ELSEVIER

International Conference on Computational Science, ICCS 2010

Automated and Parallel Code Generation for Finite-Differencing Stencils with Arbitrary Data Types

K.A. Hawick[1], D.P. Playne

Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand

Abstract

Finite-Differencing and other regular and direct approaches to solving partial differential equations (PDEs) are methods that fit well on data-parallel computer systems. These problems continue to arise in many application areas of computational science and engineering but still offer some programming challenges as they are not readily incorporated into a general standard software library that could cover all possible PDEs. Achieving high performance on numerical solutions to PDEs generally requires exposure of the field data structures and application of knowledge of how best to map them to the memory and processing architecture of a particular parallel computer system. Stencil methods for solving PDEs are however readily implemented as semi-automatically generated skeletal frameworks. We have implemented semi-automated stencil source code generators for a number of target programming languages including data-parallel languages such as CUDA for graphics processing units (GPUs). We report on some performance evaluations for our generated PDE simulations on GPUs and other platforms. In this article we focus on (diffusive) PDEs with a non-trivial data type requirement such as having vector or complex field variables. We discuss the issues and compromises involved implementing equation solvers with fields comprising arbitrary data types on GPUs and other current compute devices.

Keywords: partial differential equation, stencil, parallel code generation, GPU, CUDA

1. Introduction

Many science and engineering problems are formulated as partial differential equations (PDEs). While there are many excellent software tools from linear algebra and so forth that can and have been incorporated into software libraries, it is not so simple to construct an arbitrary library of PDEs. One very common group of PDEs are diffusive equations of the general form:

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

where all the time dependence is expressed in the time derivative (left-hand side) and the general functional $\mathcal{F}$ is typically an additive combination of spatial operators applied to powers of the field variable u. Some well-known

Email addresses: k.a.hawick@massey.ac.nz (K.A. Hawick), d.p.playne@massey.ac.nz (D.P. Playne)
[1]Corresponding author

PDE problems that fit this pattern are: the Cahn-Hilliard equation[1] which is expressed in terms of a scalar field ϕ:

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

and the Time-Dependent Ginzburg Landau equation [2] in terms of a complex scalar field u:

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

Typically many of the common and interesting spatial operators such as $\frac{\partial}{\partial x}\cdot$, $\frac{\partial}{\partial y}\cdot$,..., $\frac{\partial^2}{\partial x^2}\cdot$, $\frac{\partial^2}{\partial y^2}\cdot$, $\nabla^2\cdot$, and $(\nabla^2)^2\cdot$,... can be implemented as finite difference stencil operators with a well-defined mapping that can be applied to regular data structures such as meshes in 1-, 2-, 3-D or higher dimensions.

We have implemented a software prototype tool that can generate efficient and readily parallelisable source code for several PDEs of this type[3] and for which an in-built data type of the target programming language and its associated run-time libraries is available. We have successfully used this approach to solve PDEs such as the Cahn-Hilliard equation[1, 4], where our solvers were second- or fourth-order accurate in time and second order accurate in space with spatial operators up to the bi-harmonic operator ($\nabla^4\cdot$).

In this present paper we discuss the additional problems in generating source code for target platforms such as NVIDIA's Compute Unified Device Architecture (CUDA) programming language or the vendor consortium's Open Computing Language (OpenCL) specification, which for reasons of implementation may not have the requisite in-built type at all or may only support a slow implementation. Specific cases at the time of writing concern support for double-precision floating point calculations, but more generally these limitations apply to PDEs formulated in terms of complex numbers, and also vector field variables (or for that matter tensors). We have experimented with the Time-Dependent Ginzburg-Landau equation (TDGL)[5] which is formulated in the standard form of Equation 1 but with a complex field variable, and we are also interested in complex fluid equations[6] which can be formulated in terms of vectors of complex numbers. Other useful systems of equations that are formulated in terms of systems (vectors) of coupled equations are ecological and population models[7] such as the spatial Lotka-Volterra system[8].

The concept of automatically generating source code in this manner[9, 10] for domain specific applications problems [11] such as stencil-like problems is not a new one[12, 13] with some notable successes achieved at Lawrence Livermore by Cook and co-workers on generating Fortran source code from symbolic algebra equation specifications[14, 15, 16, 17]. Some modern problems solving environments such as Maple[18] and Mathematica[19] and Matlab[20] also offer source code generation capabilities – usually to generate either Fortran code or a proprietary scripting language. In many science and engineering applications communities Fortran and its variants are still very important target languages for this sort of simulation and projects such as FortPort have achieved success in generating Fortran source automatically[21, 22]. Up to a point such Fortran source can subsequently be optimised or have its data structures laid out appropriately in memory using standard parallel programming techniques such as compiler directives[23, 24]. However we believe that incorporating the parallel and appropriate memory layout formulation at an earlier stage of the code generation process allows a higher degree of potential performance.

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 exposed and available for a code generation tool to exploit. We are particularly interested in the potential of producing highly performance optimised programs that can be run on current generation many-core processors[25] and accelerators such as Graphical Processing Units and other data-parallel platforms. Other programming languages such as dynamic and scripting languages such as Python[26] can also be attractive targets for automatic code generation of complex problems but for the sort of application we discuss here, performance is still paramount and therefore CUDA and OpenCL seem better replacements for Fortran.

Many interesting PDEs use data fields that comprise a single scalar value at each spatial point, however many are also expressed in terms of vectors. 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. In relativity the D'Alembertian operator is a generalisation of the Laplacian in four dimensions and is sometimes written as:

$$\Box \equiv \frac{\partial^2 \mathbf{u}(x, y, z, w)}{\partial x^2} + \frac{\partial^2 \mathbf{u}(x, y, z, w)}{\partial y^2} + \frac{\partial^2 \mathbf{u}(x, y, z, w)}{\partial z^2} - \mathbf{i}\frac{\partial^2 \mathbf{u}(x, y, z, w)}{\partial w^2} \tag{4}$$

In this case the operator itself involves both a real and an imaginary component and is truly complex. For some equations the imaginary **i** pre-factor can be factored out leaving a wholly real stencil.

This issue of complex numbers is related to the other main limitation of present generation technology. 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. A 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. To an extent some of these limitations can be addressed through additional support code and run-time libraries invoked by our automatically generated skeletal stencil source code.

We describe the general approach our generator tool has taken so far in section 2 below and report on some specific computational performance figures on Graphical Processing Units (GPUS) in section 3. We discuss how these ideas for automatically generated source code for applications specific areas could be generalised to suit other problem domains and other target languages such as OpenCL, and also offer some concluding remarks and areas for further development in section 4.

2. Implementing Stencils

In this section we describe our initial prototype for a stencil generator for PDEs with simple data-types. As these stencils are part of a code-generator as opposed to an actual simulation, their purpose is not to perform any calculation but rather to generate the code that will perform this computation. However, the stencils should have the ability to have operations performed on them prior to code generation. This allows stencils to be applied to each other to create new stencils.
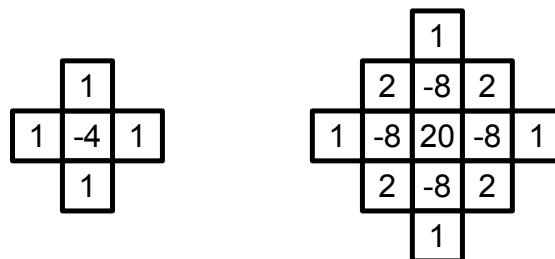

Figure 1: Two-dimensional stencils for the Laplacian operator (left) and Bi-harmonic operator (right) which are multiplied by $\frac{1}{\Delta^2}$ and are second-order accurate.

Our initial stencil definition limited stencils to simple numerical data types (int, float, double). This allows these stencil operations to be performed numerically to produce a value for each cell in the resultant stencil. These numerical stencils are defined by a series of integers defining the dimensionality and size of the stencil and a series of numerical values defining the actual values of the stencil. For example the two-dimensional Laplacian operator (see Figure 1) can be defined by: `dim = [3, 3] data = [0, 1, 0, 1, -4, 1, 0, 1, 0]` .

There are a number of operations that can be performed on these stencils to manipulate them to create new stencils. We describe the following important operations: "+", "−", "∗", "/", "expand" and "apply". The stencil arithmetic operations for "+", "−", "∗", "/" simply involve applying the numerical operation to corresponding values in the two stencils. The centres of the stencils are aligned and for cells that are present in both stencils have the numerical operation applied to them. Any cell that is present in only one stencil is included in the result but the value is left unchanged.

Rather than having to define a different stencil each time the simulation dimensionality is changed, the stencil manipulator has the ability to **expand** a one-dimensional stencil into a higher dimension. This process overlays the one-dimensional stencil rotated in each dimension on top of itself. Figure 2 shows a one-dimensional Laplacian stencil expanding to two-, three- and four-dimensions.
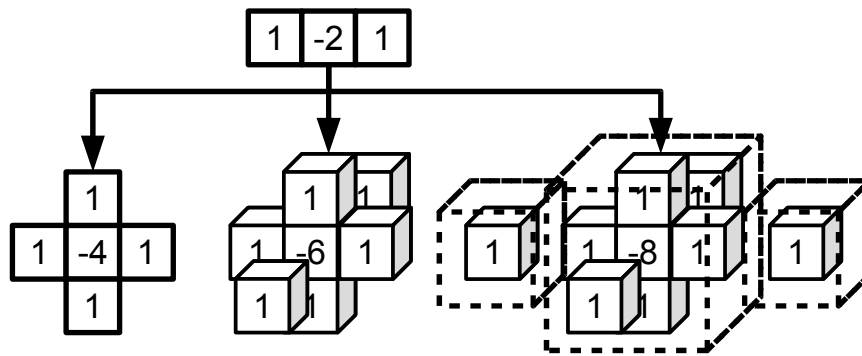

Figure 2: A one-dimensional Laplacian stencil (top) is applied to itself to create two-, three- and four-dimensional stencils.

This process of expanding stencils may not be suitable for all one-dimensional stencils. In such cases the users must define the stencil for each dimension, the code-generator will simply ignore the stencils that are not of the desired dimension. One useful operation that the stencil manipulator provides is the ability to composite or **apply** one stencil to another. This process is performed by convolving the first stencil with the second. For each cell in one stencil, the other stencil is multiplied by the cell's value and added onto the result centred around the cell's position. This operation is only supported for two stencils with the same dimensionality. Examples of this process in two- and three-dimensions can be seen in Figure 3.
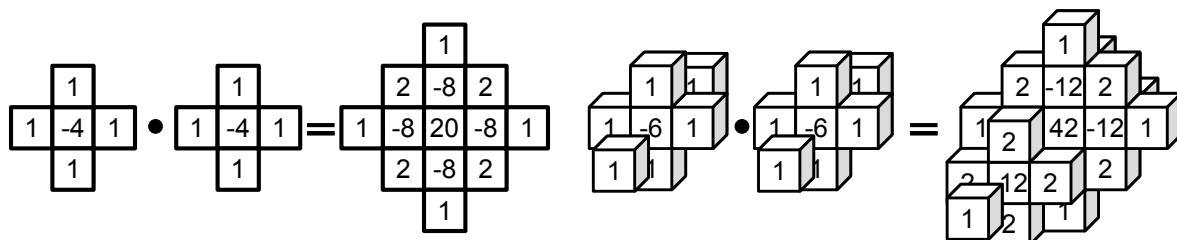

Figure 3: An example of two stencils being applied to each other in two- and three-dimensions.

The stencils are used in conjunction with the **code-generator** to turn an equation into simulation source code. The stencils are used to generate not only the actual calculation code but also used to determine which neighbouring cell values must be fetched from memory. To do this all the stencils in the equation are analysed to determine which neighbours they use and the code to fetch these values is generated. Listing 1 shows an example of code generated for

Equation 5 in two-dimensions using the Euler integration method.

$$\frac{\partial \mathbf{u}(\mathbf{r})}{\partial t} = \nabla^2 \mathbf{u} \tag{5}$$

The first-order Euler formulation is of course rarely used due to its limited numerical accuracy and range of stability, but it nevertheless often provides the most intuitive exposition of the data structure for the numerical formulation. Our generator can readily deploy higher-order (in time) numerical methods, the most commonly useful being second- or fourth-order Runge-Kutta schemes[27]. It is straightforward to add support for other time schemes to our system the only real cost being potential use of more memory if multiple-step schemes require it.

Listing 1: Generated code for accessing the required values and applying a simple scalar stencil.

```
int ym1x = input[ym1*X +  x ];
int yxm1 = input[ y *X + xm1];
int yx   = input[ y *X +  x ];
int yxp1 = input[ y *X + xp1];
int yp1x = input[yp1*X +  x ];

output[y*X + x] =  yx + (             (1*ym1x) +
                          (1*yxm1) + (−4* yx) + (1*yxp1)
                                      + (1*yp1x)               ) * dt;
```

This code assumes that the indexes of the neighbouring cells (ym1, yp1, xm1, xp1) have been calculated. The required neighbour indexes required are also found by analysing the stencils in the equation however the code to find them is not shown as the code depends on the boundary conditions of the simulation.

2.1. Stencils with Arbitrary Data Types

The simple numerical data type stencils we have currently described are sufficient for many simulations; however, simulations that use user-defined data types often require stencils of this same type. In this case defining stencils as templates is not suitable as it requires the data type to be defined at the code-generator compile time. Instead the values of the stencil are simply stored as strings allowing the user to use any data type with the assumption that it will be defined for the simulation code compile time.

To support this functionality we change the definition of the stencil data. A stencil is now defined by a string naming the data type of the stencil, a series of integers is still used to define the dimensionality and size of the stencil and a list of string is now used to define the values of the stencil. For example, a one dimensional stencil of type "complex" can be defined as:
type = "complex"
dim = [3]
data = ["complex(1,1)", "complex(-2,-2)", "complex(1,1)"]

As these values are now no longer numerical values, the stencil operations cannot compute numerical values for the resultant stencil. Instead the values are output as the calculation string that will result in the correct value. Listing 2 is an example of the previously described stencil being extended into two dimensions.

Listing 2: Generated code for accessing and applying a stencil with a user-defined type (complex).

```
complex ym1x = input[ym1*X +  x ];
complex yxm1 = input[ y *X + xm1];
complex yx   = input[ y *X +  x ];
complex yxp1 = input[ y *X + xp1];
complex yp1x = input[yp1*X +  x ];

output[y*X + x] =  yx + (              (complex( 1, 1)*ym1x) +
 (complex(1,1)*yxm1)+((complex(−2,−2)+complex(−2,−2))*yx)+(complex(1,1)*yxp1)
                                      +(complex( 1, 1)*yp1x)          ) * dt;
```

This manner of outputting the calculation string rather than the final value may have some detrimental effects on the performance of the final simulation. Most modern compilers will evaluate the calculation string to a single value,

however as this cannot be guaranteed the generator performs this functionality where possible. If the data type of the stencil is "int", "float" or "double" the stencil values will be converted to numerical types, the computation will be performed numerically and the resultant values will be converted back to strings for the final result. This effectively performs the stencil operations numerically where possible and will only use the calculation string output method when the data type is not recognised by the code-generator.

This method of generating stencils using calculation strings is far from optimal but is a suitable temporary solution. We are currently investigating methods of incorporating compiler technologies into the stencil manipulation to perform the calculations for user-defined types. This will make the generated code cleaner and in some circumstances will provide performance benefits for the generated simulations.

As finite-differencing field equation simulations are fundamentally numerical it is a requirement that any **user-defined data type** used to describe the stencils or field must support basic arithmetic operators ("+", "−", "∗" and "/"). As this code-generator creates C code it is required that the defined types be C classes with overloaded operators. Any other operators or methods used within the equation must also be defined by the user if they wish to use the data type.

The code-generator can generate code for a variety of architectures with the current restriction that they have a C style code interface. This includes single-core CPUs (C, OpenCL), multi-core CPUs (pThreads, Threading Building Blocks, OpenCL) and GPUs (CUDA, OpenCL). For specialist architectures such as GPUs, standard overloaded operators cannot be used within device calls. The standard method that overloaded operators in a user header file define host code which cannot be called from a GPU thread. Additional sets of operators must be defined that can be called from the device, in the case of CUDA this requires the operators to be defined as _*device*_. An example user-defined type with operators for C and CUDA can be seen in Listing 3.

Listing 3: Complex type with the "+" and "-" operators defined for both C and CUDA.

```
class complex {
public:
    float re;
    float im;
};
#ifdef CUDA
__device__ complex operator+(const complex &a, const complex &b) {
        return complex(a.re + b.re, a.im + b.im);
}
__device__ complex operator-(const complex &a, const complex &b) {
        return complex(a.re - b.re, a.im - b.im);
}
#else
static complex operator+(const complex &a, const complex &b) {
        return complex(a.re + b.re, a.im + b.im);
}
static complex operator-(const complex &a, const complex &b) {
        return complex(a.re - b.re, a.im - b.im);
}
#endif
```

This only defines operators for simple C and CUDA, for the data type to be used for other architectures the appropriate method of overloading the operators must be used. Another restriction for user-defined types used in simulations on specialist architectures is that pointers to data may be inaccessible. Devices such as GPUs cannot access data in host memory and thus types that use this storage mechanism must be re-designed to work correctly on devices such as GPUs.

2.2. Floating Point Precision and Type Availability

Some specialist architectures such as some GPUs are not designed to work with double precision floating point values. Architectures such as NVIDIA GPUs with compute capability 1.3 or higher do support them, they often have a significant impact on performance, being shared across streaming processors. In some applications single-precision floating point variables are sufficient for the accuracy of the simulation/integration method while in others double precision may be required. These restrictions should be very carefully considered when defining the simulation or when deploying user-defined types.

3. Example Performance Results

To test the usefulness of the arbitrary data-type stencil manipulator in conjunction with the code-generator, we compare the performance of a generated simulation vs a hand-written, optimised version. The simulation uses a user-defined complex type for the stencils and mesh with the appropriate overloaded operators. The simulation has been generated and hand-written for both a single-core C implementation and a GPU CUDA implementation. Performance data on typical mesh sizes comparing the two implementations is shown in Figure 4.

These implementations have been executed on a platform running Ubuntu 9.10 64-bit. The processor is an Intel® Core™2 Quad CPU running at 2.66GHz with 8GB of DDR2-800 memory. The GPU simulations are executed by this machine on an NVIDIA® GeForce® GTX 295 which contains two GPUs each access to 896MB of memory. Although this card contains two GPUs, only one is used as utilising both requires multiple CPU threads and explicit communication.
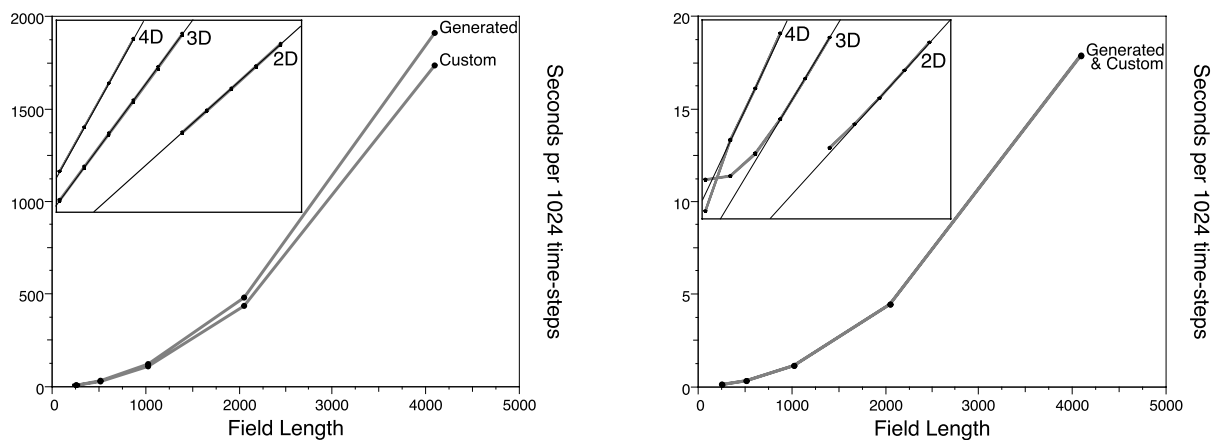

Figure 4: Performance results comparing the Generated and Custom written simulations. Single-core CPU results are on the left and GPU CUDA results are on the right. Note the difference in scale of the two graphs with approximately two orders of magnitude performance improvement attained on the GPU over a typical CPU. Ln-ln scale plots are inset showing the expected power-law scaling with lattice length for 2D, 3D and 4D.

The performance results for the single-core C implementation of the simulation show a difference in performance between the Generated simulation and the hand-written Custom implementation. This performance difference is not unexpected and considering the difference in development time we consider this an acceptable performance loss. One interesting point to note is the that there is almost no difference in performance between the GPU implementations of the simulation.

4. Discussion and Conclusions

We have reviewed some successful past projects in automatic source code generation, which predominantly have targeted Fortran as their output language. There is scope for additional heuristics to be incorporated into such automatic code generators for application domains in addition to any subsequent parallelism that can be applied to generated source code in the usual ways. We believe tightly coupled data parallel languages are perhaps easier targets for some well-defined application problems like stencil applications. We have presented results using PDEs and finite differencing although many stencil ideas also apply to image processing and other problems on regular data arrays.

We have shown here that our code-generator for field-equation simulations can generate C and CUDA source code that then runs with performance similar to that of hand-written simulations. In particular this paper has shown how this can be achieved for simulations that use arbitrary, user-defined data types. This code-generator can be easily extended to operate with a number of architectures and output targets. One such output target is OpenCL. As OpenCL allows parallel programs to be written for a variety of parallel architectures, generating OpenCL code is an attractive idea.

Generating OpenCL code moves the architecture independence from the code-generator to the generated simulation code. This would allow simulations to be generated that could easily be run on whatever resources are available on the machine. However we note that even with this platform independent simulation code, care must be taken to consider the implications that executing code on accelerator architectures can cause. This includes restrictions on user-defined types, operator overloading requirements and so forth.

We have shown that our tool can generate source code in CUDA that can perform quite well immediately on GPU platforms but that is sufficiently readable in format so that a user can subsequently hand adjust it for the vagaries of a particular platform. This seems a major design issue still to be resolved – will a user use an automatic code generator once-off and subsequently maintain the generated source code in the usual way - perhaps modifying it and hand optimising it further? Or does the tool need some sort of transitive closure and the ability to re-load its own output code to be used as a development base? We believe the former case is more tractable and useful for parallel programmers but the latter is obviously more desirable for application-domain programmers.

We believe there is great scope for interesting further automatic code generation work in: generating OpenCL target code; incorporating more heuristics into the code generator; adding support for tensor equations and other very complex systems that would be too difficult to write by hand; incorporating a symbolic algebra generator at the front end stage so we can avoid using our own markup language representations. Finally, we note that Parnas's criticism of the field of automatic programming[28] as just "a euphemism for programming in a higher level language than was available to the programmer" – while still true, can be successfully built upon by looking at application domain areas where the problem is better defined and more tractable.

5. References

[1] D. Playne, K. Hawick, Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA, in: Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09) Las Vegas, USA., no. CSTN-073, 2009.

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

[3] K. A. Hawick, D. P. Playne, Turning partial differential equations into scalable software, Tech. rep., Computer Science, Massey University (2009).

[4] K. A. Hawick, D. P. Playne, Modelling, Simulating and Visualizing the Cahn-Hilliard-Cook Field Equation, International Journal of Computer Aided Engineering and Technology (IJCAET) 2 (CSTN-075) (2010) 78–93.

[5] D. Playne, K. Hawick, Visualising vector field model simulations, in: Proc. 2009 International Conference on Modeling, Simulation and Visualization Methods (MSV'09) Las Vegas, USA., no. CSTN-074, 2009.

[6] T. Witten, P. Pincus, Structured Fluids: Polymers, Colloids, Surfactants, no. ISBN 0198526881, Oxford Univ. Press, 2004.

[7] K. Hawick, C. Scogings, Emergent spatial agent segregation, in: Proc 2008 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Sydney, 2008, pp. 34–40.

[8] A. J. Lotka, Elements of Physical Biology, Williams & Williams, Baltimore, 1925.

[9] T. Benson, P. Milligan, R. McConnell, A. Rea, A knowledge based approach to the development of parallel programs, in: Parallel and Distributed Processing, 1993. Proceedings. Euromicro Workshop on, 1993, pp. 457–463, iSBN 0-8186-3610-6. doi:10.1109/EMPDP.1993.336366.

[10] P. Milligan, R. McConnell, T. Benson, 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, pp. 385–391, ISBN: 0-8186-5370-1.

[11] T. Rus, Domain-oriented component-based automatic program generation, in: Proc. Forum on Interdisciplinary Computing, Montenegro, 2003.

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

[13] D. A. Reed, L. M. Adams, M. L. Patrick, Stencils and problem partitionings: Their influence on the performance of multiple processor systems, IEEE Transactions on Computers C 36 (7) (1987) 845–858.

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

[15] G. O. Cook, J. F. Painter, S. A. Brown, How symbolic computation boosts productivity in the simulation of partial differential equations, Journal of Scientific Computing 6 (2) (1991) 193–209, ISSN: 0885-7474.

[16] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, E. W. Bethel, Prabhat, A generalized framework for auto-tuning stencil computations, in: Proc. Cray User Group (CUG) Atlanta, Georgia,, 2009, pp. 1–11. URL http://escholarship.org/uc/item/23p6g5nj

[17] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick, Optimization and performance modeling of stencil computations on modern microprocessors, SIAM Review 51 (1) (2009) 129–159.

[18] K. Geddes, G. Gonnet, Maple, http://www.maplesoft.com/ (2009).

[19] Wolfram Research, Mathematica. available at http://www.wolfram.com (2007).
URL `Availableathttp://www.wolfram.com`

[20] The MathWorks, Matlab. available at http://www.mathworks.com (2007).
URL `Availableathttp://www.mathworks.com`

[21] R. McConnell, P. Sage, S. Rea, P. McMullan, Hot spot analysis within the FortPort migration tool for parallel platforms, Microprocessing & Microprogramming 37 (1-5) (1993) 141–144. doi:10.1016/0165-6074(93)90034-I.

[22] B. McCollum, P. H. Corr, P. Milligan, A meta-heuristic approach to parallel code generation, in: High Performance Computing for Computational Science VECPAR 2002, Vol. 2565 of LNCS, Springer, 2002, pp. 215–232. doi:10.1007/3-540-36569-9.

[23] B. Chapman, G. Jost, R. van der Pas, Using OpenMP - Portable Shared Memory Parallel Programming, no. ISBN 978-0-262-53302-7, MIT Press, 2008.

[24] H. Zima, Automatic vectorization and parallelization for supercomputers, in: R. Perrott (Ed.), Software for Parallel Computers, Chapman and Hall, 1991, Ch. 8, pp. 107–120.

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

[26] J. E. Guyer, D. Wheeler, J. A. Warren, FiPy: Partial Differential Equations with Python, Computing in Science and Engineering 11 (3) (2009) 6–15.

[27] L. F. Shampine, Some practical Runge-Kutta Formulas, Mathematics of Computation 46 (173) (1986) 135–150, ISSN: 0025-5718.

[28] D. L. Parnas, Software aspects of strategic defense systems, Comm. ACM 28 (12) (1985) 1326–1335.