Contents lists available at ScienceDirect

Computers and Fluids

journal homepage: www.elsevier.com/locate/compfluid

Productivity, performance, and portability for computational fluid dynamics applications

István Z. Reguly[a,*], Gihan R. Mudalige[b]

[a] Faculty of Information Technology and Bionics, Pazmany Peter Catholic University, Budapest, Hungary
[b] Department of Computer Science, University of Warwick, Coventry, UK

A R T I C L E I N F O

Article history:
Received 30 April 2019
Revised 1 November 2019
Accepted 2 January 2020
Available online 5 January 2020

Keywords:
Review
Domain specific language
Performance
Portability
Productivity

A B S T R A C T

Hardware trends over the last decade show increasing complexity and heterogeneity in high performance computing architectures, which presents developers of CFD applications with three key challenges; the need for achieving good performance, being able to utilise current and future hardware by being portable, and doing so in a productive manner. These three appear to contradict each other when using traditional programming approaches, but in recent years, several strategies such as template libraries and Domain Specific Languages have emerged as a potential solution; by giving up generality and focusing on a narrower domain of problems, all three can be achieved.

This paper gives an overview of the state-of-the-art for delivering performance, portability, and productivity to CFD applications, ranging from high-level libraries that allow the symbolic description of PDEs to low-level techniques that target individual algorithmic patterns. We discuss advantages and challenges in using each approach, and review the performance benchmarking literature that compares implementations for hardware architectures and their programming methods, giving an overview of key applications and their comparative performance.

1. Introduction

The hardware architectures for parallel high performance scientific computing continue to undergo significant changes. More than a decade and a half has passed since the end of CPU clock frequency scaling. This way-point for CMOS-based micro-processors, also known as the end of Dennard's scaling has resulted in a golden age for processor architecture design as increasingly complex and innovative designs are utilized to continue delivering performance gains. The primary trend has been to develop increasingly massively parallel architectures with the implicit assumption that more discrete units can do more work in parallel to deliver higher performance by way of increased throughput. As a result, we see a continuation of Moore's law - exponentially increasing transistor numbers on a silicon processor - but configured in increasing numbers of discrete processors cores. Consequently, on the one hand we see current traditional processors continuing to gain more and more cores, currently over 20 cores for high-end processors - each with larger vector units (512 bits on Intel's latest chips). On the other hand we see the widespread adoption of separate computational accelerators that excel at specific workloads, such as GPUs, with larger number of low-frequency cores, or the emergence of heterogeneous processors.

While more cores have become commonplace, feeding them with data has become a bottleneck. As the growth in the speed of memory units has lagged that of computational units, multiple levels of memory hierarchy, with significant chunks of silicon dedicated to caches to bridge the bandwidth/core-count gap have been designed. New memory technologies such as HBM and HBM2 on Intel's Xeon Phi and NVIDIA's Tesla GPUs, for example, have produced “stacked memory” designs where embedded DRAM is integrated onto CPU chips. For large datasets in particular, new non-volatile memory is becoming available. Examples include Intel's 3D Xpoint (Optane) memory, which can be put in traditional DIMM memory slots, and can be used just like traditional memory that has much higher capacity, but lower bandwidth. Supercomputers built for scientific computing have also become increasingly heterogeneous - 7 out of the 10 top machines in the world have some type of accelerator. This has led to the need to support a heterogeneous set of architectures for continued scientific delivery.

The root cause of this issue, the switch to parallelism, was aptly described by David Patterson as a Hail Mary pass, an act done

* Corresponding author.
E-mail addresses: reguly.istvan@itk.ppke.hu (I.Z. Reguly), g.mudalige@warwick.ac.uk (G.R. Mudalige).

https://doi.org/10.1016/j.compfluid.2020.104425
0045-7930/© 2020 Elsevier Ltd. All rights reserved.

in desperation, by the hardware vendors “without any clear notion of how such devices would in general be programmed” [1]. The significant impact of this decision has today changed conventional wisdoms in programming parallel high-performance computing systems [2]. If we specifically focus on Computational Fluid Dynamics (CFD), there are indeed a large number of codes ported to utilize GPUs [3–10]. These efforts focusing on migrating a codebase to a particular programming abstraction (such as CUDA or OpenACC), which does enable them to exploit GPUs, but also locks them to that architecture. For most large code-bases, maintaining two or more versions (one for CPUs and another for GPUs, etc.) is simply not a reasonable option. These challenges bring us to three key factors that should be considered when developing, or maintaining large CFD codes, particularly production codes:

1. Performance: running at a reasonable fraction of peak performance on given hardware.
2. Portability: being able to run the code on different hardware platforms/architectures with minimum manual modifications.
3. Productivity: the ability to quickly implement new applications, features and maintain existing ones.

Time and again we have seen that a general solution that delivers all three is simply not possible, programming approaches have to choose a point on this triangle. Attempts for compilers delivering some form of universal auto-parallelisation capability for general-purpose languages have consistently failed [11]; given the imperative nature of languages such as C or Fortran, compilers struggle to extract sufficient semantic information (enabling them to safely parallelize a program) from all but the simplest structures. This means that the burden is increasingly pushed onto the programmer to help compilers exploit the capabilities of the latest and purportedly greatest hardware. To make things worse, different hardware come with different low-level programming languages or extensions, and compilers.

It is of course unreasonable to expect from scientists/engineers to gain a deep understanding of the hardware they are programming for - especially given the diversity of HPC systems - and to keep re-implementing science codes for various architectures. This has led to a *separation of concerns* approach where description of what to compute is separated from how that computation is implemented. This notion is in direct contrast to the commonly used programming languages such as C or Fortran, which are inherently imperative. For example, a for/do loop written in C/Fortran explicitly describes the order in which iterations have to be executed.

Research and development of software and tools used in CFD therefore has been pushed to target individual problem domains, restricting generality, but being able to address performance, portability, as well as productivity. Classical software libraries target a small set of algorithms, such as sparse linear algebra, present a simple Application Programming Interface (API), and are highly tuned for a set of target hardware. For wider algorithmic classes, such as neighbourhood-based (stencil) operations over structured blocks or tensors, domain specific approaches, such as Domain Specific Languages (DSLs) have been developed to help separate the algorithmic description from the actual parallel implementation. At an even higher level, techniques such as DSLs have been created to allow the abstract, mathematical expression of partial differential equations - these then offer a number of discretisation and numerical algorithms to solve them.

The challenge facing CFD developers is multifaceted; have as much performance, portability and productivity as possible, by means ranging from picking an off-the-shelf solution, to going with traditional general-purpose languages and programming everything from the ground-up. The choice of the right tools is crucially important: selecting a tool/framework with just the right level of abstraction to allow the level of control desired, but remaining productive. Tools, particularly academic tools, do not always have a clear sustainability and software maintenance model. As such it may be more difficult to plan a longer-term strategy with them.

In this paper, we aim to review some of the approaches and tools that can be used to develop new CFD codes or to modernize existing ones; we take a brief look at general-purpose programming languages and parallelization approaches in Section 2, then discuss software libraries targeting some of the most common algorithmic classes for CFD in Section 3. The common property of libraries in this class is the large amount of readily-available numerical algorithms - which may then be customised to various degrees. In Section 4 we review some of the most established C++ template based performance portability libraries, which target general data-parallel or task-parallel algorithms, and themselves have few numerical algorithms implemented. We then move on to Domain Specific Languages targeting the common computational patterns in CFD in Section 5.

2. General-purpose programming approaches

In this class we consider programming APIs and extensions that have the widest scope, and allow fine control over the parallelisation of arbitrary algorithms.

There are a number of competing and complementary approaches to writing code for parallel hardware, which all place themselves at various points on the productivity-portability-performance triangle. There are general purpose programming languages such as C/C++/Fortran or extensions to such languages (e.g. CUDA) or libraries such as Pthreads that give fine-grained control over parallelism and concurrency. These allow the programmer to extract the maximum performance out of the hardware, but of course they are neither portable nor productive.

Directive-based approaches, such as OpenMP and OpenACC sacrifice some generality and fine-grained control for being significantly more productive to use - indeed, these are the two most widespread approaches to programming multi-core CPUs and GPUs in high performance computing. OpenMP has historically targeted CPUs, and aside from direct data parallelism, it has strong tools to handle concurrency. Although, OpenMP does support an offload model as of version 4.0 to run on GPUs, portability is still an issue; in practice the same directives and constructs cannot be used to target *both* the CPU and the GPU for most cases - however, this is an aspect that is being improved by implementations, and may well be a good approach in the future. OpenACC is targeting accelerators, GPUs mainly, and the offload model is fundamental to it. The standard does allow for targeting CPUs as well, but the only compiler supporting this is PGI, (owned by NVIDIA).

The OpenCL standard was introduced to address some of the portability issues - and in some respect, again it pushes the performance vs. portability trade-off onto the programmer. While it does allow fine-grained control over parallelism and concurrency, codes that do exploit this become less portable. OpenCL also struggles with productivity: it has a verbose API, which makes it more difficult to use. Additionally, support for OpenCL by various hardware vendors is mixed; NVIDIA only supports version 1.2, and Intel has varying degrees of support for its Xeon processors, and the Knights Landing Xeon Phi.

An emerging standard is SYCL [12], which can be thought of as an improved C++ version of OpenCL. In SYCL, much of the concepts remain the same, but it is significantly easier to use than OpenCL and uses a heavily templated C++ API. Naturally, similar to OpenCL, code will be portable to different platforms, but not necessarily performance portable as it has been shown for OpenCL [13–15]. SYCL may become a key standard with Intel’s introduction of OneAPI, based on SYCL, and the Xe GPU platform, which is to form a key part of the upcoming Aurora exascale supercomputer [16]. While CUDA, OpenMP, and OpenACC all support C/C++ as well as Fortran, OpenCL and SYCL do not, limiting its use in the CFD field, which still heavily uses Fortran. If indeed C/C++ based extensions and frameworks dominate the parallel programming landscape for emerging hardware, there could well be a need for porting existing Fortran based CFD applications to C/C++.

The key challenges when using general-purpose approaches include:

1. The parallel implementation tends to be very prescriptive - the more efficient an implementation is on a given hardware, the less (performance) portable it is.
2. Keeping track of and maintaining specialised code paths for different hardware.
3. Parallel implementation and data structures intertwined with science code, making it more difficult to understand and maintain.

3. Classical software libraries

In this class, we consider software and libraries that target CFD application areas, and themselves implement a diverse set of numerical algorithms.

Off-the-shelf software, such as commercial offerings from Ansys, Fluidyna, Simscale and many others arguably give the most productive approach to setting up and running CFD simulations, and they have been optimised extensively for CPUs, with certain modules offering GPU support as well. Open source packages such as OpenFOAM also give access to an array of features, though they tend to be less optimised and GPU support is sporadic [17,18], and generally not officially supported. These packages however limit the exploration of new algorithmic techniques and numerical methods, simply because either they are closed source, or they are difficult to modify - as such they lie outside the focus of our discussion in this paper.

Perhaps the largest software package, or in fact collection of packages, is the Trilinos project [19] from Sandia National Labs. Trilinos's primary goal is to offer tools for the Finite Element Method. It contains a number of capability areas; sparse linear solvers, meshing tools, parallel programming tools (such as Kokkos, discussed in the next section), discretisations, and many others. Most of these tools support distributed memory systems and classical CPU architectures, and there is increasing portability support relying on Kokkos - such as Tpetra, which can parallelise an increasing number of sparse linear algebra operations with OpenMP and CUDA [20].

A similarly prominent library is PETSc [21]. It is a library providing data structures and algorithms to assist in solving PDEs with a main focus on providing scalable linear solvers. It provides a large set of algorithms for the iterative solution of sparse linear systems, as well as some non-linear solvers. These algorithms are easy to use, and are quite robust, and have been tested and evaluated on millions of CPU cores. There is also increasing support for GPUs, with solvers based on vector and sparse matrix-vector multiplication primitives being supported, and more and more preconditioners also being added.

Most classical software libraries focus on the solution of linear systems, as the variety of algorithms, and especially the application programming interface exposed towards the user is tractable. There is a large number of such libraries that make the complex step of linear solve easily accessible - indeed in most CFD applications that use implicit methods, this step is the most time-consuming. These libraries have been heavily optimised, and the dense solvers in particular achieve a high percentage of peak machine performance. Portability remains an issue, as there is only a handful of libraries supporting GPUs. Aside from the standardised BLAS and LAPACK interfaces, most libraries have their own APIs, which makes swapping them out cumbersome.

Libraries that target dense matrix algorithms are well-established, and they often use the BLAS and LAPACK interfaces. LAPACK [22] and ScaLAPACK [23] target classical CPUs and homogeneous clusters. The PLASMA [24] library focuses on dense matrix factorisations, and introduced task based parallel execution to address the inhomogeneity in computations as well as hardware - though currently only CPUs are supported. MAGMA [25] on the other hand is the most capable dense solver package that supports GPUs - it uses an interface similar to LAPACK to better enable porting applications to use heterogeneous architectures.

Considering the inexact nature of most sparse linear solvers, there is a much richer set of algorithms and libraries and consequently programming interfaces. The aforementioned PETSc [21] and Trilinos [19] provide a wide range of functionality. There are libraries that rely heavily on C++ templates and metaprogramming to support diverse datatypes and optimisations - Armadillo [26] and Eigen [27] are such examples; these focus on classical CPU clusters. The SuiteSparse [28] library additionally supports routines on the GPU. A particularly important class of algorithms are Algebraic Multigrid methods, which is the main focus of the hypre [29] library, and the AGMG [30] library, which support CPUs only - the AmgX [31] library makes these algorithms available on the GPU as well. Another significant class of sparse linear solvers are direct solver algorithms, there are a number of libraries that provide an implementation for this, including WSMP [32], SuperLU [33], PaStiX [34], MUMPS [35], DSCPACK [36], and some include GPU implementations as well [37,38].

For the linear solution phase of applications, one should therefore use an established library in the vast majority of cases - this is the most productive approach, and these implementations also deliver high performance. Key challenges when using classical software libraries include:

1. Standardised interfaces: since most libraries only target a single hardware platform, the ability to swap out one library for another with relatively little work is important.
2. Integration with other parts of a code base: particularly with deep integration, with frequent interactions between the two parts, the efficiency of communication is important (e.g. repeated CPU-GPU memory transfers can become a bottleneck).

4. C++ template libraries

For this group, we consider libraries that facilitate the scheduling and execution of data parallel or task-parallel algorithms in general, but themselves do not implement numerical algorithms.

In sharp contrast to linear solution algorithms, the variety in the *rest* of the numerical parts of a CFD application (explicit methods, matrix assembly algorithms, etc.) is just too large to be reasonably handled with a classical library approach. Traditionally these parts of the code were written by hand in Fortran/C/C++, and oftentimes hand-ported to use OpenMP, OpenACC, CUDA, OpenCL, or similar. Clearly, having multiple variants of the same code with different parallelisations is untenable long-term. Increasingly, given the diversity in parallelisation approaches and hardware architectures, there is a need to regain productivity by separating the parallelisation concerns from the numerical algorithms. One such approach, exclusive to C++, is template libraries, which allow users to express algorithms as a sequence of parallel primitives executing user-defined code at each iteration. These libraries follow the design philosophy of the C++ Standard Template

Library [39]–indeed, their specification and implementation is often considered as a precursor towards inclusion in the C++ STL. The largest such projects are Boost [40], Eigen [27], and focusing on the parallelism aspect is HPX [41]. While there are countless such libraries, here we focus on ones that also target performance portability.

Kokkos [42] is a C++ performance portability layer that provides data containers, data accessors, and a number of parallel execution patterns. It supports execution on shared-memory parallel platforms, namely CPUs using OpenMP and Pthreads, and NVIDIA GPUs using CUDA. It does not consider distributed memory parallelism, rather it is designed to be used in conjunction with MPI. Kokkos ships with Trilinos, and is used to parallelise various libraries in Trilinos, but it can also be used as a stand-alone tool - it follows the design philosophy of the C++ STL very closely. Its data structures can describe where data should be stored (CPU memory, GPU memory, non-volatile, etc.), how memory should be laid out (row/column-major, etc), and how it should be accessed. Similarly, one can specify where algorithms should be executed (CPU/GPU), what algorithmic pattern should be used (parallel `for`, reduction, tasks), and how parallelism is to be organised. It is a highly versatile and general tool capable of addressing a wide set of needs, but as a result is more restricted in what types of optimisations it can apply compared to a tool that focuses on a more narrower application domain.

RAJA [43] is another C++ performance portability layer, from LLNL, which in many respects is very similar to Kokkos but it offers more flexibility for manipulating loop scheduling, particularly for complex nested loops. It also supports CPUs (with OpenMP and TBB), as well as NVIDIA GPUs with CUDA.

Both Kokkos and RAJA were designed by US DoE labs to help move existing software to new heterogeneous hardware, and this very much is apparent in their design and capabilities - they can be used in an iterative process to port an application, loop-by-loop, to support shared-memory parallelism. Of course, for practical applications, one needs to convert a substantial chunk of an application; on the CPU that is because non-multithreaded parts of the application can become a bottleneck, and on the GPU because the cost of moving data to/from the device.

There are a number of further libraries that use C++ templates to provide portability across different architectures, but they focus on narrower application domains, and are discussed in the next section.

Key challenges when using C++ template libraries for data parallelism include:

1. Development time and difficulty often increased by hard to read errors, and high compilation times.
2. Debugging heavily templated code is challenging.
3. Managing platform-specific code paths.

5. DSLs and eDSLs

In this category we consider a wide range of languages and libraries - the key commonality is that their scope is limited to a particular application or algorithmic domain. We only discuss DSLs that either specifically target CFD, or support basic algorithmic patterns most common in CFD.

Domain Specific Languages (DSLs) and more generally domain specific approaches, by definition, restrict their scope to a narrower problem domain, or set of algorithmic patterns. By sacrificing generality, it becomes feasible to attempt and address challenges in gaining all three of performance, portability, and productivity. There is a wide range of such approaches, starting from libraries focused on solving the Finite Element Method, to libraries focused on a handful of low-level computational patterns. Some embed themselves in a host languages (eDSLs), such as Python or C++, others develop a new language of their own.

By being more focused on an application domain, these libraries and languages are able to apply much more powerful optimisations to help deliver performance as well as portability, and because a lot of assumptions are already built into the programming interface, much less has to be described explicitly - leading to better productivity. The Achilles heel of these approaches stems from their limited applicability - if they cannot develop a considerable user base, they will become mere academic experiments that are forgotten quickly. They need to develop a community around them to help support and maintain in the long run. Therefore, there are two key challenges to building a successful DSL or library:

1. An abstraction that is wide enough to cover a range of interesting applications, and narrow enough so that powerful optimisations can be applied.
2. An approach to long-term support.

5.1. Algorithmic skeletons

A large fraction of algorithms can be considered as a sequence of basic algorithmic primitives - such as parallel for-each loops, reductions, scan operations, etc. This is a very mechanical approach to expressing algorithms, but by doing so, it forces computations into a form that is easy to parallelise. Yet, often times they are not trivial to read or write. Kokkos and RAJA can be thought of skeleton libraries supporting a small set of skeletons.

Thrust [44] (active) is a C++ template library developed by NVIDIA, for shared memory parallelism, supporting both CPUs (relying on TBB) and GPUs (with CUDA), and offers basic vector containers and a large number of parallel algorithms. SkePU 2 [45] (active) relies on C++ templates to target CPUs and GPUs (with experimental MPI support), vector and matrix data structures, and the map, reduce, map-reduce, map-overlap (stencil-type), and scan algorithmic primitives. FastFlow [46] (active) is a C++ parallel programming framework, targeting CPUs and NVIDIA GPUs, with a particular focus on controlling memory and caches.

Muesli [47] (stale, updated No. 2016) is a C++ template library targeting CPUs, GPUs, as well as MPI to support both distributed and shared memory parallelism. It supports vectors and matrices, with map, zip, fold, mapStencil algorithmic primitives. SkelCL [48] (stale, updated Sept 2016) is embedded in C++, and uses OpenCL to target various devices to exploit shared memory parallelism with vector and matrix containers, and algorithms such as map, map-reduce, reduce, scan and zip.

Furthermore, there are similar skeleton DSLs embedded into functional programming languages, such as Haskell. Functional languages have powerful tools to apply transformations and optimisations to high-level algorithms, as by design they describe what to do instead of how, yet they are significantly different from conventional (imperative) languages such as C/Fortran. Given a general lack of expertise and awareness of these languages in engineering, the utility of these for CFD is limited. It is also non-trivial to integrate these with commonly used libraries, data formats, and post-processing. Examples of DSLs targeting HPC in Haskell by supporting high-performance parallel arrays include Accelerate [49] (active), targeting GPUs and CPUs, and Repa [50] (active), targeting CPUs. hmatrix [51] (active) targets BLAS and LAPACK operations on CPUs.

5.2. DSLs For stencil computations

Another set of DSLs focus on making the description of structured or unstructured stencil-based algorithms more productive. This class of DSLs are for the most part oblivious to numerical algorithm being implemented, which in turn allows them to be used for a wider range of algorithms - e.g. finite differences, finite volumes, or finite elements. The key goal here is to create an abstraction that allows the description of parallel computations over either structured or unstructured meshes (or hybrid meshes), with neighbourhood-based access patterns. Similar DSLs can be constructed for other domains, such as molecular dynamics that help express N-body interactions, but these have limited use in CFD.

One of the first such DSLs to target CPUs as well as GPUs was Liszt [52] (stale, updated 2013) which defined its own language for expressing unstructured mesh computations. The research effort was continued and generalised to support arbitrary meshes as well as particles with Ebb [53] (stale, updated July 2016), which is embedded in the Lua and Terra languages.

Nebo [54], part of SpatialOps (stale, last updated No. 2017) targets transport phenomena on structured meshes with a DSL embedded in C++ – it targets CPUs, GPUs. Halide [55] (active) is a DSL intended for image processing pipelines, but generic enough to target structured-mesh computations [56], it has its own language, but is also embedded into C++ – it targets both CPUs and GPUs, as well as distributed memory systems. YASK [57] (active) is a C++ library for automating advanced optimisations in stencil computations, such as cache blocking and vector folding. It targets CPU vector units, multiple cores with OpenMP, as well as distributed-memory parallelism with MPI. OPS [58] (active) is a multi-block structured mesh DSL embedded in both Fortran and C/C++, targeting CPUs, GPUs and MPI - it uses a source-to-source translation strategy to generate code for a variety of parallelisations. ExaSlang [59] (active) is part of a larger European project, Exastencils, which allows the description of PDE computations at many levels - including at the level of structured-mesh stencil algorithms. It is embedded in Scala, and targets MPI and CPUs, with limited GPU support. Another DSL for stencil computations, Bricks [60] gives transparent access to advanced data layouts using C++, which are particularly optimised for wide stencils, and is available on both CPUs, and GPUs.

For unstructured mesh computations, OP2 [61] (active) and its Python extension, PyOP2 [62] (active) give an abstraction to describe neighbourhood computations, they are embedded in C/Fortran and Python respectively, and target CPUs, GPUs, and distributed memory systems.

For mixed mesh-particle, and particle methods, OpenFPM [63] (active), embedded in C++, provides a comprehensive library that targets CPUs, GPUs, and supercomputers.

A number of DSLs have emerged from the weather prediction domain such as STELLA [64] (active) and PSyclone [65] (active). STELLA, a C++ template library for stencil computations, that is used in the COSMO dynamical core [66], and supports structured mesh stencil computations on CPUs and GPUs. PSyclone is part of the effort in modernizing the UK MetOffice's Unified Model weather code and uses automatic code generation. It currently uses only OpenACC for executing on GPUs. A very different approach is taken by the CLAW-DSL [67] (active), used for the ICON model [68], which is targeting Fortran applications, and generates CPU and GPU parallelisations - mainly for structured mesh codes, but it is a more generic tool based on source-to-source translation using preprocessor directives. It is worth noting that these DSLs are closely tied to a larger software project (weather models in this case), developed by state-funded entities, greatly helping their long-term survival. At the same time, it is unclear if there are any other applications using these DSLs.

5.3. High-level DSLs for PDEs

There is a specific class of DSLs that target the solution of PDEs starting at the symbolic expression of the problem, and (semi-)automatically discretise and solve them. Most of these are focused on a particular set of equations and discretisation methods, and offer excellent productivity - assuming the problem to be solved matches the focus of the library.

Many of these libraries, particularly ones where portability is important, are built with a layered abstractions approach; the high-level symbolic expressions are transformed, and then passed to a layer that maps them to a discretisation, then this is given to a layer that arranges parallel execution - the exact layering of course depends on the library. This approach allows the developers to work on well-defined and well-separated layers, without having to gain a deeper understanding of the whole system. These libraries are most commonly embedded in the Python language, which has the most commonly used tools for symbolic manipulation in this field - although functional languages are arguably better suited for this, they still have little use in HPC. Due to the poor performance of interpreted Python, these libraries ultimately generate low-level C/C++/Fortran code to deliver high performance.

One of the most established such libraries is FEniCS [69] (active), which targets the Finite Element Method, however it only supports CPUs and MPI. Firedrake [70] (active) is a similar project with a different feature set, which also only supports CPUs - it uses the aforementioned PyOP2 library for parallelising and executing generated code.

The ExaStencils project [71] (active) uses 4 layers of abstraction to create code running on CPUs or GPUs (experimental) from the continuous description of the problem - its particular focus is structured meshes and multigrid. Saiph [72] (active) is a DSL embedded in Scala, developed at BSC, designed to target a wider range of PDEs and discretisation methods, though it is currently in an early stage, supporting finite differences and CPUs (with MPI and OpenMP) only. CDFLang [73] (stale, last updated Jan 2018) defines its own language for a set of tensor operations, which it then transforms and generates C/C++ code parallelised with OpenMP, targeting CPUs only.

DUNE/PDELab [74] (active) is another high-level library that allows the description of PDEs and their discretisation with various methods, on both structured and unstructured grids. It is embedded in C++, and its current version only supports MPI and CPUs, but research and development work towards GPU support is ongoing. OpenSBLI [75] (active) is a DSL embedded in Python, focused on resolving shock-boundary layer interactions and uses finite differences and structured meshes - it generates C code for the OPS library, which in turn can parallelise it on distributed memory machines with both CPUs and GPUs. Devito [76] (active) is a DSL embedded in Python which allows the symbolic description of PDEs, and focuses on high-order finite difference methods, with the key target being seismic inversion applications.

The most common challenges when using DSLs include:

1. In some cases, debugging can be difficult due to all the extra hidden layers of software between user code and code executing on the hardware.
2. Extensibility - implementing algorithms that fall slightly outside of the abstraction defined by the DSL can be an issue.
3. Customisability - it is often difficult to modify the implementation of high-level constructs generated automatically.

6. Characterising performance, portability, and productivity

In this section, we explore the literature that discusses how to quantify performance, portability, and productivity, and discuss a number of papers which present performance results on the libraries above. Note that given the huge body of research on performance, we restrict this discussion to papers and libraries that consider at least CPUs and GPUs, and therefore have a meaningful discussion of portability as well. Additionally, we focus on work that presents results from either production applications, or at least proxy codes of large applications.

6.1. Metrics

Quantifying even one of the three factors is exceedingly difficult. Perhaps the easiest of the three is performance - one can try and determine how efficiently a code uses the hardware it is running on. The Roofline model [77] captures efficiency based on arithmetic intensity - how many operations are carried out for each byte of data moved, and how close this is to the theoretical maximum on any given hardware architecture. As such, it sets an aspirational goal, and for complex codes (and irregular algorithms in particular), only a low fraction can be reasonably achieved. Given some reference implementations of an algorithm one can also calculate the fraction of the "best known performance".

Based on this performance metric, the definition of performance portability, as defined in Pennycook et al. [78]: "A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set.". The metric described in the paper gives a single value, $P(a, p, H)$, as a function of a given application a, running a given problem p, on a set of hardware/software platforms of interest H (where $|H|$ is the number of platforms).

$$\Psi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i\in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

which is the harmonic mean of performance efficiencies $e_i(a, p)$ on each platform. There are two common metrics for performance efficiency on a given hardware (e_i): as a fraction of some peak theoretical performance (e.g. bandwidth of computational throughput), or as a fraction of "best known performance" on the given platform. Clearly, comparing the results of this metric from different applications, and from different hardware sets is hardly objective - therefore in this paper we do not attempt to directly compare and rank libraries and software based on their published performance or portability.

The performance portability metric does not consider productivity - in the extreme case, it still considers completely separate implementations and optimisations of the same applications as one. For obvious reasons, working with such a code base is very unproductive. A "code divergence" metric was proposed by Harrell et al. [79], which quantifies the difference, relying on the number of different lines of code, between variants targeting different platforms. Code divergence D on a set of code variants is defined as follows:

$$D(A) = \binom{|A|}{2}^{-1} \sum_{\{a_i,a_j\}\subset A} d(a_i, a_j), \tag{2}$$

giving the average pairwise distances between all the variants in A (where $|A|$ is the number of variants). d is defined as the change in the number of source lines of code (SLOC):

$$d(a, b) = \frac{|SLOC(a) - SLOC(b)|}{min(SLOC(a), SLOC(b))}. \tag{3}$$

An open source tool, the Code Base Investigator [80] can be used to calculate this metric.

6.1.1. Comparative works

Unfortunately there are only a handful of works that evaluate the above metrics on codes or applications of interest to the CFD community - below we summarise their results, and also present them in tabular format in Table 1.

CloverLeaf is a Lagrangian-Eulerian explicit hydrodynamics mini application, with both 2D and 3D variants. It has been a target of several papers focusing on performance. The most recent and relevant works that discuss both performance and portability include McIntosh-Smith [81], exploring a wide variety of architectures including Intel, ARM, AMD, IBM CPUs, NVIDIA and AMD GPUs, as the NEC Aurora. They evaluate OpenMP, Kokkos, OpenACC, CUDA, and OpenCL - and report mixed results. OpenMP supports 7 out of the 12 platforms with a high performance portability score (1.0, based on fraction of best observed performance), OpenCL runs only on GPUs with a score of 0.945, OpenACC runs on NVIDIA GPUs, x86 CPUs, and Power CPUs only, with a score of 0.624, and the Kokkos implementation of CloverLeaf was only working on NVIDIA GPUs, achieving a score of 0.628.

BookLeaf is an unstructured compressible hydrodynamics proxy application, its performance and portability was evaluated in detail

Table 1
CFD software and papers with performance portability studies.

Application	Paper	Parallelisations	Performance measures
CloverLeaf	[81]	OpenMP, Kokkos, OpenACC, CUDA, OpenCL	Time, Ψ
CloverLeaf	[82,83]	OPS (OpenMP, CUDA)	Time, GB/s, GFLOPS/s
BookLeaf	[84]	MPI, OpenMP, Kokkos, RAJA, CUDA	Time, Ψ
TeaLeaf	[85]	MPI, OpenMP, OpenACC, CUDA, Kokkos, RAJA, OPS	Time, GB/s, Ψ
Bricks	[60]	OpenMP, CUDA	Time, GFLOPS, GB/s, Ψ
Nekbone	[86]	OpenMP, OpenACC	Time, GB/s
PENNANT	[87]	OpenMP, CUDA	Time
Aria	[88]	Kokkos (OpenMP, CUDA)	Time
SPARTA	[89]	Kokkos (OpenMP, CUDA)	Time
Albany	[90]	Kokkos (OpenMP, CUDA)	Time
SPARC	[91,92]	Kokkos (OpenMP, CUDA)	Time
Uintah	[93]	Kokkos (OpenMP, CUDA)	Time
ARK	[94]	Kokkos (OpenMP, CUDA)	Time
KARFS	[95]	Kokkos (OpenMP, CUDA)	Time
SAMRAI	[96]	RAJA (OpenMP, CUDA)	Time
ARES	[97]	RAJA (OpenMP, CUDA)	Time
OpenSBLI	[98]	OPS (MPI, OpenMP, CUDA)	Time, GB/s, GFLOPS/s
VOLNA-OP2	[99]	OP2 (MPI, OpenMP, CUDA)	Time, GB/s, GFLOPS/s
RR Hydra	[100,101]	OP2 (MPI, OpenMP, CUDA)	Time, GB/s, GFLOPS/s

by Law et al. [84,102], considering OpenMP, CUDA, Kokkos, and RAJA, calculating the aforementioned portability metric as well. The authors evaluate performance on a classical CPU cluster (ARCHER, Intel Xeon E5-2697 v2), and a GPU cluster (Power8 with 4 P100 GPUs). On CPUs the authors note only minor variations in performance, scaling up to 4096 compute nodes, with 10–20% slowdown compared to OpenMP only at the extreme problem sizes. Scaling up to 64 GPUs Kokkos slightly outperforming the CUDA implementation (<1%), and RAJA being slightly slower (< 2%). An extended study of performance portability to the Intel Xeon Phi and V100 GPUs show that out of the explored implementations only Kokkos and RAJA were able to run on all platforms, and both achieved above 90% architectural efficiency (bandwidth in this case), with Kokkos slightly outperforming RAJA; with a performance portability score of 0.928, compared to 0.876. The authors note that the productivity of porting a code to use Kokkos or RAJA is roughly similar to that of porting to CUDA.

TeaLeaf [103] is a structured mesh matrix-free implicit solver proxy application, solving a simple heat-conduction problem with a variety of iterative solvers - Kirk et al. [85] evaluate MPI, OpenMP, OpenACC, CUDA, Kokkos, RAJA, and OPS versions on an Intel Broadwell CPU, and Intel Xeon Phi, and an NVIDIA P100 GPU. The authors report similar productivity when porting to CUDA, Kokkos, RAJA or OPS. Performance results on the larger problem running on the CPU show Kokkos and RAJA outperforming OpenMP by 15–25%, and OPS outperforming OpenMP by 35%, due to integrated memory locality optimisations. On the Xeon Phi, OpenMP, OPS, and RAJA are within 10%, Kokkos performed 50% worse. On the P100 GPU, the hand-coded CUDA implementation performed best, with Kokkos and RAJA trailing behind by 7–15% and OPS by 20%. The paper also reports the performance portability score, with hand-coded implementations (two separate ones) achieving 0.74, OPS achieving 0.79, Kokkos 0.41, and RAJA 0.61. McIntosh-Smith also studies TeaLeaf [81] on the same platforms as reported for CloverLeaf above, calculating performance portability based on fraction of best observed performance - reporting that OpenMP runs on all of them with a performance portability score of 0.45. Kokkos runs on all but the NEC Aurora, achieving a score of 0.57. OpenACC only ran on NVIDIA GPUs and the IBM Power9, achieving a score of 0.77.

The Bricks library [60] gives access to advanced data structures tailored to the needs of high-order stencil computations. Zhao et al. [60] carry out an in-depth performance portability study, between Intel Xeon Phi, Skylake CPUs, as well as an NVIDIA P100 GPU. With increasing stencil sizes calculating the Laplacian, the code becomes increasingly computationally heavy, and the authors utilize the roofline model to determine the fraction achieved peak performance. CPU architectures achieve a consistently higher efficiency at smaller stencil sizes, largely thanks to sizable caches - utilisation goes down from 85% (KNL), 96% (Skylake), and 82% (P100) at the smallest stencil size to 41%, 43% and 62% respectively. Across all test cases, Bricks achieves a 0.72 performance portability score in double precision.

6.1.2. OpenMP and OpenACC

Daley [104] studies various compilers for OpenMP offload functionality, reporting significant differences in behaviour. While they do not report performance differences between CPU and GPU, they point out that the CPU execution fall back path of offload pragmas is currently inefficient with most compilers, therefore separate OpenMP pragmas should be used when running on the CPU and when running on the GPU.

The Nekbone code is a proxy for the large Nek5000 spectral element CFD solver, and its performance portability is evaluated by Chunduri et al. [86] - the CPU version uses OpenMP, and the GPU version OpenACC, they compare an Intel Xeon Phi, and an NVIDIA P100 GPU. They achieve 40–51% of peak computational performance on compute-intensive kernels, and 64–95% of peak memory bandwidth on data-intensive kernels on the Xeon Phi. In comparison, on the P100 GPU they achieve only 6% of peak compute, but 89–95% of peak bandwidth. Overall the Xeon Phi outperforms the GPU at all but the largest problem sizes.

PENNANT is a proxy for the LANL rad-hydro code FLAG, an implements 2D staggered-grid Lagrangian hydrodynamics on general unstructured meshes. As reported by Ferenbaugh [87], it has both CUDA and OpenMP implementations, with various degrees of granularity in the parallelism, which often has significant effects on performance. Best results are achieved with a coarse-grained approach on both CPUs and GPUs - up to 2.5x speedup is reported when comparing a Sandy Bridge (E5-2670) server and an NVIDIA K20X GPU, 5.5x when compared to a Blue-Gene/Q node, and 3x compared to the first-generation Xeon Phi.

6.1.3. Kokkos

Parts of the Aria code, an unstructured, nonlinear, multiphysics finite element solver, were ported to use Kokkos, and Brunini et al. [88] report comparable performance between one Intel Broadwell server (E5-2607 v4) and a single NVIDIA V100 GPU. They point to repeated CPU-GPU copies, particularly during development, as a key performance bottleneck.

The particle-based computations in the SPARTA Direct Simulation Monte Carlo (DSMC) code were moved to use Kokkos, as reported in [89]; with several target-specific optimisations (e.g. atomics for GPUs, and parallel reduction on CPUs). Authors report a 3 $\times$ speedup going from a Haswell server to a V100 GPU at the maximum problem size that fits in the GPU.

The Albany land ice sheet solver relies extensively on Trilinos packages, and uses Tpetra, accelerated with Kokkos in the Finite Element Assembly phase, as discussed in [90]. The authors report an up to 8.8x speedup in the node-local assembly phase between a Haswell server and a P100 GPU, but note the considerable overheads of the global assembly, which involves MPI communications, reducing the overall speedup to only about 2x. They also carry out a scalability study up to 32 nodes, reporting strong and weak scaling efficiencies of 62% and 89% respectively on Haswell, and 36% and 69% respectively on P100 GPUs.

The SPARC hypersonic CFD code developed at Sandia National Labs also uses Kokkos, as reported by Hower et al. [91,92], P100 and V100 GPUs outperform Haswell and Broadwell systems 1.5-2x, and 2-2.5x respectively on some parts of assembly, yet on other parts they are up to 2x slower. This results in no significant speedup for the whole application. Overall the system demonstrates excellent strong and weak scalability up to 64 nodes or GPUs. The authors note that re-engineering the code to use Kokkos did make most parts of the code much more performant, but they note challenges in measuring performance, building the software system for different architectures, as well as large compilation times.

Critical sections of the Uintah [93] multi-physics code were ported to use the tasking feature of the Kokkos library, and the authors evaluate scalability and performance on an Intel Xeon Phi cluster (Stampede) and an NVIDIA K20X cluster (Titan - a generation older than Stampede). The results show no loss in performance compared to the reference implementation. The authors report good scalability, particularly on the Phi, to up to 256 nodes, and the Phi outperforming the K20X by 2 $\times$.

ARK is a finite volume stratified compressible Navier-Stokes solver, targeting all Mach regimes, which was implemented using Kokkos. Padioleau et al. [94] report on its performance portability, comparing the Intel Xeon Phi, Skylake, NVIDIA K80, P100, and V100 platforms. Taking the Phi as a baseline, they report 1.5 $\times$ speedup on Skylake, 7 $\times$ on the K80, 30 $\times$ on the P100, and 53 $\times$

on the V100. The authors note that the code did not vectorise well on the CPU platforms, explaining the large performance difference between CPUs and GPUs.

KARFS (KAUST Adaptive Reacting Flows Solver) is a direct numerical simulation (DNS) code for multi-component gaseous reacting flows, which has been implemented with an MPI+Kokkos programming model [95]. The authors perform a series of benchmarks with increasing numbers of problem size and reacting species, comparing a 16-core Intel Haswell CPU with an NVIDIA K80 GPU, reporting modest speedups or even slowdowns at smaller scales, and 2–5 × speedups at larger problems. They also integrate the Cantera-CVODE-MAGMA stiff ODE solver, and report up to 2 × speedups on larger problems.

6.1.4. *RAJA*

Beckingsale et al. [96] discuss the portability of the SAMRAI structured AMR framework between CPUs and GPUs, and modify key parts of both SAMRAI and the CleverLeaf proxy application, relying on RAJA for portability. They report a 7x speedup going from the IBM Power 9 CPUs in a node (44 cores) to the 4 NVIDIA V100 GPUs.

ARES is a multiphysics ALE-AMR code at LLNL, Pearce [97] presents a port that relies on RAJA to not only utilise either CPUs or GPUs, but both in the same system, by way of a load-balanced domain decomposition that assigns work to CPUs (Intel Xeon E5-2667 v3 and IBM Power8) as well as GPUs (NVIDIA K80 and P100). The paper reports up to 40% improvement by oversubscribing GPUs, and an 18% improvement by utilising CPUs as well as GPUs. The paper also reports an up to 16 × speedup between CPU and GPU at the largest problem sizes.

6.1.5. *OP2 and OPS*

OpenSBLI [75] is a Shock-wave/Boundary-layer Interaction CFD code capable of performing Direct Numerical Simulations (DNS) or Large Eddy Simulation (LES) of SBLI problems, which has been developed using the OPS framework. Mudalige et al. [98] report on its performance, portability, and energy consumption on a variety of architectures including Intel Broadwell and Skylake, and IBM Power 8 CPUs (utilising the MPI+OpenMP capabilities of OPS), and NVIDIA P100 and V100 GPUs (utilising the CUDA capabilities of OPS). The authors report that some parts of the application are bandwidth-bound, while others are latency-sensitive, and suffer from occasional lack of vectorisation, which is why the Broadwell system outperforms Skylake 1.17 ×, IBM Power8 is 1.96 × faster thanks to the large bandwidth available, and the P100 and the V100 are 6.3 × and 9.7 × faster respectively. The authors also report architectural efficiency - for the SN, 260^3 test case the overall computational and bandwidth efficiencies are 0.14 and 0.31 respectively (Skylake), 0.24 and 0.35 (Broadwell), 0.39 and 0.29 (Power8), 0.1 and 0.39 (P100), 0.11 and 0.39 (V100), yielding a performance portability score of 0.34, based on bandwidth efficiency. Strong and weak scalability is also evaluated with up to 4096 nodes (Intel Ivy Bridge on ARCHER) or 4096 GPUs (K20X on Titan), or 256 GPUs (P100 on Wilkes2).

VOLNA-OP2 is a finite-volume non-linear shallow-water equation (NSWE) solver built on the OP2 domain-specific language capable of simulating the full life cycle of tsunamis. Reguly et al. [99] report on its performance and portability, running in Intel Skylake and Xeon Phi CPUs, as well as NVIDIA P100 GPUs, scaling up to 32 nodes or GPUs. Performance and scalability of the Skylake and Xeon Phi platforms are closely matched (within 15%), and the P100 system's relative performance highly depends on the problem size; on smaller problems there is a 1.2-2 × speedup, and at the largest problem size there is a 1.96-2.2 × speedup (1 GPU vs. 1 CPU node), which translates to a 3.2–3.6 × energy efficiency. The paper also reports detailed performance breakdowns on achieved bandwidth.

CloverLeaf, as discussed above, was also ported to OPS, and Reguly et al. [82,83] report on their performance on Intel Haswell and IBM Power8 CPUs, as well as NVIDIA K80, P100 and V100 GPUs. Their study furthermore investigates a memory-locality optimisation algorithm that can improve memory bandwidth utilisation, as well as utilise both CPU and GPU architectures in the same system. The authors evaluate achieved memory bandwidth, reporting a fraction of peak for Haswell at 0.79, 0.27 for Power 8 (1.6 × speedup), 0.82 on the K80 (4.3 × speedup), 0.86 on the P100 (11.7 × speedup), and 0.83 on the V100 (17.8 × speedup). Results from heterogeneous execution utilising both CPUs and GPUs in the system show only 10–20% degradation compared to adding the achieved bandwidths on CPU and GPU alone.

Rolls-Royce's Hydra CFD application is a finite volume Reynolds-Averaged Navier-Stokes solver used for the simulation of turbomachinery such as Rolls-Royces latest aircraft engines. Reguly and Mudalige [100,101] report on its conversion to use the OP2 library, and its performance on a variety of architectures including Intel Sandy Bridge and NVIDIA K20 GPUs, showing an up to 1.4 × speedup. The latter paper [101] studies Intel Haswell and NVIDIA K80 platforms, and achieving auto-vectorisation on unstructured mesh applications in particular, reporting a 15% improvement on the CPU by enabling specialised code-generation features. The paper gives breakdowns of achieved bandwidth for various computational stages as well, reporting an overall slowdown on the K80 of 0.95× compared to the vectorised version running on the CPU. Later benchmarks show an NVIDIA P100 outperforming an Intel Broadwell server by 2.1×. There are further benchmark data and performance analysis available on the performance of OPS and OP2 and the optimisations we have introduced not discussed here, but these are available in publications related to OPS and OP2 [105].

7. Conclusions

In this paper we have given an overview of some of the programming approaches that can be used for implementing CFD applications, with a particular focus on what level of performance, portability, and productivity can be achieved. We intend our work to be used to help pick the appropriate level of abstraction when implementing a new application. We discussed a number of tools, matching different levels of abstraction that can be used, or further developed for new applications or for porting existing ones.

We discussed some of the challenges in designing, developing, and maintaining Domain Specific Languages and tools common in this area, and while many of the libraries cited are now defunct, we argue that some of these are still worth mentioning, particularly for those who are interested in developing new DSLs.

We have also reviewed the performance benchmarking literature that compares various CPU and GPU architectures and their programming methods, giving an overview of key applications and their comparative performance. Readers wishing to develop new applications are pointed to these representative examples to make informed decisions about the choice of programming methods.

Acknowledgements

István Reguly was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013). Gihan Mudalige was supported by the Royal Society Industrial Fellowship Scheme

(INF/R1/180012). Additionally, this work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/S005072/1 - Strategic Partnership in Computational Science for Advanced Simulation (ASiMoV).

References

[1] Patterson D. The trouble with multi-core. IEEE Spectr 2010;47(7):28–32. doi:10.1109/MSPEC.2010.5491011.
[2] Asanović K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, et al. The Landscape of Parallel Computing Research: A View from Berkeley Tech. Rep.. EECS Department, University of California, Berkeley; 2006.
[3] Alimirzazadeh S, Jahanbakhsh E, Maertens A, Leguizamón S, Avellan F. GPU-accelerated 3-D finite volume particle method. Comput Fluids 2018;171:79–93. doi:10.1016/j.compfluid.2018.05.030.
[4] Diaz MA, Solovchuk MA, Sheu TW. High-performance multi-GPU solver for describing nonlinear acoustic waves in homogeneous thermoviscous media. Comput Fluids 2018;173:195–205. doi:10.1016/j.compfluid.2018.03.008.
[5] Gorobets A, Soukov S, Bogdanov P. Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers. Comput Fluids 2018;173:171–7. doi:10.1016/j.compfluid.2018.03.011.
[6] Ren F, Song B, Zhang Y, Hu H. A GPU-accelerated solver for turbulent flow and scalar transport based on the Lattice Boltzmann method. Comput Fluids 2018;173:29–36. doi:10.1016/j.compfluid.2018.03.079.
[7] Liu RK-S, Wu C-T, Kao NS-C, Sheu TW-H. An improved mixed Lagrangian–Eulerian (IMLE) method for modelling incompressible Navier–Stokes flows with CUDA programming on multi-GPUs. Comput Fluids 2019;184:99–106. doi:10.1016/j.compfluid.2019.03.024.
[8] Lee Y-H, Huang L-M, Zou Y-S, Huang S-C, Lin C-A. Simulations of turbulent duct flow with lattice Boltzmann method on GPU cluster. Comput Fluids 2018;168:14–20. doi:10.1016/j.compfluid.2018.03.064.
[9] Hashimoto T, Yasuda T, Tanno I, Tanaka Y, Morinishi K, Satofuka N. Multi-GPU parallel computation of unsteady incompressible flows using kinetically reduced local navier–Stokes equations. Comput Fluids 2018;167:215–20. doi:10.1016/j.compfluid.2018.03.028.
[10] Kao NS-C, Sheu TW-H. Development of a finite element flow solver for solving three-dimensional incompressible Navier–Stokes solutions on multiple GPU cards. Comput Fluids 2018;167:285–91. doi:10.1016/j.compfluid.2018.03.033.
[11] Singh JP, Hennessy JL. An empirical investigation of the effectiveness and limitations of automatic parallelization. In: Shared memory multiprocessing; 1992. p. 203–7.
[12] Wong M., Richards A., Rovatsou M., Reyes R.. Khronos's OpenCL SYCL to support heterogeneous devices for C++. 2016.
[13] Rul S, Vandierendonck H, D'Haene J, De Bosschere K. An experimental study on performance portability of OpenCL kernels. 2010 Symposium on application accelerators in high performance computing (SAAHPC'10); 2010.
[14] Komatsu K, Sato K, Arai Y, Koyama K, Takizawa H, Kobayashi H. Evaluating performance and portability of OpenCL programs. In: The fifth international workshop on automatic performance tuning, 66; 2010. p. 1.
[15] Pennycook SJ, Hammond SD, Wright SA, Herdman J, Miller I, Jarvis SA. An investigation of the performance portability of OpenCL. J Parallel Distrib Comput 2013;73(11):1439–50.
[16] Its Official: Aurora on Track to Be First US Exascale Computer in 2021. 2019. https://www.hpcwire.com/2019/03/18/its-official-aurora-on-track-to-be-first-u-s-exascale-computer-in-2021/.
[17] He Q, Chen H, Feng J. Acceleration of the OpenFOAM-based MHD solver using graphics processing units. Fusion Eng Design 2015;101:88–93.
[18] Malecha Z, Mirosław Ł, Tomczak T, Koza Z, Matyka M, Tarnawski W, et al. GPU-based simulation of 3D blood flow in abdominal aorta using OpenFOAM. Arch Mech 2011;63(2):137–61.
[19] Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, et al. An overview of the Trilinos project. ACM Trans Math Softw (TOMS) 2005;31(3):397–423.
[20] Hoemmen MF. Summary of current thread parallelization efforts in Trilinos' linear algebra and solvers.. Tech. Rep.. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); 2017.
[21] Balay S., Abhyankar S., Adams M.F., Brown J., Brune P., Buschelman K., et al. PETSc Web page. 2019. http://www.mcs.anl.gov/petsc; http://www.mcs.anl.gov/petsc.
[22] Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, et al. LAPACK users' guide. 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1999.
[23] Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, et al. ScaLAPACK users' guide. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1997.
[24] Buttari A, Langou J, Kurzak J, Dongarra J. A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput 2009;35(1):38–53. doi:10.1016/j.parco.2008.10.002.
[25] Tomov S, Dongarra J, Baboulin M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 2010;36(5–6):232–40. doi:10.1016/j.parco.2009.12.005.
[26] Sanderson C, Curtin R. Armadillo: a template-based C++ library for linear algebra. J Open Source Softw 2016;1(2):26.
[27] Guennebaud G., Jacob B., et al. Eigen v3. 2010. http://eigen.tuxfamily.org.
[28] Davis T., Hager W., Duff I.. SuiteSparse. 2014. http://faculty.cse.tamu.edu/davis/suitesparse.html.
[29] Falgout RD, Jones JE, Yang UM. The design and implementation of hypre, a library of parallel high performance preconditioners. In: Numerical solution of partial differential equations on parallel computers. Springer; 2006. p. 267–94.
[30] Notay Y. An aggregation-based algebraic multigrid method. Electron Trans Numer Anal 2010;37(6):123–46.
[31] Naumov M, Arsaev M, Castonguay P, Cohen J, Demouth J, Eaton J, et al. AmgX: a library for GPU accelerated algebraic multigrid and preconditioned iterative methods. SIAM J Sci Comput 2015;37(5):S602–26.
[32] Gupta A. WSMP: Watson sparse matrix package (Part-I: direct solution of symmetric sparse systems). Tech Rep RC 21886. IBM TJ Watson Research Center, Yorktown Heights, NY; 2000.
[33] Li XS. An overview of SuperLU: algorithms, implementation, and user interface. ACM Trans Math Softw 2005;31(3):302–25.
[34] Hénon P, Ramet P, Roman J. PaStiX: a high-performance parallel direct solver for sparse symmetric definite systems. Parallel Comput 2002;28(2):301–21.
[35] Amestoy PR, Duff IS, L'Excellent J-Y, Koster J. MUMPS: a general purpose distributed memory sparse solver. In: International workshop on applied parallel computing. Springer; 2000. p. 121–30.
[36] Raghavan P. DSCPACK: domain-separator codes for solving sparse linear systems. Tech. Rep.. Tech. rep. CSE-02-004. Department of Computer Science and Engineering, The ...; 2002.
[37] Sao P, Vuduc R, Li XS. A Distributed CPU-GPU Sparse Direct Solver. In: Silva F, Dutra I, Santos Costa V, editors. Euro-par 2014 parallel processing. Cham: Springer International Publishing; 2014. p. 487–98.
[38] Lacoste X, Ramet P, Faverge M, Ichitaro Y, Dongarra J. Sparse direct solvers with accelerators over DAG runtimes. Research Report. INRIA; 2012. https://hal.inria.fr/hal-00700066.
[39] Plauger P, Lee M, Musser D, Stepanov AA. C++ standard template library. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR; 2000.
[40] Schling B. The boost C++ libraries. XML Press; 2011.
[41] Kaiser H., Lelbach B.A., Heller T., Bergé A., Simberg M., Biddiscombe J., et al. STEllAR-GROUP/hpx: HPX V1.2.1: The C++ Standards Library for Parallelism and Concurrency. 2019. https://doi.org/10.5281/zenodo.2573213.
[42] Edwards HC, Trott CR, Sunderland D. Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput 2014;74(12):3202–16. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. doi: 10.1016/j.jpdc.2014.07.003.
[43] Hornung R, Jones H, Keasler J, Neely R, Pearce O, Hammond S, et al. ASC Tri-lab Co-design Level 2 Milestone Report 2015 Tech. Rep.. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States); 2015.
[44] Hoberock J., Bell N.. Thrust: A Parallel Template Library. 2010. http://code.google.com/p/thrust/.
[45] Enmyren J, Kessler CW. SkePU: a multi-backend Skeleton programming library for multi-GPU systems. In: Proceedings of the fourth international workshop on high-level parallel programming and applications, HLPP '10. New York, NY, USA: ACM; 2010. p. 5–14. doi:10.1145/1863482.1863487.
[46] Aldinucci M, Danelutto M, Kilpatrick P, Torquati M. Fastflow: high-level and efficient streaming on multicore. John Wiley & Sons, Ltd; 2017. p. 261–80. doi: 10.1002/9781119332015.ch13.
[47] Ernsting S, Kuchen H. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int J High Perform Comput Netw 2012;7(2):129–38. doi:10.1504/IJHPCN.2012.046370.
[48] Steuwer M, Kegel P, Gorlatch S. SkelCL - a portable skeleton library for high-level GPU programming. In: 2011 IEEE international symposium on parallel and distributed processing workshops and Phd forum; 2011. p. 1176–82. doi:10.1109/IPDPS.2011.269.
[49] Chakravarty MMT, Keller G, Lee S, McDonell TL, Grover V. Accelerating Haskell array codes with multicore GPUs. DAMP '11: the 6th workshop on declarative aspects of multicore programming. ACM; 2011.
[50] Keller G, Chakravarty MM, Leshchinskiy R, Peyton Jones S, Lippmeier B. Regular, shape-polymorphic, parallel arrays in haskell. SIGPLAN Not 2010;45(9):261–72. doi:10.1145/1932681.1863582.
[51] Ruiz A.. Introduction to hmatrix. 2012.
[52] DeVito Z, Joubert N, Palacios F, Oakley S, Medina M, Barrientos M, et al. 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. ACM; 2011. p. 9.
[53] Bernstein GL, Shah C, Lemire C, Devito Z, Fisher M, Levis P, et al. Ebb: a DSL for physical simulation on CPUs and GPUs. ACM Trans Graph 2016;35(2) 21:1–21:12. doi:10.1145/2892632.
[54] Earl C, Might M, Bagusetty A, Sutherland JC. Nebo: an efficient, parallel, and portable domain-specific language for numerically solving partial differential equations. J Syst Softw 2017;125:389–400. doi:10.1016/j.jss.2016.01.023.
[55] Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In: Proceedings of the 34th ACM SIGPLAN conference on programming language design and implementation, PLDI '13. New York, NY, USA: ACM; 2013. p. 519–30. doi:10.1145/2491956.2462176.
[56] Mostafazadeh B, Marti F, Liu F, Chandramowlishwaran A. Roofline guided design and analysis of a multi-stencil CFD solver for multicore performance. In: 2018 IEEE international parallel and distributed processing symposium (IPDPS); 2018. p. 753–62. doi:10.1109/IPDPS.2018.00085.

[57] Yount C, Tobin J, Breuer A, Duran A. Yaskyet another stencil kernel: A framework for HPC stencil code-generation and tuning. In: 2016 sixth international workshop on domain-specific languages and high-level frameworks for high performance computing (WOLFHPC); 2016. p. 30–9. doi:10.1109/WOLFHPC.2016.08.

[58] Reguly IZ, Mudalige GR, Giles MB, Curran D, McIntosh-Smith S. The OPS domain specific abstraction for multi-block structured grid computations. In: Proceedings of the 2014 fourth international workshop on domain-specific languages and high-level frameworks for high performance computing, WOLFHPC '14. Washington, DC, USA: IEEE Computer Society; 2014. p. 58–67. http://dl.acm.org/citation.cfm?id=2867549.2868136.

[59] Kuckuk S, Haase G, Vasco DA, Köstler H. Towards generating efficient flow solvers with the exastencils approach. Concurr Comput 2017;29(17). e4062 E4062 cpe.4062 doi: 10.1002/cpe.4062.

[60] Zhao T, Williams S, Hall M, Johansen H. Delivering performance-portable stencil computations on cpus and GPUs using bricks. In: 2018 IEEE/ACM international workshop on performance, portability and productivity in HPC (P3HPC); 2018. p. 59–70. doi:10.1109/P3HPC.2018.00009.

[61] Mudalige GR, Giles MB, Reguly I, Bertolli C, Kelly PHJ. OP2: an active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In: 2012 Innovative parallel computing (InPar); 2012. p. 1–12. doi:10.1109/InPar.2012.6339594.

[62] Rathgeber F, Markall GR, Mitchell L, Loriant N, Ham DA, Bertolli C, et al. PyOP2: a high-level framework for performance-portable simulations on unstructured meshes. In: 2012 SC companion: high performance computing, networking storage and analysis. IEEE; 2012. p. 1116–23.

[63] Incardona P, Leo A, Zaluzhnyi Y, Ramaswamy R, Sbalzarini IF. OpenFPM: a scalable open framework for particle and particle-mesh codes on parallel computers. Comput Phys Commun 2019;241:155–77. doi:10.1016/j.cpc.2019.03.007.

[64] Fuhrer O, Osuna C, Lapillonne X, Gysi T, Cumming B, Bianco M, et al. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. Supercomput Front Innov 2014;1(1):45–62. http://superfri.org/superfri/article/view/17.

[65] PSyclone Project. 2018. http://psyclone.readthedocs.io/.

[66] Baldauf M, Seifert A, Förstner J, Majewski D, Raschendorfer M, Reinhardt T. Operational convective-scale numerical weather prediction with the COSMO model: description and sensitivities. Mon Weather Rev 2011;139(12):3887–905.

[67] Clement V, Ferrachat S, Fuhrer O, Lapillonne X, Osuna CE, Pincus R, et al. The CLAW DSL: abstractions for performance portable weather and climate models. In: Proceedings of the platform for advanced scientific computing conference, PASC '18. New York, NY, USA: ACM; 2018. 2:1–2:10. https://doi.org/10.1145/3218176.3218226.

[68] Clément V, Marti P, Fuhrer O, Sawyer W. Performance portability on GPU and CPU with the ICON global climate model. In: EGU general assembly conference abstracts. In: EGU General Assembly Conference Abstracts, 20; 2018. p. 13435.

[69] Alnæs MS, Blechta J, Hake J, Johansson A, Kehlet B, Logg A, et al. The FEniCS project version 1.5. Arch Numer Softw 2015;3(100). doi:10.11588/ans.2015.100.20553.

[70] Rathgeber F, Ham DA, Mitchell L, Lange M, Luporini F, Mcrae ATT, et al. Firedrake: automating the finite element method by composing abstractions. ACM Trans Math Softw 2016;43(3) 24:1–24:27. doi:10.1145/2998441.

[71] Lengauer C, Apel S, Bolten M, Größlinger A, Hannig F, Köstler H, et al. ExaStencils: advanced stencil-code engineering. In: Lopes L, Žilinskas J, Costan A, Cascella RG, Kecskemeti G, Jeannot E, et al., editors. Euro-par 2014: parallel processing workshops. Cham: Springer International Publishing; 2014. p. 553–64.

[72] Macià S, Mateo S, Martínez-Ferrer PJ, Beltran V, Mira D, Ayguadé E. Saiph: towards a DSL for high-performance computational fluid dynamics. In: Proceedings of the real world domain specific languages workshop 2018, RWDSL2018. New York, NY, USA: ACM; 2018. 6:1–6:10. https://doi.org/10.1145/3183895.3183896.

[73] Rink NA, Huismann I, Susungi A, Castrillon J, Stiller J, Fröhlich J, et al. Cfdlang: high-level code generation for high-order methods in fluid dynamics. In: Proceedings of the real world domain specific languages workshop 2018, RWDSL2018. New York, NY, USA: ACM; 2018. 5:1–5:10. https://doi.org/10.1145/3183895.3183900.

[74] Bastian P, Blatt M, Dedner A, Engwer C, Klöfkorn R, Ohlberger M, et al. A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. Computing 2008;82(2):103–19. doi:10.1007/s00607-008-0003-x.

[75] Lusher DJ, Jammy SP, Sandham ND. Shock-wave/boundary-layer interactions in the automatic source-code generation framework OpenSBLI. Comput Fluids 2018;173:17–21. doi:10.1016/j.compfluid.2018.03.081.

[76] Lange M, Kukreja N, Louboutin M, Luporini F, Vieira F, Pandolfo V, et al. Devito: towards a generic finite difference DSL using symbolic python. In: 2016 6th workshop on python for high-performance and scientific computing (PyHPC); 2016. p. 67–75. doi:10.1109/PyHPC.2016.013.

[77] Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for floating-point programs and multicore architectures. Tech. Rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States); 2009.

[78] Pennycook SJ, Sewall JD, Lee VW. Implications of a metric for performance portability. Future Gener Comput Syst 2017;92:947–58.

[79] Harrell SL, Kitson J, Bird R, Pennycook SJ, Sewall J, Jacobsen D, et al. Effective performance portability. In: 2018 IEEE/ACM international workshop on performance, portability and productivity in HPC (P3HPC); 2018. p. 24–36. doi:10.1109/P3HPC.2018.00006.

[80] Intel. Code base investigator. https://github.com/intel/code-base-investigator.

[81] McIntosh-Smith S. Performance portability across diverse computer architectures. P3MA: 4th international workshop on performance portable programming models for manycore or accelerators; 2019.

[82] Reguly IZ, Mudalige GR, Giles MB. Loop tiling in large-Scale stencil codes at run-time with OPS. IEEE Trans Parallel Distrib Syst 2018;29(4):873–86. doi:10.1109/TPDS.2017.2778161.

[83] Siklosi B, Reguly IZ, Mudalige GR. Heterogeneous CPU-GPU execution of stencil applications. In: 2018 IEEE/ACM international workshop on performance, portability and productivity in HPC (P3HPC); 2018. p. 71–80. doi:10.1109/P3HPC.2018.00010.

[84] Law TR, Kevis R, Powell S, Dickson J, Maheswaran S, Herdman JA, Jarvis SA. Performance portability of an unstructured hydrodynamics mini-application. In: 2018 IEEE/ACM international workshop on performance, portability and productivity in HPC (P3HPC); 2018. p. 0–12.

[85] Kirk RO, Mudalige GR, Reguly IZ, Wright SA, Martineau MJ, Jarvis SA. Achieving performance portability for a heat conduction solver mini-application on modern multi-core systems. In: 2017 IEEE international conference on cluster computing (CLUSTER); 2017. p. 834–41. doi:10.1109/CLUSTER.2017.122.

[86] Sudheer Chunduri SP, Rahman R. Nekbone performance portability. The 2017 DOE COE performance portability meeting; 2017.

[87] Ferenbaugh CR. Coarse vs. fine-level threading in the PENNANT mini-app. The 2016 DOE COE performance portability meeting; 2016.

[88] Brunini V, Clausen J, Hoemmen M, Kucala A, Phillips M, Trott C. Progress towards a performance-portable SIERRA/ aria. The 2019 DOE performance, portability and productivity annual meeting; 2019.

[89] Stan Moore AS. Obtaining threading performance portability in SPARTA using Kokkos. The 2019 DOE performance, portability and productivity annual meeting; 2019.

[90] Watkins J, Tezaur I, Demeshko I. A study on the performance portability of the finite element assembly process within the Albany Land Ice solver. The 2019 DOE performance, portability and productivity annual meeting; 2019.

[91] Howard M. Performance portability in SPARC sandias hypersonic CFD code for next-generation platforms. The 2017 DOE COE performance portability meeting; 2017.

[92] Howard M, Fisher TC, Hoemmen MF, Dinzl DJ, Overfelt JR, Bradley AM, et al. Employing multiple levels of parallelism for CFD at large scales on next generation high-performance computing platforms.. In: Tenth international conference on computational fluid dynamics (ICCFD10); 2018.

[93] Holmen JK, Humphrey A, Sunderland D, Berzins M. Improving uintah's scalability through the use of portable Kokkos-based data parallel tasks. In: Proceedings of the practice and experience in advanced research computing 2017 on sustainability, success and impact. ACM; 2017. p. 27.

[94] Padioleau T, Tremblin P, Audit E, Kestener P, Kokh S. A high-performance and portable all-mach regime flow solver code with well-balanced gravity. application to compressible convection. Astrophys J 2019;875(2):128. doi:10.3847/1538-4357/ab0f2c.

[95] Prez FEH, Mukhadiyev N, Xu X, Sow A, Lee BJ, Sankaran R, et al. Direct numerical simulations of reacting flows with detailed chemistry using many-core/GPU acceleration. Comput Fluids 2018;173:73–9. doi:10.1016/j.compfluid.2018.03.074.

[96] David Beckingsale Johann Dahm PW. Porting SAMRAI to sierra. The 2019 DOE performance, portability and productivity annual meeting; 2019.

[97] Pearce O. Exploring utilization options of heterogeneous architectures for multi-physics simulations. Parallel Comput 2019;87:35–45. doi:10.1016/j.parco.2019.05.003.

[98] Mudalige G, Reguly I, Jammy S, Jacobs C, Giles M, Sandham N. Large-scale performance of a DSL-based multi-block structured-mesh application for direct numerical simulation. J Parallel Distrib Comput 2019;131:130–46. doi:10.1016/j.jpdc.2019.04.019.

[99] Reguly IZ, Giles D, Gopinathan D, Quivy L, Beck JH, Giles MB, et al. The VOLNA-OP2 tsunami code (version 1.5). Geosci Model Dev 2018;11(11):4621–35. doi:10.5194/gmd-11-4621-2018.

[100] Reguly IZ, Mudalige GR, Bertolli C, Giles MB, Betts A, Kelly PHJ, et al. Acceleration of a full-Scale industrial CFD application with OP2. IEEE Trans Parallel Distrib Syst 2016;27(5):1265–78. doi:10.1109/TPDS.2015.2453972.

[101] Mudalige GR, Reguly IZ, Giles MB. Auto-vectorizing a large-scale production unstructured-mesh CFD application. In: Proceedings of the 3rd workshop on programming models for SIMD/vector processing, WPMVP '16. New York, NY, USA: ACM; 2016. 5:1–5:8. https://doi.org/10.1145/2870650.2870651.

[102] Truby D, Wright S, Kevis R, Maheswaran S, Herdman A, Jarvis S. BookLeaf: an unstructured hydrodynamics mini-application. In: 2018 IEEE international conference on cluster computing (CLUSTER); 2018. p. 615–22. doi:10.1109/CLUSTER.2018.00078.

[103] TeaLeaf: UK Mini-App Consortium. 2015. https://github.com/UK-MAC/TeaLeaf.

[104] Daley C. Evaluation of OpenMP performance on GPUs through micro-benchmarks. The 2019 DOE performance, portability and productivity annual meeting; 2019.

[105] OP-DSL: The Oxford Parallel Domain Specific Languages. 2015. https://op-dsl.github.io.