Numerical Simulations of Unsteady Shock Wave Interactions Using SaC and Fortran-90

Daniel Rolls[2], Carl Joslin[2], Alexei Kudryavtsev[1], Sven-Bodo Scholz[2], and Alex Shafarenko[2]

[1] Institute of Theoretical and Applied Mechanics RAS SB, Institutskaya st. 4/5, Novosibirsk, 630090, Russia
[2] Department of Computer Science, University of Hertfordshire, AL10 9AB, UK

Abstract. This paper briefly introduces SaC: a data-parallel language with an imperative feel but side-effect free and declarative. The experiences of porting a simulation of unsteady shock waves in the Euler system from Fortran to SaC are reported. Both the SaC and Fortran code was run on a 16-core AMD machine. We demonstrate scalability and performance of our approach by comparison to Fortran.

1 Introduction

In the past when high performance was desired from code, high-levels of abstraction had to be comprimised. This paper will demonstrate our approach which overcomes these shortcomings: we will present the data-parallel language SaC [14] and exemplify its usage by implementing an unsteady shock wave simulator in the Euler system. SaC was developed by an international consortium coordinated by one of the authors (Sven-Bodo Scholz). We will compare the performance of our approach against Fortran by running this application on a 16-core computation server.

The language is close to C syntactically, which makes it more accessible to computational scientists, while at the same time being a side-effect free, declarative language. The latter enables a whole host of intricate optimisations in the compiler and, perhaps more importantly, liberates the programmer from implementation concerns, such as the efficiency of memory access and space management, exploitation of data-parallelism and optimisation of iteration spaces. In addition, code that was written for a specific dimensionality of arrays can be reused in higher dimensions thanks to an elaborate system of array subtyping in SaC, as well as its facilities for function and operator overloading that far exceed the capabilities of not only Fortran but the object-orientation languages as well.

SaC has already been used for many kinds of application, ranging from image-processing to cryptography to signal analysis. However, to our knowledge there has been only one occasion of programming a Computational Fluid Dymamics

V. Malyshkin (Ed.): PaCT 2009, LNCS 5698, pp. 445–456, 2009.

application in SaC namely the Kademtsev-Petviashivili system [4]. Even that example is too esoteric to support any conclusions about practical suitability of Single-Assignment C. In this paper we present for the first time the results of using SaC as a tool in solving a real, practical problem: simulation of unsteady shock waves in the Euler system.

The equations of fluid mechanics can be solved analitically for only a limited number of simple flows. As a consequence, numerical simulation of fluid flows known as Computational Fluid Dynamics (CFD) is widely used in both scientific research and countless engineering applications. Efficiency of computations and ease of code development is of great importance in CFD which is one of the most perspective fields for implementing new concepts and tools of computer science.

In Section 2 we will briefly outline the features of SaC that we would argue make it uniquely suitable for the class of applications being discussed. Section 3 delineates the numerical method being used and Section 4 discuses implementation issues we came across when porting a Fortran TVD implementatin to SaC. Our results are then presented in Section 5 and related work is discussed in Section 6 before finally Section 7 discusses the lessons learnt and concludes.

2 SaC

SaC is an array processing language that first appears to be an imperitive program like Fortran but actually has more in common with functional programming languages. A SaC function consists of a sequence of statements that define and re-define array objects. To a C programmer this looks very similar to assigning the result of expressions to arrays, but there is an important difference: what may appear to the programmer to be the "control flow" in SaC is in fact a chain of definitions that link with one another via the use of common variables, this emphasises data as opposed to control dependencies. Thus any iterative update becomes essentially a recurrence relation between the snapshots of the arrays being updated, and it is up to the compiler whether or not the arrays need to be recreated as objects in memory or whether the underlying computation may be taken in-flow. That not withstanding, analogues of control structures, such as the `IF` statement, are provided, if only with a slightly different interpretation, so the illusion of programming a control flow may be retained as far as possible. `IF` statements are expressions: this can be seen by observing that with imperitive code, control flow through conditionals can affect whether a variable is defined; however this is not valid SaC code.

Two main constructs of SaC support the kind of computation that we are concerned with in this paper:

Most of the high level constructions in this paper are compiled down to the following to constructs.

with-loop. Despite the name, which reflects some historic choices of terminology in SaC, the essence of this construct is a data-parallel array definition. The programmer supplies a specification of the index space (in an extended enumeration form) and the definition of the array value for a given index in terms of an expression with other values possibly indexed and produced by external functions. Definitions for different array values are assumed to be mutually independent, hence data-parallelism is presented to the compiler explicitly.

for loop. This is used for programming recurrences. The recurrence index is specified in the for loop together with its initial value and increment, the compiler interprets the loop body as a definition of the arrays emerging at the final step of the recurrence in terms of the arrays defined prior to the first step.

As with FORTRAN-90 small arithmetic expressions in SaC can operate on whole arrays to conveniently express elementwise operations on those arrays. E.g. `a - b * c + c` could be both an expression operating on scalars, arrays or scalars and arrays where the scalar form of the expression is applied to corresponding indicies in the arrays `a`, `b` and `c`. For consise expressivness SaC supports set notation which allows an expression to be defined for every element of a new array where each expression may depend on the index. E.g. `{ [i,j] -> matrix[j,i] }` transposes a matrix by placing element (j, i) from the original matrix into element (i, j) for all i and j.

Another feature of the language that finds its use in the application being reported is its type system, which supports subtyping. To provide an overview of this, we remark, by way of an example, that a vector can be interpreted as a two dimensional array obtained by replicating the vector as a row in the column dimension. This is a subtype of a general two dimensional array type. One consequence of this is that a function that contains a tridiagonal solver for a one-dimensional Poisson equation can be applied to a two dimensional array (acting row-wise) and then applied again column-wise by using two transpositions, all without changing a single line of code in the solver definition.

All these features make it possible to write function bodies that act on inputs of any dimension which suffer no performance loss compared to more specialized function bodies. Our code makes use of this fact to reuse function bodies for a one dimensional and two dimensional shockwave simulation.

3 Application

SaC is used to develop an efficient solver for the compressible Euler equations, which govern the flow of an inviscid perfect gas:

$$\frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0, \tag{1}$$

$$\mathbf{Q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, \qquad \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 \\ \rho uv \\ u(E+p) \end{bmatrix}, \qquad \mathbf{G} = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 \\ v(E+p) \end{bmatrix}. \tag{2}$$

Here t is time, x and y are spatial coordinates, u and v are components of the flow velocity, ρ is density, p is the pressure related to the total energy E as

$$p = (\gamma - 1)\left(E - \rho\frac{u^2 + v^2}{2}\right), \tag{3}$$

where γ is the ratio of specific heats ($\gamma = 1.4$ for air). The Euler equations are the canonical example of a hyperbolic system of nonlinear conservation laws that describes conservation of mass, momentum and energy. Numerical methods, originally developed for the Euler equations, can be also used for a wide variety of other hyperbolic systems of conservation laws, which arise in physical models describing physical phenomena in fields as varied as acoustics and gas dynamics, traffic flow, elasticity, astrophysics and cosmology. Thus, the Euler solver is a very representative example of a broad class of computational physics programs.

A salient feature of nonlinear hyperbolic equations is the emergence of discontinuous solutions such as shock waves, fluid and material interfaces. It turns their numerical solution into a non-trivial task. Modern numerical methods for solving the hyperbolic equation [9] are based on high-resolution shock-capturing schemes originated from the seminal Godunov's paper [7]. In these methods, the computational domain is divided into a number of grid cells and the conservation laws are written for each cell. The computational procedure includes three stages: 1) reconstruction (in each cell) of the flow variables on the cell faces from cell-averaged variables; 2) evaluation of the numerical fluxes through the cell boundaries; and 3) advancement of the solution from the time t^n to time t^{n+1} where $t^{n+1} = t^n + \Delta t$. These stages are successively reiterated during the time intergation of Eq (1).

The reconstruction during the first stage should avoid the interpolation across the flow discontinuities. Otherwise, numerical simulations fail because of a loss of monotonicity and numerical oscillations developing near the discontinuities. The Fortran code developed includes several techniques of monotone reconstruction, in particular, the TVD (Total Variation Diminishing) reconstructions of the 2nd and 3rd orders with various slope limiters and the 3rd order WENO (Weighted Essentially Non-Oscillatory) reconstruction, which automatically assigns the zero weight to the stencils crossing a discontinuity. The latter technique is used in the examples of flow computation below. The reconstruction is applied to the so-called (local) characterisic variables rather than to the primitive variables ρ, u, v and p or the conservative variables $\mathbf{Q}$.

The evaluation of numerical axes is performed by approximately solving the Riemann problems between two states on the "left" and "right" sides of the cell boundaries resulting from the reconstruction. The code includes a few options

for the approximate Riemann solver, below the results obtained from the shock wave simulation are presented. For time advancement (Stage 3) the 2nd or 3rd order TVD Runge-Kutta schemes are used.

As an example of flow computations both a one dimensional and two dimensional problem is described below.

3.1 One Dimensional Simulation

The Euler code was used to solve the Sod shock tube problem [16], a common test for the accuracy of computational gasdynamics code. The test consists of a one dimensional Riemann problem. At the initial moment, the diaphragm separates two resting gases with different pressures and densities. The top state is $(\rho, u, p) = (1, 0, 1)$ while the bottom state is $(\rho, u, p) = (0.125, 0, 0.1)$. Here ρ is the density, u is the flow velocity and p is the pressure. After the diaphragm rapture, a shock wave and a contact discontinuity propagates to the bottom and a rarefaction wave moves to the top. This is illustrated in Fig. 1.

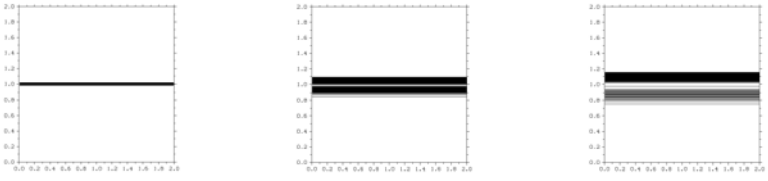

Fig. 1. The expansion of a shockwave from the center in the one-dimensional simulation where two gasses of different densities meet. The three diagrams move forward in time from left to right and show the shockwave expanding.

3.2 Two-Dimensional Simulation

Here a numerical simulation of an unsteady shock wave interaction is conducted. A schematic of flow configuration is shown in Fig. 2. The computational domain is a square divided into rectangular grid of $N_x \times N_y$ cells. A part of its left boundary is the exit section of a channel while the remaining portion of this boundary is a solid wall. The exit section of another channel comprises part of the computational domain's bottom boundary. A shock waves propagates within each of the channels and comes to the channels exits at the same moment ($t = 0$) when the computation starta. Thus, at the initial moment, the domain is filled by a quiescent gas. The boundary conditions in the exit sections of two channels are imposed in such a way that the flow variables are equal to the values behind the shock waves calculated from the Rankine-Hugoniot relations.

The computations have been conducted at the shock wave Mach numbers of $M_s = 2.2$. At this value of M_s the flow behind the shock waves is supersonic so that the flow variables in the exit sections are not changed during the computation. The size of the computational domain is $L_x = L_y = 2h$, where h is the channel width and $h = 200$ in our benchmarks.

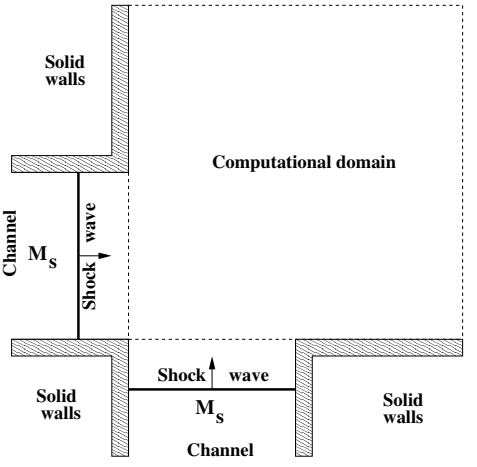

Fig. 2. A schematic of flow configuration and computational domain for the two-dimensional simulation

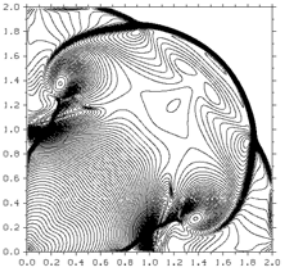

Fig. 3. A snapshot of the shockwave in the two-dimensional simulation

The results of computations are shown in Fig. 3. The interaction between the shock waves exhausting from the channels and their diffraction over solid walls generate a complex flow structure. In addition to the primary shock waves, which rapidly become approximately circular in shape, the irregular interaction of the shock waves leads to formation of a Mach stem between them and emergence of two reflected shock waves. The primary wave, the relected shock wave and the Mach stem meet in the three points, from which slipstream surfaces emanate. Behind each of the primary shock waves, there is a contact surface separating the gas exhausted out of the channel from the gas which initially filled the computatational domain. Secondary shock waves are formed closer to the exit section starting from a point on the last characteristics of the channel lips. On the later stages of evolution, the Mach stem itself becomes circular in shape and occupies a large proportion of the leading shock front while the contact surface behind it curls up into a mushroom-like structure.

4 Implementation

To illustrate the arguments from Section 2 we have selected two example functions from our TVD implementation in SAC.

4.1 dfDxNoBoundary

The function `dfDxNoBoundary` produces an array of the difference between each neighbouring pair in a vector. It takes the difference of every element in a vector but its first element with its left-neighbouring element and divides each element by a constant. The resulting vector has a length of one element less than the input vector.

As with the Fortran, in SAC the original vector is extended on both ends. The function defines two new vectors, one with the first element removed and one with the last element removed. An element-wise subtraction is applied to these new vectors (with matching indexes) and the resulting array is divided elementwise by a scalar (delta).

```
1  inline
2  fluid_cv[.] dfDxNoBoundary( fluid_cv[.] dqc, double delta)
3  {
4    return( ( drop([1], dqc) - drop( [-1], dqc) ) / delta);
5  }
```

To materialise each array in memory would be expensive; this style of programming would not be feasible for computational science if every array was copied. SAC's functional underpinnings allow it to, among other things, avoid some unnecessary calculations, memory allocation and memory copies. The style of code above often performs extremely well contrary to initial expectations.

4.2 getDT

The `GetDT` function calculates the time step to take in each iteration of the algorithm. It acts upon every element in a large array which represents the computational domain. For the two dimensional case Fortran has a nested loop structure with one loop for each dimension. The value EV is calculated each time and the largest EV value is saved. Finally this value is divided by a constant.

```
1         SUBROUTINE GetDT
2         USE Cons
3         USE Vars
4         IMPLICIT REAL*8 (A-H,O-Z)
5
6         EVmax = 0.d0
7         DO iy=IYmin,IYmax
8         DO ix=IXmin,IXmax
9           Ux = QP(1,ix,iy)
10          Uy = QP(2,ix,iy)
11          Pc = QP(3,ix,iy)
```

```
12          Rc = QP(4,ix,iy)
13          C = SQRT(Gam*Pc/Rc)
14          EV = (ABS(Ux)+C)/Dx+(ABS(Uy)+C)/Dy
15          EVmax = MAX(EV,EVmax)
16        END DO
17        END DO
18
19        DT = CFL/EVmax
20
21        END
```

The SAC version of the function is shown below. In the following code `GAM`, `DELTA` and `CFL` are constants.

```
1  inline
2  double getDt(fluid_pv[+] qp)
3  {
4    c = sqrt(GAM * p(qp) / rho(qp));
5    d = MathArray::fabs( u(qp));
6    ev = { iv -> (sum( ( d[iv] + c[iv]) / DELTA))};
7    return( CFL / maxval( ev));
8  }
```

The type of the function parameter is `fluid_pv[+]` which means an array of unknown dimensionality of `fluid_pv` values where `fluid_pv` is a user defined datatype. The syntax for an array type (`t`) can be syntactically represented as `t[x,y,z]` for an array of size `x` by `y` by `z`, `t[.,.]` for a array of two dimensions of unknown size and also `t[+]` for an array of unknown dimensionality.

The functions p and ρ extract the pressure and density from `fluid_pv` respectively. The SAC function calculates the variable `C` above using elementwise operations and then in line 6 `EV` is calculated which depends on the entire input array. With little experience with SAC this function quickly becomes easier to understand than the Fortran code. It is a functional definition (i.e. an expression) but the programmer is not obliged to use recursion on the array like a functional programmer would do with lists.

This clearer imperative-like but functional style makes data dependencies more obvious both to the programmer and to the compiler. In our simulation the SAC compiler always calculates the dimensionality needed for this function from its calls and therefore no penalty is paid for the generic type of `qp`.

5 Results

To evaluate the performance of SAC compared with Fortran we ran the 2D simulation with a 400x400 grid as described in Section 3.2. The simulation was run for 1000 time steps to ensure that the run time was sufficient to negate the start-up time of the program. We made use of a 400x400 grid as this was the size used in the original Fortran implementation. In the experiment we used the third order Runge-Kutta TVD method and first order piecewise constant reconstruction.

Compiler	Version	Arguments
SAC2C	SAC2C 16094	-L fluid -maxoptcyc 100 -O3 -mt
	stdlib 1120	-DDIM=2 -nofoldparallel -maxwlur 20
Sun Studio	8.3 Linux i386	-autopar -parallel -loopinfo -reduction
Compiler-f90	Patch 127145-01	-O3 -fast

The computer used to perform these benchmarks is a 4xQuad-Core (16 core) AMD Opteron^{TM} 8356 with 16GB of RAM. The source code is available at `http://sac-home.org`.

As the Fortran compiler uses OpenMP for parallelization, environment variables where set to control the runtime behaver of the Fortran code. Several different combinations where tried however these made a negligible difference to the runtime of the program. The options that produced the fastest runtimes, and therefore where used for the main benchmarking, were: `OMP_SCHEDULE=STATIC`, `OMP_NESTED=TRUE` and `OMP_DYNAMIC=FALSE`.

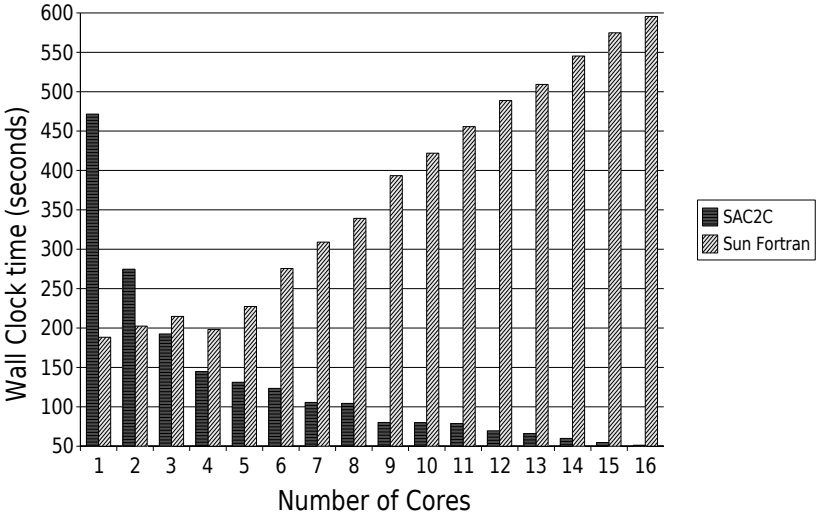

Fig. 4. Wall clock time of a 1000 time step simulation on a 400x400 grid

It can be seen in Figure 4 that SAC was much slower than the Fortran when run on just one core. However the Fortran code did not scale well with the number of cores, and as the number of cores increased performance degraded. We therefore suspect that there is added overhead of communication between the threads.

SAC does not use system calls for its inter thread communication but rather uses the programs shared memory and spin locks to allow inter thread communication with very little overhead. This low overhead allows SAC to scale well even when its problem size is to small for Fortran's auto parallelize feature to

work efficiently. There are optimizations which the SaC compiler can perform which are only possible because SaC is a functional, single assignment language. These optimizations help to allow the program to scale as SaC collates the many small operations on the arrays into fewer larger operations. This is not possible in procedural programing languages like Fortran as the compiler can not always work out the data dependences in complete detail. With a functional programing language like SaC it is possible to identify every dependency.

When the same benchmark was run with a larger 2000x2000 grid we discovered that Fortran was able to scale slightly with small numbers of cores but after just five cores it started to suffer from the overheads of inter-thread communication again.

6 Related Work

Broadly three techniques exist for producing highly parallelizable code for scientific simulations. The first technique is to carefully determine how a run should be parallelised and to explicitly write the code to do this. The message passing interface API[6] is commonly used for this. Also a threading library like Pthreads [11] could be used. Secondly, source-code annotations or directives can be used to provide information to a compiler to show it how an execution can be parallelised. Lastly compilers can try to autoparallelize code by analysing dependencies between variables. This section gives a brief overview of the three methods mentioned above and then discusses performance.

High-Performance Fortran [5] is an extension to FORTRAN-90 that allows the addition of directives to the source code to annotate distribution and locality. The Fortran code itself is written in a sequential style and already describes some operations in a data-parallel way. High-Performance Fortran compilers can then use these directives to compile to pipelined, vectorized, SIMD or message passing parallel code.

For explicitly annotating parts of a program that can be parallelized on shared memory systems the OpenMP [3] API is supported for C, C++ and Fortran. Many autoparallelizing compilers produce programs that call upon this API including the Intel and Sun Microsystems Fortran and C compilers.

ZPL [12] is a high level array processing language designed to be concise and platform independent. It allows programmers to easily describe subarrays within an array using a concept called regions. ZPL was designed with parallelism in mind and has had its performance compared with other languages for applications inclusive of computational fluid dynamics applications [13].

Parallel performance results tend to vary depending on the application, architecture and type of parallelism. For example an application that performs well on shared memory systems may not necessarily perform well when compiled to run on a distributed memory system. In addition and rather surprisingly, carefully crafted MPI applications might not necessarily have better speedups per core than implicit parallelism in high level languages. One surprising example of this is ZPL which has been shown to scale well in parallel runs with the Multigrid [1] NAS benchmark [2] and even shown prospects with CFD [13].

7 Conclusion

The results have shown that execution of high-performance applications written in SaC can achieve speedups on parallel architectures. This shows that using high-level abstractions in code operating on arrays is easier to understand. However, SaC's real strength comes into play when auto-parallelizing code.

SaC provides a powerful expressiveness where the greater learning curve is not grasping the paradigm but in resisting the temptation to try to optimize the code, and thus making use of SaC's ability to allow programs to be written with a high level of abstraction.

Programmers need a good way to express their programs so that they are quick to write, easy to understand and efficient to maintain. Up until now expressing programs in a readable form, with high levels of abstraction has come at a considerable performance penalty. However as can be seen in this paper it is possible to write programs in a clear style with high levels of abstraction while obtaining reasonable speedups that can be greater than those produced from the compilers of languages that where originally designed as sequential languages.

The results of this paper used SaC's Pthread back-end; future SaC back-ends promise even more parallelism. CUDA [10], whilst challenging to harness, has tremendous processing capabilities that will enable programs to make use of high performance, low cost processing resources found on GPUs as a potentially faster way of performing complicated simulations [15]. As part of an EU FP-7 project a back-end is being developed for SaC which produces code for an language which will compile to a many-core architecture called Microgrid [8]. This architecture will deliver considerable parallelism without the complexity that is involved with CUDA.

For future architectures parallelism will be increasingly vital. The work in this paper has shown that now that parallelism is important, it is possible to write abstract code in a high-level language and still be able to compete with traditional low-level, high performance languages like Fortran on parallel architectures.

Acknowledgments. This work was partly supported by the European FP-7 Integrated Project Apple-core (FP7-215216 — Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs).

References

1. Briggs, W.L., McCormick, S.F.: A multigrid tutorial. Society for Industrial Mathematics (2000)
2. Chamberlain, B.L., Deitz, S.J., Snyder, L.: A comparative study of the NAS MG benchmark across parallel languages and architectures. In: Supercomputing, ACM/IEEE 2000 Conference, pp. 46–46 (2000)
3. Chapman, B., Jost, G., Van Der Pas, R., Kuck, D.J.: Using OpenMP: portable shared memory parallel programming. The MIT Press, Cambridge (2007)

4. Shafarenko, A., et al.: Implementing a numerical solution of the kpi equation using single assignment c: Lessons learned and experiences. In: Implementation and Application of Functional Languages, 20th international symposium, pp. 160–170 (2005)
5. High Performance Fortran Forum. High Performance Fortran Language Specification. Rice University (1993)
6. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1. High Performance Computing Center Stuttgart (HLRS) (2008)
7. Godunov, S.K.: A difference method for numerical calculation of discontinuous equations of hydrodynamics (in russian). Mat. Sb. 47, 271–300 (1959)
8. Grelck, C., Herhut, S., Jesshope, C., Joslin, C., Lankamp, M., Scholz, S.-B., Shafarenko, A.: Compiling the Functional Data-Parallel Language sac for Microgrids of Self-Adaptive Virtual Processors. In: 14th Workshop on Compilers for Parallel Computing (CPC 2009), IBM Research Center, Zurich, Switzerland (2009)
9. Guinot, V.: Godunov-type schemes. Elsevier, Amsterdam (2003)
10. Guo, J., Thiyagalingam, J., Scholz, S.-B.: Towards Compiling SAC to CUDA. In: Proceedings of the 10th Symposium On Trends In Functional Programming, Komarno, Slovakia (June 2009)
11. Josey, A.: The Single UNIX Specification Version 3. Open Group (2004)
12. Lin, C., Snyder, L.: ZPL: An array sublanguage. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1993. LNCS, vol. 768, pp. 96–114. Springer, Heidelberg (1994)
13. Lin, C., Snyder, L.: SIMPLE performance results in ZPL. In: Pingali, K.K., Gelernter, D., Padua, D.A., Banerjee, U., Nicolau, A. (eds.) LCPC 1994. LNCS, vol. 892, pp. 361–375. Springer, Heidelberg (1995)
14. Scholz, S.-B.: Single assignement c – efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13, 1005–1059 (2003)
15. Senocak, I., Thibault, J., Caylor, M.: J19. 2 Rapid-response Urban CFD Simulations using a GPU Computing Paradigm on Desktop Supercomputers
16. Sod, G.A.: A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. Journal of Computational Physics 27(1), 1–31 (1978)