Computer Physics Communications 267 (2021) 108063

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

OpenSBLI: Automated code-generation for heterogeneous computing architectures applied to compressible fluid dynamics on structured grids ☆,☆☆

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

[a] Aerodynamics and Flight Mechanics group, University of Southampton, Boldrewood Campus, Southampton, SO16 7QF, United Kingdom
[b] Faculty of Mechanical Engineering, SRM University, AP, Andhra Pradesh, 522502, India

ARTICLE INFO

Article history:
Received 2 July 2020
Received in revised form 25 March 2021
Accepted 24 May 2021
Available online 11 June 2021

Keywords:
SBLI
CFD
GPUs
Finite-difference
Code-generation

ABSTRACT

OpenSBLI is an open-source code-generation system for compressible fluid dynamics (CFD) on heterogeneous computing architectures. Written in Python, OpenSBLI is an explicit high-order finite-difference solver on structured curvilinear meshes. Shock-capturing is performed by a choice of high-order Weighted Essentially Non-Oscillatory (WENO) or Targeted Essentially Non-Oscillatory (TENO) schemes. OpenSBLI generates a complete CFD solver in the Oxford Parallel Structured (OPS) domain specific language. The OPS library is embedded in C code, enabling massively-parallel execution of the code on a variety of high-performance-computing architectures, including GPUs. The present paper presents a code base that has been completely rewritten from the earlier proof of concept Jacobs et al. (2017) [7], allowing shock capturing, coordinate transformations for complex geometries, and a wide range of boundary conditions, including solid walls with and without heat transfer. A suite of validation and verification cases are presented, plus demonstration of a large-scale Direct Numerical Simulation (DNS) of a transitional Shockwave Boundary Layer Interaction (SBLI). The code is shown to have good weak and strong scaling on multi-GPU clusters. We demonstrate that code-generation and domain specific languages are suitable for performing efficient large-scale simulations of complex fluid flows on emerging computing architectures.

Program summary

Program Title: OpenSBLI code-generation framework for compressible fluid dynamics on heterogeneous architectures
CPC Library link to program files: https://github.com/opensbli/opensbli
Licensing provisions: GPLv3
Programming languages: Python, C/C++, OPS DSL
Nature of problem: The compressible 3D Navier-Stokes equations are solved via Implicit Large Eddy Simulation (ILES) or Direct Numerical Simulation (DNS).
Solution method: OpenSBLI [1,2] is a Python-based code-generation system that uses symbolic algebra to generate a complete CFD solver in C/C++. The basic algorithm is a stencil-based finite-difference solver on structured curvilinear meshes. Shock-capturing is performed by a selection of high-order Weighted/Targeted Essentially Non-Oscillatory (WENO/TENO) schemes. Explicit low-storage Runge-Kutta schemes are used for time-advancement.
Additional comments including restrictions and unusual features: The generated code is compliant with the Oxford Parallel Structured (OPS) [3] software library. OpenSBLI/OPS executables can be generated for the OpenMP, MPI, CUDA, OpenCL, and OpenACC parallel programming paradigms. Multi-GPU support is available via combinations of MPI with CUDA, OpenCL or OpenACC.

☆ The review of this paper was arranged by Prof. N.S. Scott.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

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

https://doi.org/10.1016/j.cpc.2021.108063

References

[1] D.J. Lusher, S.P. Jammy, N.D. Sandham, Shock-wave/boundary-layer interactions in the automatic source-code generation framework OpenSBLI, Comput. Fluids 173 (2018) 17-21.
[2] 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, J. Comput. Sci. 18 (2017) 12-23.
[3] I.Z. Reguly, G.R. Mudalige, M.B. Giles, D. Curran and S. McIntosh-Smith, The OPS Domain Specific Abstraction for Multi-Block Structured Grid Computations, in: Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC '14), Held in conjunction with IEEE/ACM Supercomputing 2014 (SC'14).

1. Introduction

The role of Computational Fluid Dynamics (CFD) in modern aerospace research is well established [1]. CFD has become an integral part of the aeronautical research and design process. CFD can complement the data obtained from wind tunnels and in-flight testing, at potentially a fraction of the cost. Furthermore, high-fidelity Large Eddy Simulations (LES) or Direct Numerical Simulations (DNS) can reveal physical insights that would be difficult to investigate experimentally. The dramatic increase in computational power over the past few decades has broadened the scope of problems that can be tackled by LES/DNS. Two of the main challenges in this field today, are the efficient utilization of computational hardware, and the development of accurate and reliable numerical methods.

A recent trend in high-performance computing (HPC), has been a shift to ever-increasing levels of heterogeneity [2]. Graphical Programming Units (GPUs) and other types of accelerators are now being applied to many diverse areas of computational science [3]. In addition to the vast available compute capacities, these emerging architectures can offer substantial improvements in power efficiency for large systems. One of the drawbacks limiting their uptake compared to conventional CPU-based platforms however, is the need for programming models suited to these architectures. Existing CFD solvers designed for CPUs often contain large amounts of legacy code, and can be inflexible to changes in computational hardware. Porting existing codes to new architectures can be a very time consuming process. One potential solution to this problem is the *'separation of concerns'* philosophy [4], applied in the present work. This approach separates the physical problem and numerical methods from their parallel implementation, allowing the researcher to focus solely on modelling the physical problem at hand, while benefiting from performance optimisations and hardware-specific knowledge from computer science [5].

The present work describes the Python-based OpenSBLI automatic code-generation system for compressible fluid dynamics. OpenSBLI is a high-order stencil-based finite-difference solver on structured meshes. The symbolic algebra library SymPy [6], is used to generate a CFD solver tailored to the equations and schemes specified by the user in a high-level Python script. An initial proof-of-concept version of this approach was presented in [7], to demonstrate the basic feasibility of code-generation for CFD. The demonstrator version was limited to smooth (shock free) subsonic problems on triply periodic domains. Given the substantial changes needed to develop the concept into a more generally useful research code for shock-wave boundary-layer interactions, the new version of OpenSBLI described in this work was started from a separate code base. It is capable of simulating complex wall-bounded flows with shockwaves, for supersonic and hypersonic CFD applications on curvilinear meshes. The purpose of this work is to describe the OpenSBLI software package and its use of code-generation techniques, present a set of a validation and verification cases, and demonstrate a 3D shockwave boundary-layer interaction representative of ongoing research [8–10].

OpenSBLI generates a complete CFD solver in the Oxford Parallel Structured (OPS) Embedded Domain Specific Language (EDSL) [11,12]. The OPS library enables execution of the code on multiple massively-parallel computing architectures. Performance of the automatically generated OPS code has been shown to as good as, or better than, hand-coded version of the same application [13,14].

Recent examples of comparable CFD codes on GPUs includes: Hydra [15], HiPSTAR [16], PyFR [17], the HTR solver [18], and STREAmS [19]. Hydra is an unstructured mesh solver that has been widely used commercially for turbo-machinery applications. It uses the OP2 EDSL [20] from the same authors as the OPS library used in this work. HiPSTAR is a high-order curvilinear finite-difference code that also originated from the University of Southampton. It has been applied to large-scale simulations of both low and high-pressure turbine cascades. PyFR [17] is a Python-based unstructured mesh framework, to solve advection-diffusion problems on streaming architectures. The code utilises a domain specific language that uses Mako templates for platform portability, allowing PyFR to be compatible with both the CUDA and OpenCL programming languages, and with OpenMP in C. HTR [18] is a hypersonic aero-thermodynamics code that includes temperature-induced thermochemical effects, and a 6th order Targeted Essentially Non-Oscillatory (TENO) scheme for shock-capturing. The code is written in the Regent programming language, using the task-based Legion system to execute the code on GPUs. Finally, STREAmS [19] is a compressible DNS solver that uses a hybridised Weighted Essentially Non-Oscillatory (WENO) scheme for wall-bounded turbulent flows. Based on an existing CPU-solver, the code uses CUDA-Fortran90 kernels to run large-scale DNS on modern GPU clusters.

One of the novel features of OpenSBLI is the use of symbolic code-generation to write the simulation code from Python. The OPS library is then used to create parallel versions of the code for a number of parallel programming paradigms. At present this includes MPI, OpenMP, OpenMP+MPI, CUDA, OpenCL, and OpenACC. Code-generation allows for a large number of numerical schemes to be contained within a compact code-base. OpenSBLI has a number of high-order accurate spatial discretization schemes. These include various orders of WENO/TENO shock-capturing [21–24], and central-differencing which can be cast in split skew-symmetric forms to improve numerical stability [25]. Time-advancement is performed by low-storage explicit 3rd and 4th order Runge-Kutta schemes [26]. The ability to define equations compactly in index notation in a high-level Python script, gives the user the flexibility to control core aspects of the solver.

Code-generation techniques have been applied to a wide range of areas within computational science, such as space-systems, robotics, and control [27], weather-modelling [28], and computational neuroscience [29]. A recent example of a comparable project to this work is Devito [30], a code-generation system aimed at

geophysical applications. Devito also uses the SymPy Python library to generate finite-difference stencils. The symbolic library has been used to manipulate and optimise expressions to improve computational efficiency [31]. While initially targeting the Intel Xeon-Phi platform, Devito has recently utilised the OPS DSL used in this work to target GPU clusters [32]. Patus [33] is another example of code-generation being used to optimise stencil-based computations for multi- and many-core architectures. Examples of code-generation applied to finite-element frameworks include the Discontinuous Galerkin methods described in [34], and the widely-used FeNICS package for solving PDEs.

The purpose of this work is to describe the main features of the OpenSBLI design, with presentation of a suite of validation and verification test cases. The work is structured as follows: Section 2 gives an overview of the numerical methods used in OpenSBLI. This includes a characteristic flux reconstruction, high-order WENO/TENO shock-capturing, central-differencing with one-sided boundary closures, and time-advancement schemes. Section 3 describes the OpenSBLI code implementation, with discussion of the core components and an example problem script. Section 3.3 shows examples of the OPS C code that results from the code-generation process. A selection of validation and verification cases are shown in section 4, plus demonstration of a large-scale transitional Shockwave Boundary-Layer Interaction (SBLI) DNS in section 5. Finally, section 6 gives a brief discussion of computational performance in the OPS DSL.

2. Numerical methods

Scale-resolving simulations of turbulence benefit from the use of high-order accurate numerical methods, which can alleviate the excessive levels of numerical diffusion associated with lower-order approximations [35,36]. In the context of compressible turbulence with shockwaves, there are two contrasting requirements placed on the numerical methods. Shock-capturing schemes stabilise the solution by adding numerical dissipation in the vicinity of flow discontinuities, but have the detrimental effect of damping small-scale turbulence [37]. This creates a requirement for very fine grids to achieve acceptable resolution of small-scale flow structures, exacerbating the already high computational cost of LES/DNS.

Common approaches include the hybrid pairing of non-dissipative central schemes with shock-capturing schemes via a shock-sensor [36]. While these have been successful, it is difficult to design shock-sensors suitable for every type of flow conditions [37]. WENO schemes [21] are family of robust high-order shock-capturing methods that have seen widespread use in compressible CFD and Magneto-Hydrodynamics (MHD). Examples of their use for SBLI includes [38]. WENO schemes have previously been shown to be superior to lower-order shock-capturing methods [39,40]. TENO schemes [23,24], are a more recent development designed for reduced numerical dissipation compared to WENO [41]. Alternative approaches includes the framework of [42], which pairs skew-symmetric central and Dispersion Relation Preserving (DRP) schemes to a dissipative WENO scheme applied as a non-linear filtering step. The non-dissipative part of the framework can be written in either the Ducros-split [25], or entropy-split [43] forms to enhance numerical stability. The present section describes the high-order numerical methods in OpenSBLI, beginning with the WENO/TENO flux reconstruction procedure for the compressible Navier-Stokes equations.

2.1. Governing equations

OpenSBLI uses numerical indices to distinguish between variables that have a dependence on dimension. For example, the Cartesian coordinate base (x, y, z) and their respective velocity components (u, v, w), are taken to be (x_0, x_1, x_2), and (u_0, u_1, u_2) respectively in the code. This approach allows the code-generation to be more flexible; all of the indexed quantities and loops are generated dynamically, based on the number of dimensions set in the problem script. The base governing equations in this work are the dimensionless compressible Navier-Stokes equations for a Newtonian fluid. Applying conservation of mass, momentum, and energy, in the three spatial directions x_i $(i = 0, 1, 2)$, results in a system of five partial differential equations to solve. These equations are defined for a density ρ, pressure p, temperature T, total energy E, and velocity components u_k as

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

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

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

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

$$q_k = \frac{-\mu}{(\gamma - 1)\, M_{\text{ref}}^2 Pr Re}\frac{\partial T}{\partial x_k}, \tag{4}$$

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

Pr, Re, and γ are the Prandtl number, Reynolds number and ratio of heat capacities respectively. The equations are non-dimensionalized by a reference velocity, density and temperature $\left(U^*_{\text{ref}}, \rho^*_{\text{ref}}, T^*_{\text{ref}}\right)$. For the SBLI cases, the characteristic length is the displacement thickness δ^* of the boundary layer imposed at the inlet. Pressure is normalised by $\rho^*_{\text{ref}} {U^*_{\text{ref}}}^2$. For cases with temperature dependent dynamic viscosity, $\mu(T)$ is evaluated using Sutherland's law

$$\mu(T) = T^{\frac{3}{2}}\left(\frac{1 + \frac{T^*_s}{T^*_{\text{ref}}}}{T + \frac{T^*_s}{T^*_{\text{ref}}}}\right), \tag{6}$$

for a reference temperature T^*_{ref}. The Sutherland temperature constant is set to be $T^*_s = 110.4$K. For a reference Mach number M_{ref}, pressure and local speed of sound are defined as

$$p = (\gamma - 1)\left(\rho E - \frac{1}{2}\rho u_i u_i\right) = \frac{1}{\gamma M_{\text{ref}}^2}\rho T \quad \text{and} \quad a = \sqrt{\frac{\gamma p}{\rho}}. \tag{7}$$

For the SBLI cases, the skin friction C_f is calculated from the wall shear stress τ_w as

$$\tau_w = \mu\left.\frac{\partial u}{\partial y}\right|_{y=0}, \qquad C_f = \frac{\tau_w}{\frac{1}{2}\rho_{\text{ref}} U^2_{\text{ref}}}. \tag{8}$$

The high-level Python interface in OpenSBLI allows users to modify the equations to be solved in the simulation. Examples of this are given in the code repository for the turbulent channel flow applications, where the governing equations are recast in split skew-symmetric formulations to improve numerical stability. It is expected that this flexibility will enable OpenSBLI to be extended to equations beyond the compressible Navier-Stokes equations presented in this work. The next section outlines the flux reconstruction applied to the convective terms in equations (1-3), for problems requiring the use of the WENO/TENO shock-capturing schemes. Heat-flux (4) and stress-tensor (5) terms are computed with central differences, as described in section 2.6. The system of equations is advanced in time using the explicit Runge-Kutta methods in section 2.7.

2.2. Flux reconstruction

The convective terms of the Navier-Stokes equations form a set of conservation laws that can be approximated by flux reconstruction methods. For a given physical dimension, a finite-difference method creates discrete representations of derivatives on a set of i grid points as in Fig. 1. Taking the example of a scalar conservation equation

$$\frac{\partial U}{\partial t} + f(U)_x = 0, \tag{9}$$

the flux term $f(U)_x$ can be approximated by computing two half-node reconstructions $\left[\hat{f}_{i+\frac{1}{2}}, \hat{f}_{i-\frac{1}{2}}\right]$, such that the flux $f(U)_x$ is replaced by

$$\frac{1}{\Delta x}\left(\hat{f}_{i+\frac{1}{2}} - \hat{f}_{i-\frac{1}{2}}\right), \tag{10}$$

for a grid spacing of Δx. For a general flux $f(U)$, splitting methods are applied to account for upstream and downstream propagating information

$$f(U) = f^{+}(U) + f^{-}(U), \tag{11}$$

where the plus and minus superscripts represent the cases [44]

$$\frac{\partial f^{+}(U)}{\partial U} > 0 \quad \text{and} \quad \frac{\partial f^{-}(U)}{\partial U} < 0. \tag{12}$$

The most common flux-splitting method is the Local Lax-Friedrich (LLF) flux

$$f^{\pm}(U) = \frac{1}{2}\left(f(U) \pm \alpha U\right), \tag{13}$$

for a given wave-speed α. For systems of equations, the flux can be applied to each component in succession. To improve the robustness of the shock-capturing [40], the reconstructions are performed in characteristic space as described in [21]. The algorithm is summarised as follows for a system of j equations.

1. Construct the flux $f(U_j)$ and solution vector U_j terms at every i grid point.
2. At each half-node $x_{i+\frac{1}{2}}$ perform the following:
   (a) Compute the average $U_{i+\frac{1}{2}}$ state with either simple or Roe averaging.
   (b) Obtain the eigensystem $R(U_{i+\frac{1}{2}})$, $R^{-1}(U_{i+\frac{1}{2}})$ and $\Lambda(U_{i+\frac{1}{2}})$ to diagonalize the equations.
   (c) Transform the solution and flux vector into characteristic space as $V_j = R^{-1}U_j$, and $g_j = R^{-1}f(U_j)$.
   (d) Apply the flux-splitting $g_j^{\pm} = \frac{1}{2}\left(g_j \pm \alpha_j V_j\right)$, where $\alpha_j = \max_k |\lambda_j|$, are the characteristic wave-speeds over the local stencil points k.
   (e) Perform the high-order WENO/TENO reconstruction at $x_{i+\frac{1}{2}}$.
   (f) Transform the flux back to physical space $R\hat{g}^{\pm}_{i+\frac{1}{2}}$.
3. Build the finite-difference approximation as in (10).
4. Repeat for all other dimensions in the problem.

The half-node flux reconstructions $\left[\hat{f}_{i+\frac{1}{2}}, \hat{f}_{i-\frac{1}{2}}\right]$ can take many forms, allowing for the construction of high-order approximations of the interface fluxes. OpenSBLI uses the WENO and TENO high-order reconstruction methods described in the next sections.

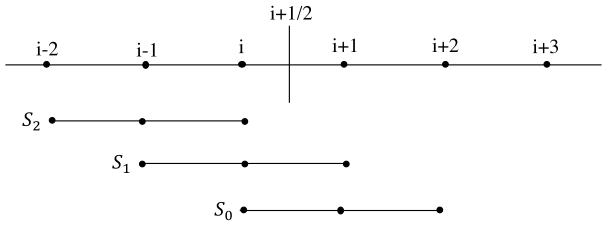

Fig. 1. Schematic of the finite-difference WENO stencils.

2.3. Weighted essentially non-oscillatory (WENO) schemes

WENO schemes construct a high-order approximation for the half-node fluxes $\left[\hat{f}_{i+\frac{1}{2}}, \hat{f}_{i-\frac{1}{2}}\right]$, by building up a convex combination of a set of smaller candidate stencils (Fig. 1). The candidate stencils are weighted based on the local smoothness of the flow. This mechanism avoids differencing over flow discontinuities, resulting in essentially non-oscillatory behaviour around shocks [21]. For a WENO reconstruction of order $2k-1$, k candidate stencils are required. In each candidate stencil an interpolation is applied such that

$$\hat{f}^{(r)}_{i+\frac{1}{2}} = \sum_{j=0}^{k-1} c_{rj} f_{i-r+j}, \qquad r = [0, k-1], \tag{14}$$

where c_{rj} are the standard ENO coefficients given in [21]. The WENO reconstruction is then formed as

$$\hat{f}_{i+\frac{1}{2}} = \sum_{r=0}^{k-1} \omega_r \hat{f}^{(r)}_{i+\frac{1}{2}}, \tag{15}$$

for a non-linear weighting ω_r. The choice of ω_r differs between the various WENO formulations. OpenSBLI uses both the original WENO-JS weightings [21], and those of the improved WENO-Z scheme [22].

The fundamental aspect of ENO/WENO reconstructions is the mechanism to select certain candidate stencils. The scheme must be capable of identifying discontinuities in the flow, to remove discontinuity-crossing stencils from the final reconstruction. The most commonly used smoothness indicator was introduced by [44], defined as

$$\beta_k = \sum_{l=1}^{r-1} \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} \Delta x^{2l-1} \left(q_k^{(l)}\right)^2 dx, \tag{16}$$

where $q_k^{(l)}$ is the l-th derivative of the $(r-1)$-th order interpolating polynomial $q_k(x)$ over a candidate stencil S_k. This smoothness indicator forms the sum of L^2 norms of the interpolating polynomial derivatives over a cell width. OpenSBLI generates the expressions for the smoothness indicators dynamically during code-generation, for the order specified by the user. The code supports generation of WENO schemes for any arbitrary odd order. As an example, for a 5th order WENO scheme the smoothness indicators are given by

$$\beta_0 = \frac{13}{12}\left(f_i - 2f_{i+1} + f_{i+2}\right)^2 + \frac{1}{4}\left(3f_i - 4f_{i+1} + f_{i+2}\right)^2, \tag{17}$$

$$\beta_1 = \frac{13}{12}\left(f_{i-1} - 2f_i + f_{i+1}\right)^2 + \frac{1}{4}\left(f_{i-1} - f_{i+1}\right)^2, \tag{18}$$

$$\beta_2 = \frac{13}{12}\left(f_{i-2} - 2f_{i-1} + f_i\right)^2 + \frac{1}{4}\left(f_{i-2} - 4f_{i-1} + 3f_i\right)^2, \tag{19}$$

where f is one of the discrete flux terms from the governing equations.

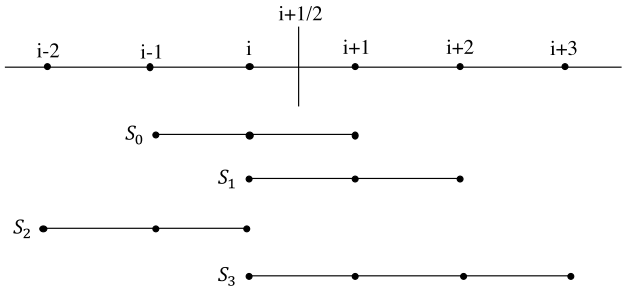

Fig. 2. Schematic of the finite-difference TENO stencils.

To construct the $(2k-1)$-th order WENO approximation, the non-linear WENO weights are normalized for $r = [0, k-1]$ as

$$\omega_r = \frac{\alpha_r}{\sum_{n=0}^{k-1} \alpha_n}, \tag{20}$$

with the smoothness indicators β_r forming part of the alpha terms as

$$\alpha_r = \frac{d_r}{(\epsilon + \beta_r)^p}, \tag{21}$$

for optimal weights d_r and constants, p, ϵ. The standard value of $p = 2$ is used, and ϵ set to a small non-zero number (10^{-6}) to avoid division by zero. Optimal weights d_r are taken from [21].

2.4. WENO-Z formulation

The WENO-Z formulation [22], [45], is a substantial improvement over the base scheme in terms of achieving lower numerical dissipation [46], while retaining robust shock-capturing. Non-linear weights ω_r from the improved formulation are

$$\omega_r^z = \frac{\alpha_r^z}{\sum_{n=0}^{k-1} \alpha_n^z}, \qquad \alpha_r^z = d_r \left(1 + \left(\frac{\tau}{\sigma_r + \epsilon}\right)^2\right), \tag{22}$$

with $(\epsilon \sim 10^{-16})$. Smoothness indicators β_r and optimal weights d_r are unchanged. WENO-Z introduces a global smoothness measure τ, representing the absolute difference in the smoothness indicators over the full reconstruction stencil. As an example, at 5th order $(k = 3)$ the global smoothness measure is calculated from [45] as $\tau = |\beta_0 - \beta_2|$. As only minor changes are required compared to the base scheme, the performance impact of WENO-Z is almost negligible [46].

2.5. Targeted Essentially Non-Oscillatory (TENO) schemes

A more significant improvement was introduced by the TENO schemes of [23,24]. TENO schemes fit into the same flux reconstruction framework as WENO, with identical flux-splitting and characteristic decompositions. TENO schemes differ from WENO however in three fundamental ways: a staggered ordering of candidate stencils as in Fig. 2, the complete removal of candidate stencils deemed to be non-smooth, and modified non-linear weights optimized for low numerical dissipation. For a K-th order TENO scheme with r candidate stencils, the non-linear weights take the form

$$\omega_r = \frac{d_r \delta_r}{\sum_{r=0}^{K-3} d_r \delta_r}, \tag{23}$$

where δ_r is a discrete cut-off function

$$\delta_r = \begin{cases} 0 & \text{if} \quad \chi_r < C_T \\ 1 & \text{otherwise} \end{cases}$$

for a tunable cut-off parameter C_T. The smoothness measures χ_r are the same as the weight normalization process as in WENO

$$\chi_r = \frac{\gamma_r}{\sum_{r=0}^{K-3} \gamma_r}, \tag{24}$$

comprised of the WENO-Z inspired form of non-linear TENO weights [23]

$$\gamma_r = \left(C + \frac{\tau_K}{\beta_r + \epsilon}\right)^q, \quad r = 0, \ldots, K-3, \tag{25}$$

with $C = 1$, and $q = 6$. Smoothness indicators β_r are unchanged from the standard Jiang-Shu formulation [44], and $\epsilon \sim 10^{-16}$. The global smoothness indicator τ_K measures smoothness over the entire stencil, and is given for 5th and 6th order TENO schemes as

$$\tau_5 = |\beta_0 - \beta_2|, \tag{26}$$

$$\tau_6 = |\beta_3 - \frac{1}{6}(\beta_0 + \beta_2 + 4\beta_1)|. \tag{27}$$

C_T is the user-specified parameter which determines whether a given candidate stencil is rejected or contributes to the flux reconstruction. Lower values of C_T are suitable for compressible turbulence simulations where minimal numerical dissipation is required, but this comes at the cost of increased spurious oscillations around shockwaves. C_T is typically taken to be between 10^{-5} and 10^{-7}, depending on the physical problem. The computational cost of the TENO schemes is approximately 15-20% greater than a WENO scheme of equivalent order [46], but offers significantly lower dissipation while retaining sharp shock capturing. OpenSBLI has both 5th and 6th order TENO schemes available.

2.6. Central schemes for heat-flux, viscous, and metric terms

Diffusive terms in OpenSBLI are computed by central-differences. Central schemes are also applied to smooth problems that do not require shock-capturing. The code-generation can produce central-difference approximations for any even-ordered central scheme. A fourth order central scheme is used throughout this work. For a grid spacing of Δx, the formula for first and second derivatives are

$$f_i' = \frac{-f_{i-2} + 8f_{i-1} - 8f_{i+1} + f_{i+2}}{12\Delta x}, \tag{28}$$

$$f_i'' = \frac{-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}}{12\Delta x^2}. \tag{29}$$

At non-periodic domain boundaries, the central-differences are replaced by one-sided derivatives. There are two 4th order boundary schemes available in OpenSBLI, to maintain a consistent order throughout the domain. The first is the scheme of [47], which uses a [-5,5] stencil at domain boundaries. The second scheme [35] modifies two points at each boundary with a [-4,4] stencil such that

$$f_0' = \frac{1}{12\Delta x}(-25f_0 + 48f_1 - 36f_2 + 16f_3 - 3f_4), \tag{30}$$

$$f_1' = \frac{1}{12\Delta x}(-3f_0 - 10f_1 + 18f_2 - 6f_3 + f_4), \tag{31}$$

$$f_{N-2}' = \frac{1}{12\Delta x}(-f_{N-5} + 6f_{N-4} - 18f_{N-3} + 10f_{N-2} + 3f_{N-1}), \tag{32}$$

$$f_{N-1}' = \frac{1}{12\Delta x}(3f_{N-5} - 16f_{N-4} + 36f_{N-3} - 48f_{N-2} + 25f_{N-1}). \tag{33}$$

For both boundary schemes the second derivatives are computed for the final two interior points from [47] using the formula

$$f_0'' = \frac{35f_0 - 104f_1 + 114f_2 - 56f_3 + 11f_4}{12\Delta x^2}, \tag{34}$$

$$f_1'' = \frac{11f_0 - 20f_1 + 6f_2 + 4f_3 - f_4}{12\Delta x^2}. \tag{35}$$

OpenSBLI also contains a metric transformation class to symbolically transform derivatives to a set of curvilinear coordinates $\xi(x, y, z)$, $\eta(x, y, z)$, and $\zeta(x, y, z)$. These are used for simulations containing stretched and curved meshes. The metric terms are evaluated by the same central interior and boundary schemes described in this section. Metric terms are computed once at the start of the simulation, to be multiplied into derivative terms to perform the coordinate transformation. Further discussion of the metric transformation in OpenSBLI is given in [10].

2.7. Explicit Runge-Kutta time-stepping

Large-scale DNS of the compressible Navier-Stokes equations has considerable memory requirements, making low-storage time-advancement schemes an attractive option to tackle challenging flows [48]. Furthermore, explicit methods avoid having to compute the expensive inversion of systems required by implicit schemes. Explicit methods with structured meshes are well suited to modern computational hardware options such as GPUs, as they avoid performance issues related to poor data locality. OpenSBLI currently has two low-storage time-stepping schemes available: a standard 3rd order Runge-Kutta scheme in the form proposed by [49], and one following the work of [50]. The second formulation is used throughout this work, and has been implemented in OpenSBLI for 3rd and 4th order, plus a 3rd order strong-stability-preserving (SSP) version to improve stability for flows containing discontinuities. Low-storage Runge-Kutta schemes require only two additional storage arrays per equation. For an m-stage scheme, time advancement of the solution vector U from time level U^n to U^{n+1} is performed at stage $i = 1, \ldots, m$ such that

$$dU^{(i)} = A_i dU^{(i-1)} + \Delta t R\left(U^{(i-1)}\right), \tag{36}$$

$$U^{(i)} = U^{(i-1)} + B_i dU^{(i)}, \tag{37}$$

$$U^{n+1} = U^{(m)}, \tag{38}$$

for a constant time-step Δt, initial conditions $U^{(0)} = U^n$ and $dU^{(0)} = 0$, and residual $R(U^{(i-1)})$. The 3rd and 4th order schemes have three and five sub-stages per iteration respectively. The coefficients A_i and B_i are taken from [26] for 3rd and 4th order, and [51] for the SSP version of the 3rd order scheme.

3. OpenSBLI code implementation

This section outlines the main design principles of the OpenSBLI code-generation system, with examples from the high-level Python user interface. The purpose of OpenSBLI is to generate a complete, customizable, CFD solver in the OPS domain specific language [11–14]. OPS is a programming abstraction for massively-parallel computation on structured multi-block meshes. The OpenSBLI/OPS workflow is summarised in Fig. 3. In step 1, OpenSBLI is used to generate an OPS C code for a given physical problem. The OPS translator is then used to perform a source-to-source translation of the base code to a number of parallel programming paradigms. The parallel code can then be compiled and executed on a target architecture. This allows the code to be executed on a range of different hardware from a single source code.

It is important to note that a code-generation system is not a prerequisite to write OPS C code. As we will see in section 3.3 however, the standardised parallel templates of OPS code lend themselves well to a code-generation approach. Code-generation also enables far greater control of the resulting simulation code, leading to performance optimisations [52] that would be difficult in hand-written code. In addition, code-generation can improve code maintainability, by utilising an object-oriented design to share common functionality between classes. This approach allows for a large number of scheme options to be implemented in a relatively compact code base.

The simulation code is tailored to the options selected by the user in the Python interface. Python was selected because it is an extremely versatile general-purpose programming language, with a rich ecosystem of external scientific libraries. One such example is the symbolic algebra library SymPy [6], which provides the fundamental building blocks of OpenSBLI. OpenSBLI has a number of symbolic data structures that inherit the functionality provided by SymPy. This functionality enables us to define, manipulate, and simplify symbolic expressions, using symbolic objects that follow the fundamental rules of mathematics.

OpenSBLI parses and expands user-defined equations, before performing a symbolic discretization procedure for the selected numerical schemes. There are classes for boundary conditions, initial conditions, and handling of simulation input/output, which are all controlled by the user in the high-level problem script. Much of the complexity of code-generation comes from the need to make the abstraction as general as possible, so that it is flexible enough to target a wide range of applications within the domain of fluid dynamics.

As the system has no inherent knowledge of the algorithms required by CFD, we have to apply sorting procedures to ensure dependencies are satisfied in the correct order in the generated simulation code. For example, the evaluation of speed of sound $a = \sqrt{\gamma p/\rho}$ requires the equation of state for pressure (7), p, to have already been calculated. Similarly, a constant declaration $\Delta x = L_x/N_x$ must come after declaration of constants L_x, N_x. The sorting procedure ensures that evaluations are written to the simulation code in the correct order. Users are also able to add their own components to the algorithm from within the Python layer [10]. The next section discusses the core components that make up the system.

3.1. Creating an OpenSBLI problem script

OpenSBLI is comprised of a number of Python classes that provide the core functionality needed to generate a CFD solver. Fig. 4 illustrates these core components. A complete description of all of the source code files is contained in the user manual that accompanies this work. In the following sections, the core components of the system are described in the order that they would appear in a user-defined problem script.

3.1.1. Defining and expanding equations

```
1 ndim, scheme = 3, "**{\'scheme\':\'Teno\'}"
2 # Define the compresible Navier-Stokes equations in
      Einstein notation.
3 mass = "Eq(Der(rho,t), - Conservative(rhou_j,x_j,%s))" %
       scheme
4 momentum = "Eq(Der(rhou_i,t) , -Conservative(rhou_i*u_j
      + KD(_i,_j)*p,x_j , %s) + Der(tau_i_j,x_j) )" %
      scheme
5 energy = "Eq(Der(rhoE,t), - Conservative((p+rhoE)*u_j,
      x_j, %s) - Der(q_j,x_j) + Der(u_i*tau_i_j ,x_j) )" %
       scheme
6 stress_tensor = "Eq(tau_i_j, (mu/Re)*(Der(u_i,x_j)+ Der(
      u_j,x_i) - (2/3)* KD(_i,_j)* Der(u_k,x_k)))"
```

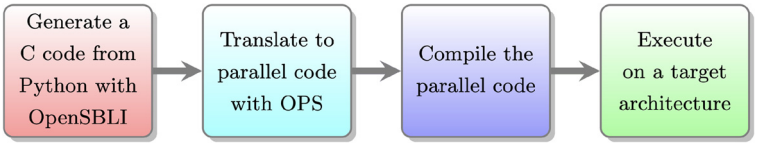

Fig. 3. The code-generation process in OpenSBLI/OPS.

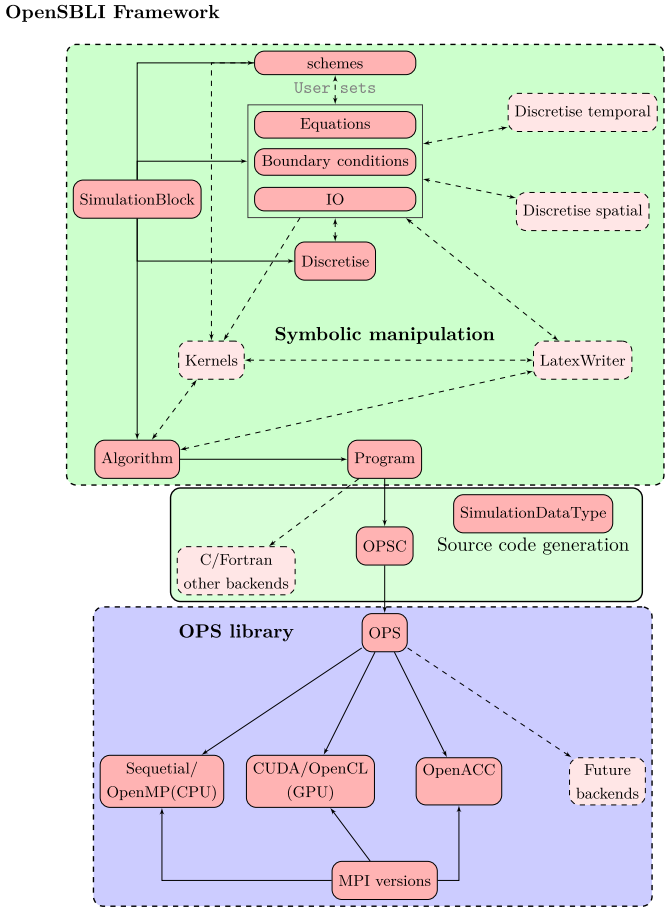

Fig. 4. Schematic of the major components in the OpenSBLI automatic source code generation framework.

```
7 heat_flux = "Eq(q_j, (-mu/((gama-1)*Minf*Minf*Pr*Re))*
      Der(T,x_j))"
8 substitutions = [stress_tensor, heat_flux]
9 constants = ["Re", "Pr", "gama", "Minf"]
```

The first step when creating a simulation in OpenSBLI is to specify the equations to be solved. The code listing shows how this is done for the compressible Navier-Stokes equations defined in section 2.1. Taking the example shown in line 3, the SymPy equation class `Eq` is used with the OpenSBLI derivative-handling classes `Der` and `Conservative`, to define the continuity equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}\left(\rho u_j\right) = 0. \tag{39}$$

Note that there is an optional argument 'scheme' passed to the spatial derivative, to specify that this derivative should be computed with the TENO scheme. Alternatively the user could have specified the use of a 'WENO' or 'Central' scheme here. If no scheme is passed, the derivative class defaults to a central derivative. Additionally, there is a `Skew` class available for skew-symmetric formulations of the central schemes. Lines 4-7 defines the momentum, energy, stress tensor, and heat flux equations in the same manner. In the first line the user selected the number of dimensions to be 3, which will be used when expanding these equations over the repeated index j. The symbol t is reserved to denote temporal derivatives. In line 8, the stress tensor and heat flux equations are stored in a list to be substituted into the momentum and energy equations. This optional substitution step is used to simplify the form of the equations. Note that if a term has more than one index, these should be separated by an underscore (e.g. τ_{ij} is written as `tau_i_j`). Finally, a list of simulation constants are defined in line 9, including Reynolds number, Prandtl number and the ratio of specific heat capacities.

```
1 # Expand the continuity equation
2 EE = EinsteinEquation()
3 continuity = EE.expand(mass, ndim, "x", substitutions,
      constants)
```

The next step is to instantiate the `EinsteinEquation` class, which contains the functionality for parsing and expanding of equations. The class contains an `expand` method, which expands the equation over the number of dimensions of the problem. A coordinate base symbol of x is given, along with any substitutions and constants as necessary. This parsing and expansion step is repeated for each of the governing equations.

```
1 # Define and expand the pressure and viscosity relations
2 pressure = "Eq(p, (gama-1)*(rhoE - rho*(1/2)*(KD(_i,_j)*
      u_i*u_j)))"
3 viscosity = "Eq(mu, T**(1.5)*(1+SuthT/RefT)/(T+SuthT/
      RefT))"
4 P = EE.expand(pressure, ndim, coordinate_symbol,
      substitutions, constants)
5 mu = EE.expand(viscosity, ndim, coordinate_symbol,
      substitutions, constants)
```

Next we have to define any relations that are required by the base equations, such as the dynamic viscosity relation (6), or the equation of state (7). In this example, Sutherland's law (6) and the pressure evaluation (7) are defined in the same manner as before, before being parsed and expanded by the `EinsteinEquation` class. A Kronecker Delta class `KD` is applied here for the pressure calculation, to sum over the square of the velocity components. OpenSBLI contains other objects providing index functionality, such as the Levi-Civita symbol.

```
1 # Store the expanded equations and relations
2 SE = SimulationEquations()
3 CR = ConstituentRelations()
4 SE.add_equations(continuity), CR.add_equations(P)
```

The governing equations and their relations have now been defined, parsed, and expanded. To distinguish between time-advanced equations and the supporting relations, we must now group them as either `SimulationEquations`, or `ConstituentRelations`. To do this, the two OpenSBLI classes are instantiated, and each of the previously expanded equations are added by the `add_equation` method. Here we are only adding one equation to each for brevity, but the method accepts lists of multiple equations at once.

3.1.2. Selection of numerical schemes

```
1 # Set the numerical schemes
```

```
2 schemes = {}
3 LLF = LLFTeno(order=6, averaging = RoeAverage([0, 1]))
4 cent = Central(4)
5 rk = RungeKuttaLS(3, formulation='SSP')
6 schemes[LLF.name], schemes[cent.name], schemes[rk.name]
      = LLF, cent, rk
```

Having defined all of the base equations, the user must now select the numerical schemes they wish to use. A standard Python dictionary is created in line 2 to hold the schemes. Lines 3-5 instantiate the 6th order LLF TENO scheme for shock-capturing with Roe-averaging, the 4th order Central scheme for viscous/heat-flux terms, and a 3rd order SSP Runge-Kutta scheme for time-advancement. The final step in line 6 is to store the schemes in the Python dictionary in the key-value syntax, where the name of the scheme is used as the key. At this point no symbolic discretization of the equations has been performed, the schemes have only been initialized.

3.1.3. Setting boundary and initial conditions

```
1 # Setting an initial condition
2 initial = GridBasedInitialisation()
3 initial.add_equations(initial_equations)
4 # Selecting a boundary condition
5 boundaries = [[0, 0] for t in range(ndim)]
6 direction, side = 1, 0
7 boundaries[direction][side] = IsothermalWallBC(direction
      , side, wall_condition)
```

To generate a complete CFD solver, we must also specify the initial and boundary conditions for the problem. The simplified code extract shows how this is done. Line 2 instantiates the `GridBasedInitialisation` class, which will be executed once at the start of the simulation ($t = 0$). The `initial_equations` variable would be a list of conditions written in the same manner as the equations in section 3.1.1. The `add_equations` method is then used in line 3 to add this set of initial conditions to the class.

Line 5 creates a list of lists to store the boundary condition classes for the problem. For a problem of dimension `ndim`, there are `ndim*2` boundary conditions that must be set. These are distinguished by having a side of 0 or 1. In the example, an isothermal wall boundary condition has been selected for the 0th (bottom) side in the 1st (y) direction of the problem. Boundary conditions are enforced on conservative variables by using the ghost (halo) points required by the shock-capturing schemes. In the case of the isothermal wall boundary shown here, the no-slip condition is first enforced on the boundary plane as

$$\rho u = \rho v = \rho w = 0. \tag{40}$$

The `wall_condition` variable is an expression for the total energy ρE, set depending on the thermal properties of the wall in question. For a constant temperature T_w isothermal wall the energy is set for a wall density ρ_w as

$$\rho E = \frac{\rho_\mathrm{w} T_\mathrm{w}}{M_\mathrm{ref}^2 \gamma\,(\gamma - 1)}, \tag{41}$$

where the wall density is obtained from solving the continuity equation (1). The ghost flow in the halo points ($i < 0$) is enforced by extrapolating temperature from the interior and wall as

$$T_{-i} = (i+1)T_\mathrm{w} - iT_1. \tag{42}$$

Density in the halos is evaluated using the extrapolated temperature values and the wall pressure p_w as

$$\rho_{-1} = \frac{M_\mathrm{ref}^2 \gamma\, p_\mathrm{w}}{T_{-i}}. \tag{43}$$

Momentum components in the halos are set by reflecting the velocity components from the interior flow with a reversed sign such that

$$\rho u_{-i} = -\rho_{-i} u_i, \tag{44}$$

$$\rho v_{-i} = -\rho_{-i} v_i, \tag{45}$$

$$\rho w_{-i} = -\rho_{-i} w_i, \tag{46}$$

where ρ_{-i} is the halo density calculated in equation (43). In the case of an adiabatic wall, the formula for a one-sided 4th order approximation of $\partial T/\partial y = 0$ is rearranged using the interior points to calculate the unknown wall temperature T_w, to enforce the zero heat-flux condition. This wall temperature is then used to set the wall energy as in equation (41). In both cases the value of the wall density is left to float by letting the scheme solve the continuity equation, which avoids over-specifying the boundary condition.

In addition to boundary conditions implemented for Navier-Stokes applications, OpenSBLI also offers a `DirichletBC` option which allows users to enforce their own custom conditions on a given boundary plane. The user specifies the equations to evaluate in the same manner as the equation definitions shown in section 3.1.1. These conditions can contain time-dependence and branching conditional expressions, which can be used to add variable forcing, or to enforce multiple spatially-dependent conditions on a single boundary plane. Alternatively, source terms localised to a boundary region can be added to the governing equations at the Python level. Further examples and a full listing of the available boundary conditions is given in [10].

3.1.4. Selecting HDF5 file I/O options

```
1 kwargs = {'iotype': "Write"}
2 h5 = iohdf5(save_every=10000, **kwargs)
3 h5.add_arrays(SE.time_advance_arrays)
4 h5.add_arrays([DataObject('T')])
```

All input/output of simulation data in OpenSBLI is handled by the parallel Hierarchical Data Format (HDF5) library [53]. HDF5 enables large data files to be stored and organised into groups containing multiple named datasets. In lines 1-2 of the example code, the OpenSBLI `iohdf5` class is instantiated with the write (output) argument. This class will control all of the output writing to disk. An optional argument `save_every` is specified, to write intermediate simulation data, for example every 10,000 iterations. The `iohdf5` class has an `add_arrays` method, which can be used to set which flow variables to write to disk. In this example the time-advance arrays (ρ, ρu, ρv, ρw, ρE), and temperature T, have been selected for the output. Restarting of simulations or the reading of a coordinate mesh is also handled by this class, with the `iotype` 'Read'.

3.1.5. Creating a block and generating the C code

```
1 # Create a simulation block
2 block = SimulationBlock(ndim, block_number = 0)
3 # Set the user options on the block
4 block.set_discretisation_schemes(schemes)
5 block.set_block_boundaries(boundaries)
6 block.setio(h5)
7 block.set_equations([CR, SE, initial])
8 # Begin the symbolic discretisation process
9 block.discretise()
10 # Create an algorithm to order the computations
11 alg = TraditionalAlgorithmRK(block)
12 SimulationDataType.set_datatype(Double)
13 # Generate the OPS C code and write it to file
14 OPSC(alg)
```

Up until this point in the script, no symbolic discretization has been performed. The user has simply been selecting the options

they require for the simulation. The core component that links everything together is called the `SimulationBlock`. A `SimulationBlock` is created in line 2, with the number of dimensions of the problem. We then set the numerical schemes, boundary conditions, HDF5 I/O, and all of the equations on the block. To begin the symbolic discretisation, we call the `block.discretise()` method. This method loops over all of the equations to be solved, and calls discretisation routines in each of the numerical methods where applicable. The continuous derivatives in the equations are converted into discrete representations, constructed from the symbolic objects that will be discussed in section 3.2.1. These discrete equations are stored in computational `Kernels`, that are discussed in section 3.2.2.

Line 11 initialises an algorithm class based on the block. The algorithm class sets out the order that individual components should appear in the simulation code. For example, precedence is given to components involved in initialising the simulation, such as the declaration of global constants and memory allocation for storage arrays. Next would be the initial conditions to set from the `GridBasedInitialisation` class, before declaration of the main iteration and Runge-Kutta sub-stage loops and their components. At the end of the algorithm would be the calls to the HDF5 library to write the simulation output to disk.

The final stages in lines 12 and 14 are to set the numerical precision of the simulation and begin the code-writing process with the `OPSC` class. The `OPSC` class is derived from the `C99CodeWriter` contained in SymPy [6]. It contains methods that return C-compliant expressions for the discrete symbolic equations stored in the computational `Kernels`. Additionally, the `OPSC` class contains templates of calls to the OPS library functions, which are populated based on the computations stored in the `Kernels`. Examples of these OPS library functions in C are given in section 3.3. Users would also input numerical values for simulation constants at this stage. Once the `OPSC` procedure is complete, a set of C/C++ codes are written out for translation and compilation with OPS as in Fig. 3. In practice, execution of the entire code-generation process takes less than a minute for the most demanding cases. The `count_ops` function in SymPy can be used to monitor operation counts for each symbolic expression. This feature can aid optimisation efforts by applying simplifications and checking the updated operation count [31]. The 3D research-representative case in section 5 has 1.5×10^4 internal representations for the discrete form of the equations.

3.2. Key OpenSBLI features and data structures

In this section we briefly discuss the OpenSBLI data structures that are used internally during the code-generation process. A more complete guide is given in the accompanying user manual and in [10]. The final part of this section highlights some of the optimisations that are possible in the symbolic engine.

3.2.1. Data structures

Equations in OpenSBLI are constructed using the `Eq` class and the data structures outlined in this section. Equations can be defined either as strings to parse (as shown in section 3.1.1), or by manually calling the following classes.

1. `DataObject`: Variables to be stored as global arrays on the entire grid. These objects are not indexed, they represent terms in the continuous symbolic equations.
2. `DataSet (e.g. T[0,0,0])`: SimulationBlock-specific versions of `DataObjects`. These objects have 'ndim' indices, which are incremented per direction to create discrete finite-difference representations.
3. `GridVariable`: Temporary local variables that are used for intermediate calculations inside computational kernels. They contain a single value at a given grid point, to perform intermediate calculations.
4. `CoordinateObject`: Direction based coordinate objects. Used for defining derivatives and metric handling of the continuous equations.
5. `ConstantObject`: Used to define constant parameters. Examples would include Reynolds number, Prandtl number, and reference quantities.

3.2.2. OpenSBLI kernels

```
1 # Create an OpenSBLI Kernel
2 kernel = Kernel(block, computation_name="%s boundary
      direction%d side%d" % (bc_name, direction, side))
3 kernel = self.set_kernel_range(kernel, block)
4 kernel.add_equation(BC_equations)
```

A key concept in OpenSBLI is that of the `Kernel`. The `Kernel` class is a way of holding the information required to perform a certain computation over a specified grid range. They are converted during the code-writing process into the OPS kernels discussed in section 3.3. Kernels are created internally as shown in the example code, where a `Kernel` object is created in line 2 for a named boundary condition. In line 3 the spatial grid range for this computation is set using grid information stored in the `SimulationBlock`. The equations to evaluate over this grid range can then be added with the `add_equation` method.

Each instance of the `Kernel` class extracts and stores information from its equations. This includes determining which of the terms in the equations require read, write, or read-write data access patterns. It must also inspect the relative data access of each quantity, to build up a set of numerical finite-difference stencils. For example, a `Kernel` for a 4th order central difference would detect that it requires read access at $\{-2, -1, 1, 2\}$ in a certain direction. The `Kernel` would also contain a work array with write access at $\{0\}$, to store the value of the derivative at each grid location. The `Kernel` performs all of the 'bookkeeping' work, that enables us to automatically generate the parallel OPS loops and C functions described in section 3.3. All of this information would otherwise have to be maintained and updated manually for each computation in a static hand-written code, which can be time consuming and error prone.

3.3. Oxford Parallel Structured (OPS) library

As previously mentioned, automated parallel abstractions can offer researchers an easy way to utilise modern hardware. In addition to x86-CPU and GPU-based architectures, ARM-based CPUs [54], Field Programmable Gate Arrays (FPGAs), and many-core accelerator cards such as the Intel Xeon Phi have all been cited as possible architectures of the future [55]. As the recent discontinuation of the Intel Xeon Phi product line [56] has highlighted however, it is not clear at this stage which, if any, of these architectures will ultimately prevail in the coming decades. As such, codes using cross-platform standardised libraries agnostic to the computational architecture, may be better placed to respond to future changes in the computational landscape.

This section provides brief examples of OPS code that was automatically generated by OpenSBLI. Full descriptions of the OPS abstraction have been reported in [11,12]. Unlike some black-box solvers, the OPS C code can be modified directly by the user. As a result, modifications can be made at either the Python, or C level, depending on the situation. New features will often be tested first in C, before being incorporated into the Python code-generation framework.

```
1 // Example OPS parallel loop
2 int iteration_range_43_block0[] = {0, block0np0, 0,
      block0np1, 0, block0np2};
3 ops_par_loop(opensbliblock00Kernel043, "Convective CD
      p_B0 x1", opensbliblock00, 3,
      iteration_range_43_block0,
4 ops_arg_dat(p_B0, 1, stencil_0_02, "double", OPS_READ),
5 ops_arg_dat(wk20_B0, 1, stencil_0_00, "double", OPS_WRITE
      ));
```

The core component of OPS is the `ops_par_loop` shown in the code example. This function provides a template to create a parallel region in OPS, based on the minimum amount of required information. Referring once more to Fig. 3, these templates are parsed by the OPS translator before compilation. The OPS translator is a Python script that reads and translates the base code to a range of parallel programming languages. For each individual `Kernel`, the computation is wrapped into parallel versions for each of the computational back-ends. The user gives OPS full control of the data, to decompose and execute the data most efficiently for a given platform. The example shown here is a kernel to perform a central derivative of $\partial p/\partial y$ at 4th order.

In line 2 we specify the iteration range to be the entire (N_x, N_y, N_z) grid. Line 3 begins the call to the parallel loop with an input function `Kernel043`, that contains the calculations to perform. The next arguments are a name for the computation, an OPS block that holds grid information, the dimensions of the problem, and the iteration range. Lines 4 and 5 specify the input/output arrays which are of type '`ops_dat`'. We must provide the names, floating point precision, and access patterns for each of the arrays. Additionally, each array has a corresponding `stencil` argument, which is a list of integers for the relative stencil access. All of this information is generated automatically by the OpenSBLI `Kernel` class discussed in section 3.2.2.

```
1 // Example OPS kernel function
2 void opensbliblock00Kernel043(const double *p_B0, double
      *wk20_B0){
3 wk20_B0[OPS_ACC1(0,0,0)] = inv_1*(-rc7*p_B0[OPS_ACC0
      (0,2,0)] - rc8*p_B0[OPS_ACC0(0,-1,0)] + (rc8)*p_B0[
      OPS_ACC0(0,1,0)] + (rc7)*p_B0[OPS_ACC0(0,-2,0)]);}
```

The second example is the corresponding C function that was called within the `ops_par_loop`. The first thing to note here is the relative indexing for the computation. The code loops over the entire grid range in parallel, where the $(0, 0, 0)$ location refers to the current grid point. The pressure array '*p_B0*' is being indexed in four points in the y direction, to build the finite-difference approximation. The result is stored to the generic work array '*wk20_B0*'. Work arrays are re-used where possible during the algorithm, to reduce the memory requirements of the code. The `OPS_ACCX` macros correspond to the ordering of the input arguments to the function. These are another feature of OPS that would have to be maintained manually without code-generation. The code-generator also extracts all rational constant factors (p/q for $p, q \in \mathbb{Z}$) found in the computational kernels, and defines them as global placeholder constants in the preamble of the simulation code. Having explained the design of OpenSBLI and the resulting OPS code, the next section shows a selection of test simulations.

4. Validation and verification

In this section, a selection of test cases are presented to demonstrate the capabilities of the OpenSBLI code for different flow configurations. The code repository contains other applications that are not discussed in detail here, as they have been covered in previous work. These applications include supersonic laminar SBLI [57], laminar duct SBLI with sidewalls [9], transitional duct SBLI with sidewalls [8], and hypersonic roughness-induced transition at Mach 6 [58]. The numerical schemes in OpenSBLI have also been assessed and validated for supersonic Taylor-Green vortex cases [41], and supersonic turbulent channel flows [59]. The code is in active use for research problems by the authors and the collaborators mentioned in the acknowledgements of this work.

A set of verification and validation cases are documented in this section for problems that test specific parts of the code implementation. The Sod shock-tube problem in section 4.1 shows the correctness of the shock-capturing and characteristic decomposition for a simple 1D case. The Shu-Osher shock-density wave interaction in section 4.2 highlights the benefits of high-order WENO shock-capturing schemes, in the context of resolving high-frequency waves. The order of convergence for the WENO-Z schemes is shown for a smooth propagating density wave in section 4.3. This case is performed on a sinusoidal mesh to verify the curvilinear coordinate transformation against an exact solution. Section 4.4 verifies the central and one-sided boundary schemes against an analytic solution for a 2D compressible laminar channel flow, for which a channel with one isothermal wall and one adiabatic wall are selected to concisely verify two no-slip wall conditions in the same problem. The 2D viscous shock-tube in section 4.5 highlights the ability of the code to capture a propagating normal shockwave, with subsequent vortex roll-ups and unsteady flow separation. Comparison is made to a reference solution at $Re = 200$. The third order explicit Runge-Kutta time-stepping scheme is used for all test cases. The following section 5 demonstrates a larger 3D SBLI DNS that contains acoustic-source body forcing, shock-reflection, flow-separation, and shock-induced transition to turbulence.

4.1. Sod shock-tube

The Sod shock-tube [60] is a classic test of shock-capturing ability for an ideal gas in one dimension. The test case consists of a Riemann problem specified by the initial left and right states separated at $x = 0.5$ by

$$(\rho, u, p) = \begin{cases} (1.0, 0.0, 1.0) & \text{if } x < 0.5 \\ (0.125, 0.0, 0.1) & \text{otherwise.} \end{cases}$$

The test acts as validation of the characteristic decomposition implementation in OpenSBLI. The initial state is advanced to a non-dimensional time of $t = 0.2$, with constant time-step $\Delta t = 1 \times 10^{-4}$. The TENO-5 scheme is selected with $N = 200$ uniformly-spaced grid points. Fig. 5 shows good agreement to the exact solution. There is an expected smearing of the steep discontinuities by the flux-splitting method at this resolution, similar to previous studies performed with these schemes [18].

4.2. Shu-Osher shock-density wave interaction

The Shu-Osher problem is a one-dimensional inviscid test case involving the interaction of a Mach 3 shock with a smooth density wave. Three different orders of the WENO-Z scheme are used in this section to highlight the benefits of higher-order shock-capturing schemes. The simulation is initialized with the discontinuous conditions given in [61,62], such that

$$(\rho, u, p) = \begin{cases} (3.857143, 2.629369, 10.33333) & \text{if } x < -4 \\ (1 + 0.2\sin(5x), 1.0, 0.0) & \text{if } x \geq -4 \end{cases}$$

with Dirichlet conditions enforced at the domain boundaries $x = [-5, 5]$. The simulation is advanced to a non-dimensional time of $t = 1.8$, with a time-step of $\Delta t = 2 \times 10^{-4}$. A reference solution was computed with a WENO-7Z scheme and a fine mesh of

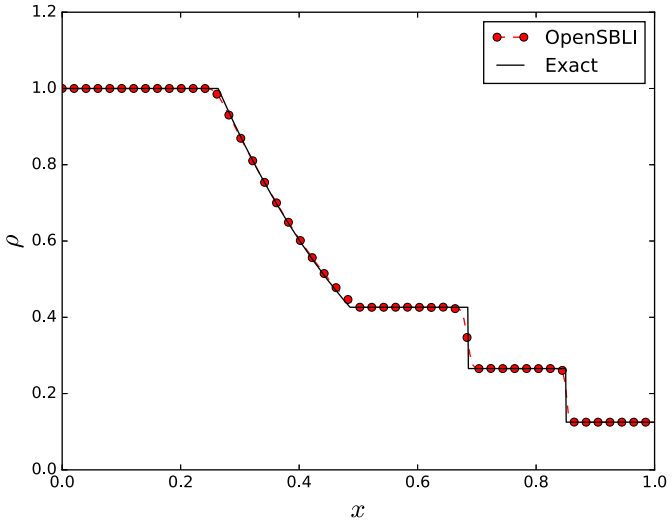

Fig. 5. Density profile for the Sod shock-tube case on $N = 200$ grid points. Comparison is made to an exact solution, plotting every other point.

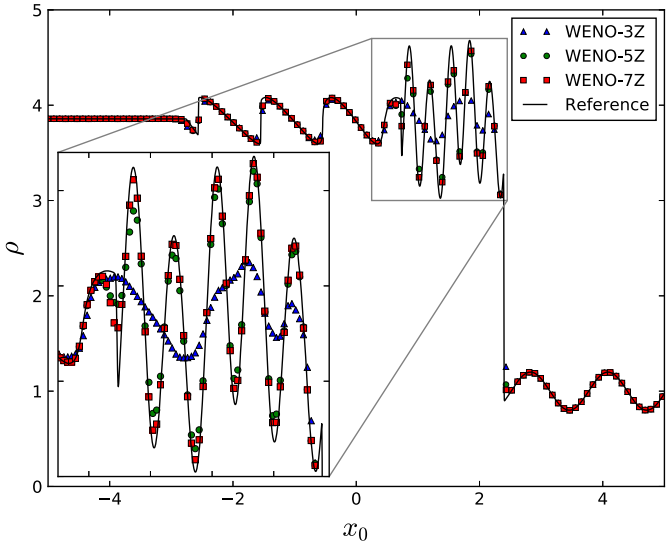

Fig. 6. Shu-Osher density distribution at a non-dimensional simulation time of $t = 1.8$, for WENO orders: 3Z (blue), 5Z (green) and 7Z (red). Reference fine mesh solution (continuous, black). Every third data point is displayed, with consecutive data shown in the 2x zoom inset. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

$N = 3200$ grid points. Fig. 6 shows a comparison of results for the WENO-3Z, WENO-5Z, and WENO-7Z schemes on an $N = 320$ grid.

The normal-shock propagates in the positive x direction from its start location at $x = -4$, interacting with the smooth imposed initial density perturbation. Shocklets are formed from the interaction in the region of $-2 < x < 1$, with a series of high-frequency waves present behind the normal-shock at $0 < x < 2$. The qualitative features of the solution are consistent with previous work such as figure 4 of [62]. While all three of the WENO-Z schemes match the reference solution for the shocklets and smooth regions of the flow, the lowest order WENO-3Z scheme struggles to resolve the high-frequency waves behind the main shock. The zoomed-inset highlights the benefits of using higher-order schemes.

4.3. Curvilinear Euler 2D-wave propagation

WENO and TENO schemes are adaptive in the sense that the number of candidate stencils (Fig. 1) used for the flux reconstruction will vary depending on the local flow conditions. The schemes will reduce to lower order approximations around shocks, and maintain the full numerical stencil in smooth regions of the flow. To be able to test the formal order of convergence of the schemes, it is therefore necessary to apply them to a smooth flow. The selected case is a smooth travelling 2D density perturbation for the Euler equations. The initial perturbation takes the form

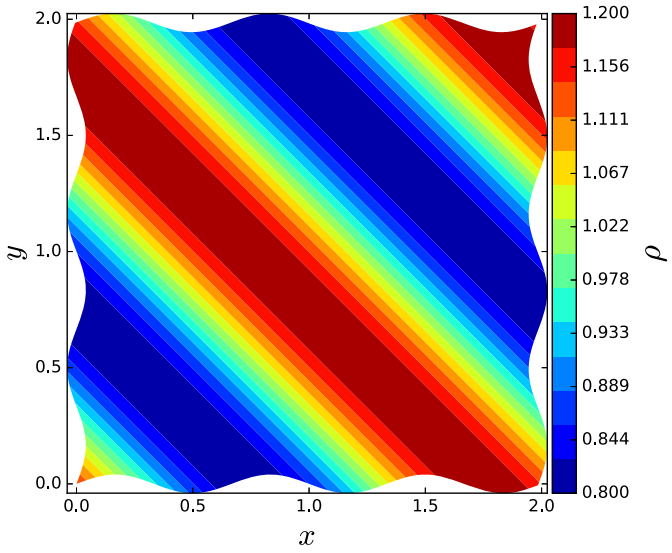

Fig. 7. Travelling 2D density wave, for the Euler equations simulated on a sinusoidal curvilinear mesh (48).

$$\rho(x, y, t) = 1 + 0.2 \sin\left(\pi\left(x + y - t\left(u + v\right)\right)\right), \tag{47}$$

with constant velocity components $u = 1.0$, $v = -0.5$, and a pressure of $p = 1.0$. Periodic boundaries are applied on all sides of the domain, with a time-step of $\Delta t = 5 \times 10^{-4}$. To verify the implementation of the Euler equations and characteristic decomposition on curvilinear meshes, a comparison is made to the analytical solution after a non-dimensional time of $t = 2.5$. A sinusoidal grid is generated such that

$$x_{i,j} = i\Delta x + A \sin\left(\frac{6\pi\, j \Delta y}{L_y}\right), \tag{48}$$

$$y_{i,j} = j\Delta y + A \sin\left(\frac{6\pi\, i \Delta x}{L_x}\right), \tag{49}$$

for $A = 0.04$, and $L_x = 2/L_x$, $L_y = 2/N_y$. The solution is shown on the curvilinear mesh in Fig. 7.

At each grid resolution, the L^1 and L^∞ error norms are computed as

$$L^1 = \frac{\sum_{i,j} \left|\rho_{\text{exact}} - \rho_{i,j}\right|}{N_x N_y}, \tag{50}$$

$$L^\infty = \left|\rho_{\text{exact}} - \rho_{i,j}\right|_{\max}. \tag{51}$$

Fig. 8 shows the rate of convergence for WENO-Z schemes of 5th and 7th order on $N = \left[32^2, 64^2, 128^2\right]$ grids. In both cases the schemes converge towards the exact solution with grid refinement, verifying the implementation of the curvilinear coordinate transformation. Both the L^1 and L^∞ convergence rates are within 10% of the formal order of accuracy for these shock-capturing schemes.

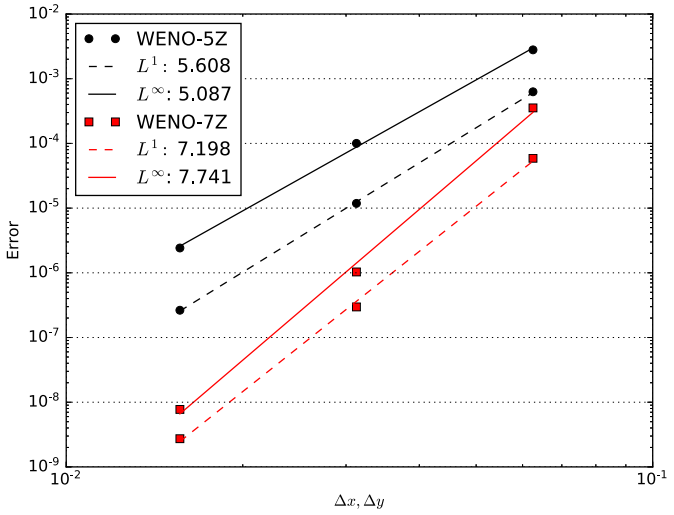

Fig. 8. Convergence of the L^1 and L^∞ error norms for WENO-Z schemes on a curvilinear mesh. The grid sizes are $N = [32^2, 64^2, 128^2]$.

4.4. Laminar compressible channel with mixed adiabatic/isothermal wall boundaries

To verify the implementation of the isothermal and adiabatic no-slip wall boundary conditions, a compressible laminar channel flow is compared to an analytical solution. The channel is comprised of a constant temperature wall at $y = -1$, and a zero heat-flux wall at $y = 1$. For constant viscosity, the streamwise velocity and temperature profiles for the analytic solution are given by

$$u = \frac{Re(1 - y^2)}{2}, \tag{52}$$

$$T = 1 - \frac{Re^2 M_{\text{ref}}^2 Pr(\gamma - 1)(y^4 - 4y - 5)}{12}. \tag{53}$$

Simulation parameters are taken to be $Re = 90$, $Pr = 0.72$, and $M_{\text{ref}} = 0.01$. The isothermal wall temperature is set to 1. At these physical values, the analytical solution predicts an adiabatic wall temperature at $y = 1$ of $T_{aw} = 1.155520$ (7 s.f.). The simulation is advanced until a non-dimensional time of $t = 1000$, with a time-step of $\Delta t = 1 \times 10^{-4}$. This test case was run as a 2D problem in a domain with size $L_x = 2\pi$, $L_y = 2$, with a uniform grid distribution of $(N_x, N_y) = (32, 64)$.

Fig. 9 shows the results for both a 4th order central scheme and a 6th order TENO scheme. Excellent agreement is observed to the analytical result for both the streamwise velocity and temperature profiles. The correct near-wall behaviour is produced for both of the schemes. At this grid resolution, the central and TENO adiabatic wall temperatures are 1.155510 and 1.155508 respectively. This corresponds to relative percentage errors for T_{aw} of 0.8×10^{-3} % and 1.0×10^{-3} % for the central and TENO schemes respectively.

4.5. 2D viscous shock-tube

The viscous shock-tube is a demanding case for shock-capturing schemes and tests the ability of the code to simulate a propagating shockwave interacting with a no-slip wall. The problem combines complex shock structures, shock reflection from a solid wall, and regions of unsteady boundary-layer separation. An in-depth study of the inviscid and viscous 2D shock-tubes was given by [64], from which the flow conditions in this section are taken. The computational domain $(x, y) = ([0, 1], [0, 0.5])$ is partitioned by an initial

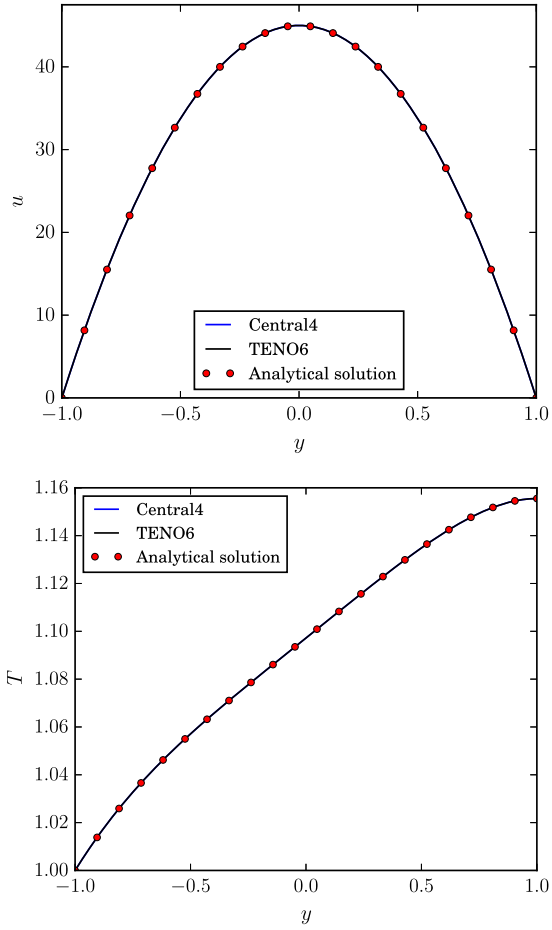

Fig. 9. Comparison of the 2D mixed-wall condition laminar channel flow to the analytical solution in equation (52). Showing profiles for the (top) streamwise velocity and (bottom) temperature.

diaphragm located at $x = 0.5$. A discontinuous initial profile is imposed which generates a normal shock that propagates in the positive x direction. The initial states to the left and right of the diaphragm are given by

$$(\rho, u, v, p) = \begin{cases} (120, 0, 0, 120/\gamma) & \text{if } x < 0.5 \\ (1.2, 0, 0, 1.2/\gamma) & \text{if } x \geq 0.5. \end{cases}$$

The reference Mach number is set to $M_{\text{ref}} = 1$, with Reynolds number $Re = 200$, and a Prandtl number of $Pr = 0.73$. The viscosity is assumed to be constant. A symmetry condition is enforced on the upper boundary of the domain, with adiabatic no-slip viscous wall conditions set on all other boundaries. The simulation is advanced with time-step of $\Delta t = 1 \times 10^{-5}$ until a non-dimensional time of $t = 1$. The grid consists of a uniform distribution of $[x, y] = [1500, 750]$ points, following the resolution of [63].

Fig. 10 shows the instantaneous density contours at $t = 1$. We observe that the shock has propagated to the $x = 1$ adiabatic wall, and reflected back to a position of $x = 0.58$. The relative motion of the shock generates a thin boundary-layer which separates after the shock reflects from the side wall. Above the separation bubble a lambda shock structure can be seen to form. Good agreement is found to the results of [63] (figure 3), with the lambda-shock triple point located at (x, y) $(0.58, 0.13)$. Fig. 11 shows a comparison of the bottom wall density profile compared to the reference data of [63]. Good agreement is observed relative to the reference solution, aside from a small overshoot in the density peak located at $x = 0.85$.

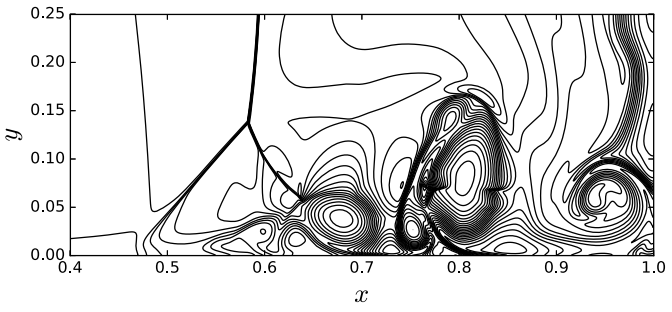

Fig. 10. Instantaneous density contours for the viscous shock-tube problem at $Re = 200$, on a $(N_x, N_y) = (1500, 750)$ mesh at a time of $t = 1$.

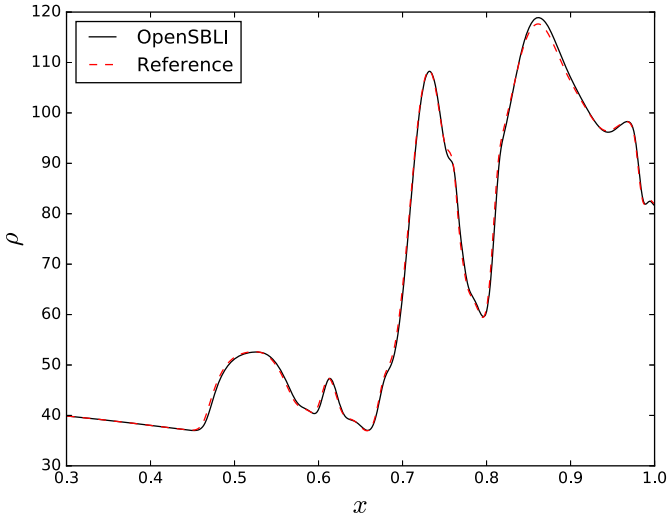

Fig. 11. Wall density for the viscous shock-tube problem at $Re = 200$, on a $(N_x, N_y) = (1500, 750)$ mesh. Compared to a reference solution [63] at a time of $t = 1$.

5. 3D transitional shockwave/boundary-layer interaction

As a demonstration of a complex large-scale computation, a DNS of a Mach 1.5 transitional SBLI is performed. The case consists of a modified version of the simulations presented in [65]. Compared to the original case, the domain is elongated in the streamwise direction to observe more of the breakdown process, and a different off-wall forcing method is used to seed the instability.

A Mach 1.5 freestream is initialized throughout the domain, with the similarity solution for a laminar boundary-layer [66] in the near-wall region. A Reynolds number based on an inlet boundary-layer displacement thickness of $Re_{\delta^*} = 750$ is used, with a reference temperature of $T_\infty = 202.17K$. Details of the laminar boundary-layer similarity solution are given in [57,9]. The similarity solution generates profiles for temperature, streamwise velocity, and wall-normal velocity, which are then used to set the conservative variables throughout. The domain has dimensions of $L_x = 375$, $L_y = 140$, and $L_z = 27.32$. Rankine–Hugoniot shock-jump conditions are enforced on the upper Dirichlet boundary at $x = 20$, corresponding to a flow deflection of 2.5° and pressure ratio $p_2/p_1 = 1.132$. Extrapolation methods are used at the inlet (for pressure) and outlet, while the span is periodic. Isothermal no-slip conditions are set on the bottom wall using the non-dimensional wall temperature of $T_w = 1.381$ (4 s.f.) from the similarity solution.

For low-supersonic boundary-layers, it has been shown that the dominant transition mechanism is the breakdown of oblique first-mode waves [67,68]. To introduce upstream disturbances to the otherwise laminar SBLI, modal time-dependent forcing is applied as an acoustic body-forcing term in the continuity equation (1). The forcing takes the form

$$\rho' = A \exp\left(-\left((x - x_F)^2 + (y - y_F)^2\right)\right) \cos(\beta z) \sin(\omega t), \quad (54)$$

for an amplitude $A = 2.5 \times 10^{-3}$, frequency $\omega = 0.1011$, and wavenumber $\beta = 0.23$. This single mode corresponds to the most unstable mode obtained from linear stability analysis of a laminar separation bubble [65]. The wavenumber $\beta = 2\pi/L_z$, corresponds to one wavelength across the width of the span. The acoustic source is located in the freestream above the boundary-layer at $x_F = 20$, $y_F = 4$. The simulation was performed with the 6th order TENO scheme, 4th order central-differencing for diffusive terms, and 3rd order Runge-Kutta time-stepping. The number of grid points is taken to be $(N_x = 2050, N_y = 325, N_z = 200)$. The grid is stretched in the streamwise and wall-normal directions such that

$$x_i = L_x \left(1 - \frac{\sinh\left(s_x \Delta_x \left(N_x - 1 - i\right)\right)/L_x}{\sinh\left(s_x\right)}\right), \quad (55)$$

$$y_j = L_y \frac{\sinh\left(s_y \Delta_y j/L_y\right)}{\sinh\left(s_y\right)}, \quad (56)$$

for grid indices i, j, and stretch factors of $s_x = 1.5$, and $s_y = 5$. In wall units this corresponds to $\Delta x^+ = 4.2$, $\Delta y^+ = 0.95$, and $\Delta z^+ = 4.5$, based on time-averaged skin-friction in the early turbulent region. A non-dimensional time-step of $\Delta t = 1 \times 10^{-2}$ was used. The simulation was initially advanced in time for 7 flow-through times of the domain to allow the SBLI to develop. Statistics were gathered every iteration for a further 60 periods of the forcing.

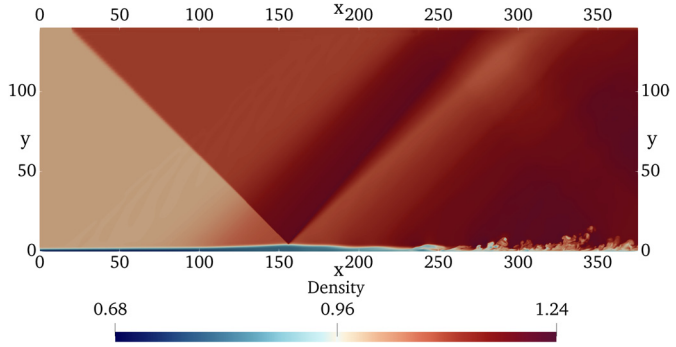

Fig. 12. An x-y slice of instantaneous density for the transitional shockwave/boundary-layer interaction.

Fig. 12 shows an (x-y) slice of instantaneous density contours, to highlight the main features of the shockwave/boundary-layer interaction. The initial oblique shockwave impinges on the flow developing over the wall, causing a thickening of the target boundary-layer. A series of compression waves are observed at $x = 100$, as the supersonic freestream adjusts to the curvature of the boundary-layer. A separation bubble is present beneath the foot of the shock reflection. This feature is clearer to see in the time-averaged skin-friction in Fig. 13 (a). Due to the adverse pressure gradient applied by the shockwave, the flow detaches from the wall at $x = 115.5$, and reattaches at $x = 192.4$. A transition to turbulence is observed in Fig. 12, downstream of the separation bubble. At $x = 200$ in Fig. 13 (a) there is a sharp increase in skin-friction, as the flow undergoes the early stages of transition. The skin-friction decreases close to the outlet, in the later stages of the turbulent breakdown. The exit skin-friction is approximately an order of magnitude higher than for a laminar version of the SBLI (e.g. figure 4.8 of [65]).

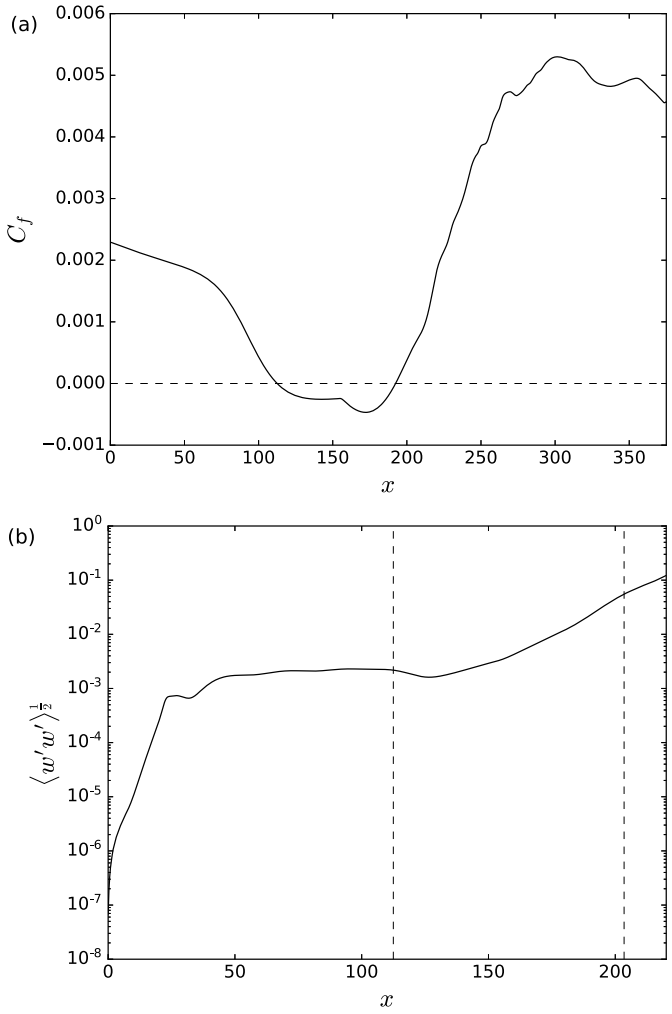

Fig. 13. (a) Streamwise distribution of the time-averaged skin-friction coefficient for the 3D SBLI case. (b) Amplification of the w velocity RMS of the disturbance along the line $y = 1$, $z = L_z/4$. The dashed lines indicate the start and end of the separation bubble.

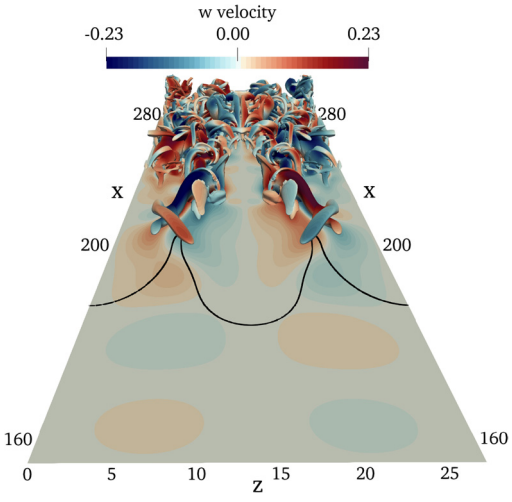

Fig. 14. Q-criterion of $Q = 5 \times 10^{-3}$ coloured by w velocity, showing the initial oblique mode breakdown of the transitional SBLI. A slice of w velocity is shown at $y = 0.5$, with the separation bubble ($u < 0$) in this plane outlined in black.

Fig. 13 (b) shows the RMS value of the w velocity component induced by the freestream acoustic forcing. The disturbance is evaluated within the boundary-layer, along the line $y = 1$, $z = L_z/4$. The two vertical dashed lines denote the start and end of the separation bubble on the line $z = L_z/4$. The SBLI amplifies the disturbance waves exponentially by two orders of magnitude, triggering a non-linear breakdown of the flow for $x > 210$. This feature is better visualized in the 3D-view of Fig. 14, where vortical structures are shown for a Q-criterion of $Q = 5 \times 10^{-3}$, coloured by the instantaneous spanwise w velocity. Beneath the Q surfaces is a slice of w velocity at $y = 0.5$, with the black line representing the $u = 0$ boundary of the separation bubble in this plane. A pair of symmetric streamwise vortices are present behind the reattachment line, which generate smaller vortex structures that remain symmetric for the range shown. Consistent with previous studies (e.g. figure 4.6 [65], figure 17 [68]), the dominant transition mechanism is observed to be an oblique-mode breakdown. This section has demonstrated that the code is capable of performing high-fidelity DNS of transitional SBLI problems, involving shock reflections, flow separation, and breakdown to turbulence.

6. Performance and scaling

In this section, the performance and scaling of the OpenSBLI/OPS code is demonstrated on multi-GPU clusters. In particular, we highlight scaling differences between two of the main schemes in the current version of the code. The performance of the OPS library has been documented extensively in previous studies, over a wide range of computational platforms. These include comparison of each of the computational back-ends to hand-written code [13], performance on ARM-based systems [54], and an in-depth comparison against a comparable structured-mesh DNS code [14]. The results in this section use a standalone OpenSBLI performance benchmark configured for scaling tests. For consistency with previous studies, the benchmark is configured to perform DNS of the 3D Taylor-Green Vortex (TGV) problem. The problem contains the transition and turbulent breakdown of an initial vortex condition on a triply periodic domain. A full description of the TGV case has been given in the context of OpenSBLI in [7,41,14]. Mudalige et al. [14] studied the performance of the Taylor-Green problem on multiple architectures using Intel's Advisor profiler. A considerable variation in the fraction of peak performance was observed, from 3% to 46% depending on the architecture and the coding strategy. The performance based on time-to-solution for different choices of storage and computational effort was analysed in [52], pointing more clearly to the value of low memory intensity algorithms that are further discussed here.

6.1. Low memory intensity algorithms

In recent decades, increases in available compute capability (FLOPS) have greatly outpaced increases in memory bandwidth [17]. This has led to many CFD codes being limited on performance by memory access. Minimising access to global memory by re-computing quantities locally on the fly is one way to alleviate this issue. Furthermore, high-order algorithms performing a large number of operations per-byte are well suited to modern hardware. For all of the simulations shown in this section, MPI+CUDA executables were generated using the OPS library discussed in section 3.3. Run-times were compared for 100 iterations of the main time loop in each case, omitting the input/output time of the simulation. Scaling results were obtained for Nvidia P100 GPU partitions on the CSD3 high-performance-computing cluster at the University of Cambridge. CUDA version 10.1 was used with -O3 optimization, and the Intel compiler (2017) and OpenMPI for inter-GPU and inter-node MPI communication.

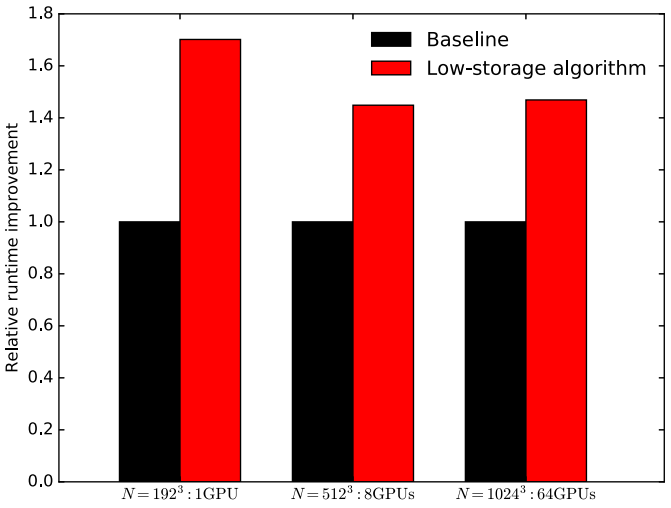

Fig. 15. Relative runtime speed-up factor of the low-memory intensity central-differencing algorithm. Simulations are performed on $N = [192^3, 512^3, 1024^3]$ grids on 1, 8, and 64 GPUs respectively.

Code-generation enables manipulation of the equations and solution algorithm that would be difficult to achieve in a hand-written code. To demonstrate the benefit of reduced memory access, we show the `StoreSome` optimisation [52] that is available in OpenSBLI for the central schemes. The standard practice in CFD is to store derivatives globally on the entire grid, to be re-accessed at later stages in the algorithm. The `StoreSome` algorithm in contrast, computes most of the derivatives locally in the kernel (section 3.2.2). Based on previous work [52], only the first derivatives of the velocity components (u, v, w) are stored in global arrays. Fig. 15 shows the runtime improvement of the `StoreSome` algorithm. Compared to the baseline version, the low-memory intensity algorithm is 1.7x faster on a single GPU. The improvement drops to 1.45x on multi-GPU configurations, as part of the total runtime is now taken up by inter-node MPI exchanges over the CPU hosts.

A secondary benefit of the algorithm that is especially pertinent on GPUs, is the reduced consumption of memory on the GPU devices. All of the storage arrays required by the simulation must be declared within the device memory on the GPU. The 16GB memory capacity of the Nvidia P100 GPUs is roughly an order of magnitude smaller than that found on a modern CPU node. This is a significant restriction in the context of large-scale DNS. The `StoreSome` algorithm reduces the total number of storage arrays from 65 to 32 for the TGV problem. This effectively doubles the maximum allowed grid size for a given number of GPUs.

6.2. Parallel GPU scaling

Once the memory capacity of a single GPU is exceeded, the problem must be split across multiple devices. The host CPU nodes are then responsible for performing MPI halo exchanges between the decomposed memory blocks. This must be implemented efficiently to achieve good performance scaling over a large number of devices. In this section the weak and strong scaling of OpenSBLI/OPS is demonstrated for the TGV problem with the 4th order central and 6th order TENO schemes.

Fig. 16 (a) shows the weak scaling of OpenSBLI on 4 to 64 GPUs. The base grid of $N = 512^3$ points on 4 GPUs is doubled with each doubling in GPU count. The central scheme achieves good weak scaling, with around 85% parallel efficiency at the largest grid size. There is a dip at the 16 GPU point, likely caused by the $(4 \times 2 \times 2)$ MPI decomposition proving suboptimal for the cubic grid distribution. The TENO6 scheme shows excellent weak scaling of 95%,

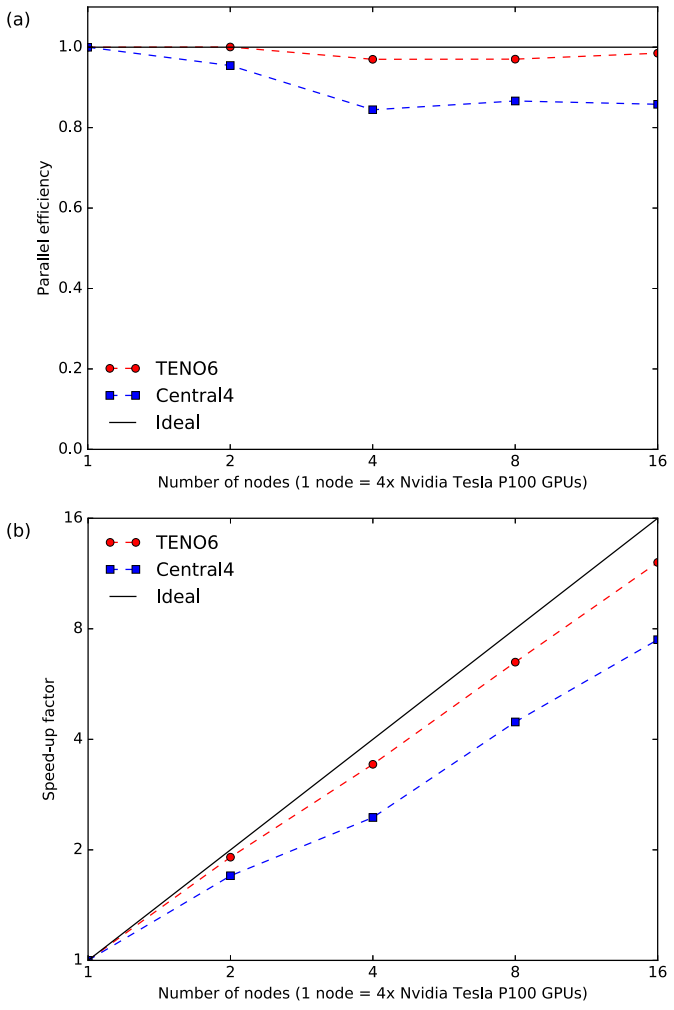

Fig. 16. OpenSBLI scaling results for the 4th order central and 6th order TENO schemes on up to 64 GPUs with MPI+CUDA. (a) Weak scaling from 4 to 64 GPUs (b) Strong scaling from 4 to 64 GPUs for a $N = 512^3$ grid.

due to the increased operation counts of shock-capturing schemes. Fig. 16 (b) shows the strong scaling of OpenSBLI on 4 to 64 GPUs. The runtime improvement is measured for increasing system resources, from an initial grid of $N = 512^3$ on 4 GPUs. The central scheme achieves a 50% speed-up on the largest GPU configuration, and shows a similar dip in scaling at 16 GPUs as observed in the weak scaling test. The 6th order TENO scheme scales well, reaching 75% of the perfect linear strong scaling.

7. Conclusions

This work has described the OpenSBLI code-generation system for compressible fluid dynamics on heterogeneous computing architectures. Based on an earlier proof of concept [7], the new code incorporates high-order shock-capturing schemes, curvilinear coordinate transformations, and a wide range of boundary conditions. The Python-based code-generation system generates code in the OPS domain specific language [12], enabling parallel execution on a wide range of computational hardware. The design and main components of the system have been discussed, with code examples demonstrating how the code is used in practice.

A suite of validation and verification cases has been presented, selected to demonstrate specific parts of the solver. A 3D DNS of a transitional shockwave/boundary-layer interaction was used to highlight the feasibility of code-generation for complex fluid flow problems. OpenSBLI was shown to exhibit good weak and strong

scaling on multiple GPUs, highlighting its suitability for large-scale DNS. Additionally, a low-memory intensity algorithm in the code demonstrated the performance benefit of reduced global memory access. Future work will use the multi-block capability of the OPS library to extend OpenSBLI for more complex domains with multiple connected grids, and to applications beyond the compressible Navier-Stokes equations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to acknowledge the contributions of Dr. Gihan Mudalige and Dr. Istvan Reguly to this project, for the development and continued support of the Oxford Parallel Structured (OPS) library used in this work. The authors would also like to thank the following people for their testing and valuable feedback on the OpenSBLI code: Julien Lefieux (ONERA), Alex Gillespie and Hiten Mulchandani (University of Southampton), Dr. Arash Hamzehloo (Imperial College London), Dr. Gary Coleman (NASA Langley), and Dr. Andrea Sansica (JAXA).

David J. Lusher was funded by an EPSRC Centre for Doctoral Training grant (EP/L015382/1). Compute resources used in this work were provided by the 'Cambridge Service for Data Driven Discovery' (CSD3) system operated by the University of Cambridge Research Computing Service (http://www.hpc.cam.ac.uk) funded by EPSRC Tier-2 capital grant EP/P020259/1, and the IRIDIS5 High Performance Computing Facility, and associated support services at the University of Southampton. The OpenSBLI code is available at https://github.com/opensbli. Data from this report will be available from the University of Southampton institutional repository. Declaration of Interests. The authors report no conflict of interest.

References

[1] P.R. Spalart, V. Venkatakrishnan, Aeronaut. J. 120 (1223) (2016) 209–232.
[2] P.M. Kogge, T.J. Dysart, in: SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–11.
[3] Nvidia GPU applications, Nvidia GPU applications catalog, https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/, 2019. (Accessed 30 November 2019).
[4] I. Ober, I. Ober, Proc. Comput. Sci. 108 (2017) 2298–2302.
[5] I.Z. Reguly, G.R. Mudalige, Comput. Fluids 199 (2020) 104425.
[6] A. Meurer, C.P. Smith, M. Paprocki, O. Čertík, S.B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J.K. Moore, S. Singh, T. Rathnayake, S. Vig, B.E. Granger, R.P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M.J. Curry, A.R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, PeerJ Comput. Sci. 3 (2017) e103.
[7] C.T. Jacobs, S.P. Jammy, N.D. Sandham, J. Comput. Sci. 18 (2017) 12–23.
[8] D.J. Lusher, N.D. Sandham, Flow Turbul. Combust. 105 (2020) 649–670, https://doi.org/10.1007/s10494-020-00134-0.
[9] D.J. Lusher, N.D. Sandham, J. Fluid Mech. 897 (2020) A18.
[10] D.J. Lusher, Shock-wave/boundary-layer interactions with sidewall effects in the OpenSBLI code-generation framework, Ph.D. thesis, University of Southampton, 2020.
[11] G.R. Mudalige, I.Z. Reguly, M.B. Giles, W. Gaudin, J.A. Herdman, A. Mallinson, High-level abstractions for performance, portability and continuity of scientific software on future computing systems - CloverLeaf 3D, 2015.
[12] I.Z. Reguly, G.R. Mudalige, M.B. Giles, D. Curran, S. McIntosh-Smith, in: The OPS Domain Specific Abstraction for Multi-Block Structured Grid Computations, WOLFHPC'14, IEEE Press, 2014, pp. 58–67.
[13] G.R. Mudalige, I.Z. Reguly, M.B. Giles, A.C. Mallinson, W.P. Gaudin, J.A. Herdman, Lect. Notes Comput. Sci. 8966 (2015) 85–104 (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).
[14] G.R. Mudalige, I.Z. Reguly, S.P. Jammy, C.T. Jacobs, M.B. Giles, N.D. Sandham, J. Parallel Distrib. Comput. 131 (2019) 130–146.
[15] I.Z. Reguly, G.R. Mudalige, C. Bertolli, M.B. Giles, A. Betts, P.H.J. Kelly, D. Radford, IEEE Trans. Parallel Distrib. Syst. 27 (5) (2016) 1265–1278.
[16] R.D. Sandberg, V. Michelassi, R. Pichler, L. Chen, R. Johnstone, J. Turbomach. 137 (5) (may 2015).
[17] F.D. Witherden, A.M. Farrington, P.E. Vincent, Comput. Phys. Commun. 185 (11) (2014) 3028–3040.
[18] Mario Di Renzo, Lin Fu, Javier Urzay, Comput. Phys. Commun. (2020) 107262.
[19] M. Bernardini, D. Modesti, F. Salvadore, S. Pirozzoli, Comput. Phys. Commun. 263 (2021) 107906, https://doi.org/10.1016/j.cpc.2021.107906.
[20] G.R. Mudalige, M.B. Giles, I.Z. Reguly, C. Bertolli, P.H.J. Kelly, in: 2012 Innovative Parallel Computing (InPar), 2012, pp. 1–12.
[21] 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, pp. 1–78.
[22] R. Borges, M. Carmona, B. Costa, W.S. Don, J. Comput. Phys. 227 (6) (2008) 3191–3211.
[23] L. Fu, X.Y. Hu, N.A. Adams, J. Comput. Phys. 305 (2016) 333–359.
[24] L. Fu, X.Y. Hu, N.A. Adams, J. Comput. Phys. 349 (2017) 97–121.
[25] F. Ducros, F. Laporte, T. Soulères, V. Guinot, P. Moinat, B. Caruelle, J. Comput. Phys. 161 (1) (2000) 114–139.
[26] M.H. Carpenter, C.A. Kennedy, Fourth-Order 2N-Storage Runge-Kutta Schemes, NASA Langley Research Center, Jun 1994.
[27] M.G. Hinchey, J.L. Rash, C.A. Rouff, Requirements to Design to Code: Towards a Fully Formal Approach to Automatic Code Generation, 2005.
[28] R. van Engelen, L. Wolters, G. Cats, IEEE Comput. Sci. Eng. 4 (3) (1997) 22–31, https://doi.org/10.1109/99.615428.
[29] I. Blundell, R. Brette, T.A. Cleland, T.G. Close, D. Coca, A.P. Davison, S. Diaz-Pier, C. Fernandez Musoles, P. Gleeson, D.F.M. Goodman, M. Hines, M.W. Hopkins, P. Kumbhar, D.R. Lester, B. Marin, A. Morrison, E. Müller, T. Nowotny, A. Peyser, D. Plotnikov, P. Richmond, A. Rowley, B. Rumpe, M. Stimberg, A.B. Stokes, A. Tomkins, G. Trensch, M. Woodman, J.M. Eppler, Front. Neuroinform. 12 (2018) 68, https://doi.org/10.3389/fninf.2018.00068.
[30] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P.A. Witte, F.J. Herrmann, P. Velesko, G.J. Gorman, Geosci. Model Dev. 12 (3) (2019) 1165–1187.
[31] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. Witte, P.H.J. Kelly, F.J. Herrmann, G.J. Gorman, Architecture and performance of devito, a system for automated stencil computation, CoRR, arXiv:1807.03032 [abs], jul 2018.
[32] V.H.M. Rodrigues, L. Cavalcante, M.B. Pereira, F. Luporini, I. Reguly, G. Gorman, S.X. de Souza, Gpu support for automatic generation of finite-differences stencil kernels, arXiv:1912.00695, 2019.
[33] M. Christen, O. Schenk, H. Burkhart, Comput. Sci. Res. Dev. 26 (3) (2011) 205–210, https://doi.org/10.1007/s00450-011-0160-6.
[34] D. Kempf, R. Heß, S. Müthing, P. Bastian, ACM Trans. Math. Softw. 47 (1) (2021) 1–31, https://doi.org/10.1145/3424144.
[35] J.A. Ekaterinaris, Prog. Aerosp. Sci. 41 (3) (2005) 192–300.
[36] S. Pirozzoli, Annu. Rev. Fluid Mech. 43 (1) (2011) 163–194.
[37] E. Johnsen, J. Larsson, A.V. Bhagatwala, W.H. Cabot, P. Moin, B.J. Olson, P.S. Rawat, S.K. Shankar, B. Sjögreen, H. Yee, X. Zhong, S.K. Lele, J. Comput. Phys. 229 (4) (2010) 1213–1237.
[38] A. Gross, H.F. Fasel, in: 54th AIAA Aerospace Sciences Meeting, AIAA SciTech Forum, American Institute of Aeronautics and Astronautics, 2016.
[39] C. Tenaud, E. Garnier, P. Sagaut, Int. J. Numer. Methods Fluids 33 (2) (2000) 249–278.
[40] C. Brehm, M.F. Barad, J.A. Housman, C.C. Kiris, Comput. Fluids 122 (2015) 184–208.
[41] D.J. Lusher, N.D. Sandham, AIAA J. 59 (2) (2021) 533–545, https://doi.org/10.2514/1.J059672.
[42] H.C. Yee, B. Sjögreen, Comput. Fluids 169 (2018) 331–348.
[43] H.C. Yee, M. Vinokur, M.J. Djomehri, J. Comput. Phys. 162 (1) (2000) 33–81.
[44] G.-S. Jiang, C.-W. Shu, J. Comput. Phys. 126 (1) (1996) 202–228.
[45] M. Castro, B. Costa, W.S. Don, J. Comput. Phys. 230 (5) (2011) 1766–1792.
[46] D.J. Lusher, N. Sandham, in: AIAA Aviation 2019 Forum, American Institute of Aeronautics and Astronautics, 2019.
[47] M.H. Carpenter, J. Nordström, D. Gottlieb, J. Comput. Phys. 365 (98) (1998) 341–365.
[48] C.A. Kennedy, M.H. Carpenter, R. Lewis, Appl. Numer. Math. 35 (3) (2000) 177–219.
[49] A.A. Wray, Minimal storage time advancement schemes for spectral methods, NASA Ames Research Center, California, Report No. MS 202, 1990.
[50] J. Williamson, J. Comput. Phys. 35 (1) (1980) 48–56.
[51] S. Gottlieb, C.-W. Shu, Math. Comput. Am. Math. Soc. 67 (221) (1998) 73–85.
[52] S.P. Jammy, C.T. Jacobs, N.D. Sandham, J. Comput. Sci. 36 (2019) 100565.
[53] The HDF Group, Hierarchical Data Format, version 5, http://www.hdfgroup.org/HDF5/ (1997-NNNN).
[54] S. McIntosh-Smith, J. Price, T. Deakin, A. Poenaru, Concurr. Comput., Pract. Exp. 31 (16) (2019) e5110.
[55] K. Bergman, T. Conte, A. Gara, M. Gokhale, M. Heroux, P. Kogge, B. Lucas, S. Matsuoka, V. Sarkar, O. Temam, Future High Performance Computing Capabilities: Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, Tech. Rep., United States, 2019, https://doi.org/10.2172/1570693, https://www.osti.gov/servlets/purl/1570693.

[56] AnandTech, The Larrabee chapter closes: Intel's final Xeon Phi processors now in EOL, https://www.anandtech.com/show/14305/intel-xeon-phi-knights-mill-now-eol/, 2019. (Accessed 16 December 2019).
[57] D.J. Lusher, S.P. Jammy, N.D. Sandham, Comput. Fluids 173 (2018) 17–21.
[58] J. Lefieux, E. Garnier, N. Sandham, in: AIAA Aviation 2019 Forum, AIAA AVIATION Forum, American Institute of Aeronautics and Astronautics, 2019.
[59] A. Hamzehloo, D.J. Lusher, S. Laizet, N.D. Sandham, Int. J. Numer. Methods Fluids 93 (1) (2021), https://doi.org/10.1002/fld.4879, https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4879.
[60] G.A. Sod, J. Comput. Phys. 27 (1) (1978) 1–31.
[61] C.-W. Shu, S. Osher, J. Comput. Phys. 77 (2) (1988) 439–471.
[62] E.M. Taylor, M. Wu, M.P. Martín, J. Comput. Phys. 223 (1) (2007) 384–397.
[63] G. Zhou, K. Xu, F. Liu, Phys. Fluids 30 (1) (2018) 16102.
[64] V. Daru, C. Tenaud, Comput. Fluids 38 (3) (2009) 664–676.
[65] A. Sansica, Stability and unsteadiness of transitional shock-wave/boundary-layer interactions in supersonic flows, Ph.D. thesis, University of Southampton, 2015.
[66] F. White, Viscous Fluid Flow, McGraw-Hill, New York, NY, 2006.
[67] H. Fasel, A. Thumm, H. Bestek, in: Transitional and Turbulent Compressible Flows - 1993, American Society of Mechanical Engineers, Fluids Engineering Division (Publication) FED, Publ. by ASME, 1993, pp. 77–92, Fluids Engineering Conference; Conference date: 20-06-1993 Through 24-06-1993.
[68] C.-L. Chang, M.R. Malik, J. Fluid Mech. 273 (1994) 323–360, https://doi.org/10.1017/S0022112094001965.