Computer Physics Communications 255 (2020) 107245

Contents lists available at ScienceDirect

Computer Physics Communications

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

A comparison of the shared-memory parallel programming models *OpenMP*, *OpenACC* and *Kokkos* in the context of implicit solvers for high-order FEM☆

Jan Eichstädt [a,*], Martin Vymazal [a], David Moxey [b], Joaquim Peiró [a]

[a] Department of Aeronautics, Imperial College London, United Kingdom of Great Britain and Northern Ireland
[b] College of Engineering, Mathematics and Physical Sciences, University of Exeter, United Kingdom of Great Britain and Northern Ireland

ARTICLE INFO

Article history:
Received 28 January 2019
Received in revised form 19 February 2020
Accepted 25 February 2020
Available online 2 March 2020

Keywords:
Shared-memory parallel programming models
OpenMP
OpenACC
Kokkos
Helmholtz equation
FEM

ABSTRACT

We consider the application of three performance-portable programming models in the context of a high-order spectral element, implicit time-stepping solver for the Navier–Stokes equations. We aim to evaluate whether the use of these models allows code developers to deliver high-performance solvers for computational fluid dynamics simulations that are capable of effectively utilising both many-core CPU and GPU architectures. Using the core elliptic solver for the Navier–Stokes equations as a benchmarking guide, we evaluate the performance of these models on a range of unstructured meshes and give guidelines for the translation of existing codebases and their data structures to these models.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

The use of computational fluid dynamics (CFD) in engineering design is now well-established practice. Such usage requires quick turnover of simulations in order to provide meaningful input into the design process. Simpler models such as Reynolds-averaged Navier–Stokes (RANS) and detached eddy simulation (DES) provide a route to computationally less expensive simulations, but at the cost of substantial reduction in accuracy, particularly for geometries admitting highly unsteady and transitional flows. Although higher fidelity models and, in particular, large eddy simulations (LES) provide a route to increased accuracy for these flows, they come at substantial increases in computational cost. Major efforts within CFD are therefore targeted at developing efficient LES codes and give the capability for overnight production runs of industrial scale LES.

One of the main issues that has been identified in producing such codes is the choice of numerical method that underpins the simulation. Low-order finite volume methods have long been the staple choice of industrial CFD software. However, such methods come with relatively low levels of arithmetic intensity: the balance between floating point operations (FLOP) and memory bandwidth. At the same time, current architectural trends in high performance computing (HPC) systems steer algorithm design towards higher arithmetic intensity, since memory latency and bandwidth continuously undergo a slower improvement compared with FLOP performance increases. This naturally limits the performance gains that can be realised on modern hardware using lower-order methods. To date therefore, the combination of low-order methods with LES models has typically not shown a sufficient increase in fidelity in the resulting solution fields to justify the additional computational cost.

This has driven CFD practitioners to contemplate alternative numerical discretisations with more favourable performance characteristics. In particular, higher order finite element methods and their variants, such as discontinuous Galerkin and flux reconstruction methods, offer an attractive alternative to classical low-order methods: numerically, low diffusion and dispersion error enable flow structures to be more accurately propagated across long convective lengths, and their higher arithmetic intensity allows for a more effective balance of floating point operations against memory bandwidth, as demonstrated for example in reference [1].

Although the theoretical performance of modern hardware is continuously improving, the hardware itself is becoming increasingly heterogeneous. Software is thus increasingly challenging to code and maintain for different architectures, particularly those that rely on a mixture of CPU and GPU hardware. This has lead to interest in how different programming models can

☆ The review of this paper was arranged by Prof. David W. Walker.

* Corresponding author.
E-mail address: jan.eichstaedt13@imperial.ac.uk (J. Eichstädt).

https://doi.org/10.1016/j.cpc.2020.107245
0010-4655/© 2020 Elsevier B.V. All rights reserved.

be constructed in order to better separate algorithmic from implementational concerns, providing performance portable and architecture independent models.

The aim of this work is to assess three such programming models, namely *Kokkos* [2], *OpenMP* [3], and *OpenACC* [4], in the context of high-order methods for CFD. Aspects to be compared are the implementational effort and code maintainability, as well as the performance and its portability across architectures. In particular, we focus on implicit time-stepping Navier–Stokes solvers, as turbulent flow physics drives the algorithmic choices towards these schemes which avoid severe CFL number restrictions associated with explicit schemes.

This work provides novelty from a number of perspectives. Firstly, although performance portability aspects of individual programming models are regularly assessed, the corresponding results are still not meaningful enough for our purpose. Common benchmark problems are most often based on kernels for BLAS operations, or so-called *mini-applications* or *mini-apps* for a variety of solver types. Examples of these can be found in a number of parallel benchmark suites, e.g. [5–9]. In this approach, stripped-down standalone versions of the core algorithms that form complete solvers are implemented. This allows for a simpler development environment, whilst still ensuring generally representative performance characteristics. Such mini-apps then allow for easier performance benchmarking and experimentations with programming models and other aspects, such as utilised hardware or compiler versions. While each mini-app solver can consist either of bespoke or of standard BLAS-type kernels, the relative weights of these kernels and their interactions will still be an important factor in the overall performance, so that isolated evaluations of individual kernels alone will not yield meaningful enough results. The mini-application that comes the closest to our purpose would be *miniFE* of the *Mantevo* suite [6]: however, this is only for a linear finite element solver, whereas we are interested in high-order finite element solvers.

In this article we will follow this approach to develop a mini-app to model the performance characteristics of a complete Navier–Stokes flow solver, as implemented within the spectral/*hp* element framework *Nektar++* [10]. While we make use of the *Nektar++* framework for pre- and post-processing, the time-stepping loop that we monitor for performance has been cleared of typical *Nektar++* features and flattened to generic *C++* code, before applying the parallel programming models. Further to this, we note that the complete Navier–Stokes implementation requires particularly heavy implementation effort that does not necessarily affect computational performance (e.g. from specialised boundary conditions). We therefore select a simpler, more lightweight solver, that still contains the relevant features of the complete solver; in our case this will be an implicit solver for the 2D Helmholtz equation. We note that this forms the essential implicit solve for both pressure Poisson and velocity correction steps within the commonly used fractional time-stepping scheme for Navier–Stokes equations [11]. The matrices arising from the discretisation are positive definite and, unless the mesh is highly distorted, well-conditioned and thus we employ the conjugate gradient method with a Jacobi preconditioner. The algorithm is executed in a matrix-free fashion, by which we mean that neither a large sparse assembled global matrix nor a series of dense local elemental matrices are constructed. Instead, the action of the matrix-vector multiplication is evaluated via the definition of the discrete Helmholtz operator, in order to reduce the memory bandwidth requirements of the algorithm and improve arithmetic intensity. As spatial discretisation we employ triangular elements with a modal continuous Galerkin (CG) formulation from [12]. The modal basis is formed with a tensor product, so that we can use the efficient sum-factorisation technique within elemental operations.

A second novel aspect of this work is the focus on implicit versus explicit time-stepping routines. Certainly the combination of using explicit time-stepping for high-order methods alongside performance portable and architecture independent methods running on both CPUs and GPUs has been demonstrated in solvers such as *PyFR* [13]. Additionally, work by other finite element groups such as *deal.II* [14] and *Dune* [15] on mostly tensor product quadrilateral and hexahedral elements has been conducted, but typically from the CPU perspective. In this work, we aim to demonstrate how architecture-independent programming can be applied from the context of implicit time-stepping.

The paper is structured as follows. In Section 2 we give a brief overview of current portable-performance programming models that we will consider in this work, alongside the architectures under which they will be applied. Section 3 outlines the mathematical model under consideration for the Helmholtz model, and its implementation is discussed in Section 4. We conduct a thorough performance evaluation in Section 5 on both many-core CPU and GPU hardware, before giving conclusions and a summary in Section 6.

2. Parallel hardware and portable programming models

Current HPC architectures are characterised by a high degree of parallelism. It does not only come 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. It is this shared memory parallel hardware and the programming model to make use of it, that we address in this work.

2.1. Multi-core and many-core architectures

The parallelism of multi-core CPUs comes on two different levels, firstly over the multiple compute cores, and secondly over the vector lanes within each core. These are addressed by the compiler using AVX instructions. The length of the vector lane of the CPU that we use for our performance benchmarks has 256 bits or 4 doubles.

Nvidia GPUs, an instance of many-core architectures, also provide their parallelism in two levels. The first level consists of a number of streaming multiprocessors (abbreviated as SM or SMX), with the second being the individual streaming processors (SP) or *CUDA cores* that lie on each SMX. Groups of 32 *CUDA* cores have to execute as an individual SIMT unit, making these comparable to vector lanes on CPU architectures.

At the same time, memory bandwidth of both these architectures is limited, particularly on GPUs. Algorithms therefore need a high arithmetic intensity – the ratio of compute operations to memory accesses – to benefit from the available computational resources. For example, the Nvidia P100 GPU has a double-precision FLOP-to-byte ratio of 9.66 FLOP/byte, based on a theoretical peak of 5304 GFLOP/s and 549 GB/s memory transfer. This level of arithmetic intensity therefore requires level-3 BLAS or similar operations to fully utilise all *CUDA* cores.

To address the parallelism of the CPU, the programming models (for example *OpenMP* or *Pthreads*) schedule one threads per core (or two in case of hyper-threading). Vector level parallelism is much harder to achieve; it can be done explicitly by using assembler-level intrinsic instructions or with a number of implicit options like compiler auto-vectorisation, by using Intel *Cilk Plus C/C++ Extensions for Array Notations* or the Intel *IPP* or *MKL* libraries [16].

Nvidia GPUs are addressed using the *CUDA* [17] programming model, which allows operations to be explicitly assigned to all parallelism levels. The use of object-orientation programming

must, however, be done with care: while seamless integration of *CUDA* kernels into *C++* code is possible, there are restrictions on what can be done within a kernel in terms of data management, which could otherwise have been conveniently hidden with object-oriented features. Most notably in our work are the limitations to offload instances of classes (for example individual mesh elements of different types) to the GPU and iterate over all instances and implicitly calling their member functions and data, especially if they stem from class inheritance and virtual functions.

All these aspects combined explain the difficulty to achieve good shared-memory parallel performance, compared with distributed memory performance, enabled by domain decomposition. A good *MPI* distributed memory implementation 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 additionally employed. Yet, the additional fine-grained level of shared memory parallelism on the hardware needs to be exploited by the application, both through parallel algorithms, as well as programming-models and compilers that can map the algorithmic parallelism to the underlying hardware. It is this aspect that often results in less ideal strong-scaling within a compute node.

2.2. Kokkos *versus* OpenMP *and* OpenACC

The lifecycles of scientific applications are rather large, often spanning more than a decade. In comparison, hardware developments are deemed to proceed at an accelerated pace. HPC hardware has witnessed the advent and disappearance of vector processing machines over the 1970s and 1980s, and intermediate dominance of single-core multi-purpose CPUs until the mid 2000s. At this time the processor clock-speed has reached its energetic limit, resulting in the development of multi-core processors and subsequently longer vector-lanes. Additionally, through the introduction of *OpenCL* [18] and *CUDA* it became possible to programme massively parallel but light-weight graphics processing units (GPUs). The recent strong demand for machine learning hardware, that requires massively-parallel high-throughput and low-precision architectures will further shift the available hardware for scientific computing. Many current scientific computing frameworks originate from the single-core CPU era and hence require updated capabilities to benefit from the increased parallelism. The evolutionary approach will allow to adjust the existing frameworks step-by-step with a compatible parallel programming model. This avoids extensive rewriting of code, that often is in the order of 100 000 lines. It is this approach that we adopt in this work.

Alternatively, the code-base could be rewritten in a different low-level language like *CUDA*. However, this only allows to address the currently popular GPU hardware, and in view of the rapidly changing HPC architectures is not a sustainable solution. The low-level language *OpenCL* is portable between CPUs and almost any device architectures including Nvidia GPUs, AMD GPUs and FPGAs. *OpenCL* could therefore under this premise be a viable option for many users. However, it requires either substantial tuning efforts to yield good performance – limiting portability in practice, or the use of specialised libraries – which would either limit performance or complexity of our algorithms.

Another alternative that has been adopted by multiple groups is to design a new framework that comprises enough flexibility to address a broad range of current and future architectures. Examples of this include *PyFR* [13] for flux reconstruction methods, *Firedrake* [19] for FEM methods, and *OpenSBLI* [20] for finite difference methods. These frameworks consist of a high-level interface to describe the system that is to be solved, with an abstraction layer underneath this which translates high-level syntax to various low-level languages that can target different backends. Using a set of optimised BLAS libraries and/or bespoke kernels for each type of architecture, alongside good auto-tuning algorithms, can yield very competitive performance.

For this work we have identified three programming models that can be added to our existing *Nektar++* framework. As a requirement, they needed to support both CPU and GPU architectures to expose their parallel capabilities, have open-access compiler support and a syntax that allows seamless implementation in an existing *C++* codebase.

Kokkos [2] is a library consisting of a performance portable abstraction layer in *C++*, with *OpenMP* [3] and *Pthreads* [21] backend for multi-core CPUs and a *CUDA* [17] backend for Nvidia GPUs. At the time of conducting our study it only supported data- and no task-parallelism, which prevents load sharing between host and device. The compiler support is very wide, spanning the *gcc*, *clang-llvm*, and *pgi* open-access compilers. The *Kokkos* syntax is based on *Kokkos* data arrays, that map the data between CPU and device memory. Its unique quality is the polymorphic layout of the 2D arrays, that can automatically switch between row- and column-major layout, to achieve coalesced memory access on different architectures. Up to three levels of hierarchical parallelism can be specified and data arrays can be explicitly placed on different cache levels or texture memories. Other similar models based on libraries with multiple backends are *RAJA* [22], which has been found in a less mature state than *Kokkos* and *OCCA* [23].

OpenMP [3] is a programming model based on compiler directives or so called *pragmas*, that can be added to legacy codebases and specify parallel constructs. Initially it only supported parallel executions on multi-core CPUs, but offloading to target devices is supported from the *OpenMP4.0* specification onwards. The specification supports both task- and data parallelism, which would allow for a real heterogeneous execution and load sharing between both host and device. The compiler support for the target offloading, however, is very sparse. The *gcc-7* compiler supports only the most basic *parallel-for* construct. The most mature open-access compiler has been the experimental *clang-ykt* compiler branch. We will show that, whilst it was possible to implement, compile and execute our benchmark problem using this compiler, it is still lacking crucial features that prevent its enrolment within a complete framework like *Nektar++*.

OpenACC is a similar programming model to *OpenMP* that is based on compiler directives, but primarily targeted at offloading to devices. It supports both task and data parallelism. The *gcc-6* compiler support very simple offloading directives and is hence not complete enough for our purpose. The *pgi* compiler, however, 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 the *pgi*-compiler, but also proper multi-core CPUs. Without using a combined programming model of *OpenACC* for the device and *OpenMP* for the host parallelism, no heterogeneous execution on host and device is possible.

A rough equivalent of *OpenACC*, but with an *OpenCL* and not a *CUDA* backend is *SYCL* [24]. However, our preliminary assessment of the supporting ecosystem and first performance results were not encouraging and we did not explore it further.

3. Method

We consider the solution of the inhomogeneous Helmholtz equation for a scalar field variable u, defined in a domain Ω with boundary $\Gamma = \Gamma_D \cup \Gamma_N$, represented by the system

$$\nabla^2 u + \lambda u = f \qquad \text{on } \Omega \tag{1}$$

$$u = u^D \qquad \text{on } \Gamma_D \tag{2}$$

$$\frac{\partial u}{\partial n} = g \qquad \text{on } \Gamma_N, \tag{3}$$

where λ is a real coefficient, f is a forcing function, Γ_D and Γ_N denote the parts of the boundary where we impose Dirichlet and Neumann boundary conditions, respectively.

Defining the inner products

$$(a, b)_\Omega = \int_\Omega ab\, d\Omega \qquad \langle a, b\rangle_\Gamma = \int_\Gamma ab\, d\Gamma, \tag{4}$$

a weak Galerkin approximation to the system is obtained through the inner product with a shape function v, i.e.

$$\left(v, \nabla^2 u\right)_\Omega + \lambda\,(v, u)_\Omega = (v, f)_\Omega \tag{5}$$

We can now apply the divergence theorem to reduce continuity requirements and get

$$(\nabla v, \nabla u)_\Omega + \lambda\,(v, u)_\Omega = (v, f)_\Omega + \left\langle v, \frac{\partial u}{\partial n}\right\rangle_\Gamma \tag{6}$$

where the boundary integral allows to directly specify the Neumann boundary conditions. The Dirichlet boundary conditions are lifted from the solution, $u = u^H + u^D$, so that we retain the unknown homogeneous term, u^H, on the left-hand-side of the equation and all the known terms are moved to the right-hand side, namely

$$\left(\nabla v, \nabla u^H\right)_\Omega + \lambda\left(v, u^H\right)_\Omega = (v, f)_\Omega + \langle v, g\rangle_{\Gamma_N} - \left(\nabla v, \nabla u^D\right)_\Omega - \lambda\left(v, u^D\right)_\Omega \tag{7}$$

We can now discretise this expression to obtain a linear system of the form

$$H\,\hat{u}^H = \hat{b} \tag{8}$$

where H is the matrix of coefficients of the discrete Helmholtz operator, $\hat{u}^H$ is a vector containing the discrete unknowns and $\hat{b}$ is the vector that results from the evaluation of the right-hand side using the known values of the forcing function and boundary conditions.

3.1. Implicit solver

The linear Helmholtz equation system is to be solved iteratively using the conjugate gradient method. This method is suitable, as the Helmholtz operator is symmetric and positive definite.

The method is initialised with an arbitrary solution $\hat{u}_0^H$:

$$r_0 = \hat{b} - H^H \hat{u}_0^H \tag{9}$$

$$w_0 = Cr_0 \tag{10}$$

$$s_0 = H^H w_0 \tag{11}$$

$$\beta = 0 \tag{12}$$

$$\alpha = \frac{r_0^T w_0}{r_0^T w_0}, \tag{13}$$

where r is the residual vector, C is a suitable preconditioner, w and s are the conjugate gradient vectors, and β and α are step sizes. The method then loops over the following steps k, until the residual is smaller than a set tolerance $\epsilon > \epsilon_{tol}$.

$$p_{k+1} = \beta p_k + w_k \tag{14}$$

$$q_{k+1} = \beta q_k + s_k \tag{15}$$

$$\hat{u}_{k+1}^H = \alpha p_{k+1} + \hat{u}_k^H \tag{16}$$

$$r_{k+1} = r_k - \alpha q_{k+1} \tag{17}$$

$$w_{k+1} = Cr_{k+1} \tag{18}$$

$$s_{k+1} = H^H w_{k+1} \tag{19}$$

$$\beta = \frac{r_{k+1}^T w_{k+1}}{r_k^T w_k} \tag{20}$$

$$\alpha = \frac{r_{k+1}^T w_{k+1}}{s_{k+1}^T w_{k+1} - r^T k + 1 w_{k+1} \cdot \beta_{k+1}/\alpha_k} \tag{21}$$

$$\epsilon = r_{k+1}^T r_{k+1} \tag{22}$$

Here p and q are search direction vectors and $\hat{u}_{k+1}^H$ is the updated solution to the linear system that is to be solved. As the preconditioner C, we use a simple diagonal Jacobi preconditioner.

3.2. Discretisation of the Helmholtz equation

We now give a brief overview of the discretisation of the Helmholtz equation, which is described further in reference [12]. The previous algorithm is applicable to any spatial discretisation, but the size, structure and numerical properties of the matrices will depend on our choice of discretisation. Here we will be using a continuous Galerkin projection over triangular elements with a hierarchical modal basis. We note that the central aspect of using a matrix-free scheme is avoiding the assembly of the complete global matrix of the Helmholtz operator. Instead we are breaking down the global operator into as a sum of elemental operators. This implies that our global vector of solution coefficients $\hat{u}^H$ needs to be mapped first to the element local solution coefficients in a scatter-type operation:

$$\hat{u}_l = \boldsymbol{A}\hat{u}_g \tag{23}$$

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

$$\hat{u}_l[i] = v_{sign}[i]\ \hat{u}_g[v_{map}[i]] \quad \text{for } i \in [0 : n_{local}) \tag{24}$$

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

$$\hat{s}_l^e = \boldsymbol{H}^e \hat{u}_l^e \quad \text{for } e \in [0 : n_{el}) \tag{25}$$

Finally, the local solution coefficients need to be mapped back to the global solution coefficients in a gather-type operation:

$$\hat{s}_g = \boldsymbol{A}^T \hat{s}_l \tag{26}$$

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

$$\hat{s}_g[v_{map}[i]] = \hat{s}_g[v_{map}[i]] + v_{sign}[i]\ \hat{s}_l[i] \quad \text{for } i \in [0 : n_{local}) \tag{27}$$

Here it can be seen that boundary modes of neighbouring elements need to be added. In a parallel implementation this basic implementation can lead to a race-condition.

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

$$\hat{s}_l^e = \boldsymbol{H}^e \hat{u}_l^e \tag{28}$$

$$\hat{s}_l^e = [\boldsymbol{L}^e + \lambda \boldsymbol{M}^e]\hat{u}_l^e \tag{29}$$

$$\hat{s}_l^e = [(\boldsymbol{D}_{x_1}\boldsymbol{B})^T \boldsymbol{W}\boldsymbol{D}_{x_1}\boldsymbol{B} + (\boldsymbol{D}_{x_2}\boldsymbol{B})^T \boldsymbol{W}\boldsymbol{D}_{x_2}\boldsymbol{B} + \lambda \boldsymbol{B}^T \boldsymbol{W}\boldsymbol{B}]\ \hat{u}_l^e \tag{30}$$

The basis matrix $\boldsymbol{B}$ with $B_{i,j} = \phi_j(\xi_i)$ denotes the backward-transform from coefficient space to physical space, where ξ denotes a coordinate inside a reference element, ξ_i denotes a set of quadrature points on this element and ϕ_j are basis functions defined on the reference element. We note the use of a hierarchical modal basis gives rise to independent basis functions so that

$\phi_{m(i,j)}(\xi) = \phi_i^a(\xi_1)\phi_j(\xi_2)$, where $m(i,j)$ is an ordering to a vector of local degrees of freedom on each element. The diagonal weight matrix $\boldsymbol{W}$ with $W_{m(i,j)} = J_{m(i,j)} w_i w_j$ includes the Jacobian J (that is, the mapping between the standard element and the curvilinear element on coordinates x), as well as the quadrature weightings w_i and w_j in each of the coordinate directions. Following reference [12] we select a Gauss–Lobatto–Legendre quadrature for w_i and a Gauss–Radau quadrature for w_j to mitigate the effects of the singularity in the collapsed coordinate Duffy mapping. The derivative matrices $\boldsymbol{D}_{x_1}$ and $\boldsymbol{D}_{x_1}$ are

$$\boldsymbol{D}_{x_1} = \boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_1}\right)\boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_1}\right)\boldsymbol{D}_{\xi_2} \tag{31}$$

$$\boldsymbol{D}_{x_2} = \boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_2}\right)\boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_2}\right)\boldsymbol{D}_{\xi_2}, \tag{32}$$

with the diagonal coefficient matrices $\boldsymbol{\Lambda}$. The block-diagonal derivative matrices $\boldsymbol{D}_{\xi_1}$ and $\boldsymbol{D}_{\xi_2}$ are

$$\boldsymbol{D}_{\xi_1} = \frac{\partial}{\partial \xi_1} \tag{33}$$

$$\boldsymbol{D}_{\xi_2} = \frac{\partial}{\partial \xi_2} \tag{34}$$

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 diagonal matrices have been precomputed and stored:

$$\boldsymbol{L}_{11} = \boldsymbol{W}\left(\boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_1}\right)\boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_1}\right) + \boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_2}\right)\boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_2}\right)\right)$$

$$\boldsymbol{L}_{12} = \boldsymbol{W}\left(\boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_1}\right)\boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_1}\right) + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_2}\right)\boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_2}\right)\right)$$

$$\boldsymbol{L}_{21} = \boldsymbol{L}_{12}$$

$$\boldsymbol{L}_{22} = \boldsymbol{W}\left(\boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_1}\right)\boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_1}\right) + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_2}\right)\boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_2}\right)\right)$$

By breaking down the discrete elemental Helmholtz operator in the described fashion, the construction of the elemental matrix operator $\boldsymbol{H}^e$ is avoided and thus memory bandwidth is reduced and algorithmic intensity is increased. Where possible, we additionally exploit the tensor-product construction of the C^0 basis to apply the sum-factorisation technique [25]. This is a widely used strategy for attaining substantial reductions in operator count for tensor-product operations, including backwards transforms and derivatives. For example, the matrix–vector product $\boldsymbol{B}\hat{\boldsymbol{u}}$ would naively be calculated as:

$$u(\vec{\xi}) = \sum_p \sum_q \hat{u}_{pq}\phi_p(\xi_1, \xi_2)$$

which has around P^4 floating-point operations. Using sum-factorisation however, and noting that $\phi_{pq}(\xi_1, \xi_2) = \psi_p^1(\xi_1)\psi_q^2(\xi_2)$, this matrix–vector product can be expressed as:

$$u(\vec{\xi}) = \sum_p \psi_p(\xi_1)\left[\sum_q \hat{u}_{pq}\psi_q(\xi_2)\right]$$

where the brackets denote temporary storage, at a cost of order P^3. Note that, in the above, these smaller one-dimensional products can be represented as matrix–matrix products. For example, taking $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ to denote the basis matrices of the one-dimensional shape functions ϕ^1 and ϕ^2, we have that

$$\boldsymbol{B}\hat{\boldsymbol{u}} = \left[\boldsymbol{B}_1\hat{\boldsymbol{u}}_{[Q_1]}\right]_{[Q_2]}\boldsymbol{B}_2^\top,$$

where $[Q_1]$ denotes the reinterpretation of a vector as a $Q_1 \times Q_2$ matrix with Q_i denoting the order of the quadrature in each direction. Existing implementations therefore focus on the use of optimised matrix–matrix routines such as the BLAS level-3 `dgemm` to perform these operations efficiently, e.g. [26], or through the use of optimised libraries for small-matrix multiplications such as *libxsmm* [27]. In our implementation, we therefore consider an application of the Helmholtz operator in a matrix-free fashion that utilises these sum-factorisation kernels for each of the components of the Helmholtz equation above, rather than storing the elemental matrices $\boldsymbol{H}^e$. More details of this technique applied to the spectral element method on triangles and other higher dimensional shape types can be found in references [12,26,28].

4. Implementation and parallelisation strategy

In the following, we discuss the individual components of the method, as introduced in Section 3. For each one we develop an individual parallel kernel, that ideally performs well on both CPU and GPU architectures. As will be seen, this is achieved apart from the *gather*-type reduction kernels, see Section 4.2.

The first kernel implements the AXPY operations specified by Eqs. (14) to (18). All three programming models, *Kokkos*, *OpenMP*, and *OpenACC* have a suitable syntax and implementation for these *parallel-for-loop*-type of equations. The same applies to the *scatter*-type operation, specified by Eq. (24).

The *gather*-type operation specified by Eq. (27) cannot readily be parallelised without causing a race-condition, as multiple local modes are added up to one global mode and hence need to write to the same memory space. How we still achieve reasonable parallel performance is discussed in Section 4.2.

The calculation of the conjugate gradient step sizes α and β and the residual ϵ, as given by Eqs. (21), (20), and (22) are classical reduction operations. These can be parallelised using the *parallel-reduce*-type syntax. Even though all three programming models support this operation, the *llvm-ykt* compiler branch for *OpenMP4.0* did not support it at the time of conducting this work. Instead, we wrote our own kernel for parallel-reductions. This kernel is based on a two level-implementation as follows. A first level that spawns multiple blocks to utilise the individual streaming multiprocessors of the GPU, so that each reduces a chunk of the overall array down to one scalar. The second level then takes all these intermediate results and performs another reduction on a single SMX to yield the final reduced scalar. We use zero-padding of the arrays on both levels to improve the performance. Although this custom kernel does not achieve the optimised performance of an optimal BLAS library implementation, it does not compromise the overall performance of the method. This will be shown in the results section (see Section 5). All of the components above have costs that roughly scale with the degrees of freedom in the system. However, the computationally most demanding part is the evaluation of the elemental Helmholtz operator given by Eq. (30). A careful evaluation of the parallel algorithm is hence performed in Section 4.1.

4.1. Interleaved data structures and kernels

Each of the elemental Helmholtz operations (see Eq. (30)) can be performed independently and thus in parallel. On a CPU without vectorisation, we would assign one thread per element or set of data, but on the GPU, there are multiple options. These are primarily explained by the constraint of GPUs and the *CUDA* programming model, which forces all 32 *CUDA* threads in one warp to perform the same mathematical or logical operation in any one cycle. Otherwise the warp will diverge so that the different operations are executed sequentially over multiple cycles, decreasing the performance. Three different work distributions can be distinguished here:

(a) Each *CUDA* kernel or GPU processes one set of data.
(b) Each *CUDA* block or streaming multiprocessor (SMX) processes one set of data.
(c) 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 our purpose here. On the Nvidia P100 GPU for example, a matrix size of 128 $\times$ 16384 on a `sgemm` kernel is necessary to get more than 50% of the peak performance of *CuBLAS* `sgemm` operations. Option (b) is mostly suited for 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, small sets of data. For good performance, however, the individual sets should all have the same size and undergo the exact same operations.

In option (b), vectors or matrices need to be sufficiently large so that the overall operation can be efficiently split into multiple warps, in which all threads perform the same elemental mathematical operation. This is because in order to be fully occupied, each SMX of a Nvidia GPU needs to schedule 32 or 64 warps at the same time. Operations over multiple *CUDA* blocks are not synchronised, so that different operations can be executed without performance penalty, as long as a good load balancing between the different *CUDA* blocks is ensured. In terms of data structures, option (b) can operate efficiently on data that is arranged one set after the other. For matrix–matrix multiplication, techniques like *tiling* are further employed to split the overall operation efficiently into elemental operations on which each thread in a warp operates on coalesced data. This option is commonly implemented as *batched* or *strided* BLAS operations. Both these operations are collections or batches of BLAS operations that are performed concurrently, so that the batches will be automatically distributed over the individual SMX. As a special case, strided operations assume the same lengths of data arrays across all batches. The *MAGMA* library [29], the Intel *MKL* library, and the *CuBLAS* library [30] all support batched `gemm` operations, the latter also strided batched operations. Again on the Nvidia P100, 128 batched single precision matrix multiplications with matrix size of 128 $\times$ 128 will result in about 50% peak performance. This is already an improvement from individual `sgemm` kernels, but lower polynomial order elemental matrix sizes are still far smaller than 128 $\times$ 128. E.g. a `gemm` operation occurring in the sum-factorisation kernel of a triangular element with polynomial order 10 has matrix sizes of 11 $\times$ 12.

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. For this approach however, the naive memory layout of having data sets consecutively in memory would violate any coalesced data access. This is because the specific data entries of each set are separated by the whole length of each data set in this case. Yet, all these specific data entries have to be loaded for the warp operations at the same time. For an efficient memory access, it follows that all specific data entries of a group of sets need to be coalesced. The group of sets is chosen as a multiple of the warp-size of 32 threads. As a result, the data sets of one such group become interleaved. The distance or stride between following entries of one data set is equal to the number of data sets that are combined in one group. Interpreting the simple data layout in which one data set is stored after each other, corresponds to a stride of one. For CPUs, this concept of explicit vectorisation over multiple elements using interleaved memory layouts has also been implemented in the *DUNE* library [15], and the *deal.II* library [14,31] to benefit from the wider vector lanes.

For level-1 and level-2 BLAS operations, stride can be specified as a variable, so that no major code adjustment is necessary. Standard level-3 BLAS operations like `dgemm` which are used for our sum-factorisation kernels, however, do not possess a stride variable, so that we decided to implement a custom strided `dgemm` function, that can operate on the interleaved memory layout. Following the function call syntax of other strided batched `gemm`, we introduced two additional variables for each of the three data arrays representing the 3 sets of matrices. The two variables are the stride between the following entries, which is equal to the size of the group, and a variable denoting the number of the group a specific thread operates on. The function call syntax and the operations the algorithms of our custom `dgemm` kernel is given in Listing 1.

Listing 1: Custom interleaved `dgemm` syntax and algorithm

```
int interleavedDgemm(char transa, char transb,
        int M, int N, int K,
        const double alpha,
        const double *A, int lda, int strideA, int instA,
        const double *B, int ldb, int strideB, int instB,
        const double beta,
              double *C, int ldc, int strideC, int instC)
{
    // one of four cases shown
    if (transa == 'T',transb == 'N') {
        for (int m = 0; m < M; ++m) {
            for (int n = 0; n < N; ++n) {
                double c_mnp = 0;
                for (int k = 0; k < K, ++k)
                    c_mnp += A[(m + k*ldA) * strideA + instA]
                    * B[(k + n*ldB) * strideB + instB];
                C[(m + n*ldC) * strideC + instC] =
                alpha * c_mnp +
                beta * C[(m + n*ldC) * strideC + instC];
            }
        }
    }
}
```

Having developed an efficient and working implementation for option (c), and because the performance of option (b) would be severely limited due to the small matrix sizes considered here, we proceed with option (c) throughout this work.

There is another limitation of using a batched function as in option (b), because a new kernel needs to be scheduled for each of these BLAS functions. This is not ideal in conjunction with our framework, in which the code is structured in a way that describes the operations to be executed for each element or data set. Ideally, the data will be loaded into the cache at the beginning of such extended kernels and then operated on by a sequence of BLAS functions and, at the end, stored back into global memory. Using *CuBLAS* functions that start a kernel for each BLAS function will require multiple memory copies between global memory and cache, compromising efficiency. Further, it would require to refactor the code so that the extended kernel is broken up into chunks corresponding to one BLAS function each.

We further found that explicitly using the interleaved memory layout and strided kernel has only been strictly necessary for kernels in which dynamic memory allocation within the kernel is specified. This is the natural way to write with the *Kokkos* syntax and the natural extension from our initial CPU code. The variable length of arrays depending on the element polynomial order can more readily been accounted for using dynamic memory

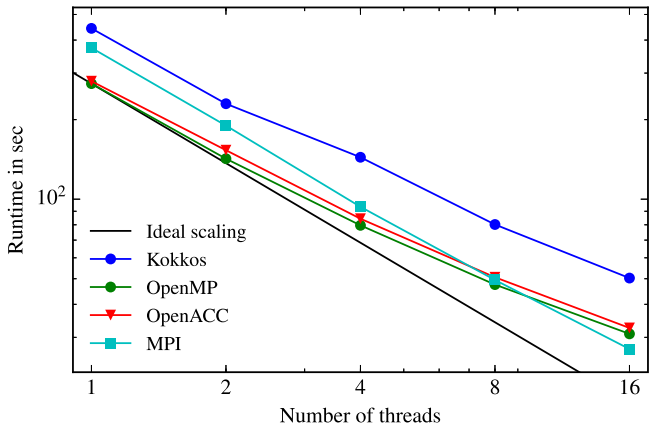

Fig. 1. Strong scaling on 16-core CPU.

Algorithm 1 Initial local-to-global reduction algorithm

procedure LocalToGlobalInitial($\hat{s}_g$, $\hat{s}_g$, v_{sign}, v_{map})
  for all global coefficients j **do in parallel**
    $\hat{s}_g[j] \leftarrow 0$
  end for
  for all elements e **do in serial**
    for all coefficients c **do in parallel**
      $i \leftarrow e * n_{coeffs} + c$
      $\hat{s}_g[v_{map}[i]] \leftarrow \hat{s}_g[v_{map}[i]] + v_{sign}[i] * \hat{s}_l[i]$
    end for
  end for
end procedure

allocation. However, the *OpenMP*-target compiler did not cover this functionality and the *OpenACC* implementation was poorly performing. Here we had to write our kernels with static memory allocation and use templating to account for the varying array lengths of varying polynomial orders. We found that in this case the compiler optimisation can implicitly deal with this situation and create an interleaved memory layout.

4.2. Parallel element colouring and reduction operations

The local-to-global mapping, as given in Eq. (27) is an operation that is not trivial to parallelise. This *gather*-type or reduction operation can be expressed as a double for-loop, as in Algorithm 1. The inner loop over all local coefficients or modes can be implemented in parallel, and the outer loop over all elements needs to be in serial to avoid a race-condition.

Although we found this algorithm to perform very well on the CPU, its performance was limited on the GPU since not even a single SMX could be fully utilised even at reasonable element polynomial orders. To enable more parallelism we introduced an initial elemental colouring of the mesh. This ensures that no neighbouring elements are in the same colourgroup, and hence no write conflict of multiple local modes to global modes can occur. With this colouring in place, it is now safe to parallelise the for-loop over all elements within one colourgroup and introduce a third outermost serial loop over all colourgroups, as in Algorithm 2. Our colouring algorithm uses a re-balancing step so we yield colourgroups that are typically within 25% of the average group size.

The algorithm requires a global synchronisation between the different colourgroups, which can be realised using two options on the GPU. This is either launching a new kernel per colourgroup,

Algorithm 2 Parallel local-to-global reduction algorithm using element colouring

procedure LocalToGlobalParallel($\hat{s}_g$, $\hat{s}_g$, v_{sign}, v_{map}, C)
  for all global coefficients j **do in parallel**
    $\hat{s}_g[j] \leftarrow 0$
  end for
  for all colorsets $c \in C$ **do in serial**
    for all elements $e \in c$ **do in parallel**
      for all coefficients c **do in parallel**
        $i \leftarrow e * n_{coeffs} + c$
        $\hat{s}_g[v_{map}[i]] \leftarrow \hat{s}_g[v_{map}[i]] + v_{sign}[i] * \hat{s}_l[i]$
      end for
    end for
  end for
end procedure

or launching a single kernel for all colourgroups, but only utilise one SMX, since it is not possible to synchronise between SMX in a single kernel. The first option comes with a greater overhead, whereas the second utilises only a fraction of the GPU. For reasonably large mesh sizes, each colourgroup kernel receives enough work so that the first option will be more performant. Another challenge occurs as the inner parallel-loop is relatively skinny and the outer parallel loop is relatively fat, which does not map well to the GPU hardware. However, since the two loops are perfectly nested, a good compiler implementation will be able to collapse the two loops, resulting in better performance.

For all CPU executions we utilise the initial Algorithm 1, for all GPU executions we utilise the mesh-colouring approach and the parallel Algorithm 2.

4.3. Ease of implementation and maintainability of the programming models

All three tested programming models, *Kokkos*, *OpenACC*, and *OpenACC* advertise their capability to incrementally parallelise and port legacy applications written in *C++* to multicore CPUs and GPUs. While this is certainly the case, we noted a few obstacles that indicate this process is not as seamless. In fact, care has to be taken when the initial code-base features many object-oriented programming structures. While they can be parallelised over the cores of a CPU, they do not map to the *CUDA* programming model, that underlies all GPU backends of the considered programming models.

For example, with *CUDA* it is not possible to parallelise over an array of element objects, in which each object holds its data as member-variables and calls its overloaded member-functions, depending on which type of element it is. In this case, the functions and hierarchies of objects need to be de-tangled and implemented in a more straightforward *C*-syntax, in which the corresponding variables and functions for each element are taken care of explicitly. This can result in a complete re-writing of the involved compute-heavy kernels.

Similarly, encapsulated data-structures cannot be mapped directly to the GPU memory. Instead the raw pointers would need to be passed, which in turn requires all functions further down the call-tree to be adapted for the plain data-type. *Kokkos* uses its own encapsulated data-arrays, that map between different memory spaces. It follows that, if in the case of *Nektar++*, encapsulated data-arrays are employed, a large initial implementation effort is necessary for all three programming models. If the legacy application already uses plain *C* arrays, *OpenMP* and *OpenACC* would be favourable.

The limitation of using dynamic memory allocation within the kernels also requires the introduction of templating for variable array lengths. This is a further step away from initial OOP-oriented *C++* code-bases.

When it comes to the actual syntax of specifying parallel loops, the three programming models are comparable, but the *Kokkos* syntax is slightly more complex than that of the *OpenMP* and *OpenACC* compiler directives. *Kokkos* allows for the possibility to explicitly specifying an array to be placed into shared memory, which can be rather cumbersome to optimise when dealing with different element polynomial orders and attempting to align this placement with the shared memory size restriction. *OpenACC* can place arrays into shared memory implicitly, whereas the tested *OpenMP-clang-ykt* implementation does not cover this functionality. In addition, the *OpenMP* and *OpenACC* syntaxes are so similar that a change between them should be straightforward.

5. Performance evaluation

Our test case involves the solution of the Helmholtz equation

$$\nabla^2 u + \lambda u = -(\lambda + 2\pi^2)\cos(\pi x)\cos(\pi y) \tag{35}$$

with $\lambda = 2.5$, on a square domain $\Omega = \{(x, y) : -1 \leq x \leq 1, -1 \leq y \leq 1\}$. On the sides $x = -1$ and $x = 1$ we apply the Dirichlet boundary condition

$$u = \cos(\pi x)\cos(\pi y) \tag{36}$$

and the Neumann boundary condition

$$\frac{\partial u}{\partial n} = \pi \cos(\pi x)\sin(\pi y) \tag{37}$$

is imposed on the sides $y = -1$ and $y = 1$.

We compare the same algorithms as presented in Section 3 that solve this equation implemented with three different programming models: *Kokkos*, *OpenMP* and *OpenACC*. We use the same codebase for both CPU and GPU applications and only at compile time specify the relevant compile flags for each architecture. The only algorithmic difference between architectures is the parallelism in the local-to-global reduction operation discussed in Section 4.2.

The individual meshes for the spatial discretisation consist of triangular straights-sided elements, but different element polynomial orders between 2nd and 11th order. The number of degrees of freedom in each mesh is fixed at 4 million, so that the meshes consist of between 66k to 1975k elements from highest to lowest order respectively. We iterate the conjugate gradient loop until the solution is converged to a tolerance of 10^{-9}.

In terms of compiler toolchains, for our *Kokkos* implementation we use the GNU compiler *gcc-5.4.0* with optimisation flag -O3. For the *OpenACC* implementation we use the PGI compiler version *pgc++-18.4* with optimisation flag -O3. The *OpenMP* implementation is compiled with the experimental *llvm-clang* compiler branch *clang-ykt* and optimisation flag -O3.

At the time of conducting this work the *clang-ykt* branch has not been feature complete. Even though the *OpenMP4.0* specifications support *parallel-reduce* operations, it has not been implemented, yet. As a workaround, we wrote an own kernel for parallel reductions, that is based on a two level-implementation.

The *llvm-ykt* compiler branch also did not support linking to external libraries at the time of conducting this work. Thus, our *OpenMP*-GPU implementation had to be completely decoupled from the *Nektar++* framework.

For a better performance on the CPU we set the environmental variable *OMP_PROC_BIND=true*. For the GPU versions we always use *CUDA8.0*.

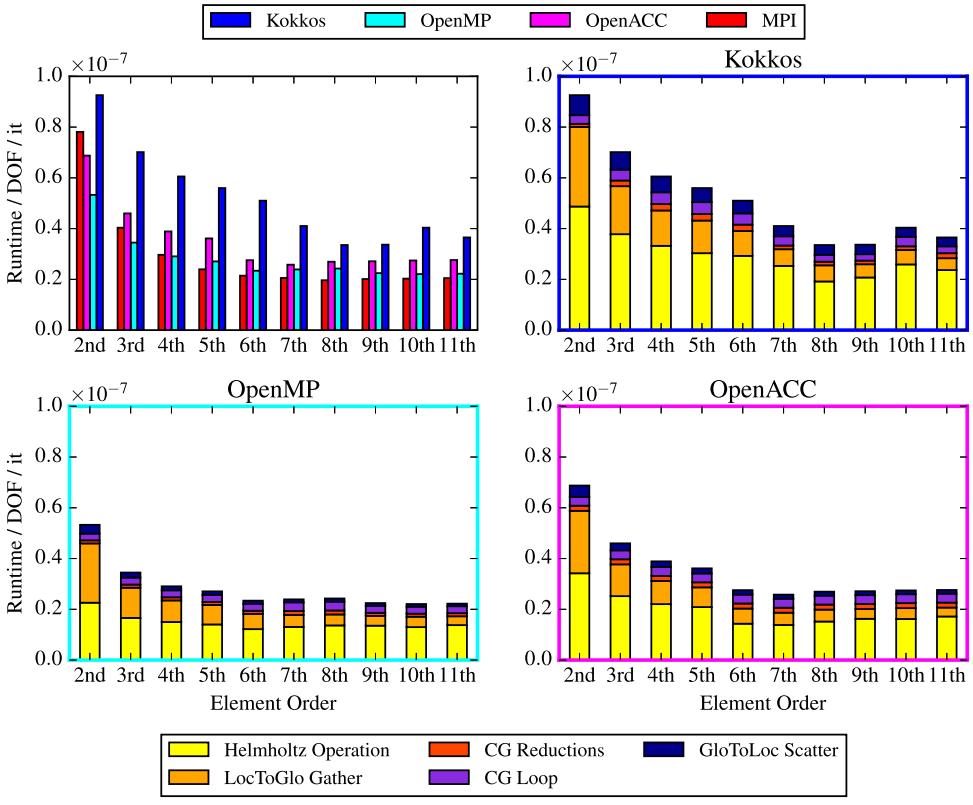

Fig. 2. Runtime comparison of different CPU implementations over a range of elemental orders.

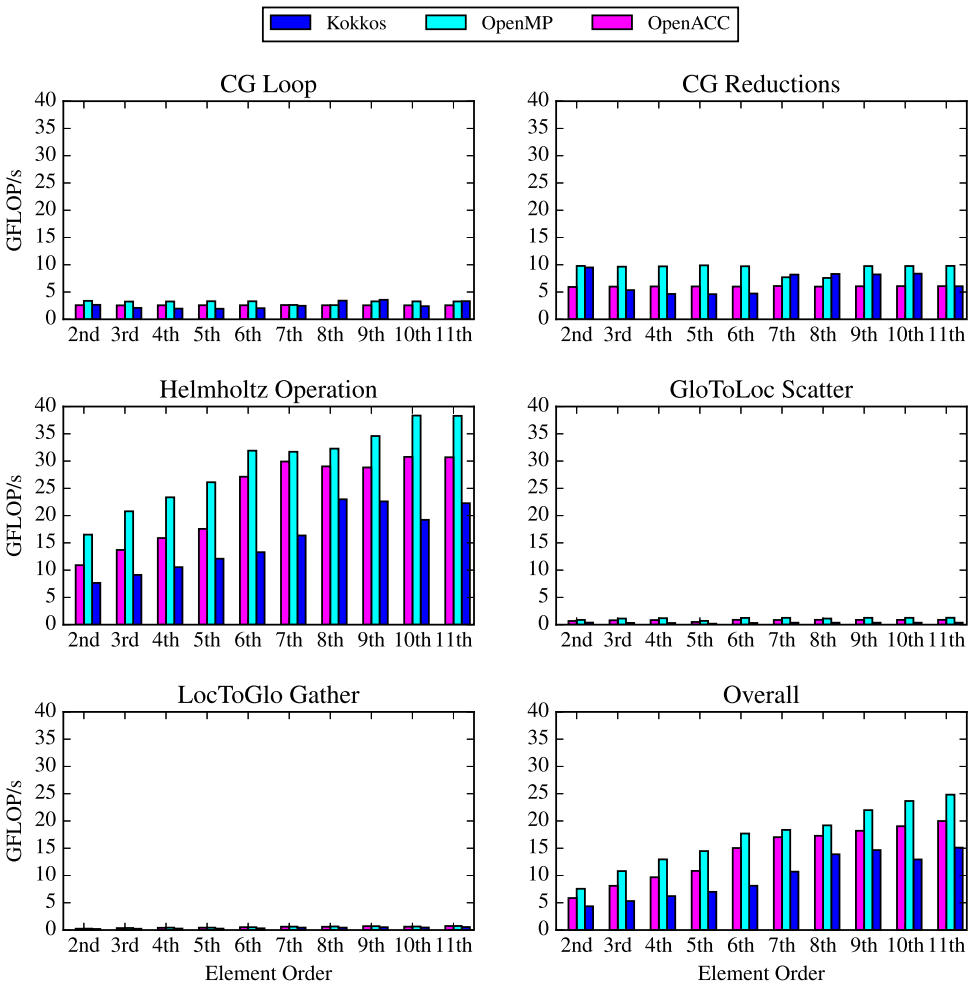

Fig. 3. Achieved FP64 FLOPs for the main kernels on 16-core CPU.

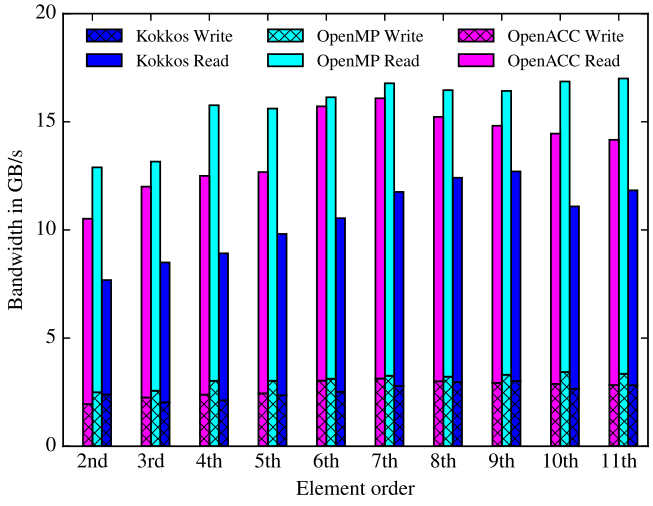

Fig. 4. DRAM memory bandwidth on 16-core CPU.

5.1. Comparison of Kokkos, OpenMP and OpenACC on multi-core architectures

For the multi-core system, we use an Intel E5-2690 v0 CPU with 16 cores and a base frequency of 2.9 GHz and 64 GB RAM with 1600 Hz. The peak FP64 performance considering FMA and AVX is 371.2GFLOPs, the maximum memory bandwidth is 51.2 GB/s, the thermal design point (TDP) is 135 W.

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

5.1.1. Strong scaling exercise

As is customary for multi-core CPU applications, we conduct a strong scaling exercise between 1 and 16 threads on a mesh consisting of 7th-order triangles with 116k elements. This allows us to evaluate the performance of each model and identify potential performance overheads. The results are given in Fig. 1. Here we see that the *OpenMP* and *OpenACC* implementations perform better than the *Kokkos* implementation; however their relative performance varies with polynomial order and will be clearer to evaluate in this context in Section 5.1.2. More important is the observation that all three implementations scale very similarly with increasing numbers of threads and do not reach the almost ideal scaling of our *MPI* implementation. This is probably due to the reduction operations that do not allow us to fully utilise all threads continuously.

5.1.2. Efficiency of different polynomial orders

For this test series we use meshes with elements of varying polynomial orders where the degree of freedom count is fixed across the polynomial orders. We utilise all 16 threads on our

multi-core CPU. The runtimes for one iteration of the conjugate gradient solve of all four applications scaled by the exact number of DOFs are given in the upper left subplot of Fig. 2. The runtimes improve with increasing element order up to about 6th order and then plateau. The low element orders have a lower FLOP count per DOF, but also a lower FLOP-per-byte ratio or arithmetic intensity. As our applications are memory-bound, they do not benefit from a lower FLOP count *per se*. Overall, the *MPI* implementation is the fastest across the range of polynomial orders, but closely matched by the *OpenMP* and the *OpenACC* versions. The *Kokkos* version gives the worst performance generally, with around twice the runtime.

The remaining three subplots of Fig. 2 present the runtime-splits between the five different kernels of the application, as discussed in Section 4. The elemental Helmholtz operation consumes about 50% of the overall runtimes. The local-to-global reduction also shows bad performance for the lower elemental orders, as the involved arrays are too short in this case. Overall, the splitting of runtime between the *Kokkos*, *OpenMP*, and *OpenACC* is comparable.

5.1.3. Detailed performance evaluation

In order to evaluate how well the underlying CPU hardware is utilised we consider two metrics, the achieved double precision (FP64) FLOPs and the DRAM memory bandwidth. The FLOPs that the different kernels have achieved are given in Fig. 3. The gather and scatter operations, as well as the conjugate gradient reductions and for-loops perform very poorly, because they are strongly memory bound. Only very few operations are performed on each data element, which is read and written back to DRAM memory. Only for the parallel reduction kernel data can be re-used from higher cache levels, leading to faster loading to registers. The gather and scatter operations perform especially poorly since the data is usually not read and written in continuous cache blocks. The Helmholtz operation is the critical kernel and achieves much better performance of up to 38GFLOP/s for the *OpenMP* version, 31GFLOP/s for the *OpenACC* version, but only 23GFLOP/s for the *Kokkos* version. For the *OpenMP* version, this translates to utilising more than 10% of peak FLOPs. The trend for higher FLOPs with higher element orders is very clear, as it requires more compute operations per loaded byte, i.e. a higher arithmetic intensity. The overall FLOPs performance is dominated by the Helmholtz operation kernel, but pulled down by about one third by the less performant kernels (see Fig. 3).

The DRAM memory bandwidth is given in Fig. 4. Generally it can be seen from the data that especially the *OpenMP* and to a slightly lower degree also the *OpenACC* implementation utilise the memory more efficiently than the *Kokkos* implementation. As a trend, higher polynomial orders are processed more efficiently, as can be expected due to the larger arrays. The *OpenMP* implementation achieves a combined *read* and *write* memory bandwidth of 17 GB/s, which amounts to about 33% of the theoretical bandwidth.

5.2. Comparison of Kokkos, OpenMP and OpenACC on GPU architectures

As the GPU system we use a Nvidia Tesla P100 with 60 streaming multiprocessors (SMX), a peak FP64 performance of 5304GFLOPS, a maximum bandwidth of 549G B/s, and a TDP of 300 W.

5.2.1. Efficiency of different polynomial orders

We employ the same test methodology as in the multi-core CPU case in Section 5.1.2. The runtimes for one iteration scaled by the DOFs are given in Fig. 5. Across the whole range of

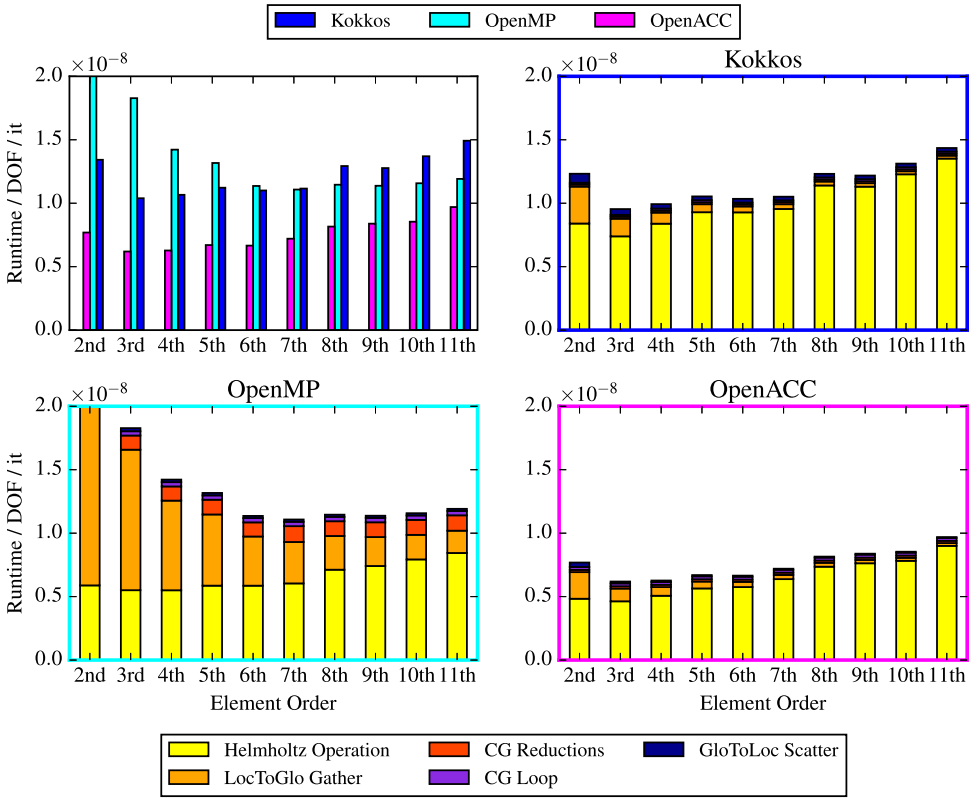

Fig. 5. Runtime comparison of different GPU implementations over a range of elemental orders on P100.

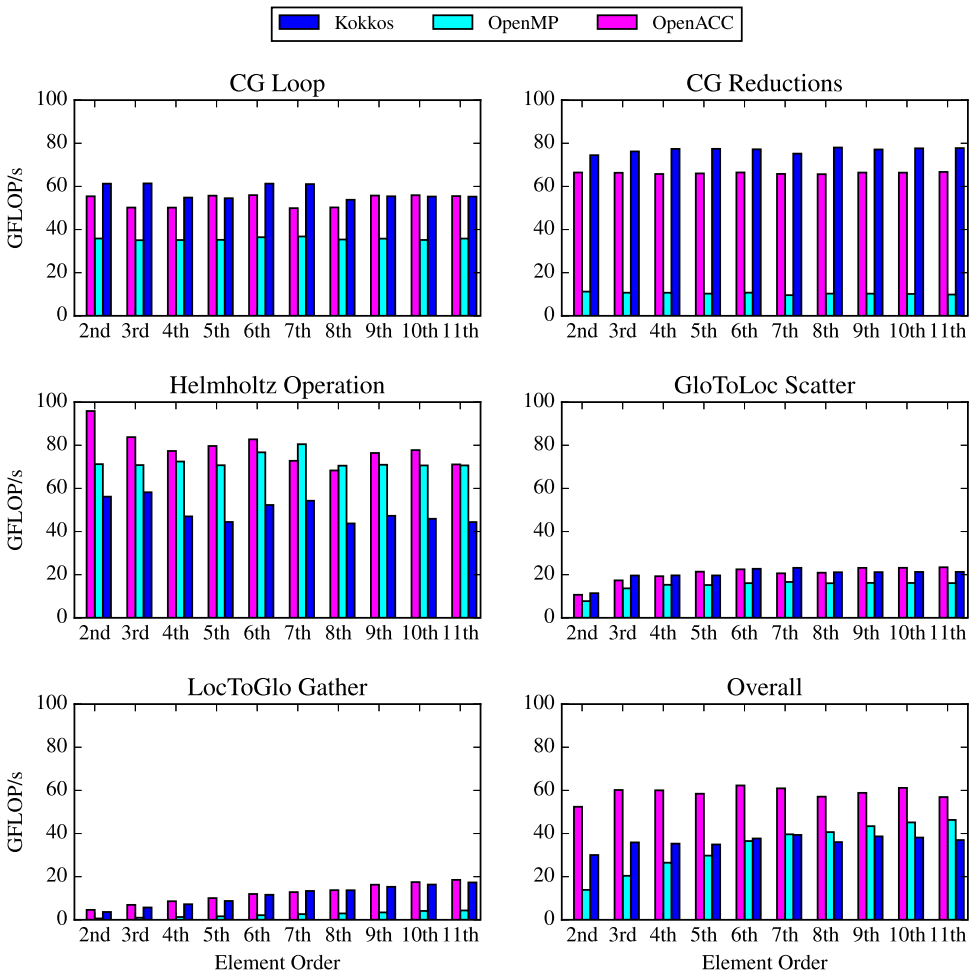

Fig. 6. Achieved FP64 flops for the main kernels.

elemental orders, the *OpenACC* implementation is the most performant, while the variation between elemental orders is low. The runtime splits show a much larger proportion of time spent in the elemental Helmholtz operation, that varies between about 65% for 2nd order and 90% for 11th order. The performance for the *OpenMP* implementation is only about 25% worse for high polynomial orders, but suffers greatly for low polynomial orders. It becomes clear that primarily the bad performance of the local-to-global reduction is responsible and only secondarily the lower performance of our custom-written CG-reduction kernel. The embarrassingly parallel elemental Helmholtz kernel is *on par* with the *OpenACC* version. The performance of the *Kokkos* implementation is about 40% worse than the *OpenACC* implementation across the range of elemental orders. Examining the runtime splits of the *Kokkos* version, no individual kernel can be pointed out that would be responsible, indicating a general performance issue with this version.

5.2.2. Detailed performance evaluation

Runtimes alone do not tell us how well the implementation is utilising the underlying hardware. To this end we first provide the achieved double precision (FP64) FLOP/s in Fig. 6. Comparing the three different programming models, it can be seen again that *Kokkos* does not achieve the same performance as *OpenACC* and *OpenMP* for the Helmholtz operation kernel and hence for the overall application. Furthermore, the poor performance of the *OpenMP* reduction/gather kernels becomes obvious. In general, the achieved FLOP/s translate to utilising actually less than 2% of the theoretical FP64 performance, which indicates that the memory bandwidth is actually the performance bottleneck.

To prove this, we provide the DRAM memory bandwidth in Fig. 7. Here it can be seen that our implementations operate very close to the theoretical maximum across al elemental orders for the CG-Loop, the CG-reductions and the global-to-local scatter kernels. The most important kernel, the Helmholtz operation, is not fully utilising the DRAM bandwidth indicating that either the bandwidth of a higher level memory or cache level is limiting the overall performance or the memory latency results in a stall. The three programming models perform very similar, just the *OpenMP* application is falling off slightly.

Comparing the utilisation or bandwidth of the global (Fig. 8) against the local and shared memory (Fig. 9) for the Helmholtz operation kernel, we can see that the *Kokkos* kernel is utilising more of the slower global memory (residing on DRAM, L2 and L1 cache) than the faster local and shared memory that resides only on the L1 cache. This could explain the performance deficit of the *Kokkos* implementation. It has to be acknowledged that a better performance could be achieved if tuning the Helmholtz operation kernel for each elemental order by explicitly specifying the utilisation of L1 cache for certain arrays. This tuning approach

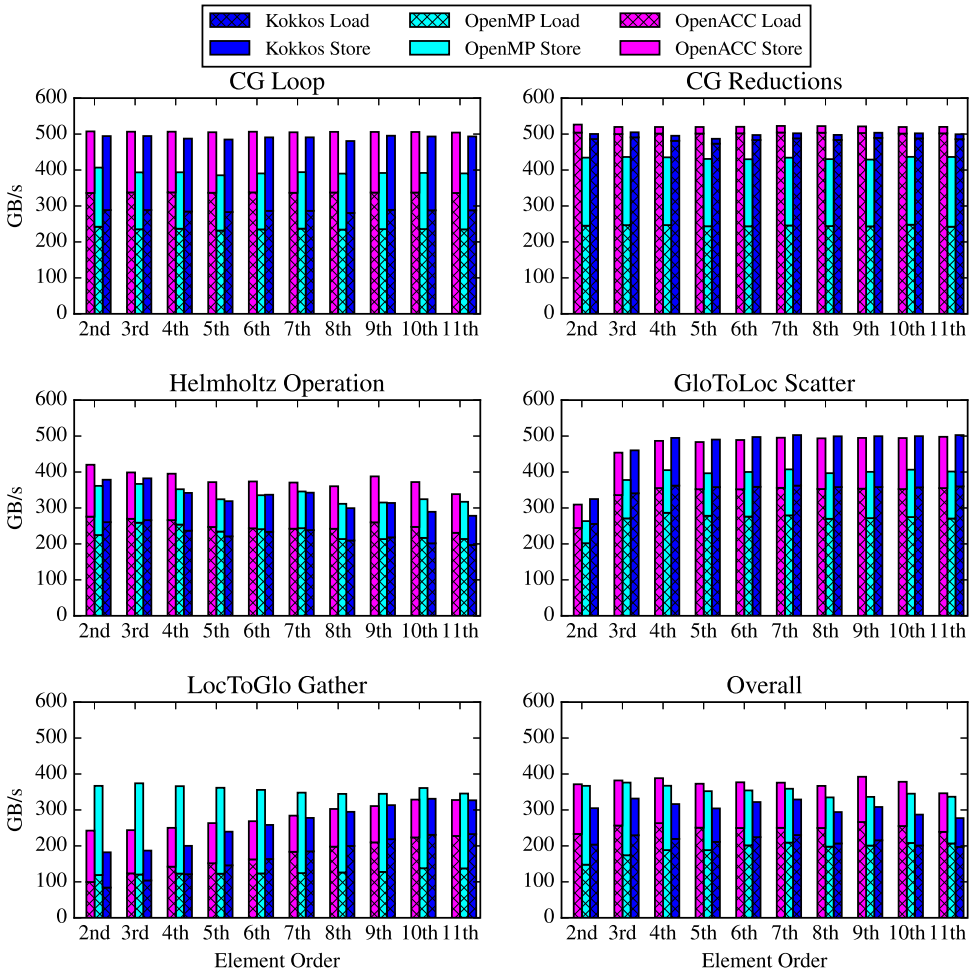

Fig. 7. DRAM memory bandwidth for the main kernels.

however defeats the idea of a performance portable programming model.

It can be concluded that our applications are heavily memory bound. It shows that just porting a legacy CPU-optimised algorithm cannot fully utilise modern GPUs. More effort to reduce the memory dependency of the algorithms and hence increase the arithmetic intensity are necessary.

5.3. Cost comparison between different architectures

We use system acquisition costs to identify the most efficient architectures in terms of run-time and operational costs.

The acquisition costs are the sum of the CPU or GPU by itself plus a hosting system. We estimate the current price of the Nvidia Tesla P100 GPU as $6000, the Intel E5-2690 v0 CPU with 2 sockets or 16 cores as $800, and the hosting system as $2000. We use a time frame of 3 years for complete linear depreciation, to obtain hourly prices for both the complete CPU and the GPU system. The costs of code execution are then the product of run-time with hourly system costs. This follows the approach presented in reference [32]. The result is given in Fig. 10. It illustrates very well the benefit of employing GPU systems for the presented type of algorithms, as both the runtime and costs are lower on the P100, the costs by 20%, the runtimes by 72%, considering the most performant cases. Considering only the two *OpenMP* implementations, though, the cost of utilising the P100 GPU is higher than for the CPU, although it is faster. In terms of both cost and time performance metrics, the *OpenACC* version for the GPU is the most beneficial. This result would only become more pronounced, when optimising the algorithms further and replacing pre-computations and loading from memory with on-the-fly calculations to reduce the required memory bandwidth that so far slows down the GPU implementations to a higher degree than the CPU implementations.

6. Conclusions

There are two main conclusions to be drawn from this study.

Firstly, when comparing three different performance-portable programming models, namely *Kokkos*, *OpenACC* and *OpenMP* using open-access compilers in a high-order FEM setting, we conclude that the choice of programming model is still a complex question.

All three programming models require very similar implementational efforts. There are syntactical differences, which are minor between *OpenMP* and *OpenACC* (and would enable a seamless switch between the two) and slightly deeper compared with *Kokkos*, mostly due to the requirement of using *Kokkos*-specific arrays. *Kokkos* further allows addressing many aspects of performance tuning which are connected with native *CUDA* applications, allowing more individual trade-offs between portability and performance. The main current drawback of the *OpenMP* model is the lack of robust open-access compiler implementations. The best compiler currently available, which is the *llvm-ykt* branch, still lacks a parallel reduction functionality, has no linking support for external libraries, and showed bad performance of the

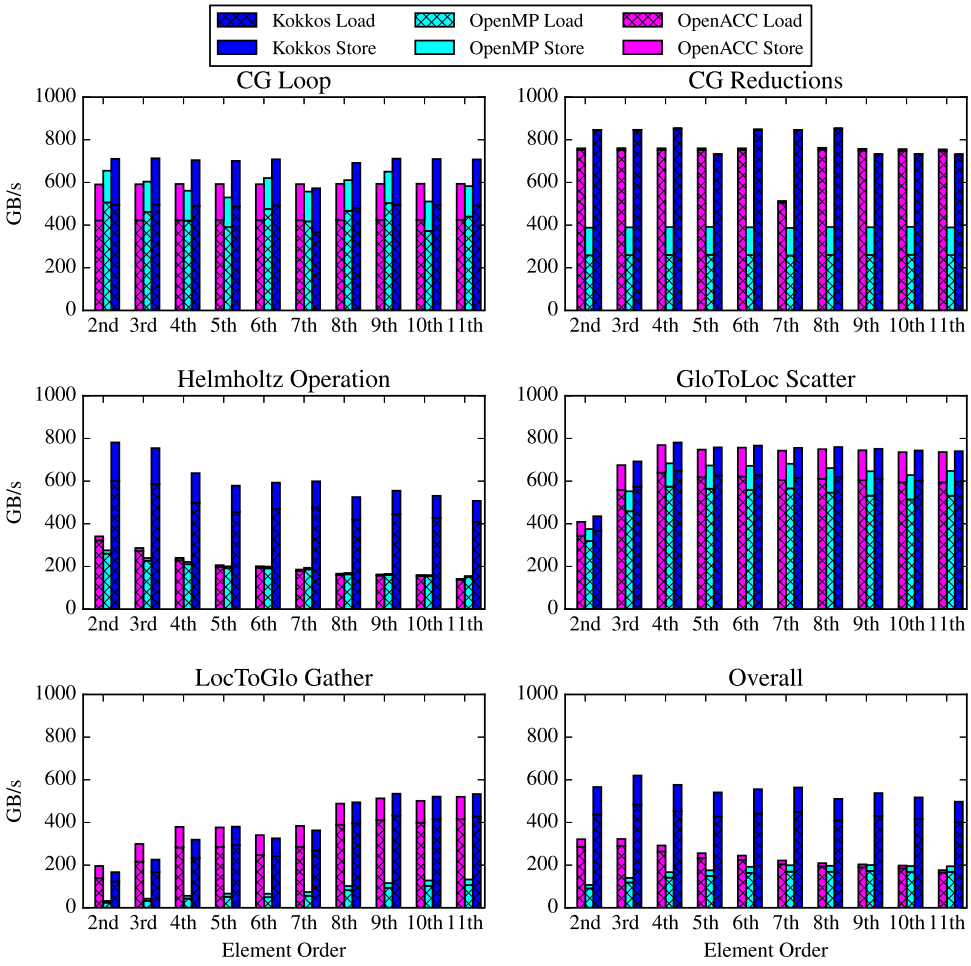

Fig. 8. Global memory bandwidth for the main kernels.

gather-type operation, probably due to the implementation of the *collapsing*-clause and the implicit compiler choice for block-sizes. The *OpenACC* model has shown the best overall performance results, yielding the fastest runtimes on Nvidia GPUs, and only slightly slower runtimes on Intel CPUs than *OpenMP*.

Secondly, this study has helped us to identify the steps to be undertaken in order to port a legacy CPU code to modern multi-core CPUs and many-core GPUs and achieve good performance. The mapping of the algorithmic parallelisation to the hardware parallelisation, together with the right data layout, plays a fundamental role in achieving reasonable performance on modern hardware. Here we have employed a very fine grained parallelism in which we process a single mesh element on each thread. This requires the processing of elements to be vectorised either on the AVX vector lanes or over the *CUDA* threads of a warp. This vectorisation in turn necessitates the use of interleaved data structures and custom BLAS functions to operate on them. Reduction operations can become a major bottleneck in massively parallel applications. We employ mesh colouring to counteract this aspect and regain more parallelism.

The performance of our three applications is very reasonable on multi-core CPUs, but lacks the ability to fully leverage the arithmetic potential of GPUs. To do so, it is not enough to translate the traditional algorithms, that often just minimise FLOP count, as we did in this study. Instead of pre-computing data and loading it when required, more data should be recomputed on the fly, as it can often be faster than loading a large array from DRAM memory, when latencies are of the order of 10^4 cycles. For example, the framework *OpenSBLI* [20] offers the option to variably execute either or a mix of these two approaches.

These observations show that performance portability might not be realisable in practice just by employing a performance portable programming model. The algorithmic choices between current CPUs and GPUs that have to be made now are fundamentally different. To deal with this increasing code complexity, auto-tuning frameworks and just-in-time compilation, that can make algorithmic choices based on the underlying architecture at runtime, should be considered.

Declaration of competing interest

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

Acknowledgements

JE acknowledges the constructive discussions with Niki Loppi that led to the development of the strided kernels. JE also acknowledges the support through the President's Scholarship of Imperial College London, United Kingdom. MV acknowledges support from the EU Horizon 2020 project ExaFLOW (Grant No. 671571). DM and JP acknowledge support from the EPSRC, United Kingdom under Grant EP/R029423/1. The Quadro P5000 GPU used for this work has been kindly donated by the NVIDIA corporation, United States.

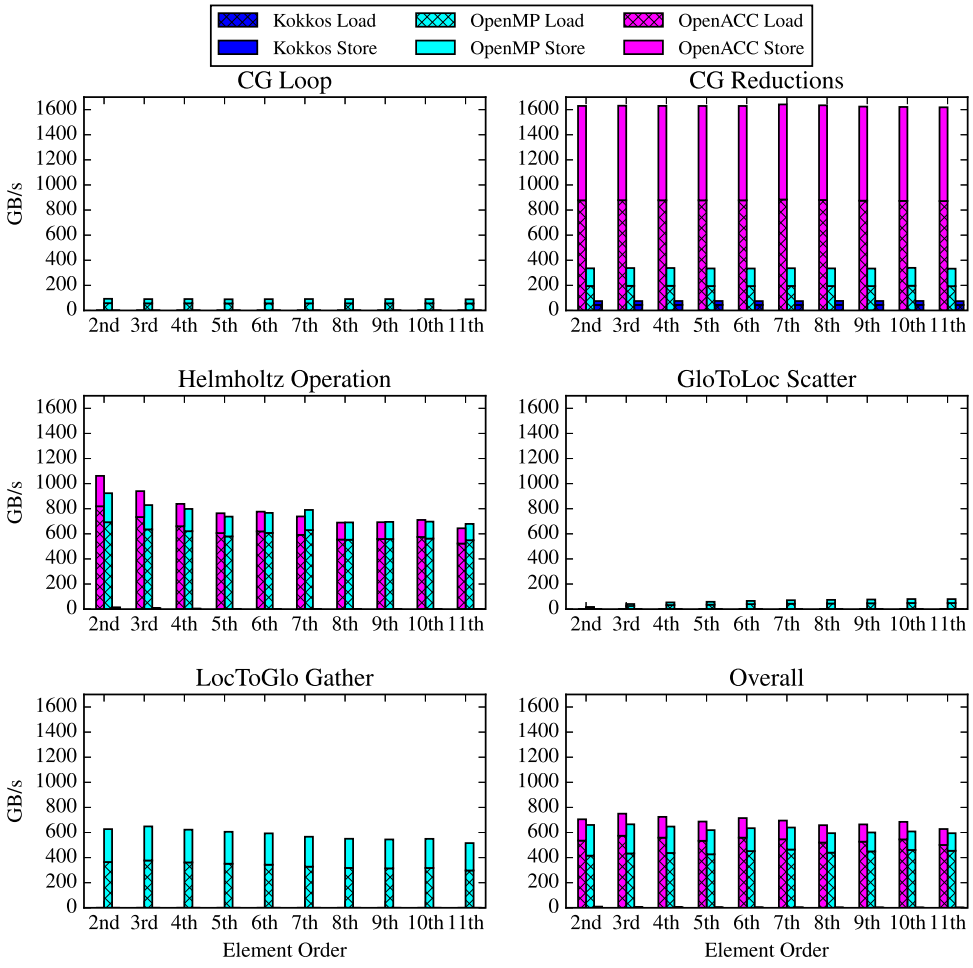

Fig. 9. Local and shared memory bandwidth for the main kernels.

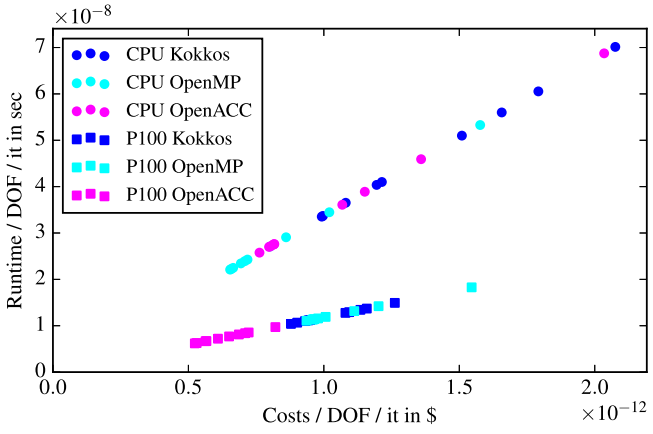

Fig. 10. Cost vs time comparison of the CPU and the GPU system.

References

[1] J.-E.W. Lombard, D. Moxey, S.J. Sherwin, J.F.A. Hoessler, S. Dhandapani, M.J. Taylor, AIAA J. 54 (2) (2016) 506–518, http://dx.doi.org/10.2514/1.J054181.

[2] H. Carter Edwards, C.R. Trott, D. Sunderland, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216, http://dx.doi.org/10.1016/j.jpdc.2014.07.003.

[3] OpenMP 4.5 specifications, 2015, URL http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[4] OpenACC programming and best practices guide, 2015, URL http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf.

[5] J.J. Dongarra, P. Luszczek, A. Petite, Concurrency Comput. Pract. Exp. 15 (9) (2003) 803–820, http://dx.doi.org/10.1002/cpe.728.

[6] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist, R.W. Numrich, Sandia Report SAND2009-5574, Tech.Rep. September, Sandia National Labs, 2009, URL https://mantevo.github.io/pdfs/MantevoOverview.pdf.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, K. Skadron, Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, IEEE, 2009, pp. 44–54, http://dx.doi.org/10.1109/IISWC.2009.5306797.

[8] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 63–74, http://dx.doi.org/10.1145/1735688.1735702, arXiv:1107.1714v2.

[9] J.A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G.D. Liu, W.-m.W. Hwu, IMPACT Technical Report IMPACT-12-01, Tech. Rep., University of Illinois at Urbana-Champaign, 2012, URL http://impact.crhc.illinois.edu/parboil/parboil.aspx.

[10] C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, S.J. Sherwin, Comput. Phys. Comm. 192 (2015) 205–219, http://dx.doi.org/10.1016/j.cpc.2015.02.008.

[11] G.E. Karniadakis, M. Israeli, S.A. Orszag, J. Comput. Phys. 97 (2) (1991) 414–443, http://dx.doi.org/10.1016/0021-9991(91)90007-8.

[12] G. Karniadakis, S. Sherwin, Spectral/hp Element Methods for Computational Fluid Dynamics, second ed., Oxford University Press, 2005, http://dx.doi.org/10.1093/acprof:oso/9780198528692.001.0001.

[13] F. Witherden, A. Farrington, P. Vincent, Comput. Phys. Comm. 185 (11) (2014) 3028–3040, http://dx.doi.org/10.1016/j.cpc.2014.07.011.

[14] M. Kronbichler, K. Kormann, ACM Trans. Math. Software 45 (3) (2019) 1–40, http://dx.doi.org/10.1145/3325864.
[15] P. Bastian, C. Engwer, J. Fahlke, M. Geveler, D. Göddeke, O. Iliev, O. Ippisch, R. Milk, J. Mohring, S. Müthing, M. Ohlberger, D. Ribbrock, S. Turek, in: H.-J. Bungartz, P. Neumann, W.E. Nagel (Eds.), Software for Exascale Computing - SPPEXA 2013-2015, Springer International Publishing, Cham, 2016, pp. 3–23, http://dx.doi.org/10.1007/978-3-319-40528-5_1.
[16] Intel Corporation, Using AVX without writing AVX code, 2012, URL https://software.intel.com/en-us/articles/using-avx-without-writing-avx-code.
[17] Nvidia, CUDA8.0 release note, 2016, URL http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Release_Notes.pdf.
[18] Khronos Group, OpenCL overview, 2019, URL https://www.khronos.org/opencl/.
[19] F. Rathgeber, D.A. Ham, L. Mitchell, M. Lange, F. Luporini, A.T.T. McRae, G.-T. Bercea, G.R. Markall, P.H.J. Kelly, ACM Trans. Math. Software 43 (3) (2015) 24:1 – 24:27, http://dx.doi.org/10.1145/2998441.
[20] C.T. Jacobs, S.P. Jammy, N.D. Sandham, J. Comput. Sci. 18 (2017) 12–23, http://dx.doi.org/10.1016/j.jocs.2016.11.001.
[21] The Open Group, POSIX threads, 1997, URL http://pubs.opengroup.org/onlinepubs/007908799/xsh/threads.html.
[22] R. Hornung, H. Jones, J. Keasler, R. Neely, A. Kunen, O. Pearce, RAJA Overview, Tech. Rep. LLNL-TR-677453, Lawrence Livermore National Laboratory, 2015, URL https://github.com/LLNL/RAJA.
[23] D.S. Medina, A. St-Cyr, T. Warburton, OCCA: A unified approach to multi-threading languages, 2014, pp. 1–25, ArXiv, URL http://arxiv.org/abs/1403.0968.
[24] Khronos Group, SYCL overview, 2016, URL https://www.khronos.org/sycl/.
[25] S.A. Orszag, J. Comput. Phys. 37 (1) (1980) 70–92, http://dx.doi.org/10.1016/0021-9991(80)90005-4.
[26] D. Moxey, C.D. Cantwell, R.M. Kirby, S.J. Sherwin, Comput. Methods Appl. Mech. Engrg. 310 (2016) 628–645, http://dx.doi.org/10.1016/j.cma.2016.07.001.
[27] A. Heinecke, G. Henry, M. Hutchinson, H. Pabst, International Conference for High Performance Computing, Networking, Storage and Analysis, SC, 2017, pp. 981–991, http://dx.doi.org/10.1109/SC.2016.83.
[28] C.D. Cantwell, S.J. Sherwin, R.M. Kirby, P.H.J. Kelly, Comput. & Fluids 43 (1) (2011) 23–28, http://dx.doi.org/10.1016/j.compfluid.2010.08.012.
[29] T. Dong, A. Haidar, P. Luszczek, S. Tomov, A. Abdelfattah, J. Dongarra, MAGMA Batched: A Batched BLAS Approach for Small Matrix Factorizations and Applications on GPUs, ICL Tech Report 08/2016, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, 2016, p. 37996.
[30] Y. Shi, U.N. Niranjan, A. Anandkumar, C. Cecka, Proceedings - 23rd IEEE International Conference on High Performance Computing, HiPC 2016, 2017, pp. 193–202, http://dx.doi.org/10.1109/HiPC.2016.031.
[31] G. Alzetta, D. Arndt, W. Bangerth, V. Boddu, B. Brands, D. Davydov, R. Gassmoeller, T. Heister, L. Heltai, K. Kormann, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, J. Numer. Math. (2019) URL https://doi.org/10.1515/jnma-2018-0054.
[32] J. Eichstädt, M. Green, M. Turner, J. Peiró, D. Moxey, Comput. Phys. Comm. 229 (2018) 36–53, http://dx.doi.org/10.1016/j.cpc.2018.03.025.