Performance Analysis of a High-level Abstractions-based Hydrocode on Future Computing Systems

G.R. Mudalige[1], I.Z. Reguly[1], M.B. Giles[1], A.C. Mallinson[3], W.P. Gaudin[2], and J.A. Herdman[2]

[1] Oxford e-Research Centre, University of Oxford, 7, Keble Road Oxford OX1 3QG
{gihan.mudalige, istvan.reguly}@oerc.ox.ac.uk, mike.giles@maths.ox.ac.uk
[2] High Performance Computing, UK AWE plc. Aldermaston, UK.
{Wayne.Gaudin, Andy.Herdman}@awe.co.uk
[3] Department of Computer Science, University of Warwick, UK
acm@dcs.warwick.ac.uk

Abstract. In this paper we present research on applying a domain specific high-level abstractions (HLA) development strategy with the aim to "future-proof" a key class of high performance computing (HPC) applications that simulate hydrodynamics computations at AWE plc. We build on an existing high-level abstraction framework, OPS, that is being developed for the solution of multi-block structured mesh-based applications at the University of Oxford. OPS uses an "active library" approach where a single application code written using the OPS API can be transformed into different highly optimized parallel implementations which can then be linked against the appropriate parallel library enabling execution on different back-end hardware platforms. The target application in this work is the CloverLeaf mini-app from Sandia National Laboratory's Mantevo suite of codes that consists of algorithms of interest from hydrodynamics workloads. Specifically, we present (1) the lessons learnt in re-engineering an industrial representative hydro-dynamics application to utilize the OPS high-level framework and subsequent code generation to obtain a range of parallel implementations, and (2) the performance of the auto-generated OPS versions of CloverLeaf compared to that of the performance of the hand-coded original CloverLeaf implementations on a range of platforms. Benchmarked systems include Intel multi-core CPUs and NVIDIA GPUs, the Archer (Cray XC30) CPU cluster and the Titan (Cray XK7) GPU cluster with different parallelizations (OpenMP, OpenACC, CUDA, OpenCL and MPI). Our results show that the development of parallel HPC applications using a high-level framework such as OPS is no more time consuming nor difficult than writing a one-off parallel program targeting only a single parallel implementation. However the OPS strategy pays off with a highly maintainable single application source, through which multiple parallelizations can be realized, without compromising performance portability on a range of parallel systems.

1 Introduction

High performance computing (HPC) is currently in a period of enormous change. For many years, increased performance was achieved through higher clock frequencies, but that trend was brought to an abrupt halt by the corresponding increase in energy consumption. The clear direction now is towards improved performance through increasing parallelism, even reducing the clock frequency a little to improve the energy efficiency, which is becoming a key concern. However, there is no clear consensus yet on the best architecture for HPC. On the one hand there are many-core accelerators such as GPUs and the new Intel Xeon Phi, usually with 16-64 functional units, each of which can be viewed as a vector processor with many elements (cores) performing the same operation at the same time but with different data. On the other hand, we have mainstream Intel/AMD CPUs with very large caches and a more modest number of functional units (cores) each with their own vector components (e.g. AVX units), or the IBM BlueGene systems which are based on a large network of relatively small but energy-efficient CPUs. In the future, we may also have interesting energy-efficient designs from ARM [9] and other companies [19] which have achieved great energy efficiency for mobile and embedded applications, and are now targeting HPC which increasingly shares similar goals.

In the light of these developments, an application developer faces a tough problem. Optimizing their application for execution on a particular platform requires an increasing amount of platform-specific knowledge, and possibly a major re-write to reduce data communications. At the same time, there is considerable uncertainty about which platform to target; it is not clear which architectural approach is likely to "win" in the long-term, and it is not even clear in the short-term which platform is best for any given application.

Currently the common approach for utilizing novel hardware, or different many-core accelerators is to manually port the legacy application, in many cases by converting key compute kernels to utilize the accelerators. In some cases a major ground-up rewrite is required, for example if you need to reduce data communications to efficiently utilize the new hardware. The conversion process is highly error-prone and takes significant amounts of developer effort to program, validate and optimize. It is unreasonable for domain scientists to be engaged in such optimization work that will require them to port the application for each new generation of systems. Thus "future proofing" HPC applications for their continued performance and portability on a diverse range of hardware and future emerging systems is of critical importance.

One such approach, is the use of domain specific high-level abstractions (HLAs), such as domain specific languages (DSLs) and active libraries [31, 13]. The key idea is to provide the application developer with a set of domain specific constructs to declare the problem to be computed, without specifying its implementation [18]. It is then the task of a lower implementation level to apply automated techniques for translating the specification into different implementations for different hardware and software platforms. The use of such a development strategy has previously been shown to have significant benefits both for

developer productivity and gaining near-optimal performance [28, 14]. However, currently these still remain as experimental research projects and have not yet been adopted by a wider HPC community. Partly the reason is a lack of DSLs or high-level frameworks that are actively used for creating production level applications. On the other hand, previous work has only developed such frameworks for a few application domains.

The research in this paper is thus motivated by the need to explore further the utility of high-level abstraction frameworks for future proofing parallel scientific simulation applications from a range of application domains. Here we focus on a hydro-dynamics application, belonging to an important class of codes which form a key part of the HPC workload at many organizations such as the AWE. We make use of a previously developed mini-application called CloverLeaf [8], which implements algorithms of interest related to this workload. This research explores the performance of CloverLeaf after re-engineering the application based on a domain specific HLA framework. CloverLeaf is open source software and forms part of Sandia National Laboratory's Mantevo project [5]. With the use of an unrestricted application as a proxy, our aim is to demonstrate to a wider HPC audience the performance portability resulting from an HLA based development and how this strategy might help in addressing various scientific simulation challenges on future emerging systems.

The CloverLeaf mini-application has been previously manually ported [17, 20, 16] to execute on many parallel platforms. These include parallelizations based on single-instruction-multiple-data (SIMD, e.g. SSE and AVX) and shared memory multi-threading for multi-core CPUs (e.g. OpenMP), single instruction multiple thread (SIMT, e.g. CUDA, OpenCL and OpenACC) for GPUs and the Intel's Xeon Phi and distributed memory parallelization (e.g. MPI) for clusters of CPUs/GPUs. Recently the code was re-written [10] with a domain specific high-level abstraction framework, called OPS which resulted in a single high-level application source. Automated code generation techniques of OPS were then used to generate a range of parallel implementations. In this paper we compare the performance of the resulting parallelizations to that of the original hand-tuned CloverLeaf applications. Unlike previous work, the availability of highly optimized, manually hand-tuned parallel versions gives us a unique opportunity to compare and contrast the high-level development process both in terms of developer productivity and performance portability. Our research demonstrates, through performance analysis and benchmarking on a range of hardware and software systems, the benefits of the HLA approach giving significant insights into high-level methods for "future proofing" HPC applications. The main contributions of this paper are twofold:

1. We present lessons learnt in re-engineering an industrially representative hydro-dynamics application to utilize the OPS high-level framework and subsequent code generation to obtain a range of parallel implementations. Through OPS we generate code targeting OpenMP thread level multi-core parallelism, single-instruction multiple-thread (SIMT) many-core parallelism

using CUDA, OpenCL and OpenACC and distributed memory parallelism with MPI.
2. The performance of the OPS versions of CloverLeaf is compared to that of the performance of the original CloverLeaf implementations on a range of platforms. These include the latest Intel multi-core CPUs (Sandy Bridge), NVIDIA GPUs (Kepler K20c), a Cray XC30 distributed memory cluster (Archer [7]) and a large Cray XK7 GPU cluster (Titan [11]). Key performance bottlenecks are analyzed and further optimizations are discussed.

The rest of this paper is organized as follows: in Section 2 we briefly present the OPS abstraction, its API, design and code generation process; in Section 3, a benchmarking and performance analysis of the of the application is carried out comparing the OPS based CloverLeaf with the original hand-tuned version; Section 4 will briefly detail related work in this area and compare them to our contributions in this paper. Finally Section 5 notes future work and conclusions.

2 OPS

Previous work at the University of Oxford developed a high-level abstraction framework called OP2 [6] targeting the domain of unstructured mesh based applications. With OP2 we 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 work includes a number of performance analysis studies on standard CFD benchmark applications [23] as well as a full industrial-scale application from the production work-load at Rolls-Royce plc. [28].

OPS (Oxford Parallel Library for Structured-mesh solvers) follows much of the design of OP2, but targets the domain of multi-block structured applications. Multi-block structured mesh applications can be viewed as an unstructured collection of structured mesh blocks. As CloverLeaf is a single block-structured mesh code, it only required OPS's single block API to re-engineer the application. The structured mesh domain is distinct from the unstructured mesh applications domain due to the implicit connectivity between neighboring mesh elements (such as vertices, cells) in structured meshes/grids. The key idea is that operations involve looping over a "rectangular" multi-dimensional set of grid points using one or more "stencils" to access data.

OPS is designed to appear as a classical software library with a domain specific API. It then uses source-to-source translation techniques to parse the API calls and generate different parallel implementations. These can then be linked against the appropriate parallel library enabling execution on different back-end hardware platforms. The aim is to generate highly optimized platform specific code and link with equally efficient back-end libraries utilizing the best low-level features of a target architecture. The next section briefly illustrates the OPS API using examples from CloverLeaf.

```
/* Declare a single structured block */
ops_block cgrd = ops_decl_block(2, "clover grid");

int size[2]  = {x_cells+5, y_cells+5};
double* dat = NULL;

/* Declare data on block */
ops_dat density0, energy0, ..., pressure, volume;

density0=ops_decl_dat(cgrd,1,size,dat,"double","density0");
energy0 =ops_decl_dat(cgrd,1,size,dat,"double","energy0");
......
pressure=ops_decl_dat(cgrd,1,size,dat,"double","pressure");
volume  =ops_decl_dat(cgrd,1,size,dat,"double","volume");
```

Fig. 1. OPS API example for declaring blocks, data and stencils

2.1 The OPS API

The CloverLeaf mini-app involves the solution of the compressible Euler equations, which form a system of four partial differential equations. The equations are statements of the conservation of energy, density and momentum and are solved using a finite volume method on a structured staggered grid. The cell centers hold internal energy and density while nodes hold velocities. The solution involves an explicit Lagrangian step using a predictor/corrector method to update the hydrodynamics, followed by an advective remap that uses a second order Van Leer up-winding scheme. The advective remap step returns the grid to its original position. The original application [8] is written in Fortran and operates on a 2D structured mesh. It is of fixed size in both x and y dimensions.

OPS separates the specification of such a problem into four distinct parts: (1) structured blocks, (2) data defined on blocks, (3) stencils defining how data is accessed and (4) operations over blocks. Thus the first aspect of declaring such a single-block structured mesh application with OPS is to define the size of the regular mesh over which the computations will be carried out. In OPS vernacular this is called an `ops_block`. OPS declares a block with the `ops_decl_block` API call by indicating the dimension of the block (2D in this case) and assigning it a name for identification and runtime checks (see Figure 1).

CloverLeaf works on a number of data arrays (or fields) which are defined on the 2D structured mesh (e.g. density, energy, x and y velocity of particles). OPS allows users to declare these using the `ops_decl_dat` API call; the `density0,energy0, ... pressure` and `volume` are `ops_dat`s that are declared through this API. A key idea is that once a field's data is declared via `ops_decl_dat` the ownership of the data is transfered from the user to OPS, where it is free to rearrange the memory layout as is optimal for the final parallelization and execution hardware. In contrast, each of the original CloverLeaf implementations explicitly involve the allocation and management of memory specific to each parallel implementation at the application source level. In this example a NULL pointer of type `double` is passed as an argument. CloverLeaf

```
1 DO k=y_min-2,y_max+2
2   DO j=x_min-2,x_max+2
3     pre_vol(j,k)=volume(j,k)+
4                  (vol_flux_x(j+1,k)-vol_flux_x(j,k)+
5                   vol_flux_y(j,k+1)-vol_flux_y(j,k))
6     post_vol(j,k)=pre_vol(j,k)-
7                  (vol_flux_x(j+1,k)-vol_flux_x(j,k))
8   ENDDO
9 ENDDO
```

Fig. 2. Original loop from `advec_cell` kernel

initializes these values later, as part of the application itself. When a NULL array is supplied, OPS will internally allocate the required amount of memory based on the type of the data array and its size. On the other hand an array containing the relevant initial data can be used in declaring an `ops_dat`. In the future we will provide the ability to read in data from HDF5 files directly using a `ops_decl_dat_hdf5` API call. Note above in an `ops_decl_dat` call, a single double precision value per grid element is declared. A vector of a number of values per grid element could also be declared (e.g. a vector with three doubles per grid point to store x,y and z velocities).

All the numerically intensive computations in the structured mesh application can be described as operations over the block. Within an application code, this corresponds to loops over a given block, accessing data through a stencil, performing some calculations, then writing back (again through the stencils) to the data arrays. A loop from the `advec_cell` routine in CloverLeaf's reference implementation [8] is detailed in Figure 2, operating over each grid point in the structured mesh. Note that here the data arrays are all declared as Fortran allocatable 2D arrays. The loop operates in column major order.

An application developer declares this loop using the OPS API as illustrated in Figure 3 (lines 31-37), together with the "elemental" kernel function (lines 2-14). The elemental function is called a "user kernel" in OPS to indicate that it represents a computation specified by the user (i.e. the domain scientist) to apply to each element (i.e. grid point). User kernels are usually placed on a separate header file, which gets included in the file declaring the `ops_par_loop`. By "outlining" the user kernel in this fashion, OPS can factor out the declaration of the problem from its parallel implementation. The macros `OPS_ACC0, OPS_ACC1, OPS_ACC2` etc. will be resolved to the relevant array index to access the data stored in `density0, energy0, pressure` etc.[4] The explicit declaration of the stencil (lines 19-28) additionally will allow for error checking of the user code. In this case we use three stencils, one consisting of a single point referring to the current element, the second accessing the $(1, 0)$ stencil and the third accessing the $(0, 1)$ stencil. More complicated stencils can be declared giving the relative position from the current `(0,0)` element. The `ops_par_loop` declares

[4] A similar approach is used in the C kernel implementations of the original CloverLeaf application

```
/*user kernel*/
inline void advec_cell_kernel1_xdir (
    double *pre_vol, double *post_vol,
    const double *volume,
    const double *vol_flux_x,
    const double *vol_flux_y){

pre_vol[OPS_ACC0(0,0)] = volume[OPS_ACC2(0,0)] +
  (vol_flux_x[OPS_ACC3(1,0)]- vol_flux_x[OPS_ACC3(0,0)]+
   vol_flux_y[OPS_ACC4(0,1)]- vol_flux_y[OPS_ACC4(0,0)]);

post_vol[OPS_ACC1(0,0)] = pre_vol[OPS_ACC0(0,0)] -
  (vol_flux_x[OPS_ACC3(1,0)]- vol_flux_x[OPS_ACC3(0,0)]);
}
//mesh execution range
int rangexy[] = {x_min-2,x_max+2,y_min-2,y_max+2};

//declare stencils
ops_stencil S2D_00, S2D_00_P10, S2D_00_0P1;
/*single point stencil*/
int s2D_00[] = {0,0};
S2D_00 = ops_decl_stencil( 2, 1, s2D_00, "00");

/*2 point stencils*/
int s2D_00_P10[] = {0,0, 1,0};
S2D_00_P10 = ops_decl_stencil(2,1,s2D_00_P10,"0,0,:1,0");
int s2D_00_0P1[] = {0,0, 0,1};
S2D_00_0P1 = ops_decl_stencil(2,1,s2D_00_0P1,"0,0,:0,1");

/*parallel loop declaration*/
ops_par_loop(advec_cell_kernel1_xdir,
  "advec_cell_kernel1_xdir", clover_grid, 2, rangexy,
   ops_arg_dat(work_array1,S2D_00,"double",OPS_WRITE),
   ops_arg_dat(work_array2,S2D_00,"double",OPS_WRITE),
   ops_arg_dat(volume,S2D_00,"double",OPS_READ),
   ops_arg_dat(vol_flux_x,S2D_00_P10,"double",OPS_READ),
   ops_arg_dat(vol_flux_y,S2D_00_0P1,"double",OPS_READ));
```

Fig. 3. Loop from `advec_cell` converted to use the OPS API

the structured block to be iterated over, its dimension, the iteration range and the `ops_dat`s involved in the computation. `OPS_READ` indicates that `density0` will be read only. The actual parallel implementation of the loop is specific to the parallelization strategy involved. OPS is free to implement this with any optimizations necessary to obtain maximum performance. The `ops_arg_dat(..)` in Figure 3 indicates an argument to the parallel loop that refers to an `ops_dat`. A similar function `ops_arg_gbl()` enables users to indicate global reductions.

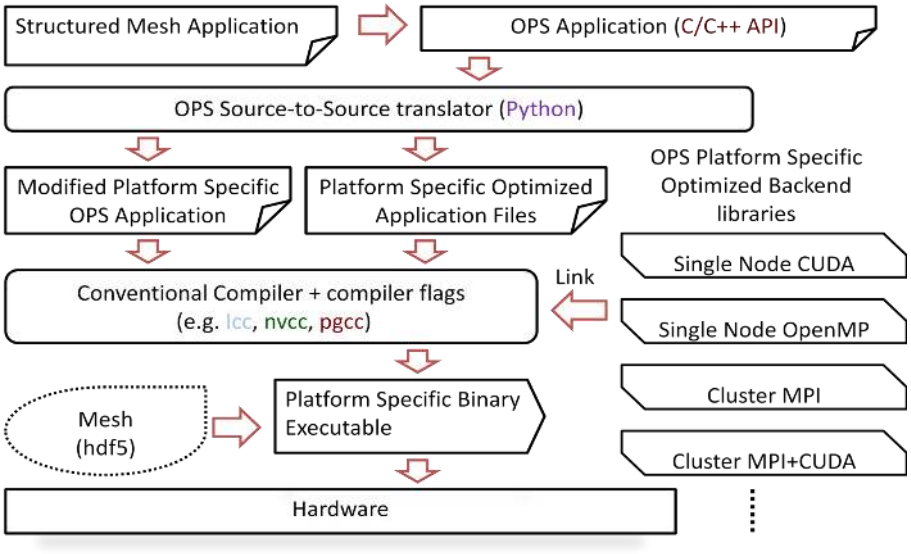


Fig. 4. OPS code generation and build process

2.2 Porting CloverLeaf to OPS

The original CloverLeaf 2D application written in Fortran 90 was converted to the OPS API by manually extracting the user kernels, outlining them in header files and converting the application to the OPS's C/C++ API. All effort was taken to keep the naming conventions of routines and files as similar to the original as possible. After conversion, the OPS CloverLeaf version consists of 80 `ops_par_loop`s spread across 16 files with about 7000 lines of code. This application can be code generated to obtain a range of parallel implementations. In comparison each of the original CloverLeaf implementations are self contained separate parallel implementations, one for each of MPI+CUDA, MPI+OpenMP etc. The original CloverLeaf reference implementation (i.e. the MPI+OpenMP parallelization) consists of about 7000 lines of source code. The OPS back-end library (implemented in C and C++) which currently supports parallelizing over with OpenMP, CUDA, OpenACC, OpenCL and MPI including common support functions for all these parallelizations and other utility functions, plus the code generation tools, in total consists of about 15000 lines of source code. However, the important fact to note here is that the back-end libraries and code generation tools are generic to be applicable to any application developed with the OPS API, not just CloverLeaf.

Once converted to the OPS API, an application can be validated as a single threaded implementation, simply by including the header file `ops_seq.h` and linking with OPS's sequential back-end library. The header file and the library implement API calls for a single threaded CPU and can be compiled and linked using conventional (platform specific) compilers (e.g. gcc, icc) and executed as a serial application.

The serial developer version allows for the application's scientific results to be inspected before code generation takes place. It also validates the OPS API calls and provides feedback on any errors, such as differences between declared stencils and the corresponding user kernels or differences between data types. All such feedback is intended to reduce the complexity of programming and ease debugging. There is opportunity at this stage to add further checks and tests to increase developer productivity, for example report cases where the iteration range of a loop written by a developer attempts to access elements beyond the number of grid points in any dimension of an `ops_dat`. Including the developer header file and linking with OPS's distributed memory (MPI) back-end libraries can also be used to obtain a low performance MPI parallelization of the application for testing purposes. The full CloverLeaf developer version can be found under the OPS git-hub repository [10].

The manual conversion of the original application to the OPS API required no more effort than what is typically required by a developer proficient in a given parallel computing model (OpenMP, CUDA etc.) for directly porting to a different parallel implementation. However once converted, the use of OPS to generate different parallelizations of the application was trivial. Therefore we believe that the conversion is an acceptable one-off cost for legacy applications attempting to utilize the benefits of high level frameworks such as DSLs or Active Libraries. As we will show in this paper, the advantages of such frameworks far outweigh the costs, by significantly improving the maintainability of the application source, while making it possible to also gain near optimal performance and performance portability across a wide range of hardware.

Once the application developer is satisfied with the validity of the results produced by the sequential application, parallel code can be generated. The build process to obtain a parallel executable as illustrated in Figure 4 follows that of OP2's code generation process [23]. The API calls in the application are parsed by the OPS source-to-source translator which will produce a modified main program and back-end specific code. These are then compiled using a conventional compiler (e.g. gcc, icc, nvcc) and linked against platform specific OPS back-end libraries to generate the final executable. As mentioned before, there is the option to read in the mesh data at runtime. The source-to-source code translator is written in Python and only needs to recognize OPS API calls; it does not need to parse the rest of the code. We have deliberately chosen to use Python and a simple source-to-source translation strategy to significantly simplify the complexity of the code generation tools and to ensure that the software technologies on which it is based have long-term support. The use of Python makes the code generator easily modifiable allowing for it to even be maintained internally within an organization. Furthermore, the code generated through OPS is itself human readable which helps with maintenance and development of new optimizations.

OPS currently supports parallel code generation for execution on (1) single threaded vectorized CPUs, (2) multi-threaded CPUs/SMPs using OpenMP, (3) NVIDIA GPUs using CUDA and OpenACC, (4) OpenCL devices such as AMD

Table 1. Single Node Benchmark systems

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
Compilers and Flags	Intel CC 14.0.0 Intel MPI 4.1.3 -O3 IEEE_FLAGS[1]	CUDA 6.0 IEEE_FLAGS[1] -gencode arch=compute_35, code=sm_35 -O3 NVIDIA OpenCL PGI compiler 14.2 (for OpenACC)

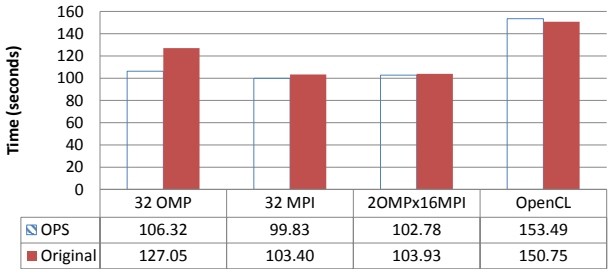
Table 2. Distributed memory Benchmark systems

System	Archer	Titan
Node Architecture	2×12-core Intel Xeon E5-2697 2.70GHz (Ivy Bridge)	16-core AMD Opteron 6274 + NVIDIA K20X
Memory per Node	64GB	32GB + 6GB/GPU (ECC on)
Interconnect	Cray Aries	Cray Gemini
OS	CLE	CLE
Compilers and Flags	Cray C Compilers 8.2.1 cray-mpich/6.1.1 -O3 -Kieee	Cray C Compilers 8.2.2 -cray-mpich/6.3.0 -O3 -hgnu -O3 -arch=sm_35 PGI Compiler 13.10-0

GPUs, the Intel XeonPhi, etc. (5) distributed memory clusters of single threaded CPUs using MPI (6) a cluster of multi-threaded CPUs using MPI and OpenMP and (7) a cluster of GPUs using MPI and CUDA. A more complete discussion of the code generation and optimizations for the multi-core CPU, NVIDIA GPU and MPI parallelizations is given in [10]. In the next section we delve directly into the performance of each of these generated versions.

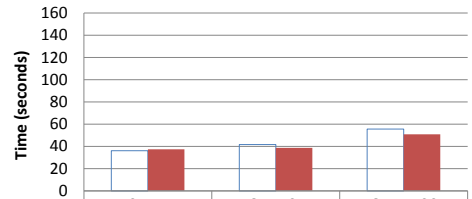
3 Performance

In this section, we present quantitative results exploring the performance portability and scaling of CloverLeaf developed with OPS and compare it to the performance of the various original implementations. Table 1 and Table 2 provide details of the hardware and software specifications of the benchmark systems. The first two systems, Broomway and K20 are single node systems which we use to benchmark the multi-threaded CPU and GPU performance respectively. The



	32 OMP	32 MPI	2OMPx16MPI	OpenCL
OPS	106.32	99.83	102.78	153.49
Original	127.05	103.40	103.93	150.75

(a) Broomway (Intel Xeon E5-2680 CPU)



	CUDA	OpenCL	OpenACC
OPS	36.13	41.64	55.59
Original	37.47	38.77	50.89

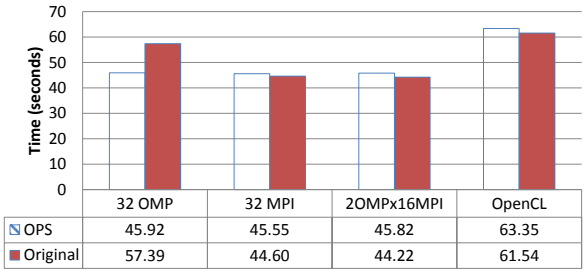
(b) K20 (NVIDIA Tesla K20c)

Fig. 5. CloverLeaf performance - 960×960 mesh (≈ 2955 iterations)

third system is the UK national supercomputing resource - Archer [7] which we use to benchmark OPS's distributed memory performance. The final system is Titan [11], the large scale K20x GPU based Cray XK7 system at ORNL. To be consistent with the compiler flags recommended for gaining accurate results from the original CloverLeaf application, we enforce IEEE floating-point mathematics compliance on each compiler and benchmark[1].

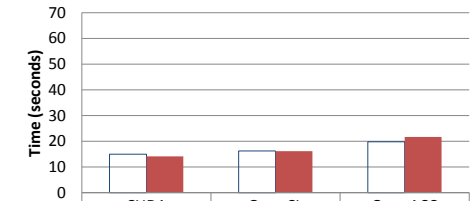
On the single node systems we present the total runtime of the hydro loop of CloverLeaf for the 960×960 (`clover_bm.in`) and 3840×3840 (`clover_bm16_short.in`) mesh input decks. Figure 5 and Figure 6 present times taken by the main hydro iteration loop to solve these problems. The MPI and OpenMP results are from the dual socket Intel CPUs on Broomway while the CUDA and OpenACC results are from the NVIDIA K20c GPU. We also ran the OpenCL version of the application on both the CPU and GPU. To reduce the NUMA effects on performance, both the original and OPS OpenMP versions were executed with the `KMP_AFINITY` environmental variable set to `compact`. We found that this gave the best performance on this two socket CPU node. Additionally, the MPI processes were bound to a specific core using the `numactl` command at runtime, again to reduce NUMA issues on the two socket CPU node.

[1] On Intel compilers, IEEE_FLAGS=-ipo -fp-model strict -fp-model source -prec-div -prec-sqrt



	32 OMP	32 MPI	2OMPx16MPI	OpenCL
OPS	45.92	45.55	45.82	63.35
Original	57.39	44.60	44.22	61.54

(a) Broomway (Intel Xeon E5-2680 CPU)



	CUDA	OpenCL	OpenACC
OPS	15.01	16.27	19.82
Original	14.14	16.19	21.67

(b) K20 (NVIDIA Tesla K20c)

Fig. 6. CloverLeaf performance - 3840×3840 mesh (≈ 87 iterations)

We see that on the Intel CPU node for both problems with the exception of the OpenMP only parallelization, the OPS version executes within 10% of the original implementation's runtime. The OPS's OpenMP parallelization gives better performance. We believe that this is due to OPS explicitly partitioning the iteration space and allocating them to be computed by the available OpenMP threads. In the original version allocating work to threads is handled automatically by OpenMP. The best runtime for the 960×960 mesh is achieved using OPS's pure MPI version, which is about 3% faster than the best runtime achieved with the original MPI version. The OpenCL runtime on the CPUs are about 30% worse than the OpenMP versions, however OPS matches the runtime of the original CloverLeaf OpenCL version. The poor OpenCL performance on the CPU may be due to NUMA affects as the OpenCL runtime does not yet have facilities for explicitly placing and binding threads to cores. A further reason could be poorer vectorization from OpenCL compared to vectorization achieved with SIMD pragmas using the Intel compiler. On the NVIDIA K20c GPU with CUDA, OpenCL and OpenACC all application versions perform approximately the same. The CUDA version gives a speedup of 3× over the best runtime on the two socket Intel CPU node.

The code generated with OPS additionally consists of profiling instrumentation for capturing `ops_par_loop` execution times and achieved bandwidths.

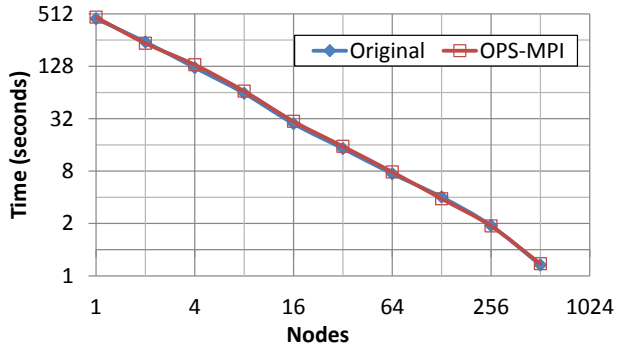
Table 3. Single node performance - 960×960 mesh (≈ 2955 iterations)

Loop	Broomway CPU Node (2 × Intel Xeon E5-2680) $Peak_{Flops} = 345.6GFlops/s$, $Peak_{BW} = 102.4GB/s$ $DGEMM_{Flops} = 304GFlops/s$, $STREAM_{BW} = 78GB/s$					
	32 OpenMP		32 MPI		OpenCL	
	GFlops/s	GB/s	GFlops/s	GB/s	GFlops/s	GB/s
viscosity	81.79	23.72	86.48	20.14	81.86	23.66
accelerate	25.67	43.45	30.42	50.81	21.45	36.17
pdv	48.92	53.55	42.46	47.28	38.59	42.10
ideal_gas	43.92	50.19	57.79	63.29	25.52	29.06
flux_calc	9.44	50.33	12.19	64.27	5.18	27.52
advec_mom	17.04	46.69	21.39	63.79	10.44	28.51
advec_cell	25.70	44.02	30.04	54.88	18.52	31.61

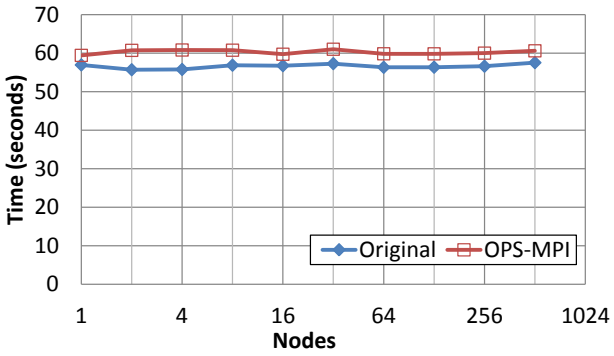
Loop	K20 GPU (NVIDIA K20c) $Peak_{Flops} = 1.17TF/s$, $Peak_{BW} = 208GB/sec$ $DGEMM_{Flops} = 625GFlops/s$, $BWTest_{BW} = 165GB/s$					
	CUDA		OpenCL		OpenACC	
	GFlops/s	GB/s	GFlops/s	GB/s	GFlops/s	GB/s
viscosity	248.55	72.08	250.26	72.57	176.35	51.16
accelerate	90.84	153.75	69.37	117.41	39.36	66.64
pdv	126.78	138.80	122.20	133.78	131.52	103.08
ideal_gas	127.43	145.64	125.71	143.67	52.85	122.82
flux_calc	29.11	155.27	23.66	126.18	25.10	133.93
advec_mom	47.08	129.01	42.52	116.54	39.43	108.10
advec_cell	76.09	130.33	74.53	127.66	64.70	110.85

This information, together with details of approximate number of double precision floating-point operations executed per `ops_par_loop` (gathered through a profiler) enables us to compute the achieved floating-point operation rates and memory bandwidths. Table 3 details this achieved performance per single node on the CPU and GPU systems for each of the related parallelizations. Only the results for the most time consuming routines are given in the table. As a comparison we note the achieved DGEMM (double precision generic matrix-matrix multiply [15]) floating point operation rate on both the CPU and GPU, the STREAM [22] memory bandwidth achieved on the CPU node, and the resulting bandwidth from NVIDIA's bandwidthTest [2] benchmark. The peak achievable performance (Number of Cores×Average frequency×Operations per cycle for Intel CPUs and for NVIDIA K20c GPU [3]), for each platform is also presented.

On the two socket Intel CPU node, Broomway, we see that some loops achieve over 80% of the STREAM memory bandwidth (with MPI). However, only a small fraction out of the 304 GFlops/s DGEMM floating-point operation rate is achieved. On the K20c GPU, the achieved fraction of peak bandwidth is even higher, with loops in `flux_calc` obtaining 155.27 GB/s (with CUDA), which is over 90% of the bandwidth achieved with the bandwidthTest benchmark. Again,



(a) Strong Scaling 15360×15360 mesh - Runtime

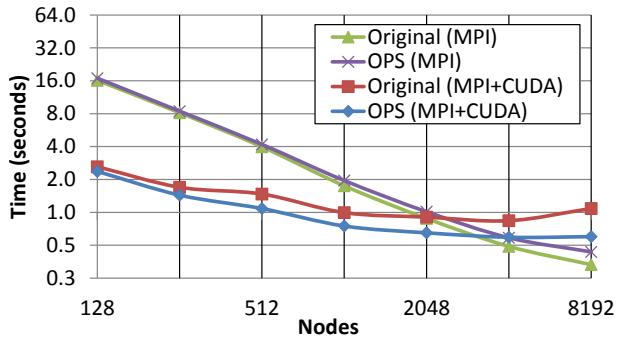


(b) Weak Scaling 2×3840×3840 mesh per node - Runtime

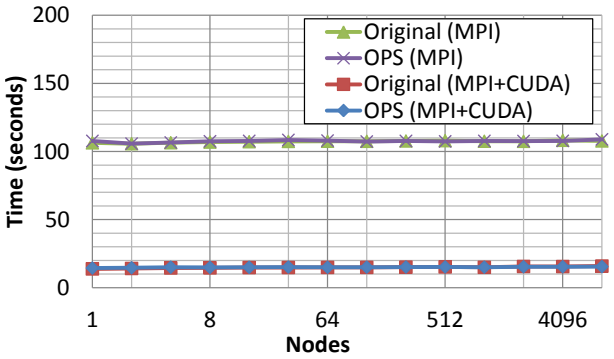
Fig. 7. CloverLeaf scaling performance on Archer (≈ 87 iterations)

the achieved floating-point rate is significantly smaller compared to the GPU's theoretical and practical peak rates. Thus we can say that the CloverLeaf application is much more bandwidth limited, than compute limited. OpenCL parallelization on the CPU performs considerably less well than MPI and OpenMP. However on the K20c GPU, OpenCL was as good as the CUDA implementation.

Next, we benchmark the scaling performance of the distributed memory parallelization, on two large-scale clusters. The first is on Archer, a Cray XC30, on which we benchmark CloverLeaf's pure MPI performance. Figure 7 details the results from this system for both strong scaling and weak scaling on up to 1024 nodes (12,288 cores). The strong scaling mesh consists of 15360^2(≈ 230 million) grid points, while for weak scaling a mesh size of 3840^2 is assigned per socket of a node (i.e. for the 2 socket Archer node a mesh of 2×3840^2 is assigned per node). We see that again, OPS CloverLeaf version's runtime at increasing scale matches that of the original MPI version to within less than 10%. This is true for



(a) Strong Scaling 15360×15360 mesh - Runtime



(b) Weak Scaling 3840×3840 mesh per node- Runtime

Fig. 8. CloverLeaf scaling performance on Titan (≈ 87 iterations) - 1 MPI process per node for MPI+CUDA, 8 MPI processes per node for pure MPI

both configurations. A closer look at the compute time vs communications time reveal that for both strong and weak scaling the time spent in communications, including message set up costs and time to communicate messages is less than 10% of the total run time for any execution on Archer. Profiling the number of MPI messages sent/received in both OPS and original Cloverleaf versions reveal that OPS performs 4× more MPI messages than the original version. This is due to the finer granularity of each `ops_par_loop`, each of which only sends MPI messages for data sets belonging to it. In contrast the original version only does halo exchanges in the `update_halo` routine, aggregating all the MPI messages that need to be sent/received for all subsequent loops. In other words, OPS communicates messages as and when required (i.e. on demand) which only enables a much smaller number of halos to be aggregated.

Figure 8 presents the benchmarking results from Titan. One node in Titan contains one NVIDIA K20x GPU, thus we have allocated one MPI process per

node when executing the MPI+CUDA parallelizations. The figure also plots the run times gained on this system with the MPI only parallelization. In this case, we have allocated 8 MPI processes per node, the reason being that on Titan, there is only one AMD Interlagos CPU consisting of 16 cores, where two cores share one floating point operation unit (FPU). For the weak scaling runs the mesh size allocated per Titan node is 3840×3840 as there is only one CPU socket per node on Titan.

The OPS MPI+CUDA results again match the original CloverLeaf application's hand tuned MPI+CUDA version and demonstrates that the HLA approach to OPS's development has not resulted in any performance degradation. However, comparing OPS's MPI only version to that of the original, OPS loses about 30% performance at 8K nodes. We believe that the reason is due to OPS's on-demand MPI messaging strategy which at the very large scale results in significantly larger number of messages. The latency of these messages dominates the runtime due to the very low amount of compute performed on each MPI process. Currently we are exploring further message aggregation strategies for improving performance of OPS to resolve this issue.

The MPI only version strong-scales better than the MPI+CUDA version, where beyond 2K nodes on Titan, MPI+CUDA does not give any additional speedups. We believe that this is almost certainly due to the cost of the PCIe latencies dominating the computation of the small problems at the higher node sizes. Even using NVIDIA's GPU direct, which can be utilized with OPS for MPI+CUDA applications did not give any notable benefits. The MPI-only versions do not suffer from this issue. However MPI+CUDA achieves a higher speedup (up to 8×) at very low node counts, which then subsequently diminishes at scale. With weak scaling this 8× speedup is maintained at increasing scale. Additionally, at the higher node scales, the same performance loss experienced with OPS when strong-scaling does not occur with weak-scaling. We believe that in this case, the amount of computation carried out per MPI process is large enough to hide the MPI message latencies.

4 Related Work

Several similar research projects have shown the significant benefits of utilizing high-level frameworks such as domain specific languages (DSLs) or active libraries. These include Firedrake [1], FENiCS [25] and Liszt [14], OP2 [6] for unstructured mesh applications and Paraiso [24], Ypnos [26], Pochoir [30] and SBLOCK [12] for explicit stencil based applications (structured mesh applications).

Ypnos [26] is a functional, declarative domain specific language, embedded in Haskell and extends it for parallel structured grid programming. The language introduces a number of domain specific abstract structures, such as *grids* (representing the discrete space over which computations are carried out), *grid patterns* (stencils) etc. in to Haskell, allowing different back-end implementations, such as C with MPI or CUDA. Similarly, Paraiso [24] is a domain-specific

language embedded in Haskell, for the automated tuning of explicit solvers of partial differential equations (PDEs) on GPUs, and multi-core CPUs. It uses algebraic concepts such as tensors, hydrodynamic properties, interpolation methods and other building blocks in describing the PDE solving algorithms. In contrast SBLOCK [12] uses extensive automatic source code generation very much similar to the approach taken by OP2 and OPS, and expresses computations as kernels applied to elements of a set.

Pochoir [30] is a compiler and runtime system for implementing stencil computations on multi-core processors. The main aim of the project is to generate cache efficient multi-threaded CPU code for structured mesh (i.e. stencil) computations. The OPS project also aims to implement cache efficient, "tiling" algorithms through lazy-execution techniques in the future. The work presented in this paper is created from static source-to-source translation techniques to investigate the performance of the resulting code that we believe will be improved via tiling.

Liszt [14] from Stanford University implements a domain specific language (embedded in Scala [4]) for the solution of unstructured mesh based partial differential equations (PDEs). A Liszt application is translated to an intermediate representation which is then compiled by the Liszt compiler to generate native code for multiple platforms. The aim, as with OP2, is to exploit information about the structure of data and the nature of the algorithms in the code and to apply aggressive and platform specific optimizations. Performance results from a range of systems (a single GPU, a multi-core CPU, and an MPI based cluster) executing a number of applications written using Liszt have been presented in [14].

The FEniCS [25] project defines a high-level language, UFL, for the specification of finite element algorithms. The FEniCS abstraction allows the user to express the problem in terms of differential equations, leaving the details of the implementation to a lower level library. Although well established finite element methods could be supported by such a declarative abstraction, it lacks the flexibility offered by frameworks such as OP2 for developing new applications/algorithms. Currently, a runtime code generation, compilation and execution framework that is based on Python, called PyOP2 [27], and a larger framework that supports finite element application development called Firedrake [1, 21] is being developed at Imperial College London.

Another related project of note is Delite [29] a compiler framework and runtime for developing parallel embedded domain-specific languages (DSLs) where the aim is to enable the rapid construction DSLs for a given domain.

5 Conclusions

In this paper, we explored the performance of a Hydrodynamics mini-app called CloverLeaf, after re-engineering it to use the OPS domain specific high-level abstractions framework. OPS provides an API for developing multi-block structured mesh applications and uses code generation techniques to translate an application to a range of parallel implementations.

The OPS based CloverLeaf's performance was compared to that of the various original hand-tuned versions on a number of single-node multi-core/many-core platforms and distributed memory cluster systems. OPS based CloverLeaf's performance on single node systems matched the original versions to within 10% for most parallelizations and sometimes out-performed it by up to about 20%. The achieved memory bandwidth on single node systems showed that the OPS implementations achieve over 80% of the advertised peak bandwidth of each system for some parallel loops. However only a small fraction of the peak floating-point rates are reached on all single node systems. This points to the fact that CloverLeaf is much more constrained by bandwidth than the compute capability of a system. Distributed memory parallelizations on both the Archer (Cray XC30) and Titan (Cray XK7) systems showed excellent scalability, matching that of the original application on both strong- and weak-scaling configurations. However we found that OPS's MPI implementation exchanges about 4× more shorter messages than that of the original. Further MPI message aggregation strategies for OPS are currently being explored to improve strong-scaling performance.

Nevertheless, our experience clearly shows that the development of parallel HPC applications through the careful factorization of a parallel program's functionality and implementation, using a high-level framework such as OPS, is no more time consuming nor difficult than writing a one-off parallel program targeting only a single parallel implementation. However the OPS strategy pays off with a highly maintainable single application source without compromising performance portability on parallel systems on which it will be executed. It also lays the groundwork for providing support for execution on future parallel systems. We believe such an approach will be an essential paradigm shift for utilizing the ever-increasing complexity of novel hardware and software technologies.

6 Acknowledgements

This research is funded by the UK AWE plc. under project "High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems".

The OPS project is funded by the UK Engineering and Physical Sciences Research Council projects EP/K038494/1, EP/K038486/1, EP/K038451/1 and EP/K038567/1 on "Future-proof massively-parallel execution of multi-block applications" and EP/J010553/1 "Software for Emerging Architectures" (ASEArch) project. This paper used the Archer UK National Supercomputing Service from time allocated through UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on "Multi-layered Abstractions for PDEs".

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Cloverleaf development is supported by the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Pro-

grame) and also the Royal Society through their Industry Fellowship Scheme (IF090020/AM).

We are thankful to Endre László at PPKE Hungary for his contributions to OPS, David Beckingsale at the University of Warwick and Michael Boulton at the University of Bristol for their insights into the original CloverLeaf application and its implementation.

References

1. The Firedrake Project, http://www.firedrakeproject.org/
2. Nvidia CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/cuda-samples/#bandwidth-test
3. Nvidia Tesla Kepler Family Datasheet, http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf
4. The SCALA Programming Language, http://www.scala-lang.org/
5. The Mantevo Project (2012), http://mantevo.org/
6. OP2 for Many-Core Platforms (2013), http://www.oerc.ox.ac.uk/research/op2
7. Archer - UK national high performance computing facility (2014), http://www.archer.ac.uk/
8. AWE cloverleaf (2014), http://warwick-pcav.github.io/CloverLeaf/
9. The montblanc project (2014), http://www.montblanc-project.eu/
10. OPS for Many-Core Platforms (2014), http://www.oerc.ox.ac.uk/projects/ops
11. Titan cray xk7 (2014), https://www.olcf.ornl.gov/titan/
12. Brandvik, T., Pullan, G.: SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms. In: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology. pp. 1181–1188. CIT '10, IEEE Computer Society, Washington, DC, USA (2010)
13. Czarnecki, K., Eisenecker, U.W., Glück, R., Vandevoorde, D., Veldhuizen, T.L.: Generative Programming and Active Libraries. In: Selected Papers from the International Seminar on Generic Programming. pp. 25–39. Springer-Verlag, London, UK (2000)
14. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: a Domain Specific Language for Building Portable Mesh-based PDE Solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 9:1–9:12. SC '11, ACM, New York, NY, USA (2011)
15. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. 16(1), 1–17 (Mar 1990), http://doi.acm.org/10.1145/77626.79170
16. Gaudin, W., Mallinson, A., Perks, O., Herdman, J., Beckingsale, D., Levesque, J., Jarvis, S.: Optimising Hydrodynamics applications for the Cray XC30 with the application tool suite. In: The Cray User Group 2014, May 4-8, 2014, Lugano, Switzerland (2014)
17. Herdman, J.A., Gaudin, W.P., McIntosh-Smith, S., Boulton, M., Beckingsale, D.A., Mallinson, A., Jarvis, S.: Accelerating hydrocodes with openacc, opecl and cuda. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:. pp. 465–471 (Nov 2012)

18. Howes, L.W., Lokhmotov, A., Donaldson, A.F., Kelly, P.H.J.: Deriving Efficient Data Movement from Decoupled Access/Execute Specifications. In: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers. pp. 168–182. HiPEAC '09, Springer-Verlag, Berlin, Heidelberg (2009)
19. Lindtjorn, O., Clapp, R., Pell, O., Fu, H., Flynn, M., Fu, H.: Beyond traditional microprocessors for geoscience high-performance computing applications. Micro, IEEE 31(2), 41 –49 (March - April 2011)
20. Mallinson, A., Beckingsale, D., Gaudin, W., Herdman, J., Jarvis, S.: Towards portable performance for explicit hydrodynamics codes. In: International Workshop on OpenCL (IWOCL 13). Atlanta, USA. (May 2013)
21. Markall, G.R., Slemmer, A., Ham, D.A., Kelly, P.H.J., Cantwell, C.D., Sherwin, S.J.: Finite element assembly strategies on multi- and many-core architectures. International Journal for Numerical Methods in Fluids 71, 80–97 (2013), http://dx.doi.org/10.1002/fld.3648
22. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (dec 1995)
23. 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), 669–692 (2013)
24. Muranushi, T.: Paraiso : An Automated Tuning Framework for Explicit Solvers of Partial Differential Equations. Computational Science & Discovery 5(1), 015003 (2012)
25. Ølgaard, K.B., Logg, A., Wells, G.N.: Automated Code Generation for Discontinuous Galerkin Methods. CoRR abs/1104.0628 (2011)
26. Orchard, D.A., Bolingbroke, M., Mycroft, A.: Ypnos: Declarative, Parallel Structured Grid Programming. In: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming. pp. 15–24. DAMP '10, ACM, New York, NY, USA (2010)
27. Rathgeber, F., Markall, G.R., Mitchell, L., Loriant, M., Ham, D.A., Bertolli, C., Kelly, P.H.J.: PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:. pp. 1116–1123 (2012)
28. 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. (under review) ACM Transactions on Parallel Computing (2013), available at http://arxiv-web3.library.cornell.edu/abs/1403.7209
29. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Transactions on Embedded Computing Systems (TECS) 13(4s), 134 (2014)
30. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The pochoir stencil compiler. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 117–128. SPAA '11, ACM, New York, NY, USA (2011)
31. Veldhuizen, T.L., Gannon, D.: Active Libraries: Rethinking the roles of compilers and libraries. In: In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO98). SIAM Press (1998)