Towards performance-portable high-order implicit flow solvers

Jan Eichstädt[*]
Imperial College London, South Kensington Campus, London SW7 2AZ, United Kingdom

David Moxey[†]
University of Exeter, Innovation Centre, Exeter EX4 4RN, United Kingdom

Joaquim Peiró[‡]
Imperial College London, South Kensington Campus, London, United Kingdom

We discuss the steps required to adapt legacy flow, or structural, solvers to modern CPU and GPU architectures using a portable programming model. These steps are illustrated using a high-order mesh optimiser and an implicit Helmholtz solver as examples. We show that satisfactory performance can be achieved in both architectures using such a framework, and highlight the importance of developing efficient data structures.

I. Introduction

Major efforts within computational fluid dynamics (CFD) are targeted at overnight production runs of industrial-scale large-eddy simulations. Current state-of-the-art solver capabilities for the most accessible problems like compressor-cascade simulations, though, are still requiring about an order of magnitude longer wall-clock times, independently of the size of the computing system at hand. It is the aspiration of various initiatives such as the *ExaFLOW* computing project [1] or the *Exascale Computing Project* [2] to bridge that gap within the next decade, to enable industrial LES simulations for applications like full aircraft configurations over the whole flight envelope, astrophysics, small modular reactors, industrial scale chemical reactors, cosmological hydrodynamics, or wind plants in complex terrain to name but a few. Our aim is to increase the algorithmic efficiency of these solver capabilities, while also having any current and future high-performance-computing (HPC) systems in mind.

Current architectural trends in HPC systems show a massive increase in FLOPS, which are distributed over thousands of compute cores, but the increase in memory bandwidth and capacity, as well as communication speed lacks behind. These trends demand highly parallel algorithms with high arithmetic intensity and efficient data structures, since it is not the FLOP count, but memory access and communication costs that are the bottlenecks in applications. More traditional algorithms have primarily targeted minimum operation counts, which now require a re-evaluation.

A second aspect is finding ways to express these parallel algorithms using suitable programming models. The life-cycles of scientific applications are large, often spanning more than a decade. It is hence not suitable to rewrite the application's code-base each time a new class of hardware needs to be addressed. Instead, it will be more suitable to find a path to evolutionary adapt such code-bases, in order to minimise the implementational effort and keep the code maintainable, yet performant.

Accordingly, we are investigating in this work *how* to adapt and re-design legacy CFD frameworks to exploit such massively parallel hardware. We choose methods that have the potential for parallelisation and will still be good algorithmic choices in the future. In this spirit, we consider two fundamental algorithms that are relatively simple to implement but comprise many typical algorithmic features of CFD methods: a relaxation-type optimisation method and an implicit Helmholtz solver. On a more abstract level, these methods share many algorithmic commonalities independently of the spatial discretisations and the physics to be modelled, be that fluids or structural solvers.

A. Two fundamental building blocks for CFD methods

We consider that an unstructured spatial discretisation with high-order spectral element methods is a promising path for unsteady flow simulations, particularly LES, as they offer higher accuracy per degree of freedom with low dispersion and diffusion errors, while offering geometric flexibility and the ability to model curved domain boundaries accurately.

[*]PhD-Scholar, Department of Aeronautics
[†]Lecturer, Department of Engineering
[‡]Reader, Department of Aeronautics

The optimisation method we have chosen to highlight some of the implementation issues of portable programming frameworks is at the core of the high-order mesh generation process. High-order meshes are most often generated starting from linear meshes. Next, the exact and often curved CAD boundaries are imposed upon all elements next to the boundary, which can lead to highly deformed and potentially invalid elements. Subsequently, these mesh deformations need to be propagated into the interior of the domain with a suitable algorithm. Here we consider a variational framework that comprises relaxation-type optimisation methods and encompasses a variety formulations of deformation energy, see references [3] and [4]. The applied algorithms are similar in character to explicit time-stepping methods, in that the optimisation of element coordinates is performed on localised stencils and not over the whole domain. It is therefore possible to parallelise multiple localised optimisation steps, as long as the stencils do not overlap. This is readily achieved by employing a suitable colouring algorithm during the preprocessing phase to ensure there is no conflicts when the operations are performed in parallel.

Our second choice, the Helmholtz solver, is a core component of incompressible Navier-Stokes solvers employing an operator splitting scheme, in which the advection terms are decoupled from the pressure and diffusion terms as introduced, for instance, in reference [5], and implemented within the *Nektar++* framework [6]. The advection terms are evolved in time using an explicit time-stepping, whereas the pressure and diffusion terms are recast as Helmholtz systems and solved by implicit time-stepping. Many more prototypical applications in acoustics or seismology, to name but a few, are also based on the efficient solution of the Helmholtz equation. The most efficient approach for its solution is based on a Fourier decomposition, but this approach is limited to regular Cartesian meshes. The use of implicit methods is almost compulsory when simulating low Mach number flows with compressible solvers as this often results in very stiff systems, which makes the use of explicit methods impractical due to their severe stability restrictions. The implementation of implicit schemes on massively parallel architectures is however far more challenging because they lack the inherent locality of explicit schemes, which maps well to massively parallel architectures. However, to circumvent this issue we adopt a matrix-free scheme that avoids constructing and storing the complete global system matrix and thus offers increased locality and a smaller memory footprint.

B. Massively parallel computing hardware

Current state-of-the-art HPC hardware is characterised by a high degree of parallelism. This comes not only in the more traditional form of distributed memory parallelism, as would be the case for clusters of single-core CPUs, but also in the form of intra-node or shared memory parallelism over multiple compute cores.

The parallelism of multi-core CPUs comes on two different levels: over the multiple compute cores, and over the vector lanes within each core. These typically cover 2 to 4 double precision floats and are addressed by the compiler using AVX instructions. To address the parallelism of the CPU, the programming models (be it *OpenMP* [7] or *Pthreads* [8]) specify one (or two in case of hyper-threading) threads per core. The vector level parallelism can only be achieved implicitly by compiler auto-vectorisation or explicitly by using assembler-level AVX instructions.

Nvidia GPUs, an instance of many-core architectures, also provide their parallelism on two levels: a number of streaming multiprocessors (SMX), and the individual compute cores on each SMX. Groups of 32 compute cores have to execute as an individual SIMT unit, making it comparable to vector lanes on CPUs. Implementations on Nvidia GPUs are carried out using the *CUDA* [9] programming model, that allows to specify any operation on all parallelism levels explicitly. However, there is not much support for object-oriented programming instructions.

At the same time, memory bandwidth and latency of both these architectures, and especially for GPUs, is limited and therefore algorithms need a high arithmetic intensity (the ratio of compute operations to memory accesses) to benefit from the compute resources. As data is typically loaded in contiguous chunks, a good spatial memory allocation of data is crucial.

All these aspects combined explain the difficulty in achieving good shared-memory parallel performance, compared with distributed memory parallelism, which is typically enabled by domain decomposition. A good distributed-memory implementation using *MPI* [10] should achieve perfect weak and strong scaling, as long as the problem size per *MPI*-rank is sufficient and communication costs can be hidden. However, to further reduce the execution time per problem size, shared memory parallelism need to be employed, too, by using a shared-memory programming model and suitable algorithms.

C. A portable programming model

Many current scientific computing frameworks originate from the single-core CPU era and hence require updated capabilities to benefit from the increased parallelism of modern hardware. Approaching this with an evolutionary approach would allow to adjust the existing frameworks step-by-step with a compatible parallel programming model. We adopt this approach because it avoids extensive rewriting of the code that, for the CFD codes that we are targeting, often contain of the order of 100,000 lines or more. Alternatively, the code-base would need to be completely rewritten from scratch, either in a different low-level language like *CUDA* or *OpenCL* [11], or using a meta-programming approach in which code is developed in a high-level language like *Python* and compiled to different low-level codes targeting different hardware, a method that for example *PyFR* [12] adopts. A highly optimised low-level implementation will most probably achieve a better performance compared with a portable programming model. The actual performance penalty highly depends on the compiler at hand and can vary between less than 10% or very rarely up to an order of magnitude. In any event, each choice of programming model will always be a compromise between performance and implementational effort.

Here we employ two different programming models that facilitate parallelisation for existing code frameworks. As a requirement, they need to support both CPU and GPU architectures to expose their parallel capabilities, have open-access compiler support and provide a syntax that allows seamless implementation in an existing *C++* code-base.

OpenACC [13] is a programming model based on compiler directives, similar to *OpenMP*, but primarily targeted at off-loading to GPU devices. The *pgi* compiler that supports Nvidia GPUs has a very mature ecosystem and will be employed in this work. More recently, not only a host fallback is supported by it, but also a proper multi-core CPU compilation target. As a second option we employ the *Kokkos* [14] library, that allows to add functors for parallel execution within the code-base. It can then be compiled for CPUs using its *OpenMP* backend or for GPUs using its *CUDA* backend.

D. Outline of this paper

We will present the core algorithms of the mesh optimiser and Helmholtz solver in the *Methods* section. We will then discuss the steps required for porting a legacy implementation to modern parallel hardware in the section on *Implementation procedure and parallelisation strategy*. This will include considerations on efficient data structures and approaches to parallelise different kernels. Finally, we will assess their performance on a variety of parallel systems in the section on *Performance evaluations* section and draw some *Conclusions*.

II. Methods

This section briefly describes the core algorithms of the energy optimiser and the Helmholtz solver, with an emphasis on their commonalities.

A. Minimisation of the energy functional

In producing a high-order curvilinear mesh by deforming a straight-sided mesh, whilst maintaining its validity and quality, we minimise an energy functional, $\mathcal{E}$, which depends on the mesh deformation metric W. The deformation is measured by the mapping gradient between an ideal element and the deformed element $\nabla\boldsymbol{\phi}(\boldsymbol{y})$, with $\boldsymbol{y}$ being the coordinates of the mesh nodes. This energy is evaluated as the sum of contributions from all mesh elements e in the domain Ω, i.e.

$$\mathcal{E}(\nabla\boldsymbol{\phi}(\boldsymbol{y})) = \sum_{e=1}^{N_{el}} \int_{\Omega^e} W(\nabla\boldsymbol{\phi}(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}, \tag{1}$$

Instead of optimising the node positions in a global approach, we apply a relaxation method that solves a set of local optimisation problems. This is possible because the energy functional of each element is only influenced by the positions of the elemental nodes it owns. For example, a node that is interior to one element will only affect the energy contribution of that element. Figure 1 illustrates this concept: nodes of the same colour can be processed concurrently as the respective operations involved in the evaluation of the energy functional are independent. This is the crucial concept that later allows to efficiently parallelise this optimisation method. It further relies on a good initial colouring algorithm.

The optimisation of a free-to-move node i towards lower energy functionals becomes

$$\mathcal{E}_i(\nabla\boldsymbol{\phi}) = \sum_{e \ni i} \int_{\Omega^e} W(\nabla\boldsymbol{\phi}(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y} \tag{2}$$

where $e \ni i$ denotes the set of all elements that own node i and hence influence the energy functional of node i. The problem is then essentially reduced to multiple evaluations of an integral quantity over one mesh element.

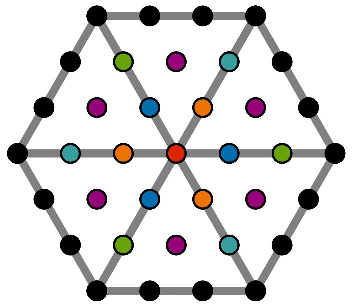

Fig. 1 Node colouring scheme for a domain of six triangular elements. Nodes of the same colour can be processed concurrently.

B. Optimisation method

The minimisation of the energy functional is performed using a Newton method with truncated steps. This method requires the evaluation of the gradient vector $\boldsymbol{G}$ and the Hessian matrix $\boldsymbol{H}$ of the energy functional $\mathcal{E}$, where the derivatives are calculated analytically with respect to the position $\boldsymbol{y}$ of the node to be optimised. The coordinates of the free-to-move nodes are then updated in optimisation step k according to

$$\boldsymbol{y}^{k+1} = \boldsymbol{y}^{k} - \alpha \boldsymbol{H}^{-1}\boldsymbol{G}, \tag{3}$$

with α being a step size parameter with $0 < \alpha \leq 1$, that is established in a reverse line search, which involves additional evaluations of the energy functional. Multiple other optimisation methods could be applied here, which however, would regularly rely on evaluating an integral value, like the energy functional, and its derivatives. Depending on the physics of the energy quantity and the spatial discretisation, the details of that evaluation might vary. For more details on the overall mesh optimisation method, see reference [15].

C. Helmholtz equation

We solve the inhomogeneous Helmholtz equation for a scalar field variable u given by the system

$$\nabla^2 u + \lambda u = f \qquad \text{in } \Omega \tag{4}$$

$$u = u_0 \qquad \text{on } \Gamma_D \tag{5}$$

$$\frac{\partial u}{\partial n} = g \qquad \text{on } \Gamma_N, \tag{6}$$

where λ is a real coefficient, f is a forcing function, and Γ_D and Γ_N denote the regions in the boundary where Dirichlet and Neumann boundary conditions are set.

We apply a Galerkin approximation to the system, and lift the Dirichlet boundary conditions, to yield a linear system with a Helmholtz operator $\boldsymbol{H}$, homogeneous solution coefficients $\hat{\boldsymbol{u}}^H$ and right-hand-side coefficients $\hat{\boldsymbol{b}}$, that include all boundary condition terms and the forcing function, as:

$$\boldsymbol{H}\hat{\boldsymbol{u}}^H = \hat{\boldsymbol{b}}. \tag{7}$$

D. Implicit iterative solver

The linear Helmholtz equation system is solved iteratively using the conjugate gradient method. This method is suitable because the matrix that arises from the Galerkin discretization of the Helmholtz operator is symmetric and positive definite. After an initialisation step, the method then proceeds by iteration until convergence. The step k in the iteration loop reads

$$\boldsymbol{p}_{k+1} = \beta \boldsymbol{p}_k + \boldsymbol{w}_k \tag{8}$$

$$\boldsymbol{q}_{k+1} = \beta \boldsymbol{q}_k + \boldsymbol{s}_k \tag{9}$$

$$\hat{\boldsymbol{u}}^H_{k+1} = \alpha \boldsymbol{p}_{k+1} + \hat{\boldsymbol{u}}^H_k \tag{10}$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha \boldsymbol{q}_{k+1} \tag{11}$$

$$\boldsymbol{w}_{k+1} = \boldsymbol{C}^{-1} \boldsymbol{r}_{k+1} \tag{12}$$

$$\boldsymbol{s}_{k+1} = \boldsymbol{H} \boldsymbol{w}_{k+1} \tag{13}$$

$$\beta = \frac{\boldsymbol{r}^T_{k+1} \boldsymbol{w}_{k+1}}{\boldsymbol{r}^T_k \boldsymbol{w}_k} \tag{14}$$

$$\alpha = \frac{\boldsymbol{r}^T_{k+1} \boldsymbol{w}_{k+1}}{\boldsymbol{s}^T_{k+1} \boldsymbol{w}_{k+1} - \boldsymbol{r}^T_{k+1} \boldsymbol{w}_{k+1} \cdot \beta_{k+1}/\alpha_k} \tag{15}$$

$$\epsilon = \boldsymbol{r}^T_{k+1} \boldsymbol{r}_{k+1} \tag{16}$$

where $\boldsymbol{r}$ is the residual vector, $\boldsymbol{C}^{-1}$ a suitable preconditioner, $\boldsymbol{w}$ and $\boldsymbol{s}$ are the conjugate gradient vectors, β and α are step sizes $\boldsymbol{p}$ and $\boldsymbol{q}$ are search direction vectors and $\hat{\boldsymbol{u}}^H_{k+1}$ is the updated solution to the linear system that is to be solved. The iteration stops when the residual ϵ is smaller than a set tolerance ϵ_{tol} Here we use a simple diagonal Jacobi preconditioner.

The details of other iterative solvers will be different, but they will most often still consist of the same functions, vector additions and multiplications, reductions, and multiplications with an operator matrix.

E. Continuous Galerkin discretisation

We now give a brief overview of the discretisation of the Helmholtz equation, which is described further in reference [16]. The previous parts of the described method are independent of the spatial discretisation, so they would be applicable for finite difference or finite volume methods, too. In this work, we use a continuous Galerkin projection over triangular elements with a C^0 hierarchical modal basis. An important part of attaining performance at higher orders is to consider that the global vector of solution coefficients $\hat{\boldsymbol{u}}^g$ can be mapped to the element local solution coefficients $\hat{\boldsymbol{u}}^l$ in a *scatter*-type operation (see reference [17]), so that

$$\hat{\boldsymbol{u}}_l = \boldsymbol{A} \hat{\boldsymbol{u}}_g. \tag{17}$$

Since the global-to-local matrix $\boldsymbol{A}$ is very sparse, it is implemented using a mapping vector $\boldsymbol{v}_{map}$ and a sign vector $\boldsymbol{v}_{sign}$ of length n_{local} as

$$\hat{\boldsymbol{u}}_l[i] = \boldsymbol{v}_{sign}[i]\ \hat{\boldsymbol{u}}_g[\boldsymbol{v}_{map}[i]] \quad \text{for } i \in [0 : n_{local}). \tag{18}$$

On each element e the elemental Helmholtz operator can then be applied as:

$$\hat{\boldsymbol{s}}^e_l = \boldsymbol{H}^e \hat{\boldsymbol{u}}^e_l \quad \text{for } e \in [0 : n_{el}) \tag{19}$$

Each of these elemental operations can be performed independently and in parallel.

Finally, the local solution coefficients $\hat{\boldsymbol{s}}^l$ need to be mapped back to the global solution coefficients $\hat{\boldsymbol{s}}^g$ in a gather-type operation

$$\hat{\boldsymbol{s}}_g = \boldsymbol{A}^T \hat{\boldsymbol{s}}_l. \tag{20}$$

Equivalently to the global-to-local operation, this is implemented as

$$\hat{\boldsymbol{s}}_g[\boldsymbol{v}_{map}[i]] = \hat{\boldsymbol{s}}_g[\boldsymbol{v}_{map}[i]] + \boldsymbol{v}_{sign}[i]\ \hat{\boldsymbol{s}}_g[i] \quad \text{for } i \in [0 : n_{local}) \tag{21}$$

Here it can be seen that boundary modes depend on those of neighbouring elements, which in a parallel implementation can lead to a race condition.

Alternatively, using a discontinous Galerkin projection, the mapping between local and global solution coefficients is not required. This is because no C^0 continuity is prescribed, but in turn a flux evaluation between elements needs to be calculated.

F. Elemental operations

The elemental Helmholtz operation is the combination of the Laplacian operator $\boldsymbol{L}$ and the mass operator $\boldsymbol{M}$ and is given by

$$\hat{\boldsymbol{s}}_l^e = \boldsymbol{H}^e \hat{\boldsymbol{u}}_l^e \tag{22}$$
$$= [\boldsymbol{L}^e + \lambda \boldsymbol{M}^e] \hat{\boldsymbol{u}}_l^e \tag{23}$$

Three matrices per triangular element of size P by $P+1$, where P is the elemental polynomial order, are precomputed and stored. The evaluation of these matrices involves a combination of quadrature weights and mapping coefficients. More details can be found in Appendix A. In order to increase the arithmetic intensity, these matrices could be re-computed to different degrees within each iteration, depending on the compute-to-memory performance of the underlying hardware. This is a feature implemented in the *OpenSBLI* framework [18] which simulates a different physical problem, but is otherwise very similar from an algorithmic point of view.

III. Implementation procedure and parallelisation strategy

In the following, we discuss the individual parts of the two methods, as introduced in the methods section. For each part we develop an individual parallel kernel, that ideally performs well on both CPU and GPU architectures using a portable programming model. Even though each kernel is based on either of the two specific introduced methods, they will be the basis of numerous other fluids and structural solvers. The challenges that we faced when parallelising our method will thus be transferable to parallelisation efforts in other codes. This section will loosely follow the procedure that needs to be undertaken when parallelising or porting a legacy code to GPUs.

A. Data structures

In most legacy applications, algorithms, as well as data structures, have been developed with CPU architectures in mind. Yet, in order to achieve satisfactory performance on GPUs, modified data structures and algorithms need to be employed, as outlined in more detail in the subsection on elemental kernel operations. To this end, it is paramount to regain control over the placement of data in memory. This has been possible with traditional programming languages like *C* or *FORTRAN*, that only allowed for *plain old data* structures. With the advent of object-oriented programming features in the single-core CPU era, many scientific applications also introduced object-oriented data structures. While they are conceptually simple to implement, maintain and even extend, the control over data placement in memory is minimal.

The two programming models that directly address GPUs, *OpenCL* and *CUDA*, both require *plain old data* structures. While they support some object-oriented programming features, it is not possible to directly copy an object with all its instance variables to the GPU memory or create a loop that directly iterates over all objects of a class and accesses its member variables. Portable programming models like *OpenACC* or *Kokkos* will in a first compilation step also create *CUDA* code, that is subsequently compiled by the *nvcc* compiler. In extension, these programming models, too, do not support object-oriented data structures. While these high-level programming models advertise to enable a seamless porting of legacy code to GPUs, this is thus in practise not always the case and in our view an often neglected aspect.

For the porting exercise of the mesh optimiser and the Helmholtz solver we took the implementations within the *Nektar*++ framework [6] as the starting point. This framework is heavily based on object-oriented programming features. Most of its code-base serves as multiple layers of libraries to its solvers, which sit on top of the pyramid. Mesh elements, for example, are objects of different inherited element classes with their respective mappings and quadrature points as member variables, that are accessed with shared pointers.

The first step for us has accordingly been to flatten out the object-oriented data structures to *plain old data* arrays, which turned out to be the most demanding step. Any legacy code, that still uses traditional data structures is under this premise at a huge advantage. This step might involve to *conceptually* re-evaluate and re-factor most data structures to de-tangle the hierarchical object structures, so that in the end all former member variables are taken care of explicitly. It also involves implementing the changes, which can be very cumbersome if algorithms are spread over many levels in a function call tree. Whereas before, classes with member functions and variables may have ensured variables are accessible at the right places, now all involved functions and levels need to explicitly pass variables to the bottom level, where they are processed. Porting the Helmholtz solver has not involved a conceptually different data storage, as the changes have been mostly syntactically. For the mesh optimiser it involved both aspects.

1. Data structures for the mesh optimiser

The initial CPU implementation of the mesh optimiser organized the data using shared pointers to objects and its member variables. This way the complex associations between the set of nodes and elements could be organised associatively. Each free-to-move node was an object of the *node* class and each mesh element was an object of the *element* class. The objects needed to be shared because nodes on the surface of elements are part of all its neighbouring elements. Hence, a given *node* object had one or more *element* objects connected to it. At the same time, a given *element* object owned multiple free-to-move *node* objects and other fixed nodes to fill up the ranks.

As can be explained with Equation 2, all nodes of a given element need to be included for the evaluation of the energy functional, which is the most important computation. We therefore re-factored the node coordinate data to store all coordinates of one element contiguously in three (X,Y,Z) two-dimensional arrays sorted by element ID and local-node ID. This leads to an about threefold duplication of stored data, due to multiple copies of the element boundary nodes, but results in a contiguous and hence more efficient memory access, which is more important. Even though the coordinates of each free-to-move node are updated only once per optimisation step, the coordinate arrays of *all* elements that own the node need to be updated. The information for this update is provided by a complex indexing structure, that uses multiple *plain old data* integer arrays. For more details, see the appendix of reference [15].

B. Simple kernels

A first kernel implements the AXPY operations specified by Equation 8 to Equation 12 of the Helmholtz solver. *OpenACC* and most other parallel programming models have a suitable syntax and implementation for these *parallel-for-loop*-type of equations. *Scatter*-type operations, as specified by Equation 18, are also trivially parallelised using a *parallel-for* loop. All of the components above have costs that roughly scale linearly with length of the arrays or the degrees of freedom in the system. Smaller problem size pay a relative penalty due to the overhead of launching the kernel. Alternatively, a native *BLAS* kernel for CPUs or a *CuBLAS*-kernel [19] for GPUs could be employed. Overall, these type of equations and kernels do not pose any difficulty for parallelising. However, these kernels are have a very low arithmetic intensity, and are thus always memory bound on all current architectures.

C. Gather kernels and colouring algorithms

The *gather*-type operation specified by Equation 21 of the Helmholtz solver cannot be readily parallelised without causing a race-condition, as multiple local modes are summed to obtain one global mode, therefore needing to write to the same location in memory. In this section we will explain how it is still possible to achieve satisfactory performance on parallel hardware. As mentioned before, these operation occur in a C^0 continuous Galerkin discretisation and could be avoided when using discontinuous Galerkin, finite difference or finite volume discretisations. The costs of these kernels scale roughly with the degrees of freedom in the overall system, but have a very high penalty for low element orders when using continuous Galerkin methods, connected with avoiding the race-condition.

An initial *gather*-type operation can be expressed as a nested for-loop. The outer loop over all elements needs to be in serial to avoid a race-condition, where neighbouring elemental modes need to be summed and stored in the same variable. The inner loop over all local coefficients or modes of one element can be implemented in parallel, though, as these modes are never added to each other. This algorithm performs well on the CPU, but not on the GPU, because for regular elemental polynomial orders there are substantially less coefficients to process in parallel to even occupy a single streaming multiprocessor (SMX). It is hence crucial to accelerate this kernel by enabling more parallelism.

This can be achieved by colouring the mesh in as few colours as possible, but in a way that no two neighbouring elements share the same colour. All elements with the same colour can then be processed in parallel, as it is ensured that no write conflict or race-condition can appear when adding neighbouring modes. There are numerous mesh or graph-colouring algorithms to choose from. We chose an algorithm that re-balances the size of the colourgroups, so that they are all of similar magnitude and of a minimum size in order to fully occupy the hardware at any point. It is then possible to parallelise the for-loop over all elements within one colourgroup and introduce a third outermost serial loop over all colourgroups, as shown in algorithm 1. As can be easily seen, the algorithm requires a global synchronisation between colourgroups, to ensure each of them is completed before the next one starts. Current GPUs however, can only ensure such synchronisation with the termination and launch of a new kernel. Hence, as many kernels as colourgroups need to be scheduled in serial, which adds typically around 30% overhead to the entire group of kernels. In the context of the Helmholtz solver, we employ the initial algorithm for all CPU executions, whereas for all GPU executions we employ the parallel *gather* algorithm based on the mesh colouring.

Algorithm 1 Parallel local-to-global gather algorithm using element colouring

```
1: procedure LocalToGlobalParallel($\hat{s}_g, \hat{s}_g, v_{sign}, v_{map}, C$)
2:     for all global coefficients $j$ do in parallel
3:         $\hat{s}_g[j] \leftarrow 0$
4:     end for
5:     for all coloursets $c \in C$ do in serial
6:         for all elements $e \in c$ do in parallel
7:             for all coefficients $c$ do in parallel
8:                 $i \leftarrow e * n_{coeffs} + c$
9:                 $\hat{s}_g[v_{map}[i]] \leftarrow \hat{s}_g[v_{map}[i]] + v_{sign}[i] * \hat{s}_l[i]$
10:            end for
11:        end for
12:    end for
13: end procedure
```

D. Reduction kernels

The calculation of the conjugate gradient step sizes α and β and the residual ϵ, as given by Equation 15, Equation 14 and Equation 16 of the Helmholtz solver, are classical reduction operations. These can be parallelised using the *parallel-reduce*-type syntax, which is readily available for GPUs with the *OpenACC* programming model and its compilers.

For the GPU, the compiler will translate the reduction to a tree-like algorithm, where at each step multiple data points are processed in parallel, until only one final data point is left. The computational cost will thus be logarithmic up to the point of completely occupying the hardware and only then scale linearly with increased degrees of freedom. Smaller problem sizes hence pay a relative penalty. Similar to the *AXPY* algorithm, these kernels have a low arithmetic intensity and are hence memory bound. A good compiler will however efficiently re-use data that has been copied from global memory to local memory or registers for the next step.

On the CPU, this operation can be more efficiently performed, due to better data caching. This opens up considerations for initially pure GPU implementations, to copy a data array to CPU memory, perform the reduction on the CPU, and copy the reduced value back to GPU memory. This can be beneficial if the memory transfer can be hidden and the computations can overlap, i.e. if the reduction is performed on the CPU, while the GPU is already processing the next iteration step. The conjugate gradient algorithm, for example, could be adapted in a way that it uses values for α, β and the residual ϵ calculated from the previous iteration. This obviously alters the algorithm slightly and will probably make its convergence slower, but results in a more efficient implementation. The trade-off would still need to be investigated in more detail.

E. Elemental kernel evaluations

The most computationally demanding part of the considered methods is the evaluation of the elemental Helmholtz operator, as detailed in Appendix A, and the evaluation of the energy functional in the mesh optimiser, as detailed in reference [15]. Both have operator counts akin to a level 2 BLAS operation for each element, so that they can achieve a higher arithmetic intensity and achieve a higher occupancy of the hardware's compute resources. In a naive implementation these operators or kernels will still be memory-bound, though. It follows that ensuring an efficient memory access is more important than finding an algorithm that just minimises the number of compute-operations. An optimal performance of a GPU application is achieved, when the choice of data structures and algorithms is properly coupled, as we discuss in the following.

1. Hierarchical parallelisation approaches

Each of the elemental Helmholtz operations or elemental evaluations of the energy functional in the mesh optimiser can be performed independently and thus in parallel. There are however multiple options to map the algorithmic parallelism to the parallel level of the hardware. As explained before, both modern CPUs and GPUs have two levels of parallelism, the outer over the number of cores or SMX, the inner over the vector lanes or *CUDA* warps. All data entries within the vector lane or warp need to execute as one SIMD unit, so they execute the same mathematical or logical operation at any one compute cycle. This makes the processors more efficient, but poses a considerable challenge to the algorithmic design.

The elemental operation itself can be computed in parallel, too, at least when using high-order finite element discretisations, as within *Nektar++*. For classic finite element or finite volume methods this would not be the case. High-order finite element methods allow to compute the different modes or nodal points within each element in parallel. For 2D elements these are rather few, but for 3D element can be relatively large, a 4th order hexahedra for example already has $5^3 = 125$ modes. The consideration which parallelising approach to pursue, will thus depend on the dimension and polynomial order of elements within the domain.

Three different work distributions on the GPU can now be discussed:

1) each *CUDA* kernel or GPU processes one set of data;
2) each *CUDA* block or streaming multiprocessor (SMX) processes one set of data;
3) each *CUDA* thread or core processes one set of data.

Option (a) is only suitable for a single, very large operation and hence can be discarded for reasonable elemental orders. For purely spectral codes, it might be a suitable option, though. Option (b) is mostly suited to operations on rather few, but large sets of data. The individual sets might vary in size and might undergo different operations, without inducing performance breakdowns. Option (c) is only suited for operations on many, rather small sets of data. For good performance, however, the individual sets should all have the same size and undergo the exact same operations. As a general guideline, option (b) is suited to 3D elements of higher polynomial order, whereas option (c) is suited for 2D elements of low polynomial order.

We have taken option (b) for our implementation of the mesh optimiser for tetrahedral elements with nodal discretisation of 2nd to 5th order. This way there were always enough nodal points to undergo the same parallel operations for each warp or block. Option (b) is additionally more suitable, as the reverse line search in the optimiser means that each element could undergo a different number of iterations within the optimisation kernel. Without a substantial change of the algorithm that would ensure each node undergoes the same number of steps in the reverse line search, option (c) could not even be implemented.

In option (c), all data sets need to have the same size and undergo the same overall operation, so that 32 threads can always be efficiently combined into a warp. Here, a sufficiently large number of the small data sets is required to fully occupy the number of concurrent warps that need to be scheduled. This is the case for the elemental Helmholtz operation, as each element undergoes exactly the same operation and each mesh domain will consist of sufficiently many elements.

In our initial Helmholtz implementation dynamic memory allocation was utilised to account for varying polynomial orders of elements. However, the *OpenACC* implementation for the GPU was poorly performing, so we switched to static memory allocation within each kernel. This way the memory allocations are known at compile time and the compiler can create more efficient kernels. We then resorted to a templating approach to still accommodate for varying array lengths of varying polynomial orders. We consider it the best solution in this case, but it means another step away from an object-oriented programming style.

IV. Performance evaluation

Within this section we are presenting some performance results of different parallel implementations of the mesh optimiser and the implicit Helmholtz solver.

A. Mesh optimiser test case

For the strong scaling exercise of the mesh optimiser, we use an initially valid, but non-optimised mesh of 33K third-order tetrahedral elements and optimise it for 10 iterations. For this exercise we employ a 24 core Intel(R) Xeon(R) E5-2670v3 @ 2.30GHz processors with a maximum FP64 performance of 883GFLOPs and 128GB RAM with 1600Hz. As the compiler we use gcc 4.9.2 with the `-O3` optimisation flag. We compare three different CPU versions of the code, (a) the benchmark *Nektar*++ implementation with a *pthread* thread-scheduling and extensive object-oriented data structures, (b) a similar implementation using the *Kokkos* thread-scheduling with its *OpenMP* backend, and (c) a complete *Kokkos* version with its *OpenMP* backend using only *plain old data* structures.

The *Kokkos* programming model further allows to specify a *CUDA* backend for execution on Nvidia GPUs. We hence conduct a further test series comparing different CPU and GPU architectures using four different tetrahedral meshes of 2nd to 5th polynomial order with 96k elements to fully occupy the hardware. As a *manycore* accelerator system we employ an Intel Xeon Phi 7210 of the *Knights Landing* architecture, consisting of 64 cores operating at 1.3GHz with a theoretical FP64 peak performance of 3072GFLOPs. As GPUs we employ a Nvidia Tesla K40, a

Nvidia GTX 1070, and a Nvidia Pascal P100 with FP64 peak performance of 1680GFLOPs, 238.44GFLOPs, and 5304.32GFLOPs, respectively. The executables have been compiled using gcc 4.8.5 with the `-O3` optimisation flag and limiting the register count using the `-maxregcount 128` flag.

1. Cost evaluation

Despite their similarities, CPUs and GPUs are still different architectures, which makes it difficult to compare their performance in a fair way. Just considering runtimes would ignore the often much higher acquisition costs of Nvidia Tesla GPUs. Similarly, considering percentages of achieved FLOPs or memory bandwidth alone, would ignore the greater difficulty of using the GPU memory or cache hierarchy efficiently. We hence propose to look at two fundamental constraints: runtimes *and* costs accrued during these runtimes. We establish these costs by investigating *bare-metal* cloud computing prices per hour for similar systems that we employed for our test-cases. More details on this method can be found in reference [15].

B. Helmholtz solver testcase

We specify a generic testcase and solve the inhomogeneous Helmholtz equation on a square domain $\Omega = \{(x, y) \in \mathbb{R}^2 | -1 \leq x \leq 1, -1 \leq y \leq 1\}$ and solve the following equation:

$$\nabla^2 u + 2.5u = -(\lambda + 2\pi^2)\cos(\pi x)\cos(\pi y) \qquad \text{on } \Omega \tag{24}$$

$$u = \cos(\pi x)\cos(\pi y) \qquad \text{on } x = \{-1, 1\} \tag{25}$$

$$\frac{\partial u}{\partial n} = \pi\cos(\pi x)\sin(\pi y) \qquad \text{on } y = \{-1, 1\}. \tag{26}$$

Our meshes consist of triangular straights-sided elements, but varying element polynomial orders between 2nd and 11th order and around 4 Million degrees of freedom for each mesh, which will ensure the GPU hardware is fully occupied.

Our applications are written with the *OpenACC* programming model, which allows us to use the same code-base for CPU and GPU architectures. As discussed, we only use varying algorithms for the *gather* kernel. At compile time, we set the relevant flags for each architecture, `-ta=multicore` for multicore CPUs or `-ta=tesla` for Tesla type Nvidia GPUs. For all cases, we use the pgc++-18.4 compiler with optimisation flag `-O3`. For the multi-core system, we use a 16-core Intel(R) Xeon(R) E5-2690 v0 @ 2.9Ghz processor and 64GB RAM with 1600Hz. Its peak FP64 performance is 371.2GFLOPs with a maximum memory bandwidth of 51.2GB/s. For the GPU system we use a Nvidia Tesla P100 with a peak FP64 performance of 5304GFLOPS and a maximum bandwidth of 549GB/s.

As a benchmark we also present the results obtained with our basic *MPI* implementation within *Nektar++*. It uses the same algorithms as the CPU version, but realises its parallel execution on a multicore CPU using domain decomposition.

C. Effect of using *plain old data* structures

The strong-scaling timings for all three different CPU versions of the mesh-optimisation testcase are shown in Figure 2. It can be seen, that the benchmark implementation scales almost linearly, whereas the two *Kokkos* versions do not scale that well. This can only be due to a non-optimal thread-scheduling in the *Kokkos* programming model. The important observation however is, that replacing object-oriented data structures with *plain old data* structures significantly improves performance on multicore CPUs, buy a factor of 9.5. It underlines how important an efficient data access is to achieve good performance on all modern hardware, given that most applications are memory bound.

The same effect can be observed for the Helmholtz solver. For a strong scaling exercise using the 16-core CPU and a 7th order triangular mesh with 116k elements, we are comparing our benchmark *MPI* implementation with our *OpenACC* implementation. The *OpenACC* version does not scale as well as the *MPI* version, probably due to the reduction and gather operations. For low numbers of cores, however, it is faster, indicating that replacing object-oriented data structures with efficient *plain old data* structures is beneficial.

D. Benefits of portability to GPU systems

Once a code has been fully adjusted for using a portable programming model, it can be readily compiled for different compute architectures. The performance of the mesh-optimiser on a number of systems is shown in Figure 4.

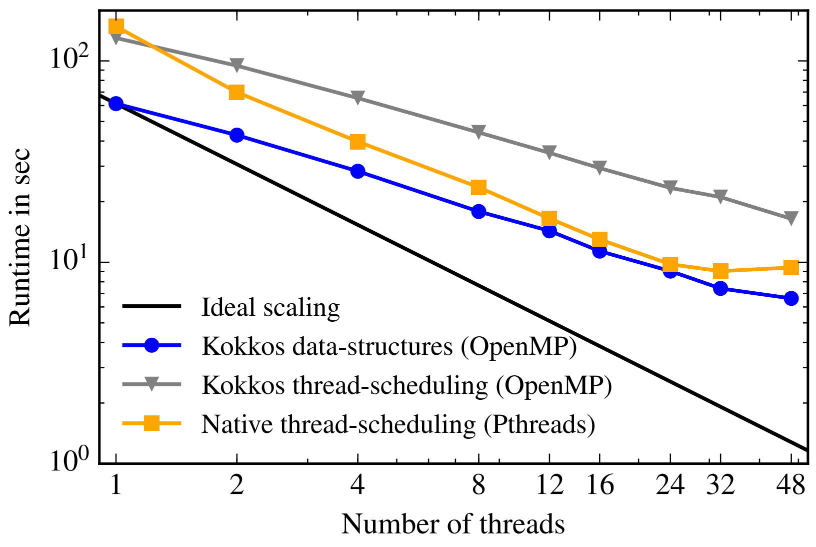

Fig. 2 Strong scaling of different CPU implementations of the high-order mesh optimisation method.

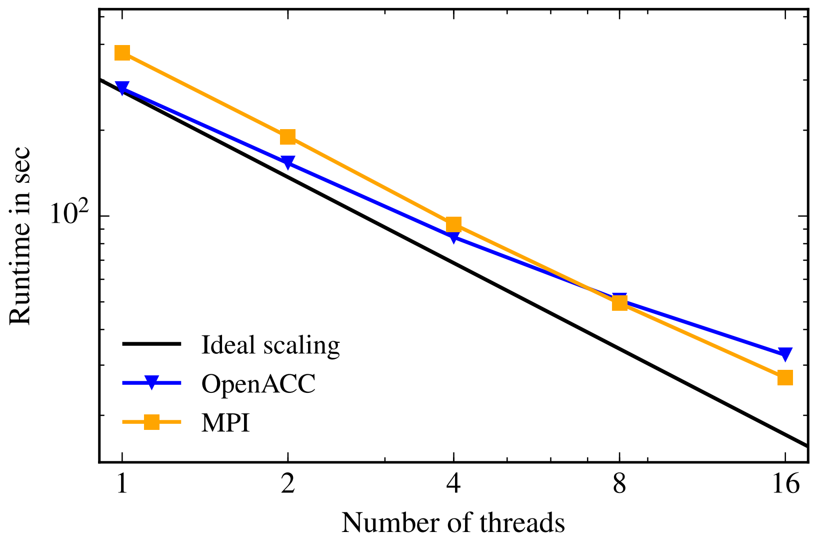

Fig. 3 Strong scaling of the overall Helmholtz solver on a 16-core CPU

It clearly shows, that using a Tesla P100 or GTX 1070 GPU system is more beneficial than using the multicore CPU system. Depending if time or costs are more important, one of these two GPUs could be chosen over the other. The older generation Kepler K40 GPU is poorly performing due to its much slower memory, that again is the bottleneck in this application. It has also been possible to run the code on the more exotic *Knights Landing* system, but without realising any performance benefits compared with the two latest GPU systems.

1. Detailed performance evaluation

In order to evaluate how well the different kernels utilise the GPU hardware, we present the achieved percentage of peak FP64 FLOPs in Figure 5 and of the DRAM bandwidth in Figure 6. 60% to 95% of the runtime is spend in the elemental Helmholtz kernel, that subsequently dominates the overall results metric. The performance for different polynomial orders is very similar. Lower polynomial orders are processed less efficiently on the hardware, but also require less operations per degree of freedom. The result values of less than 2% of peak FLOPs are not optimal, whereas the almost complete utilisation of the DRAM bandwidth shows that most kernels are throttled by this memory level.

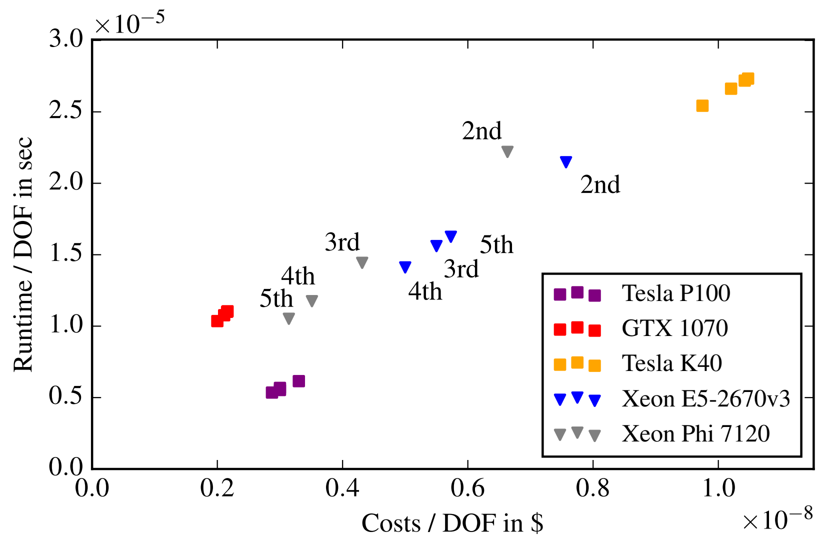

Fig. 4 Comparison of costs and runtime per DOF for different systems and varying element order of the mesh optimiser.

The CG Loop, CG Reduction, the *gather* and the *scatter* function all deal with large arrays and perform only a few operations on them, so that very few data entries can be re-used and would allow for intermediate storage in cache. This is different for the elemental Helmholtz kernel, in which data values are re-used and thus the bottleneck of this kernel will be the bandwidth of a higher cache level. The resulting values of the gather kernel include the overhead of the repeated kernel launches and are hence lower.

The elemental Helmholtz kernel could be improved further, by reducing the load operations from DRAM memory. This can be achieved by re-designing the algorithm so that more values like mappings and quadrature metrics are computed on-the-fly, rather than precomputed and loaded.

V. Conclusions

We have presented a methodology to port a legacy CPU code to modern CPU and GPU architectures and achieve good performance.

If the legacy code employs object-oriented data structures, these need to be re-factored to *plain old data* structures first. This is necessary because current GPU systems and their compilers can only deal with plain data structures, and hence all portable programming models based on them have to adhere to this principle, too. As in the case of our mesh-optimiser example, this can lead to substantial algorithmic and syntactical changes in the code-base. These efforts can be very beneficial, though, decreasing runtimes by an order of magnitude in our case.

Reduction or *gather*-algorithms can pose another challenge for efficient GPU implementations. They can be accelerated though, using colouring algorithms that avoid obtaining race-conditions. These are not necessary for CPU implementations, so that *gather* algorithms provide an example, for which algorithmic divergence between architectures is beneficial.

In our two examples, we still pre-compute and store repeatedly used values, as in most legacy applications. These will however result in applications that are heavily memory bound and thus do not perform optimally on GPUs. It would be beneficial to recompute these values in each iteration and variably adjust the arithmetic intensity depending on the underlying hardware.

The multiple levels of parallelism of the hardware and the algorithms requires a careful evaluation on how to map the workload of elemental kernels. Depending on the amount of work each element could either be assigned to a *CUDA* block or even a *CUDA* thread. We chose the first option for our 3D mesh optimiser and the latter for our 2D Helmholtz solver.

We can conclude that performance portable programming models allow to evolutionary adapt legacy code-bases. However, they are still only a compromise between implementational efforts, portability and performance. Alternative approaches or combinations of approaches should be investigated. These could be auto-tuning frameworks and

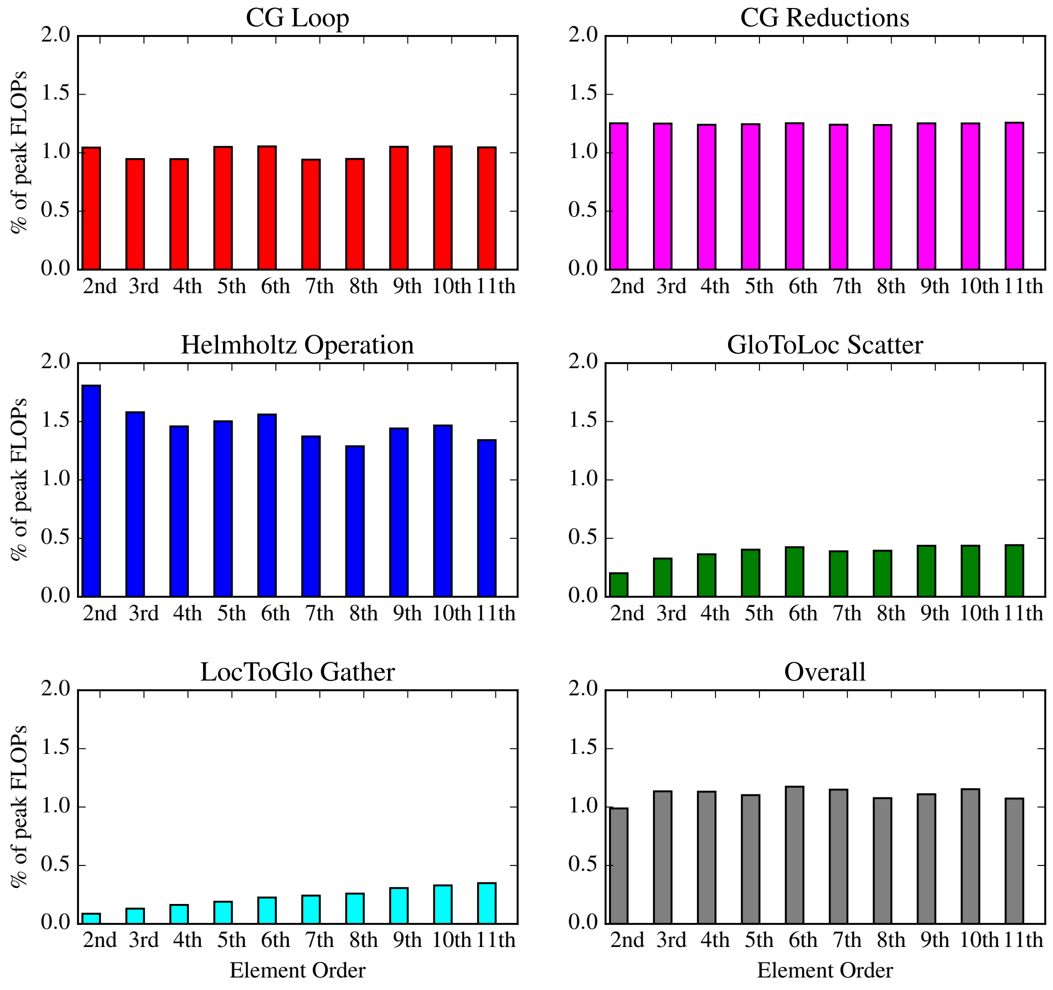

Fig. 5 Achieved FP64 FLOPs for the main kernels of the Helmholtz solver on a P100 GPU.

just-in-time compilations that test different algorithmic variants on a given hardware and automatically decide on the most performant one. These variants could then be coded either in a portable programming language like *OpenACC* or alternatively also directly in a low-level language to gain more performance.

Acknowledgements

JE acknowledges the support by EPSRC and the President's Scholarship of Imperial College London. DM acknowledges support from the EU Horizon 2020 project ExaFLOW (grant 671571). The Quadro P5000 GPU used for this research was donated by the NVIDIA Corporation.

References

[1] "ExaFLOW project," , 2018. URL `http://exaflow-project.eu/index.php`.

[2] "Exascale Computing Project," , 2018. URL `https://exascaleproject.org/`.

[3] Turner, M., Peiró, J., and Moxey, D., "A Variational Framework for High-Order Mesh Generation," *Procedia Engineering*, Vol. 163, 2016, pp. 340–352. doi:10.1016/j.proeng.2016.11.069.

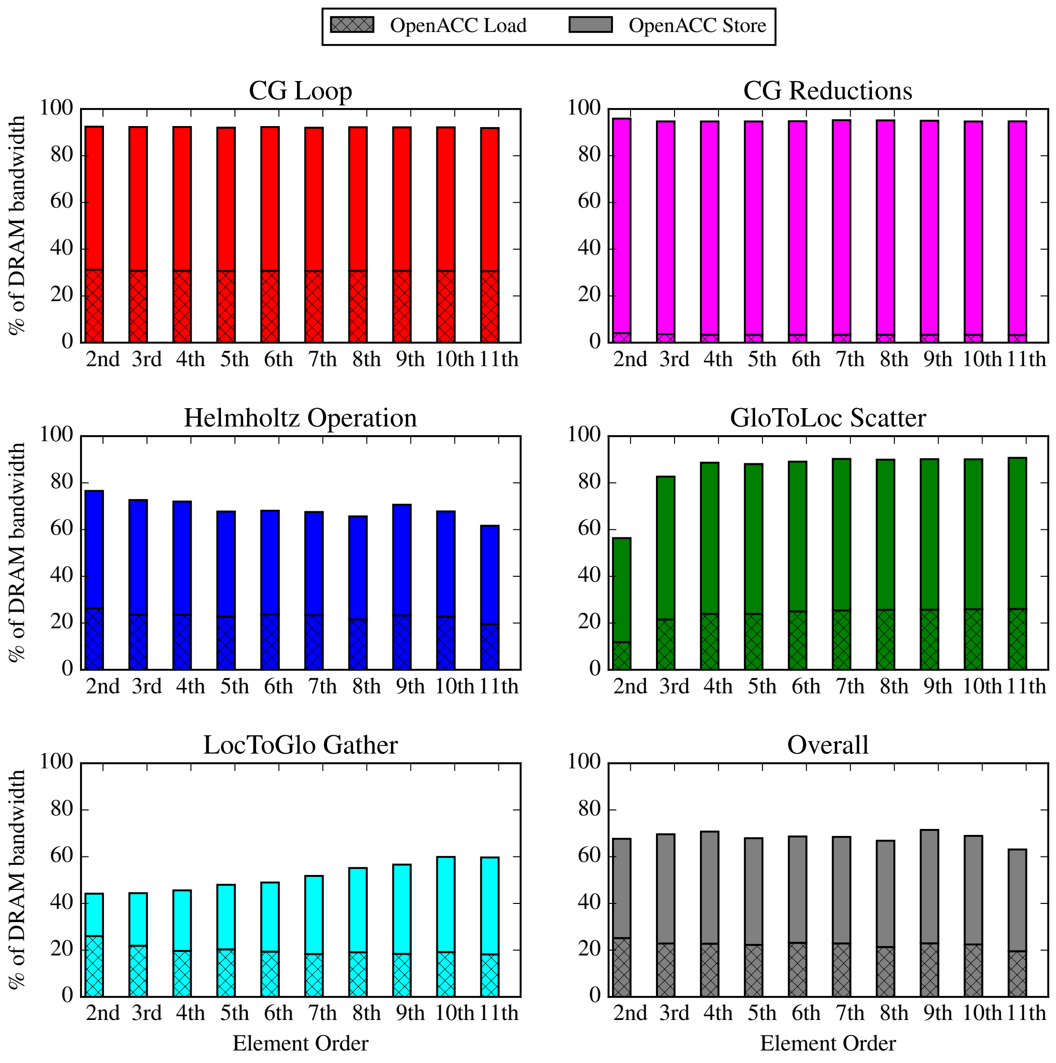

Fig. 6 Load and store DRAM memory bandwidth for the main kernels of the Helmholtz solver on a P100 GPU.

[4] Turner, M., Moxey, D., Sherwin, S. J., and Peiró, J., "Automatic Generation of 3D Unstructured High-Order Curvilinear Meshes," *ECCOMAS Congress 2016 VII European Congress on Computational Methods in Applied Sciences and Engineering*, 2016, pp. 5–10.

[5] Karniadakis, G. E., Israeli, M., and Orszag, S. A., "High-Order Splitting Methods for the Incompressible Navier-Stokes Equations," *Journal of Computational Physics*, Vol. 97, No. 2, 1991, pp. 414–443. doi:10.1016/0021-9991(91)90007-8, URL `http://dx.doi.org/10.1016/0021-9991(91)90007-8`.

[6] Cantwell, C. D., Moxey, D., Comerford, A., Bolis, A., Rocco, G., Mengaldo, G., De Grazia, D., Yakovlev, S., Lombard, J. E., Ekelschot, D., Jordi, B., Xu, H., Mohamied, Y., Eskilsson, C., Nelson, B., Vos, P., Biotto, C., Kirby, R. M., and Sherwin, S. J., "Nektar++: An open-source spectral/hp element framework," *Computer Physics Communications*, Vol. 192, 2015, pp. 205–219. doi:10.1016/j.cpc.2015.02.008, URL `http://www.sciencedirect.com/science/article/pii/S0010465515000533`.

[7] "OpenMP 4.5 Specifications," , 2015. URL `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

[8] The Open Group, "POSIX Threads," , 1997. URL `http://pubs.opengroup.org/onlinepubs/007908799/xsh/threads.html`.

[9] Nvidia, "CUDA8.0 Release Note," , 2016. URL `http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Release_Notes.pdf`.

[10] "MPI 3.1 Specifications," , 2015. URL `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[11] Khronos Group, "The OpenCL Specification," , 2015. URL `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`.

[12] Witherden, F. D., Farrington, A. M., and Vincent, P. E., "PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, Vol. 185, No. 11, 2014, pp. 3028–3040. doi:http://dx.doi.org/10.1016/j.cpc.2014.07.011, URL `http://www.sciencedirect.com/science/article/pii/S0010465514002549`.

[13] "OpenACC Programming and Best Practices Guide," , 2015. URL `http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf`.

[14] Carter Edwards, H., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, 2014, pp. 3202–3216. doi:10.1016/j.jpdc.2014.07.003.

[15] Eichstädt, J., Green, M., Turner, M., Peiró, J., and Moxey, D., "Accelerating high-order mesh optimisation with an architecture-independent programming model," *Computer Physics Communications*, Vol. 229, 2018, pp. 36–53. doi:10.1016/j.cpc.2018.03.025, URL `https://doi.org/10.1016/j.cpc.2018.03.025`.

[16] Karniadakis, G., and Sherwin, S., *Spectral/hp Element Methods for Computational Fluid Dynamics*, 2nd ed., Oxford University Press, 2005.

[17] Cantwell, C. D., Sherwin, S. J., Kirby, R. M., and Kelly, P. H. J., "From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements," *Computers and Fluids*, Vol. 43, No. 1, 2011, pp. 23–28. doi:10.1016/j.compfluid.2010.08.012, URL `http://dx.doi.org/10.1016/j.compfluid.2010.08.012`.

[18] Jacobs, C. T., Jammy, S. P., and Sandham, N. D., "OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures," *Journal of Computational Science*, Vol. 18, 2017, pp. 12–23. doi:10.1016/j.jocs.2016.11.001, URL `http://dx.doi.org/10.1016/j.jocs.2016.11.001`.

[19] Nvidia, "cuBLAS Library," , 2016. URL `http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf`.

[20] Orszag, S. A., "Spectral methods for problems in complex geometries," *Journal of Computational Physics*, Vol. 37, No. 1, 1980, pp. 70–92. doi:10.1016/0021-9991(80)90005-4.

[21] Moxey, D., Cantwell, C. D., Kirby, R. M., and Sherwin, S. J., "Optimising the performance of the spectral/hp element method with collective linear algebra operations," *Computer Methods in Applied Mechanics and Engineering*, Vol. 310, 2016, pp. 628–645. doi:10.1016/j.cma.2016.07.001.

A. Appendix A: Elemental Helmholtz Operator

The elemental Helmholtz operation for a 2D element is the combination of the Laplacian operator and the mass operator given as

$$\hat{\boldsymbol{s}}_l^e = \boldsymbol{H}^e \hat{\boldsymbol{u}}_l^e = [\boldsymbol{L}^e + \lambda \boldsymbol{M}^e] \hat{\boldsymbol{u}}_l^e = [(\boldsymbol{D}_{x1}\boldsymbol{B})^T \boldsymbol{W} \boldsymbol{D}_{x1}\boldsymbol{B} + (\boldsymbol{D}_{x2}\boldsymbol{B})^T \boldsymbol{W} \boldsymbol{D}_{x2}\boldsymbol{B} + \lambda \boldsymbol{B}^T \boldsymbol{W} \boldsymbol{B}]\ \hat{\boldsymbol{u}}_l^e$$

Here the action of the basis matrix $\boldsymbol{B}$ with entries $B_{i,j} = \phi_j(\xi_i)$ denotes the backward-transform from coefficient space to physical space. The diagonal weight matrix $\boldsymbol{W}$ with $W_{i,j} = J_{i,j} w_i^1 w_j^2$ includes the Jacobian J, that is the mapping $\chi^e : \Omega_{st}^{tri} \to \Omega^e$ between the standard element defined on $\Omega_{st}^{tri} = \{\vec{\xi} \mid \xi_1, \xi_2 \geq -1, \xi_1 + \xi_2 \leq 1\}$ and the curvilinear element on coordinates $\vec{x} \in \Omega^e$, as well as the quadrature weightings w_i^1 and w_j^2.

Note that the use of the C^0 basis leads to a discretisation where triangles are represented under a collapsed coordinate system through a square-to-triangle Duffy transformation, which permits the use of a tensor-product quadrature even for triangular elements. To avoid issues with singular points of the Duffy transform, we use Gauss-Lobatto-Legendre quadrature for w^1 and Gauss-Radau quadrature for w^2. The derivative matrices $\boldsymbol{D}_{x1}$ and $\boldsymbol{D}_{x2}$ with respect to coordinate directions x_1 and x_2 are given as

$$\begin{aligned}
\boldsymbol{D}_{x1} &= \boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_1}\right)\boldsymbol{D}_{\xi 1} + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_1}\right)\boldsymbol{D}_{\xi 2} \\
\boldsymbol{D}_{x2} &= \boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_2}\right)\boldsymbol{D}_{\xi 1} + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_2}\right)\boldsymbol{D}_{\xi 2},
\end{aligned}$$

with the diagonal coefficient matrices $\boldsymbol{\Lambda}$, which depend on the derivatives of the mapping χ^e. Finally, the block-diagonal derivative matrices $\boldsymbol{D}_{\xi 1}$ and $\boldsymbol{D}_{\xi 2}$ denote the actions of the derivative with respect to the standard element, so that

$$\boldsymbol{D}_{\xi i} = \frac{\partial}{\partial \xi_i}\ ;\ i = 1, 2.$$

To minimise the operator count for an efficient CPU implementation, the elemental matrices $\boldsymbol{W}$ for the mass operator have been precomputed and stored. For the elemental Laplacian operators, the following matrices have been precomputed and stored:

$$\boldsymbol{L}_{ij} = \boldsymbol{D}_{xi}^{T}\boldsymbol{W}\boldsymbol{\Lambda}\left(\frac{\partial \xi_j}{\partial x_i}\right)\ ;\ i, j = 1, 2.$$

Where possible, we additionally exploit the tensor-product construction of the C^0 basis to apply the sum-factorisation technique [20]. This is a widely used strategy for attaining substantial reductions in operator count or tensor-product operations, including backwards transforms and derivatives. More details of this technique applied to the spectral element method on triangles and other higher dimensional shape types can be found in references [16, 17, 21].