. This is expressed by:

$$\frac{1}{V}\int_V u(\mathbf{r},t)\,d\mathbf{r} = c_A \tag{5}$$

where V is the system volume, and c_A the concentration of atomic species A. This conservation law implies that the local concentration field obeys a continuity equation of the form:

$$\frac{du(\mathbf{r},t)}{dt} + \nabla.\mathbf{j}(\mathbf{r},t) = 0 \tag{6}$$

which defines a concentration current $\mathbf{j}(\mathbf{r},t)$, assumed to be proportional to the gradient of the *local* chemical potential difference $\mu(\mathbf{r},t)$ with constant of proportionality m, the mobility.

$$\mathbf{j}(\mathbf{r},t) = -m\nabla\mu(\mathbf{r},t) \tag{7}$$

The chemical potential difference is, by definition:

$$\mu(\mathbf{r},t) = \frac{d\mathcal{F}\{u(\mathbf{r},t)\}}{du(\mathbf{r},t)} \tag{8}$$

where $\mathcal{F}$ is the Landau functional. Differentiating this functional with respect to u and assuming a scalar mobility yields the chemical potential difference as:

$$\mu(\mathbf{r},t) = \left.\frac{\partial f}{\partial u}\right|_T - \frac{R^2}{d}k_bT\nabla^2u(\mathbf{r},t) \tag{9}$$

which when substituted into the continuity equation 6 gives the Cahn-Hilliard equation[31] for the concentration field.

$$\frac{\partial u(\mathbf{r},t)}{\partial t} = m\nabla^2\left(\left.\frac{\partial f\left(u(\mathbf{r},t)\right)}{\partial u}\right|_T - K\nabla^2u(\mathbf{r},t)\right) \tag{10}$$

where the parameter K is defined as:

$$K = \frac{R^2}{d} k_b T \tag{11}$$

Expanding equation 10 we obtain finally:

$$\frac{\partial u}{\partial t} = \mathrm{m}\nabla^2 \left(-\mathrm{b}u + \mathrm{U}u^3 - \mathrm{K}\nabla^2 u\right) \tag{12}$$

Writing the equation in a form more amenable to managing multiple numerical parameters we then have:

$$\frac{\partial u}{\partial t} = \mathrm{m}\nabla^2 \left(A_1 u + A_3 u^3 + A_{2n+1} u^{2n+1} - \mathrm{K}\nabla^2 u\right) \tag{13}$$

Where numerically: $A_1 \equiv -\mathrm{b} = -1, A_3 \equiv \mathrm{U} = +1, A_5 = -1, A_7 = +1, ...$, and $A_{2n} \equiv 0, n = 0, 1, 2, 3, ..., \forall n$, for as many terms as we care to make use of in the free energy approximation polynomial. Our equation parser allows us to change n relatively trivially and generate efficient, correct and yet readable code to address different members of the resulting family of equations.

4 GPU Architectural Feature Summary

Since data-parallel GPUs form a key target for the generated code we report on in this paper, we provide a brief outline summary of the key architectural features of typical GPUs. GPUs can be used for many parallel applications in addition to their originally intended purpose of graphics processing algorithms. Relatively recent innovations in high-level programming language support and accessibility have caused the wider applications programming community to consider using GPUs in this way. The term general-purpose GPU programming (GPGPU) [38] has been adopted to describe this rapidly growing applications level use of GPU hardware.

GPUs contain many, low power, SIMT (Single Instruction Multiple Thread [39]) processor cores. GPUs can manage many millions of threads in hardware and schedule them for execution on the processor cores. The disadvantage of this architecture is that memory access performance is highly dependent on access patterns. GPUs contain several types of memory that must be explicitly used by the programmer. However, the new NVIDIA FERMI architecture GPUs have automatic caching on the main memory, easing restrictions on memory use.

Figure 2 shows the essential software architecture of a GPU program written in NVIDIA's Compute Unified Device Architecture (CUDA) language. The CPU code (written in plain ordinary "C") treats the GPU (CUDA) code as special purpose subroutines or "GPU kernels" which are

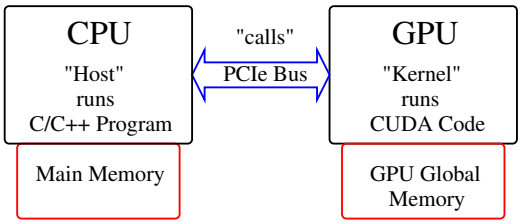

Figure 2: CPU/GPU Programming Architecture

run on the GPU and report back to the calling host CPU program. Any data that is used by the kernels must be copied into the GPU global memory. In previous work [40, 41] we have discussed the performance improvements that can be gained by using the optimised GPU memory types.

5 Parser and Generator Structure

In this section we describe the various components of our parser-generator software prototype - known as "Simulation Targeted Automatic Reconfigurable Generator of Abstract Tree Equations (STARGATES)." Our system creates simulation code from a mathematical description of a partial-differential field equation. It will parse an ASCII representation of the equation in a mathematical form. A tree representing the equation can be created from this parsed textual representation. This tree is combined with information about integration methods and stencils to create an abstract "simulation tree" that represents all the vital information about the simulation. This tree can be traversed to generate code that performs the simulation in any desired output programming language. Some additional configuration information about the simulation must also be supplied to define properties of the simulation such as system size, dimensionality etc. The architectural structure of the STARGATES software prototype is shown in Figure 3.

The equation file containing a mathematical description of the equation is given as the first input to the system. The Equation Parser will read this file and construct a tree which represents the equation. This equation tree is transformed into a simulation tree by combining it with a selected integration tree supplied by the Integration Library and with Stencils from the Stencil Library. This simulation tree can be inspected and traversed by an Output Generator which (using the config file for the specific simulation parameters) will generate output simulation code. We thus have a mechanism to manage and separate code generation information from run-management information.

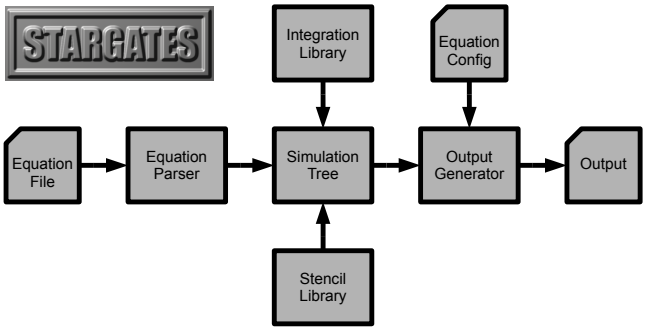

Figure 3: The diagram showing the structure and logical flow of STARGATES. The system takes an equation description and a configuration file as input and the output generator produces an output file.

The advantage of this structure is that the output generators are dependent only on the Simulation Tree. This means that it is a relatively simple process to create a new target language generator as the output depends purely on the way the output generator interprets the information in the simulation tree. In this present paper we focus on the generation of CPU serial C/C++ code and data-parallel GPU CUDA code.

5.1 Equation Parser

STARGATES allows the user to write the equation in ASCII in a mathematical form, and the **Equation Parser** component reads this ASCII representation and constructs a tree representing the equation. This equation tree will contain all the fundamental problem information required by the rest of the system to generate output code to perform that simulation. Parsing mathematical equations is potentially an open ended problem but as indicated we are able - for our prototype tool - to restrict the equation forms we are addressing to some specific patterns, and make the problem tractable.

To write the **Equation Parser**, we have made use of the compiler generator technology ANTLR [42]. ANTLR is a relatively modern tool building upon historical developments[43] including the well known lexing/parsing tools: lex/yacc[44] and flex/bison[45, 46]. ANTLR allows us to specify a relatively simple grammar from which ANTLR will automatically generate a Lexer and a Parser. The grammar shown here supports the declaration of parameters and fields and equations with simple mathematical operators $+, -, *, /$ as well as parameters and stencils. A simplified version of our Grammar is shown in Listing 1.

Listing 1: Simple equation ANTLR grammar.

```
DIGIT     : '0'..'9';
CHAR      : 'a'..'z'|'A'..'Z'|'_';
ID        : CHAR (CHAR|DIGIT)*;
NUM       : (DIGIT)+ ('.' (DIGIT)+ )?;
DERIVATIVE
          : 'd/dt';

file      : (statement)+ EOF!;
statement
          : (declaration | equation);
declaration
          : ID '[' ']' ID ';'
          | ID ID ';' ;
equation
          : DERIVATIVE ID '=' additive ';' ;
additive
          : multiplicative (('+' ^ | '-' ^) multiplicative)*;
multiplicative
          : unary (('*' ^ | '/' ^) unary)*;
unary
          : atom
          | MINUS atom;
atom      : NUM
          | ID
          | '(' additive ')'
          | ID '{' additive '}' ;
```

This grammar is sufficient to parse equations of the form:

```
float M;
float B;
float U;
float K;
float[] u;
d/dt u = M * Laplacian{(-B*u + U*(u*u*u) - K*Laplacian{u})};
```

where the "equation" start-point defines a first order time-differential equation, whose right hand side has a number of spatial calculus operators as well as algebraic combinations of the fundamental field and parameters.

After the initial equation is parsed, the tokens are converted into a tree which can then be parsed by a ANTLR tree parser. This tree parser constructs a tree representing the equation out of objects that are each equivalent to a component of the equation (parameters, operators, stencils etc). Given the example of the Cahn-Hilliard equation (See equation 2), when this equation is parsed, the ANTLR tree parser generates the tree shown in Figure 4.

This tree contains all of the information about the Cahn-Hilliard equation needed by STARGATES. The tree will be given to the generator which will traverse through the tree to gather the information required to generate the language specific code to calculate the equation. However, this tree does not contain information about how the equation is to be integrated as integration methods are independent of specific equations. Information about integration methods is stored in Integration Trees.

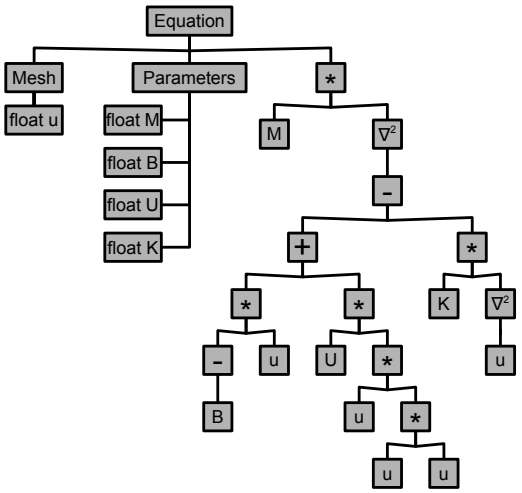

Figure 4: The tree that is created by the ANTLR tree parser for the Cahn-Hilliard equation.

5.2 Integration Library

The Integration Library is responsible for supplying STARGATES with necessary information about selected integration methods. This information is in the form of integration trees which are very similar to the equation trees discussed in the previous section. Each integration tree contains all the information required about a specific integration method including the required meshes, parameters and of course the equations that define the method itself. In the current prototype version of STARGATES, the integration trees are created by a hard-coded functions within the Integration Library. However, it is intended that in the future a parser and lexer will be incorporated into the Integration Library that will allow these integration trees to be constructed from intermediate ASCII representations.

The simplest numerical integration method available is the (first order) Euler method. This integration method can be expressed as:

$$y_{n+h} = y_n + f(y_n) \times h \tag{14}$$

Although in almost all cases this method is unstable or only works with very small time steps (h), it is still useful as an illustrative example to see how the time-integration code generation and libraries work.

The tree structure that represents this simple integration method is constructed by the simulation generator and results in the tree shown in Figure 5.

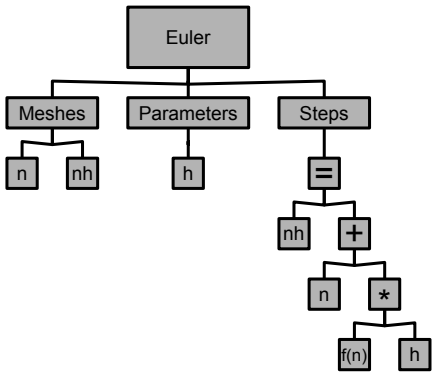

Figure 5: The tree constructed by the Integration Library that represents the Euler integration method.

Practically, standard higher order methods such as the Runge-Kutta methods are preferred for most problems, and an advantage of our simulation generator approach is that different methods can readily be tried out on a problem without recourse to much tedious and error prone recoding by hand. In the modern literature *The* Runge-Kutta method often refers to the 4^{th} order or RK4 method; however, Runge-Kutta really refers to a whole family of integration methods of various orders. For the Cahn-Hilliard example we discuss in this paper second-order time is quite sufficient. The simulation generator has a function to build the integration tree for the 2^{nd} order Runge-Kutta method (sometimes called the midpoint method). This two-part integration method is given in equation 15.

$$\begin{aligned} y_{n+\frac{1}{2}h} &= y_n + f(y_n) \times \frac{1}{2}h \\ y_{n+h} &= y_n + f(y_{n+\frac{1}{2}h}) \times h \end{aligned} \tag{15}$$

The Integration Library component constructs the integration tree shown in Figure 6 to represent the Runge-Kutta 2^{nd} order method.

These integration trees are constructed and then combined with the equation tree that is being integrated by the method. The equation trees are inserted into the integration tree at the f(_) nodes. The target field identified by integration method is then substituted into the equation tree. These trees then contain all of the information about the equation and integration method that is needed to generate the simulations.

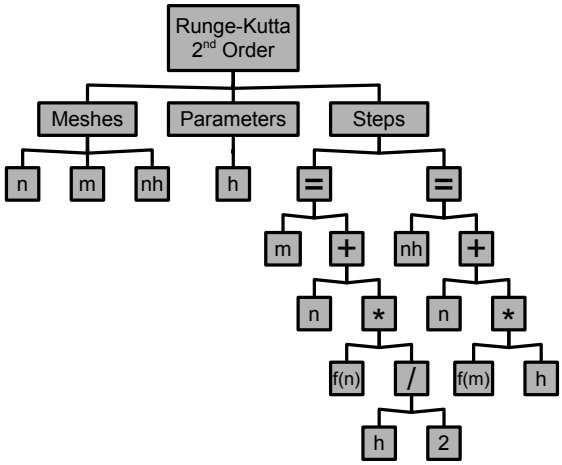

Figure 6: The tree that represents the Runge-Kutta 2^{nd} order integration method. It required three meshes *n, m, nh* to store the field and intermediate information and a single parameter h for the time-step of the method.

5.3 Stencil Library

The Stencil Library is responsible for generating and manipulating the stencils in the equation. An advantage of the generator approach is that the problem can be specified independently of whether it is to be solved on a two-, three- or higher-dimensional mesh. Some PDE problems that arise in other areas of physics can be in higher dimensions and it is useful to be able to separate the dimension from other problem details and thus generate software for arbitrary dimensions. We have discussed hyper-dimensionality library support apparatus in [47], and the code generator can use this. The library must provide stencils of the correct dimensionality as defined by the equation configuration. Currently the Stencil Library has functions to generate a specific set of stencils with various dimensionalities. We have already published in [41] a method in which user-defined stencils with arbitrary data-types can be used within this simulation generator. For the purposes of this present paper we demonstrate the generator with straightforward two-dimensional mesh problems.

One of the most important functionalities the Simulation Generator has is to apply stencils to each other. This operation allows the Stencil Library to rearrange the equation to expand all of the stencils with the expression such that there are no stencils within other stencil expressions. This allows the simulation update to be performed in a single step rather than having to generate intermediate expressions. It also allows complex, higher-order

stencils to be automatically constructed from simple user-defined stencils.

For example, the Equation tree shown in Figure 4 will be rearranged by the Stencil Library to the following tree (Figure 7). Note how the ∇^2 within the other ∇^2 node has been converted to a ∇^4 node.

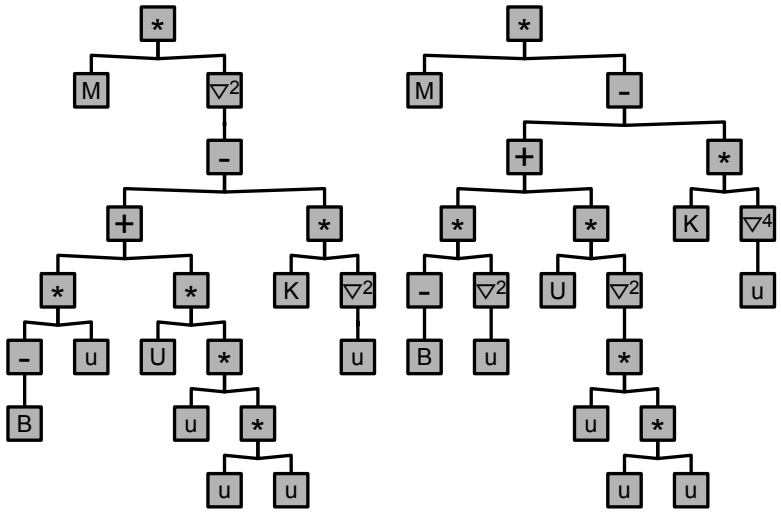

Figure 7: Tree representing the Cahn-Hilliard equation after it has been refactored by the Stencil Library. No stencil node sub-tree contains another stencil node.

To convert the two ∇^2 stencils into a ∇^4 stencil, the stencils are applied to each other. This can be performed numerically by summing a copy one stencil for each cell in the other stencil with each copy being multiplied by the corresponding cell in the other stencil. An example of this process in two- and three-dimension for applying two ∇^2 stencils to each other can be seen in Figure 8.

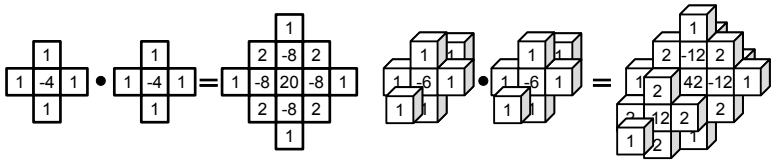

Figure 8: Two ∇^2 stencils being applied to each other to create on ∇^4 stencil. Shown in two- and three-dimensions.

The stencils produced by Stencil Library are not tied to any indexing scheme, rather they just contain the information about the stencil size,

dimensionality and actual values. Indexing schemes and access methods are defined purely in the output generator and thus the stencils remain independent of the target language.

5.4 Output Generator

The output generator is responsible for traversing the Simulation Tree and creating the language-specific output code. These generators glean the information they need from these trees to generate their language-specific implementation of the simulation. Different generators can be created that use the same target language but create simulations with a different structure, for example two generators could both use CUDA but one generates single-GPU simulations while the other creates simulations that use multiple-GPUs.

These two simulations could come from the same generator and simply have a parameter (that comes from the simulation configuration file) that says whether the simulation should use a single- or multi-GPU structure. This choice is specific to the generators and does not affect the rest of the simulation generator.

The advantage of this approach is that the front-end parsing and simulation tree construction for the simulations remains the same regardless of the output generator used. When a new architecture or language is released, a new generator can be written that will allow all of a users simulations to be migrated to make use of that new architecture or language.

This makes it much easier to adopt a new language or architecture without the need to rewrite the entire simulation base. This is a far easier and more extensible programming model than maintaining separate code versions for each simulation and architecture.

6 Results

We have currently implemented two code generator stages, one for single-threaded C++ and one for CUDA using global memory. In the following section we present some code that these generators have produced and compare the performance of the code to hand-written versions.

6.1 Example Code Output

Here we present the code generated by two output generators for the Cahn-Hilliard equation (see Equation 2). One of the generators builds a single-threaded C++ program and the other generates a simple CUDA simulation using global memory. We have only shown fragments of the code to show how the two generators produce code specific to their target languages.

Listing 2 shows the main function and integration method for the Cahn-Hilliard equation using the Runge-Kutta 2^{nd} order integration method. The generator creates and intialises the main mesh of the equation u. It also creates the three meshes required by the RK2 method un, um and unh. Also shown in the Listing is the function to perform the integration steps, in this code both of the RK2 steps are performed in one C++ function.

Listing 2: The code generated by the C++ generator for the Cahn-Hilliard equation using RK2 integration method.

```
int main() {
  float *u = new float[Y * X];
  for(int iy = 0; iy < Y; iy++) {
    for(int ix = 0; ix < X; ix++) {
      u[iy*X + ix] = (uniform() * 2.0) - 1.0;
    }
  }
  float *un = new float[Y * X];
  float *um = new float[Y * X];
  float *unh = new float[Y * X];
  float h = 0.01;
  memcpy(un, u, Y * X * sizeof(float));
  memcpy(um, u, Y * X * sizeof(float));
  memcpy(unh, u, Y * X * sizeof(float));

  for(int t = 0; t < 1024; t++) {
    rk2(un, um, unh, h);
    swap(un, unh);
  }
  memcpy(u, un, Y * X * sizeof(float));
}

void rk2(float *un,float *um,float *unh,float h) {
  for(int iy = 0; iy < Y; iy++) {
    for(int ix = 0; ix < X; ix++) {
      ...
    }
  }
  for(int iy = 0; iy < Y; iy++) {
    for(int ix = 0; ix < X; ix++) {
      ...
    }
  }
}
```

The code produced by the CUDA generator has several important differences that reflect the difference between the CPU and GPU architecture. First of all, the meshes used by the integration method are allocated on the device and the main mesh is copied to it by a Host-Device copy. The second major difference is the integration functions, as these functions are executed in parallel (note the difference syntax for configuring the function calls) the two RK2 steps must be performed as separate functions to avoid race conditions.

Listing 3: The main function and integration methods generated by the

CUDA generator for the Cahn-Hilliard equation.

```
int main() {
  float *u = new float[Y * X];
  for(int iy = 0; iy < Y; iy++) {
    for(int ix = 0; ix < X; ix++) {
      u[iy*X + ix] = (uniform() * 2.0) - 1.0;
    }
  }
  float *un;
  cudaMalloc((void**)&un, Y * X * sizeof(float));
  float *um;
  cudaMalloc((void**)&um, Y * X * sizeof(float));
  float *unh;
  cudaMalloc((void**)&unh, Y * X * sizeof(float));
  float h = 0.01;
  cudaMemcpy(un, u, Y * X * sizeof(float), cudaHostToDevice);
  cudaMemcpy(um, u, Y * X * sizeof(float), cudaHostToDevice);
  cudaMemcpy(unh, u, Y * X * sizeof(float), cudaHostToDevice);
  dim3 block(BLOCK_X, BLOCK_Y);
  dim3 grid(X/block.x, Y/block.y);
  for(int t = 0; t < 1024; t++) {
    rk2_a<<<grid,block>>>(un, um, unh, h);
    rk2_b<<<grid,block>>>(un, um, unh, h);
    swap(un, unh);
  }
  memcpy(u, un, Y * X * sizeof(float));
}

__global__ void rk2_a(float *un,float *um,float *unh,float h) {
  int k =
  (threadIdx.z*(gridDim.y*blockDim.y*gridDim.x*blockDim.x)) +
  (((blockIdx.y*blockDim.y)+threadIdx.y)*(gridDim.x*blockDim.x)) +
  (blockIdx.x*blockDim.x)+threadIdx.x;
  int ix =  k % X;
  int iy =  (k/X) % Y;

  ...
}
__global__ void rk2_b(float *un,float *um,float *unh,float h) {
  int k =
  (threadIdx.z*(gridDim.y*blockDim.y*gridDim.x*blockDim.x)) +
  (((blockIdx.y*blockDim.y)+threadIdx.y)*(gridDim.x*blockDim.x)) +
  (blockIdx.x*blockDim.x)+threadIdx.x;
  int ix =  k % X;
  int iy =  (k/X) % Y;
  ...
}
```

Since both the C++ and CUDA generator stages use C-like syntax, the code to perform the actual equation is the same for both the C++ and CUDA generators. This code (with whitespace formatted to be easier to read) is shown in Listing 4. This code calculates the change in one spatial cell for the Cahn-Hilliard equation. We have tried to make the variable names and code layout closer to human readable choices than some code generators do since the programmer may decide to adopt the generated code and include it in a code package that is subsequently human-maintained rather than regenerated without being ever subsequently viewed by a programmer.

Listing 4: The same equation calculation code generated by both the C++ and CUDA generators.

```
unh[iy*X + ix] = un[iy*X + ix] + M*(
(-B)*(                    (unym1x) +
            (unyxm1) + (-4*unyx) + (unyxp1) +
                      (unyp1x))+
U*(                       (unym1x*unym1x*unym1x) +
(unyxm1*unyxm1*unyxm1)+(-4*unyx*unyx*unyx)+(unyxp1*unyxp1*unyxp1) +
                      (unyp1x*unyp1x*unyp1x))-
K*(                             (unym2x) +
          (2*unym1xm1) + (-8*unym1x) + (2*unym1xp1) +
(unyxm2) + (-8*unyxm1) + (20*unyx) + (-8*unyxp1) + (unyxp2) +
          (2*unyp1xm1) + (-8*unyp1x) + (2*unyp1xp1) +
                             (unyp2x)))*h;
```

The code that the generator produces obviously does not contain every possible optimisation as humans are usually much better at identifying which optimisations are applicable for particular simulations. However, if the pattern of possible optimisation is identified, it could subsequently be incorporated into the output generator. These optimisations belong in the generator as optimisations are specific to the target language. We anticipate that our architecture will give us a strong platform for further experiments in advanced optimisation of this sort.

6.2 Performance Comparison

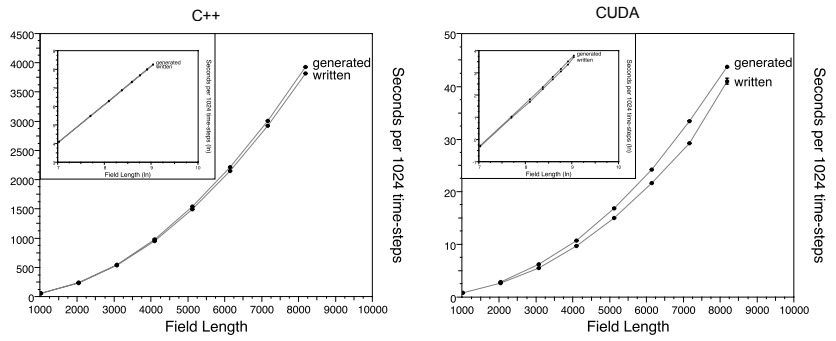

Figure 9: A performance comparison between hand-written and generated simulation codes. Results are shown for C++ of the left and CUDA on the right, ln-ln plots are shown inset.

One of our major requirements for STARGATES is that it should generate fast, efficient simulation code. To test how well this requirement has been fulfilled, we have compared the performance of the generated simulations to our existing, hand-written versions. The results we present here

are for two-dimensional simulations with field-lengths of N={1024, 2048, 3072, 4096, 5120, 6144, 7168, 8192}. These performance results can be seen in Figure 9.

It can be seen that all the simulations both hand-written and generated scale with the expected $O(N^2)$. Also as expected, the hand-written versions perform slightly faster than the generated version. This is due to the specific optimisations that the programmer can identify are applicable to the simulation. The relative performance of the generated versus hand-written simulations for both target languages can be seen in Figure 10.

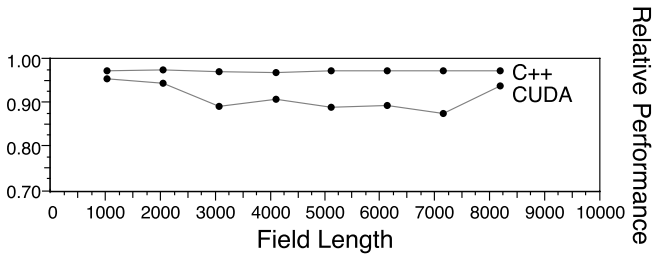

Figure 10: Relative performance of the generated simulations compared to the hand-written versions. The results are in the range of 0.85-1.0, showing that the generated simulations can perform between 85-100% of the computation performed by the hand-written versions in the same time period.

It should be noted here that the optimisations that allow the hand-written simulations to perform faster could be incorporated into the output generator and are not fundamental changes to the model. It is possible to build optimisations into the output generators that are either always applied or applied to some simulations based on input from the configuration file. Even so, these simple output generators still produce code that performs efficiently and have performance comparable to hand-written code.

7 Discussion

The STARGATES design allows language-specific optimisations to be incorporated into the system with ease. Because the output generators are purely responsible for traversing the simulation tree and generating output code, optimisations can be added without change to any other part of the system. This means that if a method of identifying when an optimisation is applicable (or an option is added to the configuration file) then it can be included into the generated code.

In general our design philosophy is to defer decisions that the programmer might want to make about details for a particular "run" as far down the simulation tree as possible, and associated with this, to separate as far as possible the different specifications. So the equation parsing language should be separable from the particular equation parameters, and the code generation options and optimisation choices are also separated as much as possible. We have been through various early stage software prototypes where a monolithic architecture was used and as we have learned more about the processes involved we have managed to aim at a cleaner more separable set of components for STARGATES.

We learned about ANTLR relatively late in our conceptualisation process, but it has helped considerably in providing a higher level parser generator apparatus. In particular the concept of separating out the tree walker generation stages is much easier using ANTLR.

7.1 Higher Order Stencils

A great deal of practical experience is often needed to decide on the right choice of numerical method to employ with a particular PDE. Sometimes it is possible to determine analytically from the equation what methods will be required for numerical stability but sometimes this issue can only be addressed empirically through numerical experimentation. Although we have simplified some of the numerical methods discussion for this paper, which focuses more on the overall architecture of STARGATES, it is of course a goal that we can incorporate a scalable library of numerical methods. We have implemented the simple Euler, mid-point and third- and fourth-order Runge-Kutta methods and are presently working on the more modern fifth order Dormand and Prince methods[48]. There is a range of mathematical and numerical "lore" available in the literature and a good framework would support the possibilities of greater numerical experimentation with state-of-the-art methods.

There is also a good body of work reported in the literature on higher order spatial calculus operators[49]. We have employed second order stencils for the Laplacians in the example we report, but for some PDE problems involving higher order field terms a higher order stencil is also warranted. Our generator is capable of supporting stencils for fourth or sixth order Laplacians in principle.

We have not discussed boundary conditions in this present paper and this is obviously a very important issue for specifying many PDE problems and generating solver codes. The work we report here made use of simple periodic boundaries but we have done some work with the Cahn-Hilliard equation with a mix of fixed and periodic boundaries. This is particularly important for flow problems or where some global field such as gravity or

some other preferred direction is present. We are still working on how boundary condition information can be incorporated into the simulation tree without compromising the principle of data separation.

7.2 Data Types

Another important aspect of numerical experimentation is to determine what precision is necessary. As discussed in Section 4, GPUs come with different levels of floating point support. Generally speaking present generation devices perform a 32-bit floating point precision considerably more cheaply than 64-bit double precision calculations. A simplistic approach to generating numerical solver code is to take all equations as working on simple scalar floating point field variables of a particular precision such as 32-bit floats or 64-bit doubles. There are two obvious limitations to this philosophy.

Firstly many PDEs are expressed in terms of vectors fields and not scalars and while a vector variable such as velocity for example could be expressed separately as in 3-dimensions as a system of 3 separate but coupled equations, this unnecessarily complicates the problem formulation and indeed can hide some potential optimisation information. Specifically, in solving a vector field equation typically each vector element is worked on at once but the separate x, y and z components might all be needed in memory at once to take care of cross-terms in the calculation. This will affect the optimal way to lay the field variables out in memory – in terms of $v[x][y][z]$; or $v[z][y][x]$ or separately as $vx[]$, $vy[]$, and $vz[]$.

Similarly for some purposes even a scalar field may be modelled as a complex number with separate real and imaginary parts. In some target languages there may be a complex data type but quite commonly in the C syntax related family of programming languages, the concept of a complex data type has to be implemented separately using separate real and imaginary floating point variables.

This is related to the other main limitation. It is by no means the case that a definite precision and fundamental floating point data type is ideal for all PDEs that a stencil generating apparatus might address. Furthermore it is still the case that many available devices do not have all the desired available floating point resolutions that might be desired - and in many cases even if they are all available in principle, the performance that is attainable varies drastically. At the time of writing almost all mainstream CPUs available offer optimum performance on 64 bit double precision. This is not the case for many accelerator devices such as GPUs which may be capable of double precision only as a "special operation" and which are only optimal for 32-bit floating point. Many GPUs share double precision units amongst a group of cores rather than having duplicated FPU hardware on

each core. While this issue may resolve partially in time, it may be the case in time that 64-bit FPU is standard and some devices offer 128-bit FPU as a special but sub optimal option.

In summary therefore, a flexible and portable software apparatus cannot ignore this issue and must support some selection of data types and associated compromise decision-making by the user. One solution is to structure the type information in a way that the user can specify details to the auto-generating tool. In particular this must consist of specifying how to initialise, as well as how to store the chosen individual data type for a particular PDE problem. At present we incorporate this as type specification information at the equation grammar level. The work we report in this present paper uses single precision (float) data which is adequate for the Cahn-Hilliard work.

8 Conclusions and Future Work

In summary, we have described how a staged parser and tree-walking code generator can produce data-parallel software fro modern accelerator devices such as GPUs that is both speed optimized as well as human-readable and maintainable. This is possible as we have focused on a very specific form of application domain problem - that of solving regular partial differential equations using finite difference equations. We have shown that the speed performance of the generated code is very close to that attainable by expert programmer hand-generated software.

One important outcome of this work for us is the ability to investigate whole families of problems rather than having to focus on just one hand-coded one. Problems like the Cahn-Hilliard equation or the Time-Dependent Ginzburg-Landau equation have a number of choices embedded in them that, while compactly expressible in mathematics, lead to quite different software formulations. A tool like STARGATES opens up a number of feasible investigations in computational physics that would otherwise be quite time consuming - and in the past have consumed a whole PhD each in terms of coding, testing and general research effort.

A more general outcome of this work however is the software architecture for scientific problem domain specific languages that can be parsed and can have output code generated in a number of different target languages and associated platforms. We particularly note the promise of modern compiler generator tools such as ANTLR and the benefits of using them rather than attempting a monolithic single stage parser-generator tool.

GPUs work quite well for this problem (of course) which is why we are writing about them. A great deal of effort has been expended by programmers over recent years in coming to terms with the features of

new and emerging multi core processors and accelerator devices. Up to a point compiler optimisation technology has coped quite well with the use of complex and many register optimisations. It would seem a fair comment that compiler developers are still addressing the issues of multi core CPUs. Languages like NVIDIA's CUDA do expose the workings of GPUs to the programmer and likewise open language standards such as OpenCL[50] do allow data-parallelism to be exploited. Nevertheless it is by no means something that can be left to an automatic parallelising compiler. Our approach of breaking the application problem down with an extra layer of a domain specific language is likely to be important and necessary for many problem areas for some time. We hope that our approach will allow more agile deployment on the next emerging generation of accelerators and data parallel devices with rather less intense programmer effort required.

We also intend to experiment with code generator stages that produce message passing code harnesses; with conventional threads libraries such as pThreads[51] and Intel's Thread Building Blocks[52] that can both support multi-core CPUs; but also with some of the more parallel specific language extensions and ideas such as OpenMP code directives. We hope to be able to release a STARGATES version that can support all these multiple target generator stages as well as support experimentation with hybrid approaches.

References

[1] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers Principles, Techniques, and Tools. Addison-Wesley (1986) ISBN 0-201-10194-7.

[2] Cooper, K.D., Torczon, L.: Engineering a Compiler. Number ISBN 1-55860-698-X. Morgan Kaufmann (2004)

[3] Grune, D., Bal, H.E., Jacobs, C.J.H.: Modern Compiler Design. Number ISBN 0-471-97697-0. Wiley (2000)

[4] Srikant, Y., Shankar, P., eds.: The Compiler Design Handbook - Optimizations and Machine Code Generation. Second edn. Number ISBN 1-4200-4382-X. CRC Press (2008)

[5] Bozkus, Z., Choudhary, A., Fox, G.C., Haupt, T., Ranka, S.: Fortran 90D/HPF compiler for distributed-memory MIMD computers: design, implementation, and performance results. In: Proc. Supercomputing '93, Portland, OR (1993) 351

[6] Polychronopoulos, C.D.: Parallel Programming and Compilers. Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers (1988)

[7] Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Compiler Construction. Volume 6011. Springer (2010) 244–263

[8] Tofte, M.: Compiler Generators - What they can do, what they might do, and what they will probably never do. Number ISBN 0-387-51471-6 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1990)

[9] NVIDIA® Corporation: CUDA^{TM} 2.1 Programming Guide. (2009) Last accessed May 2009.

[10] McMullin, P., Milligan, P., Corr, P.: Knowledge assisted code generation and analysis. In: High-Performance Computing and Networking. Volume 1225 of LNCS., Springer (1997) 1030–1031

[11] Cook, G.O.: Code Generation in ALPAL Using Symbolic Techniques. In: Int. Conf. on Symbolic and Algebraic Computation, Berkeley, CA, USA., ACM/SIGSAM (1992) 27–35 ISBN:0-89791-489-9.

[12] Cook, G.O., Painter, J.F., Brown, S.A.: How symbolic computation boosts productivity in the simulation of partial differential equations. Journal of Scientific Computing **6** (1991) 193–209 ISSN: 0885-7474.

[13] Zima, H.: Automatic vectorization and parallelization for supercomputers. In Perrott, R., ed.: Software for Parallel Computers. Chapman and Hall (1991) 107–120

[14] Benson, T., Milligan, P., McConnell, R., Rea, A.: A knowledge based approach to the development of parallel programs. In: Parallel and Distributed Processing, 1993. Proceedings. Euromicro Workshop on. (1993) 457–463 ISBN 0-8186-3610-6.

[15] Milligan, P., McConnell, R., Benson, T.: The Mathematician's Devil: An Experiment In Automating The Production Of Parallel Linear Algebra Software. In: Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on. (1994) 385–391 ISBN: 0-8186-5370-1.

[16] The MathWorks: Matlab. available at http://www.mathworks.com (2007)

[17] Wolfram Research: Mathematica. available at http://www.wolfram.com (2007)

[18] Ganapathi, A., Datta, K., Fox, A., Patterson, D.: A case for machine learning to optimize multicore performance. In: First USENIX Workshop on Hot Topics in Parallelism (HotPar'09), Berkeley, CA, USA. (2009)

[19] Cardenas, A.F., Karplus, W.J.: PDEL - A Language for Partial Differential Equations. Comm. of the ACM **13** (1970) 184–191

[20] Logg, A.: Automating the finite element method. Arch. Comput. Methods Eng. **14** (2007) 93–138

[21] Logg, A., Wells, G.N.: Dolfin: Automated finite element computing. ACM Trans. Math. Soft. **37** (2010) 1–28

[22] Dongarra, J.J., Croz, J.D., Duff, I.S., Hammarling, S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Soft. **16** (1990) 18–28

[23] Dongarra, J.J., Whaley, R.C.: A users' guide to the BLACS v1.1. Technical report, Univ of TN, Knoxville (1997)

[24] Bischof, C.H., Dongarra, J.J.: A linear algebra library for high-performance computers. In Carey, G.F., ed.: Parallel Supercomputing: Methods, Algorithms and Applications. Wiley (1989) 45–55

[25] Choi, J., Dongarra, J.J., Pozo, R., Walker, D.W.: Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In: Proc. of the Fourth Symp. the Frontiers of Massively Parallel Computation, IEEE Computer Society Press (1992) 120–127

[26] James, H.A., Patten, C.J., Hawick, K.A.: Stencil methods on distributed high performance computers. Technical Report DHPC-010, Advanced Computational Systems CRC, Department of Computer Science, University of Adelaide (1997)

[27] Reed, D.A., Adams, L.M., Patrick, M.L.: Stencils and problem partitionings: Their influence on the performance of multiple processor systems. IEEE Transactions on Computers C **36** (1987) 845–858

[28] Kamil, S., Chan, C., Williams, S., Oliker, L., Shalf, J., Howison, M., Bethel, E.W., Prabhat: A generalized framework for auto-tuning stencil computations. In: Proc. Cray User Group (CUG) Atlanta, Georgia,. (2009) 1–11

[29] Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Review **51** (2009) 129–159

[30] Datta, K., Murphy, M., amd S. Williams, V.V., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In: Proc. ACM/IEEE Conf. on Supercomputing (SC'08'). (2008)

[31] Cahn, J., Hilliard, J.: Free energy of a non-uniform system III. Nucleation in a two point compressible fluid. J.Chem.Phys. **31** (1959) 688–699

[32] Hawick, K.A., Playne, D.P.: Modelling, Simulating and Visualizing the Cahn-Hilliard-Cook Field Equation. International Journal of Computer Aided Engineering and Technology (IJCAET) **2** (2010) 78–93

[33] Cahn, J.W., Hilliard, J.E.: Free Energy of a Nonuniform System. I. Interfacial Free Energy. The Journal of Chemical Physics **28** (1958) 258–267

[34] Tuszynski, J., Skierski, M., Grundland, A.: Short-range induced critical phenomena in the Landau-Ginzburg model. Can.J.Phys. **68** (1990) 751–755

[35] M.A.Carpenter, E.Salje: Time dependent Landau theory for order / disorder processes in minerals. Mineralogical Magazine **53** (1989) 483–504

[36] Lotka, A.J.: Elements of Physical Biology. Williams & Williams, Baltimore (1925)

[37] Volterra, V.: Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. Mem. R. Accad. Naz. dei Lincei, Ser VI **2** (1926)

[38] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. In: Eurographics 2005, State of the Art Reports. (2005) 21–51

[39] NVIDIA® Corporation: CUDA™ 2.0 Programming Guide. (2008) Last accessed November 2008.

[40] Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience **21** (2009) 2400–2437 CSTN-065.

[41] Hawick, K., Playne, D.: Automated and parallel code generation for finite-differencing stencils with arbitrary data types. In: Proc. Int. Conf. Computational Science, (ICCS), Workshop on Automated Program Generation for Computational Science, Amsterdam June 2010. Number CSTN-106 (2010)

[42] Parr, T.: The Definitive ANTLR Reference - Building Domain-Specific Languages. Number ISBN 978-0-9787392-5-6. Pragmatic Bookshelf (2007)

[43] Aho, A.V., Ullman, J.D.: Principles of Compiler Design. Number ISBN 0-201-00022-9. Addison-Wesley (1977)

[44] Levine, J.R., Mason, T., Brown, D.: LEX and YACC. 2nd edn. O'Reilly (1992) ISBN 1-56592-000-7.

[45] Paxson, V., Estes, W., Millaway, J.: The Flex Manual - Lexical Analysis with Flex. Version 2.5.35 edn. (2007)

[46] Levine, J.: flex and bison: Text Processing Tools. O'Reilly Media (2009) ISBN: 978-0-596-15597-1.

[47] Hawick, K.A., Playne, D.P.: Hypercubic Storage Layout and Transforms in Arbitrary Dimensions using GPUs and CUDA. Technical Report CSTN-096, Computer Science, Massey University (2010) Accepted for and to appear in Concurrency and Computation: Practice and Experience.

[48] Dormand, J., Prince, P.J.: A family of embedded runge-kutta formulae. J. of Computational and Applied Maths. **6** (1980) 19–26

[49] Cohen, G.C.: Higher-Order Numerical Methods for Transient Wave Equations. Number ISBN 3-540-41598-X in Scientific Computation. Springer (2002)

[50] Khronos Group: OpenCL - Open Compute Language (2008)

[51] IEEE: IEEE Std. 1003.1c-1995 thread extensions. (1995)

[52] Reinders, J.: Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism. 1st edn. Number ISBN 978-0596514808. O'Reilly (2007)