BLOCK STRUCTURED COMPRESSIBLE NAVIER STOKES SOLUTION USING THE OPS HIGH-LEVEL ABSTRACTION

Satya P. Jammy*, Gihan R. Mudalige†, Istvan Z. Reguly†, Neil D. Sandham* and Mike Giles †

*Faculty of Engineering and the Environment, University of Southampton, Southampton SO17 1BJ, UK
e-mail: s.p.jammy@soton.ac.uk

†Oxford e-Research Centre, University of Oxford, 7, Keble Road Oxford OX1 3QG, UK
Web page: http://www.oerc.ox.ac.uk/projects/ops

Key words: Future-proof CFD, GPU computing

Abstract. In this paper we report the development and validation of a compressible solver with shock capturing, using a domain-specific high-level abstraction framework, OPS, that is being developed at the University of Oxford. OPS uses an active library approach for block-structured meshes, capable of generating codes for a variety of parallel implementations with different parallelization strategies. Performance results on various architectures are reported for the 1D Shu-Osher test case.

1 Introduction

High Performance Computing (HPC) systems and architectures are currently evolving rapidly. Traditional single processor-based CPU clusters are moving towards multi-core/multi threaded CPUs with at least four CPU cores per single silicon chip. At the same time new architectures, based on many-core processors such as GPU's and Intel's Xeon Phi, are emerging as important systems and further developments are expected with energy-efficient designs from ARM and IBM. Given the rapid changes, porting existing solvers to extract the full computational power of new architectures is a major challenge for legacy CFD codes and maintainability of the codes is becoming more difficult. One way to address this problem is to use domain-specific high-level abstractions (HLA), such as domain-specific languages (DSLs) and active libraries.

This research explores the capability of one such Domain Specific Active Library, called OPS (Oxford Parallel Library for Structured mesh solvers) [1] for future proofing of legacy CFD codes. Here, we focus on a hydrodynamic code for solving the compressible Navier-Stokes equations on block-structured grids with shock capturing. Such legacy codes form a key part of HPC workload and are used extensively for studying various aspects of shock wave boundary layer interaction (SBLI), aero-thermodynamics, aerofoils and many others (see for example, [2] and references therein). For this purpose, we develop a shortened version of the Southampton SBLI code [2] written from scratch in C++ using the OPS API (application program interface), named as SHSGC (Supersonic Hypersonic Solver on

GPUs and CPUs). This results in a single high-level application source code. Using the automatic code generation techniques of OPS, a range of parallel implementations (MPI, OpenMP, CUDA, OpenCL and OpenACC) on various architectures are generated. In this research, we present the validation and performance results of SHSGC on Intel multi-core CPUs and NVIDIA GPUs (Keppler K20c) for the Shu-Osher problem.

The rest of the paper is organised as follows: Section 2 presents the numerical method used; in Section 3 we briefly present the OPS abstraction, its API for key elements in the solution algorithm, and the automatic code generation process; in Section 4 the validation and performance of various implementations of SHSGC is presented; finally, Section 5 notes the conclusions and future work.

2 Numerical method

The governing equations are the compressible Navier-Stokes equations, which are solved using fourth-order central schemes for the spatial derivatives, coupled with a TVD scheme for capturing shocks and a third-order low-storage Runge-Kutta temporal scheme. As the developed solver is a shortened version of the Southampton SBLI code, only multi-block, 1D, inviscid, viscous and TVD scheme with a choice of limiters are incorporated.

3 OPS

Previous work at the University of Oxford developed a high-level abstraction framework called OP2 [3] that targeted the domain of unstructured mesh-based applications. With OP2 it was demonstrated that both developer productivity as well as near-optimal performance could be achieved on a wide range of parallel hardware. Research published as a result of this includes performance analysis of standard CFD applications [5], as well as a full industrial-scale application from Rolls-Royce plc[6].

OPS is designed similar to OP2, but targets the domain of multi-block structured applications. OPS has its own domain specific API, which uses source-to-source translation to parse the API calls and generate different parallel implementations. The next section illustrates various OPS API calls that are used in SHSGC.

3.1 OPS API calls for SHSGC

As outlined in Section 2, SHSGC uses finite differences to solve the governing equations on a structured grid. The algorithm consists of declaring the blocks, data and operations or calculations such as evaluating derivatives or primitive variable, that should be performed on each block to advance the solution in time. Below we discuss the implementation of various key components used in developing SHSGC. In addition some original C code is shown, to give the reader an overview of the modifications required while writing source code using OPS.

The OPS API calls discussed here are for a single block; multiple blocks are treated by OPS as unstructured collection of structured blocks. Multi-block implementation details can be found in [1] and the references therein.

Figure 1 shows the OPS API call for declaring blocks and the data sets. In the para-

```
1 // Declare a block
2 ops_block shsgc_grid = ops_decl_block(ndim, "shsgc_grid");
3 /* Size of the data array*/
4 int size[1] = {nxp};
5 int d_p[1]  = {2};
6 int d_m[1]  = {-2};
7 double* dat = NULL;
8 /* Declare data on block */
9 ops_dat rho, rhoU, ..., rhoE;
10 rho  = ops_decl_dat(shsgc_grid,ndim,size,d_m,d_p,dat,"double","rho") ;
11 rhoU = ops_decl_dat(shsgc_grid,ndim,size,d_m,d_p,dat,"double","rhoU");
12 rhoE = ops_decl_dat(shsgc_grid,ndim,size,d_m,d_p,dat,"double","rhoE");
```

Figure 1: OPS API for declaration of blocks, data sets

graphs below the text in bold letters refers to either an OPS calling function or arguments to it. In the block declaration API call (lines 1-2 in the figure) **ndim** is the number of dimensions of the block (in this case 1) and **shsgc_grid** is the name of the block.

To declare data on the blocks, the OPS API (**ops_decl_dat**) is shown in lines $7-10$ of figure 1. Data set declaration in OPS requires information about the block on which it should be declared, the number of dimensions (**ndim**), the size of the array (**size**), the number halo points in the positive and negative direction (**d_p, d_m**), data (**dat**), as well as its data type and the name (used for debugging). If a **NULL** pointer is provided as the data, OPS automatically allocates the required amount of memory, depending on the shape of the array and the data-type provided.

```
1 /* Original C loop to evaluate Primitive variables in 1D */
2 for (int i = 0; i < nxp; i++) {
3   u[i] = rhoU[i]/rho[i];
4   p[i] = (gamma-1.0f) * (rhoE[i] - 0.5f * rho[i](pow(u[i],2)));
5   }
6 /* Evaluation of first derivative in a C loop*/
7 for (int i = 2; i < nxp-2; i++) {
8   deru[i] = (u[i+2] + 8.0f*u[i+1] - 8.0*u[i-1] - u[i-2])/(12.0f*dx);
9  }
```

Figure 2: C loop for evaluating primitive variables and derivatives

After declaring blocks and data on the blocks, computations are performed on the blocks. For example, the evaluation of primitive variables or derivatives on the entire grid is achieved using a for-loop. For loops are implemented in OPS using **ops_par_loop**. To convert a for-loop into OPS API, the application developer needs to write the calling function (**ops_par_loop**) and the user kernel that contains the computations. These concepts are explained below using two example for-loops: evaluation of primitive variables and differentiation using a fourth-order central scheme.

Figure 2 shows the C loops for the examples and figure 3 shows their OPS API equivalent loops and kernel functions. The OPS equivalent of the C loop shown in figure 2 lines 2-5 is given in figure 3 lines 1-18. The inputs to the **ops_par_loop** are the computational

```
1 /* OPS loop to evaluate Primitive variables in 1D */
2 /*kernel*/
3 void primitive_kernel(const double *rho,const double *rhoU, const double
4   *rhoE, double *u, double *p){
5   u[OPS_ACC3(0)] = rhoU[OPS_ACC1(0)]/rho[OPS_ACC0(0)];
6   p[OPS_ACC4(0)] = (gamma-1.0f) * (rhoE[OPS_ACC2(0)] -
7                    0.5f * rho[OPS_ACC0(0)](pow(u[OPS_ACC3(0)],2)));
8   }
9 int s1D_0[]    = {0};
10 S1D_0          = ops_decl_stencil( 1, 1, s1D_0, "0");
11 int range = {0-halo,nxp+halo}
12 ops_par_loop(primitive_kernel,"primitive_kernel",shsgc_grid,ndim,range,
13              ops_arg_dat(rho, S1D_0, "double", OPS_READ),
14              ops_arg_dat(rhoU, S1D_0, "double", OPS_READ),
15              ops_arg_dat(rhoE, S1D_0, "double", OPS_READ),
16              ops_arg_dat(u, S1D_0, "double", OPS_WRITE),
17              ops_arg_dat(p, S1D_0, "double", OPS_WRITE)
18              )
19 /* Evaluation of derivatives*/
20 void xder1_kernel(const double *inp, double *out) {
21   double dix = 1/(12.0f*dx);
22   out[OPS_ACC1(0)] = (inp[OPS_ACC0(-2)] - inp[OPS_ACC0(2)]  + 8.0f *(
23     inp[OPS_ACC0(1)] - inp[OPS_ACC0(-1)] )) * dix;
24 }
25 int s1D_5[]    = {-2,-1,0,1,2};
26 S1D_5          = ops_decl_stencil( 1, 5, s1D_5, "5");
27 int range = {0-halo,nxp+halo}
28 ops_par_loop(xder1_kernel,"xder1_kernel",shsgc_grid,ndim,range,
29               ops_arg_dat(u, S1D_5, "double", OPS_READ),
30               ops_arg_dat(derxu, S1D_0, "double", OPS_READ)
31              )
```

Figure 3: For-loop implementation in OPS API

kernel, block, number of dimensions of the block, grid range on which the loop should be implemented and the **ops_arg_dat** variables. The **ops_arg_dat** API call should have the data-access stencil. The stencil access for the evaluating primitive variables is shown in lines 9-11 in figure 3. The stencil is defined by the relative position of the data from the grid point. Similarly, the stencil for evaluating the derivatives is shown in lines 25,26 in figure 3. This helps in error checking of the data stencils accessed in the computations. In the user kernel, the data should be accessed using **var[OPS_ACCn(loc)]**, where **var** is the name of the variable in the user kernel, **n** is the number of the variable (**var**) in the inputs list to the kernel (C-style indexing) and **loc** is the stencil location relative to the grid point. The various inputs that can be provided to the user kernel are described

	Left $x <= -4$	**Right** $x > -4$
ρ	3.857143	$1 + 0.2\sin(5\pi x)$
u	2.629369	0
p	10.3333	1

Table 1: Initial conditions for the Shu-Osher problem

in detail in [1]. Similarly, the other loops in SHSGC are defined.

3.2 OPS translator

The OPS translator is written in Python, with the high-level application source code provided as input to the translator. It parses the OPS API calls and generates optimised platform-specific OPS application files. These application files are linked with the OPS platform-specific optimised backend libraries at compile time to generate optimised binary executable files. Details of the implementation of the OPS translator can be found in [1]. In the next section we discuss the validation and performance results of SHSGC.

4 Results

The high-level SHSGC solver developed is translated using the OPS translator to automatically generate platform specific optimised MPI, OpenMP, CUDA and OpenCL parallelizations. Below, we discuss the validation and performance of the solver on various architectures for a benchmark CFD test case.

4.1 Shu-Osher problem

The Shu-Osher test case is a hydrodynamic test case, which simulates the evolution of a normal shock interacting with downstream sinusoidal density fluctuation ($\rho = 1 + \epsilon\sin(x/\lambda)$) as the initial condition. where, λ is the wavelength and ϵ is the amplitude of the initial fluctuations and the initial shock foot is located at $x = \lambda$ [4]. The problem is solved on a domain $[-5, 5]$, with 2504 grid points. The left and right states of the initial conditions are given in table 1, the initial location of the shock foot is at $x = -4$. The left and right boundary conditions are set to supersonic inlet and supersonic outlet respectively. The solver is run for both inviscid and viscous cases with $Re = 1 \times 10^5$ for the latter.

The solution is evolved from the initial state until $t = 1.8s$. Figure 4 shows the results from the present solver compared with the results from [4] at $t = 1.8s$. Similar results are observed for various auto generated parallelisations.

4.2 Performance

To evaluate the performance at various grid sizes, the Shu-Osher test case is scaled to higher grid sizes. The simulations are performed until $t = 1.8s$ and the total runtime of the solver on various architectures is reported in table 3. The CPU and GPU simulations

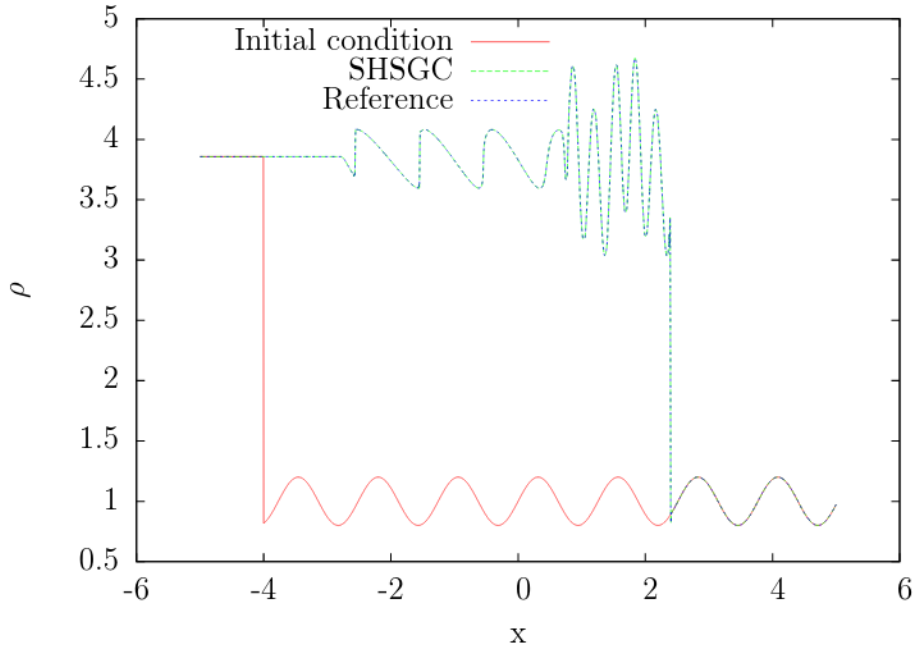

Figure 4: Density plot for Shu-Osher test case with solutions from SHSGC and reference [4] at $t = 1.8$, along with the initial condition

are performed on Intel CPU and NVIDIA GPU respectively, with details given in table 2. From the total runtime of the simulations, it can be concluded that the MPI parallelization performs better than OpenMP parallelization by a factor of 1.5. For OpenCL and CUDA implementations on the GPU, CUDA performs better than the OpenCL implementation.

The CUDA parallelization show a speed up of ~ 6 compared to 12 MPI processes and a speed-up of ~ 9 compared to 12 OpenMP parallelization for a grid size of two million as shown in figure 5. Similar speed-up results are observed on the GPU using OpenCL.

5 Conclusions

In this paper, we have explored the future proofing of a representative compressible hydrodynamic solver by re-engineering it in the OPS domain-specific high-level abstraction

System	Broomway	K20
Node Architecture	2 × 8-core Intel Xeon E5-2680 2.70GHz (Sandy bridge)	NVIDIA Tesla K20c
Memory per Node	64GB	5GB/GPU (ECC off)
OS	Red Hat Enterprise Linux 6	Red Hat Enterprise Linux 6.4

Table 2: Benchmark systems used in the simulations

Grid size (Millions)	12 MPI	12 OpenMP	OpenCL on GPU	CUDA
0.0025	2.92	4.61	3.75	3.37
0.1	66.05	76.47	16.69	16.13
0.2	136.79	167.13	30.09	29.15
2	1738.83	2429.44	271.51	264.59

Table 3: Total runtime of the simulation on various architectures

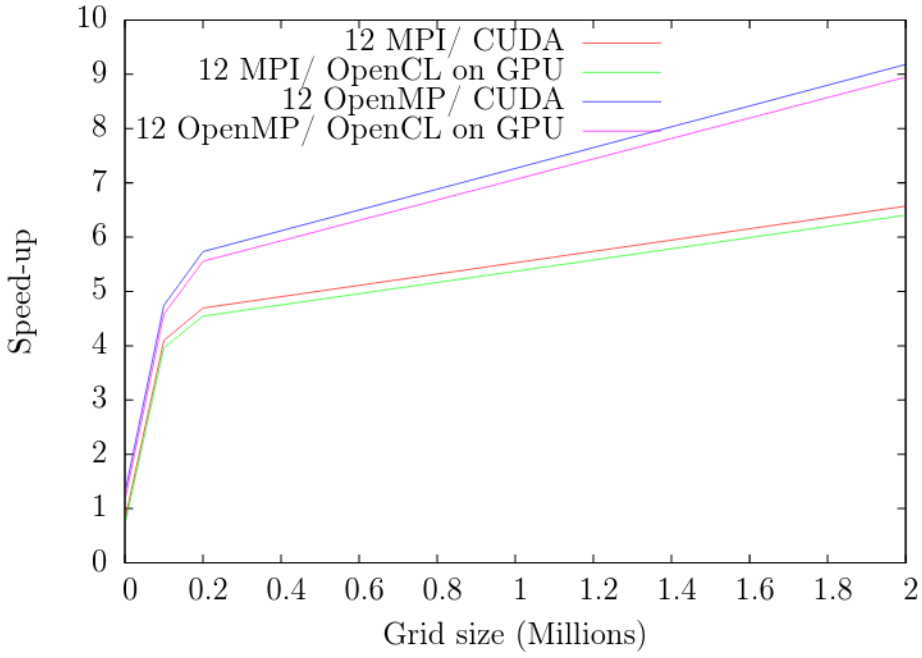

Figure 5: Speed up of CUDA, OpenCL solvers compared with MPI and OpenMP solvers

framework API. The high-level source code is translated into various parallel implementations on a variety of architectures using the automatic code generator (OPS translator). The performance results show that the optimised CUDA and OpenCL implementations achieve a speed-up of 6-9 compared with optimised 12 MPI implementation. Using OPS can help in the future proofing and maintainability of legacy CFD codes, as only a single source code is required for multiple target architectures.

6 Acknowledgements

This research is funded by the UK Engineering and Physical Sciences Research Council projects EP/K038494/1 and EP/K038567/1 on "Future-proof massively-parallel execution of multi-block applications".

REFERENCES

[1] Istvan Z. Reguly, Gihan R. Mudalige, Michael B. Giles, Dan Curran and Simon McIntosh-Smith. "The OPS domain specific abstraction for multi-block structured grid computations". *Proceedings of the 4th international workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing(WOLFHPC '14)*

[2] Nicola De Tullio, and Neil D. Sandham "Influence of boundary-layer disturbances on the instability of a roughness wake in a high-speed boundary layer". *Journal of Fluid Mechanics* (2015) 763:136-165

[3] OP2 for Many-Core Platforms (2013), http://www.oerc.ox.ac.uk/research/op2

[4] Sergio Pirozzoli, "Conservative hybrid compact-WENO schemes for shock-turbulence interaction", *Journal of Computational Physics* (2002) 178:81-117

[5] Mudalige, G., Giles, M., Thiyagalingam, J., Reguly, I., Bertolli, C., Kelly, P., Trefethen, A.: Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. Parallel Computing 39(11), 669âĂŞ692 (2013)

[6] Reguly, I.Z., Mudalige, G.R., Bertolli, C., Giles, M.B., Betts, A., Kelly, P.H.J., Radford, D.: Acceleration of a full-scale industrial CFD application with op2. ACM Transactions on Parallel Computing (2013), available at http://arxiv-web3.library.cornell.edu/abs/1403.7209