Shock-wave/boundary-layer interactions in the automatic code generation framework OpenSBLI

David J. Lusher[a,*], Satya P. Jammy[a], Neil D. Sandham[a]

[a] *Aerodynamics and Flight Mechanics Group, Faculty of Engineering and the Environment, University of Southampton, Southampton, SO17 1BJ, United Kingdom*

Abstract

Laminar shock-wave/boundary-layer interactions were simulated using OpenSBLI, a Python-based source code generation framework. Shock-capturing was performed by a 5^{th} order finite-difference Weighted Essentially Non-Oscillatory (WENO)-Z scheme applied in characteristic space. Oblique shock conditions were imposed for a shock angle of $\theta = 32.58°$ and Mach 2 free-stream, impinging on a laminar flat-plate boundary-layer. Performance of the code was assessed on different architectures for CPU, GPU and Xeon Phi.

Keywords: SBLI, WENO, Code-generation, GPU, Xeon Phi

1. Introduction

Shock-wave/boundary-layer interactions (SBLI) are a significant factor in the consideration of transonic/supersonic aircraft designs, with the induced adverse pressure gradients leading to detrimental boundary-layer thickening and separation of the flow. Recent studies of laminar SBLI include [1], [2], the first of which was simulated in the legacy Fortran code 'SBLI'. OpenSBLI [3] is a modern Python-based 'future-proof' version of the SBLI code, generating C code tailored to a user specified problem; the symbolic algebra library SymPy is used extensively throughout. Code written out by OpenSBLI is parallelised by source-to-source translation via the OPS library [4]. OPS is a Domain-Specific-

*Corresponding author.
E-mail address: D.Lusher@soton.ac.uk

Language (DSL) for multi-block structured grid computations, producing parallel code for a wide range of computational platforms. Other examples of DSLs include Pluto [5], Mint [6], and Devito [7], however unlike OPS these are often restricted to a single computational platform. OPS generates code for MPI, OpenMP, MPI+OpenMP, CUDA, OpenCL and OpenACC, plus MPI versions for multi-GPU clusters.

Separation of numerical algorithms from their parallel implementation allows for greater compatibility with emerging computational architectures, avoiding the time consuming and error-prone process of parallelisation and code porting. Furthermore, a code generation approach grants the user greater flexibility in how a code is written out and structured. An example within OpenSBLI was given in [8], where algorithms were tuned during code generation to improve compute intensity and reduce the bottleneck of slow global memory accesses.

In this work, reflection of an oblique shock-wave of shock angle 32.58° impinging on a laminar flat-plate boundary-layer is performed via numerical simulation of the two-dimensional compressible Navier-Stokes equations, for a Mach 2 inlet as proposed by [9]. This work demonstrates the feasibility of SBLI simulations within a code-generation framework, and details the implementation of the characteristic based Weighted Essentially non-Oscillatory (WENO) schemes used for shock-capturing in OpenSBLI.

2. WENO reconstruction for systems of hyperbolic equations

This section contains a review of WENO schemes as implemented in OpenSBLI for shock-capturing, focusing on WENO applied to systems of hyperbolic equations. Detailed discussion of the theory behind WENO can be found in [10], with this section restricted to a 5^{th} order formulation applied to the convective terms of the compressible Navier-Stokes equations. OpenSBLI uses numerical indices to distinguish variables, for example the Cartesian coordinate base (x, y, z) and standard velocity components (u, v, w) are taken to be (x_0, x_1, x_2), and (u_0, u_1, u_2) respectively. The governing equations are given

in non-dimensional index form with density ρ, pressure p, temperature T, and velocity components u_k as

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_k}\left(\rho u_k\right) = 0, \tag{1}$$

$$\frac{\partial}{\partial t}\left(\rho u_i\right) + \frac{\partial}{\partial x_k}\left(\rho u_i u_k + p\delta_{ik} - \tau_{ik}\right) = 0, \tag{2}$$

$$\frac{\partial}{\partial t}\left(\rho E\right) + \frac{\delta}{\delta x_k}\left(\rho u_k\left(E + \frac{p}{\rho}\right) + q_k - u_i\tau_{ik}\right) = 0, \tag{3}$$

with heat flux q_k and stress tensor τ_{ij} defined as

$$q_k = \frac{-\mu}{\left(\gamma - 1\right) M_\infty^2 PrRe}\frac{\partial T}{\partial x_k}, \tag{4}$$

$$\tau_{ik} = \frac{\mu}{Re}\left(\frac{\partial u_i}{\partial x_k} + \frac{\partial u_k}{\partial x_i} - \frac{2}{3}\frac{\partial u_j}{\partial x_j}\delta_{ik}\right), \tag{5}$$

where Pr, Re and γ are the Prandtl number, Reynolds number and ratio of heat capacities respectively. Dynamic viscosity $\mu\left(T\right)$ is given by Sutherland's law

$$\mu\left(T\right) = T^{\frac{3}{2}}\left(\frac{1 + \frac{T_s}{T_\infty}}{T + \frac{T_s}{T_\infty}}\right), \tag{6}$$

with freestream and Sutherland temperatures taken to be $T_\infty = 288.0$K and $T_s = 110.4$K. For a freestream Mach number M_∞, pressure and local speed of sound are defined as

$$p = \frac{1}{\gamma M_\infty^2}\rho T, \qquad a = \sqrt{\frac{\gamma p}{\rho}}. \tag{7}$$

2.1. WENO stencil construction

WENO schemes rely on an adaptive stencil formed by taking a convex combination of the smaller candidate ENO stencils shown in Figure 1, each assigned a weighting proportional to the local smoothness of the solution over those grid points. Stencils containing discontinuities are weighted close to zero and their contribution to the reconstruction is minimal. This weighting procedure avoids differencing over discontinuous regions of the solution field, hence achieving non-oscillatory behaviour of the numerical scheme. The basis for a WENO scheme is to provide a half-node flux reconstruction $\hat{f}_{i+\frac{1}{2}}$, such that

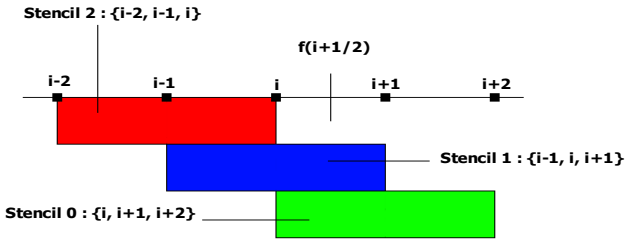

Figure 1: Stencil for a 5^{th} order WENO scheme, upwind biased reconstruction.

$$\hat{f}_{i+\frac{1}{2}} = \sum_{r=0}^{k-1} \omega_r \hat{f}^{(r)}_{i+\frac{1}{2}}, \tag{8}$$

where ω_r are the non-linear WENO weights quantifying the local smoothness, and $\hat{f}^{(r)}_{i+\frac{1}{2}}$ are the standard ENO interpolations given in [10]. For a given WENO reconstruction of 5^{th} order, we have $k = 3$, and $r = [0, 1, 2]$ weighted candidate ENO stencils. To form a downwind biased flux, reflection of the stencils is taken about the $x_{i+\frac{1}{2}}$ interface. Non-linear weights ω_r from the improved WENO-Z formulation [11] are chosen to be

$$\omega_r^z = \frac{\alpha_r^z}{\sum_{n=0}^{2} \alpha_n^z}, \qquad \alpha_r^z = d_r \left(1 + \left(\frac{\tau_5}{\sigma_r + \epsilon}\right)^2\right), \qquad \tau_5 = \left|\sigma_0 - \sigma_2\right|, \tag{9}$$

with the parameter τ_5 being the absolute difference between the smoothness indicators $[\sigma_0, \sigma_2]$, and ϵ is a small parameter ($\sim 10^{-16}$) to avoid division by zero. Smoothness indicators σ_r and optimal weights d_r take the form [10]:

$$\sigma_0 = \frac{13}{12}\left(f_i - 2f_{i+1} + f_{i+2}\right)^2 + \frac{1}{4}\left(3f_i - 4f_{i+1} + f_{i+2}\right)^2, \tag{10}$$

$$\sigma_1 = \frac{13}{12}\left(f_{i-1} - 2f_i + f_{i+1}\right)^2 + \frac{1}{4}\left(f_{i-1} - f_{i+1}\right)^2, \tag{11}$$

$$\sigma_2 = \frac{13}{12}\left(f_{i-2} - 2f_{i-1} + f_i\right)^2 + \frac{1}{4}\left(f_{i-2} - 4f_{i-1} + 3f_i\right)^2, \tag{12}$$

$$d_0 = \frac{3}{10}, \quad d_1 = \frac{3}{5}, \quad d_2 = \frac{1}{10}. \tag{13}$$

2.2. Characteristic WENO reconstruction

A simple approach for systems of equations is to apply WENO component by component to the governing equations (1-3), in terms of either the primitive or

conservative variables. These methods however can be inadequate for problems involving strong shocks [12], for which the more robust characteristic decomposition is used. This procedure requires the diagonalisation of the inviscid parts of equations (1-3), briefly outlined here. For an $m \times m$ system with conservative variable vector U the Jacobian $f'(U)$ has m real eigenvalues

$$\lambda_1(U) \leq \cdots \leq \lambda_m(U), \tag{14}$$

and a set of eigenvectors $r(U)$ that form the columns of the matrix

$$R(U) = \left(r_1(U), \ldots, r_m(U)\right), \tag{15}$$

to diagonalise the Jacobian such that

$$R^{-1}(U) f'(U) R(U) = \Lambda(U). \tag{16}$$

Finite-difference WENO schemes can be applied direction by direction independently, with spatial derivatives formed as the difference between half-node flux reconstructions at $\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right]$. For the case of an x_0 derivative with grid spacing Δx_0, this would result in the approximation

$$\frac{\partial U_j}{\partial x_0} \sim \frac{1}{\Delta x_0}\left(\hat{f}_{i+\frac{1}{2}} - \hat{f}_{i-\frac{1}{2}}\right), \tag{17}$$

where $\hat{f}$ is the sum of the split flux $\hat{f} = \hat{f}^+ + \hat{f}^-$, computed using local Lax-Friedrichs flux splitting

$$f^{\pm}(U_j) = \frac{1}{2}\left(f(U_j) \pm \alpha_j U_j\right), \qquad \alpha_j = \max|\Lambda_l(U_j)|. \tag{18}$$

The α_j term is the characteristic wave-speed for the j^{th} component of the system, evaluated as the maximum eigenvalue (14) over the l WENO stencil points.

3. Implementation in OpenSBLI

To facilitate the implementation of shock-capturing routines within OpenSBLI, the version 1.0 code presented in [3] was overhauled to enable greater flexibility for multiple numerical schemes of differing types. This section gives an overview of the WENO algorithm for spatial discretisation, including example code to demonstrate how a problem is specified within the framework.

3.1. Algorithm for characteristic WENO

1. Eigensystems (14)-(16) are instantiated in symbolic form, for a chosen number of dimensions. These are used to initialize the characteristic flux splitting scheme with a choice of simple or Roe averaging.
2. For a grid index i, an averaged state in the primitive variables of the eigensystem (14)-(16) is formed to obtain $R_{i+\frac{1}{2}}$, $R^{-1}_{i+\frac{1}{2}}$ and $\Lambda_{i+\frac{1}{2}}$.
3. Transformation of the solution vector U_j and flux vector $f(U_j)$ into characteristic variables is applied as $R^{-1}_{i+\frac{1}{2}}U_j$ and $R^{-1}_{i+\frac{1}{2}}f(U_j)$. Flux splitting (18) is applied to get flux components: $g^{\pm}_j = \frac{1}{2}\left(R^{-1}_{i+\frac{1}{2}}f(U_j) \pm \alpha_j R^{-1}_{i+\frac{1}{2}}U_j\right)$. The j^{th} wave-speed component α_j is the maximum eigenvalue $\max_{U_j}|\Lambda_{i+\frac{1}{2}}|$ evaluated over the WENO stencil points i. In the final code each element of these matrix-vector multiplications is stored as a thread local variable, to be reused throughout the WENO reconstruction and avoiding slow global memory accesses.
4. The WENO reconstruction (8) is applied to the characteristic flux components over the WENO stencil points.
5. Characteristic reconstructions are transformed back to physical space with the averaged right eigenvector matrix $R_{i+\frac{1}{2}}$ as $\hat{f}^{\pm}_{i+\frac{1}{2}} = R_{i+\frac{1}{2}}\hat{g}^{\pm}_{i+\frac{1}{2}}$.
6. The finite-difference approximation of the derivative is built for each component of the system as in (17).
7. Steps 2-6 are repeated for all further dimensions in the problem.

Viscous terms of the Navier-Stokes equations are computed in this paper with 4^{th} order central differencing in the interior of the domain, replaced by a 4^{th} order one-sided Carpenter scheme [13] at domain boundaries. Time-stepping is performed by a low-storage explicit 3^{rd} order Runge-Kutta scheme.

3.2. Example OpenSBLI user specified problem code

A cut down version of a problem specification file to generate a code in OpenSBLI is presented, restricted in this case to the continuity equation using WENO schemes; additional schemes, time-stepping and boundary conditions are omitted for brevity but follow a similar structure.

```
1  mass = "Eq(Der(rho,t), - Conservative(rhou_j,x_j,**{scheme:Weno}))"
2  simulation_eq = SimulationEquations(); ndim = 2 # Equation class.
3  simulation_eq.add_equations(EinsteinEquation().expand(mass, ndim,
       coordinate_symbol="x", substitutions=[], constants=[]))
4  LLF = LLFWeno(weno_order=5, formulation='Z', RoeAverage([0, 1]))
5  block = SimulationBlock(ndim, block_number=0) # Create a block.
6  block.set_equations([simulation_eq]) # Set equations on the block.
7  block.set_discretisation_schemes({LLF.name: LLF})
8  block.discretise() # Discretise the equations on the block.
9  alg = TraditionalAlgorithmRK(block) # Create solution algorithm.
10 OPSC(alg) # Generate OPS C code.
```

The continuity equation is defined in line 1 as a string, which is parsed and expanded in line 3 by the *EinsteinEquation* class. During parsing the equation is formed using the SymPy equality class *Eq*, and the derivative and conservative derivative operators in OpenSBLI: *Der*, and *Conservative* of type WENO. The expanded equation is added to the *SimulationEquation* class, that contains the equations to be solved by the simulation. In line 4 the characteristic scheme is instantiated with local Lax-Friedrich flux splitting, Roe averaging and WENO-Z of the specified order. Time-stepping and arbitrary order central differencing schemes would be instantiated here in a similar fashion, for full details see [3]. To apply the numerical methods, an OpenSBLI *SimulationBlock* is instantiated in line 5, with simulation equations and the schemes to solve them set on the block in lines 6-7. Similarly, boundary and initial conditions would be set on the block at this stage.

Discretisation of the governing equations is performed in line 8, whereby the continuous symbolic derivatives are replaced by their discrete representation for the chosen numerical scheme and order. The computations to be performed are stored as equations in a computational *Kernel*, which describe the calculations to be executed relative to a generic grid point. The algorithm class in line 9 collects all of the *Kernels* for the equations on a block, and orders them based

on their dependencies within the RK3 time-stepping scheme. Arrays, constants and other dependences to be declared within the program are extracted and stored as attributes to the algorithm class. Each stage of the algorithm has components, giving a developer the flexibility to add additional components at a desired place within the algorithm.

Once the algorithm is generated, the final step is to call the code-writer class in line 10. During code writing each stage of the algorithm is converted into OPS-compliant C code, with for example declaration of data structures and parallel regions written out as calls to the OPS library. The OPS C code is then translated to various parallel implementations using the OPS library ready for compilation. To generate code for another language such as Fortran, only the code-writer class needs to be modified. All other components of the OpenSBLI framework are independent of the final code writing process, to make the framework future-proof.

4. Application: Katzer 2D shock-wave/boundary-layer interaction

Katzer [9] performed simulations of SBLI with free-stream Mach numbers ranging from 1.4 to 3.4. For this validation the Mach 2 case is taken, with Reynolds number $Re_x = 3{\times}10^5$ based on the distance from the inlet to shock impingement location. A shock is generated by enforcing Rankine-Hugoniot jump conditions at $x_0 = 40$ for a shock angle of $32.58°$. At this shock strength with inlet pressure p_1, the outlet pressure is equal to $p/p_1 = 1.4$ at the wall.

The domain is of size $[x_0, x_1] = [400, 115]$, so that reflections on the upper boundary from the initial reflected shock fall outside of the computational domain and do not influence the interaction region. A pressure extrapolation inlet is applied at the left boundary, no-slip isothermal wall conditions at the bottom, and Dirichlet and zeroth-order extrapolation used on the upper and outlet boundaries respectively. At the start of the simulation the domain is initialised with a similarity solution of the compressible boundary-layer equations [14].

The Reynolds number based on the inlet displacement thickness is $Re_\delta =$

950. Sutherland's law is used for dynamic viscosity with the free-stream and Sutherland temperatures $T_\infty = 288.0$K and $T_s = 110.4$K, and the Prandtl number is taken to be 0.71. Isothermal wall conditions are held at a constant non-dimensional temperature of $T_w = 1.676$ (4 s.f.), equal to the adiabatic wall temperature from the similarity solution. Simulations were advanced with time-step $dt = 0.04$ until the length of the separation region converged, corresponding to a non-dimensional time of $t = 1.3 \times 10^4$.

4.1. Results

To validate the code implementation, comparison to the reference solution published in [1] was performed. A (609×255) grid stretched in the wall normal x_1 direction was used, with length L_{x_1} and stretch factor β as

$$x_1 = \frac{L_{x_1} \sinh(\beta\eta)}{\sinh(\beta)}, \qquad \eta \sim U[0,1] \qquad \beta = 5. \tag{19}$$

Figure 2 shows Mach number contours for the final flow field, a well resolved shock-wave impinges on the boundary-layer, causing thickening of the profile and the development of a separation bubble.

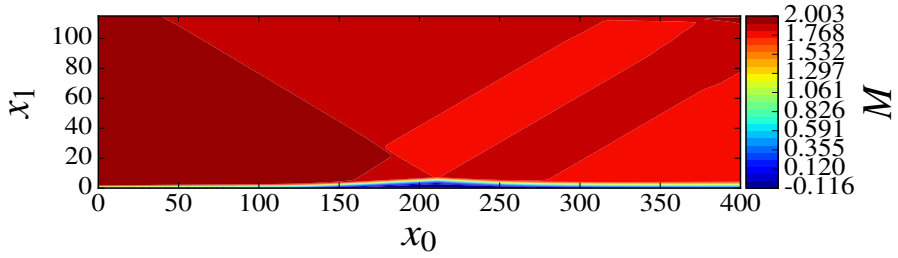

Figure 2: Contours of Mach number, 609×255 stretched grid.

Figure 3 shows the skin-friction distribution in x_0 along the wall, with the negative region corresponding to the reversed flow within the separation bubble. The expected asymmetric shape of the separation bubble is observed, with excellent agreement to the reference solution of [1]. Figure 4 provides a similar picture, demonstrating the pressure rises at the regions of separation and reattachment, and the correct pressure ratio of $p/p_1 = 1.4$ at the outlet.

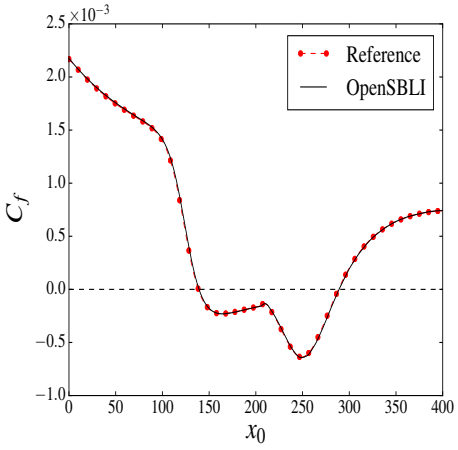

Figure 3: Wall-normal skin friction.

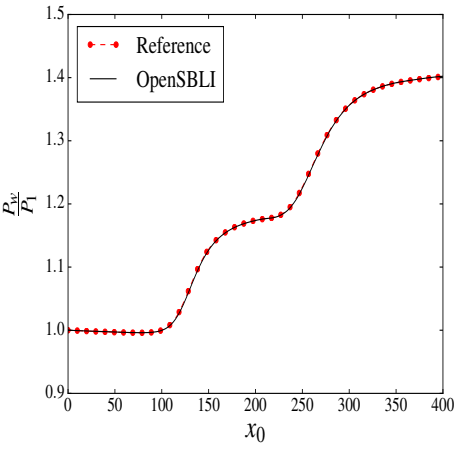

Figure 4: Normalised wall pressure.

5. Computational architecture comparison

A runtime comparison was performed for a 3D spanwise periodic version of the SBLI case presented in the previous section, on a $(300 \times 400 \times 25)$ grid. The 3D case was selected as a more representative computational workload for future studies, whereas 2D grids are too small in resolution to offer a useful performance comparison. Secondly, it also demonstrates the ease of extension to 3D problems within OpenSBLI; the equations are expanded to the number of dimensions selected by the user. Table 1 shows the runtime on various architectures after 500 iterations, with a speed-up factor relative to one node (24 CPU cores, MPI) on the UK's national ARCHER HPC facility. MPI codes were compiled with the Intel compiler optimised with function inlining and -O3, CUDA code was compiled using GCC/NVCC and -O3 optimisation.

The Intel Xeon Phi with 256 threads enabled achieved a speed-up of 1.84 relative to the CPU node, utilizing a hybrid MPI/OpenMP OPS generated code. Similar performance was found for the NVIDIA K20X and K40 GPUs, both proving to be fast alternatives to a conventional CPU node. Significant performance improvements were seen for the Pascal based NVIDIA P100, achieving a 4.65x speed-up relative to a previous generation NVIDIA K40 GPU. Further

performance comparisons of OPS generated code is given in [15], with details of the implementation and memory management.

Target architecture (processes/threads)	Time (s)	Speed-up
Intel Xeon E5-2697 24 core node (CPU, 24 MPI)	413.1	1.00
Intel Xeon Phi 7210 (64 MPI x 4 OMP, AVX512)	224.7	1.84
NVIDIA Tesla K20X (GPU, CUDA)	233.9	1.77
NVIDIA Tesla K40 (GPU, CUDA)	204.6	2.02
NVIDIA Tesla P100 (GPU, CUDA)	44.0	9.39

Table 1: Runtime comparison. Ivy-Bridge 24 CPU cores (MPI) is taken as the baseline time.

6. Conclusions

Shock-wave/boundary-layer interactions were carried out in the automatic source code generator OpenSBLI, with excellent agreement to a reference solution observed. To be able to perform simulation of SBLIs, the capabilities of OpenSBLI had to be greatly enhanced for this report. Robust WENO-based shock-capturing, a characteristic transform and a high-order boundary treatment were among the features added and validated here.

Coupling of OpenSBLI to the OPS library enables the code to run on multiple architectures from a single source code. Strong performance was seen on many-core architectures, highlighting the potential performance benefits GPUs and accelerators have over conventional CPU systems. Recent trends in high performance computing have seen a shift towards hybrid systems with large deployments of GPUs, as they offer substantial performance gains at a fraction of the energy cost. Modern approaches to computational science including code generation and DSL abstractions offer an efficient way of targeting emerging architectures, and minimising the development cost to the scientific researcher.

7. Acknowledgements

DJL is funded by an EPSRC Centre for Doctoral Training grant (EPSRC grant EP/L015382/1). SPJ is supported by a European Commission Horizon 2020 project grant entitled ExaFLOW: Enabling Exascale Fluid Dynamics Simulations (grant reference 671571). The authors acknowledge the use of the UK National Supercomputing Service (ARCHER), with computing time provided by the UK Turbulence Consortium (EPSRC grant EP/L000261/1), and the support of the NVIDIA PSG Cluster with donation of the Tesla GPUs. Data from this report is available at `https://doi.org/10.5258/SOTON/D0458`.

References

[1] A. Sansica, N. Sandham, Z. Hu, Stability and Unsteadiness in a 2D Laminar Shock-Induced Separation Bubble, 43rd AIAA Fluid Dynamics Conference.

[2] J. Sivasubramanian, H. F. Fasel, Numerical Investigation of Shock-Induced Laminar Separation Bubble in a Mach 2 Boundary Layer, 45th AIAA Fluid Dynamics Conference 2641 (2015) 1–36.

[3] C. T. Jacobs, S. P. Jammy, N. D. Sandham, OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures, Journal of Computational Science 18 (2017) 12 – 23.

[4] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, S. McIntosh-Smith, The OPS Domain Specific Abstraction for Multi-block Structured Grid Computations, WOLFHPC '14, IEEE Press, 2014, pp. 58–67.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral program optimization system, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2008.

[6] D. Unat, X. Cai, S. B. Baden, Mint: realizing CUDA performance in 3D stencil methods with annotated C, Proceedings of the international conference on Supercomputing (2011) 214–224.

[7] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, G. Gorman, Devito:Towards a Generic Finite Difference DSL Using Symbolic Python, PyHPC '16, IEEE Press, pp. 67–75.

[8] S. P. Jammy, C. T. Jacobs, N. D. Sandham, Performance evaluation of explicit finite difference algorithms with varying amounts of computational and memory intensity, Journal of Computational Science.

[9] E. Katzer, On the lengthscales of laminar shock/boundary-layer interaction, Journal of Fluid Mechanics 206 (-1) (1989) 477.

[10] C.-w. Shu, Essentially Non-Oscillatory and Weighted Essentially Non-Oscillatory Schemes for Hyperbolic Conservation Laws Operated by Universities Space Research Association, ICASE Report (97-65) (1997) 1–78.

[11] R. Borges, M. Carmona, B. Costa, W. S. Don, An improved weighted essentially non-oscillatory scheme for hyperbolic conservation laws, Journal of Computational Physics 227 (6) (2008) 3191–3211.

[12] C. Brehm, M. F. Barad, J. A. Housman, C. C. Kiris, A comparison of higher-order finite-difference shock capturing schemes, Computers and Fluids 122 (2015) 184–208.

[13] M. H. Carpenter, J. Nordström, D. Gottlieb, A Stable and Conservative Interface Treatment of Arbitrary Spatial Accuracy, Journal of Computational Physics 365 (98) (1998) 341–365.

[14] F. White, Viscous fluid flow, McGraw-Hill, New York, NY, 2006.

[15] G. R. Mudalige, I. Z. Reguly, M. B. Giles, W. Gaudin, A. Mallinson, O. Perks, High-level abstractions for performance, portability and continuity of scientific software on future computing systems (2014).