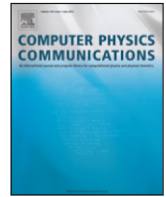
Computer Physics Communications 229 (2018) 36–53

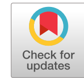
Contents lists available at ScienceDirect

Computer Physics Communications

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



Accelerating high-order mesh optimisation with an architecture-independent programming model



Jan Eichstädt [a,*], Mashy Green [a], Michael Turner [a], Joaquim Peiró [a], David Moxey [b]

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

A R T I C L E   I N F O

Article history:
Received 7 November 2017
Received in revised form 24 January 2018
Accepted 27 March 2018
Available online 5 April 2018

Keywords:
High-order mesh optimisation
Architecture-independent programming model
Kokkos
Portability
Parallel hardware
Variational framework

A B S T R A C T

Heterogeneous manycore performance-portable programming models and libraries, such as *Kokkos*, have been developed to facilitate portability and maintainability of high-performance computing codes and enhance their resilience to architectural changes. Here we investigate the suitability of the *Kokkos* programming model for optimising the performance of the high-order mesh generator *NekMesh*, which has been developed to efficiently generate meshes containing millions of elements for industrial problem involving complex geometries. We describe the variational approach for *a posteriori* high-order mesh optimisation employed within *NekMesh* and its parallel implementation. We discuss its implementation for modern manycore massively parallel shared-memory CPU and GPU platforms using *Kokkos* and demonstrate that we achieve increased performance on multicore CPUs and accelerators compared with a native *Pthreads* implementation. Further, we show that we achieve additional speedup and cost reduction by running on GPUs without any hardware-specific code optimisation.

© 2018 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

High-order spectral element methods are gaining support within the computational fluid dynamics (CFD) community. They offer improved solution accuracy for a given computational cost due to their exponential convergence and show very low dispersion and diffusion errors, giving these methods an edge over traditional low-order methods [1]. Although the use of high-order methods is becoming increasingly common in academic studies, a significant bottleneck in their more widespread adoption in industrial applications is the availability of robust high-order meshing capabilities for complex three-dimensional geometries, and their efficiency on current and future high-performance computing (HPC) systems [2].

The standard approach to generate a high-order mesh is to deform an initial coarse linear mesh, which can be obtained using one of the many available linear meshing tools, to conform with the curved boundary specified by the CAD geometry. This *a posteriori* process will likely yield very distorted or inverted elements close to the boundary, as the introduction of curvature into the element frequently leads to self-intersection. We therefore require a second step, that corrects invalid elements through a boundary-induced mesh deformation, so that curvature is introduced into elements connected and in close proximity to the curved surface. Several different techniques for this step have been proposed in the literature, which can be broadly classified into two categories: *elastic analogies* where the mesh is treated as a solid body and the curvature acts a force on the body, e.g. [3–5], and *energy minimisation techniques* in which a functional representing mesh distortion is minimised to optimise mesh quality and correct invalid elements, e.g. [6,7]. Alternatively, high order meshes can be adapted by combining mesh curving and mesh topology changes, as presented for example in Ref. [8].

These techniques in general are computationally expensive, since they require either the solution of a partial differential equation or a non-linear optimisation to obtain the corrected mesh. In an industrial setting, where geometries are typically extremely complex and meshes can consist of millions or billions of elements, this process can be computationally prohibitive. Modern design lifecycles also demand the generation and optimisation of meshes in the order of minutes or hours, typically on only a single high-performance workstation.

Although modern high-performance computing systems are providing more computational power than ever seen before in terms of the number of floating-point operations per second (FLOPS) that can be performed, this increased performance comes in the form of

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

https://doi.org/10.1016/j.cpc.2018.03.025
0010-4655/© 2018 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

heterogeneous multi- and manycore architectures. There are several challenges involved in being able to effectively use the FLOPS such platforms provide. Developers must find appropriately scalable algorithms and data structures that align to these architectures. Different architectures also typically require the use of different programming techniques to achieve maximum performance. This means that maintaining sustainable and useable software, as well as being able to support possible future architectures, poses a formidable problem. These difficulties mean that, to date, this power has yet to be realised in the area of high-order mesh optimisation. We do however note that in the area of linear mesh generation, first implementations for increased efficiency that run on general purpose GPUs have been started to be presented, as seen for example in Ref. [9].

The purpose of this work is to investigate how the variational framework introduced by Turner, Peiró and Moxey in Refs. [10] and [11], which encompasses many of the *a posteriori* techniques introduced above, can effectively make use of modern *heterogeneous* architectures. To achieve this in a sustainable manner that does not require the maintenance of a codebase for each architecture, we make use of the performance portable programming model *Kokkos* [12], which is one of the models that have been introduced to overcome the inherent difficulties of architecting programs for several architectures. The use of this model allows us to use a single, shared codebase to support a range of vastly different modern hardware, including 'traditional' multicore CPUs (e.g. Intel Xeon CPUs), manycore coprocessors (e.g. Intel Xeon Phi processors) and general purpose GPUs (e.g. Nvidia GPUs). Some scientific applications have already been implemented in *Kokkos*, e.g. [9,13,14], all developed within Sandia National Laboratories. The authors ported serial or *MPI* legacy code to *Kokkos* and conclude that it is a suitable and performance portable paradigm.

The paper aims to answer two important questions:

- How well are the meshing algorithms laid out in the variational framework suited for manycore architectures, when compared with multicore architectures?
- How well does the *Kokkos* programming model perform in the context of this algorithm in terms of attaining maximum performance of the hardware?

In the context of the first question, to achieve optimal performance on manycore hardware, algorithms need to possess a high degree of task and data parallelism to take advantage of the ever rising processor core counts. We will show in the following sections that the variational mesh optimiser offers a high level of parallelism and can exploit these hardware trends. Together with its robustness and flexibility, it combines both critical factors for the success of high-order meshing capabilities on modern hardware. We then address the second question through a series of performance tests on a variety of modern hardware, which examines factors such as strong and weak scalability, relative speedups between architectures, assessing hardware utilisation, and modelling the operating costs of different hardware as a function of degree of freedom.

This paper makes a number of contributions. The most significant in terms of our application area is the demonstration of a methodology that is capable of effectively using manycore CPU and GPU hardware to substantially reduce high-order mesh optimisation runtimes. To the best of the authors' knowledge, this is the first time that this has been presented in the literature, particularly in the context of GPU-accelerated high-order mesh optimisation. Additionally, through the discussion of the techniques used to accelerate our code, we also provide valuable insight into the steps required to port existing code onto new architectures. Although in this case we are using the *Kokkos* model, the experiences described here are relatively broad and extend past the specific choice of model being used. This work is therefore valuable in the context of existing HPC users who intend to port their code to new architectures and who aim to promote sustainability of their software.

The rest of this paper is organised as follows. Section 2 briefly outlines the formulation of the variational framework introduced in Ref. [11] and describes how the method can be effectively parallelised. Section 3 introduces *Kokkos* and the practical steps taken to accelerate the optimisation method. In Section 4, we then perform a thorough performance analysis of the optimised code to demonstrate the effectiveness of the strategy on a variety of architectures, including a standard multicore CPU system, a manycore CPU and various GPU platforms. We conclude the paper in Section 5 with an overview and discussion of the results and the wider impact of the work.

2. Variational framework for high-order mesh optimisation

We start with a brief overview of the variational framework outlined in Ref. [11]. The motivation for the framework is to reform PDEs arising from *a posteriori* techniques based on solid body analogies into an energy minimisation problem through the calculus of variations. In this manner, we are able to utilise a range of optimisation approaches from the literature under a single framework by minimising a functional $\mathcal{E}$. The techniques outlined below are implemented within the meshing tool *NekMesh* [15], which is part of the spectral/*hp* element framework *Nektar++* [16].

A general overview of the process is as follows. As a first step of the mesh optimisation, we obtain a linear mesh of a given CAD geometry using either the internal *NekMesh* mesh generation tools, or employ external programs such as *Gmsh* [17]. We then apply the variational framework inside *NekMesh* in order to transform the linear mesh into a high-order mesh, correct invalid elements, and generally improve the mesh quality. The sections below outline the mathematical background of the framework and describe the parallelisation strategy, which has been used in Ref. [11] to demonstrate good performance and robustness in the processing of meshes of the order of 10 million degrees of freedom.

The definition of the energy functional to be minimised relies on a mesh deformation tensor $\nabla\boldsymbol{\phi}$, where $\boldsymbol{\phi}$ is a mapping between an 'ideal' straight-sided mesh Ω_I and the boundary-conforming curvilinear mesh Ω, as shown in Fig. 1. We then write $\boldsymbol{\phi} : \Omega_I \to \Omega$, so that the Cartesian coordinates $\boldsymbol{x} \in \Omega$ are related to the ideal coordinates $\boldsymbol{y}$ through the relationship $\boldsymbol{x} = \boldsymbol{\phi}(\boldsymbol{y})$. The energy functional can then be defined as the integral

$$\mathcal{E}(\nabla\boldsymbol{\phi}(\boldsymbol{y})) = \int_{\Omega_I} W(\nabla\boldsymbol{\phi}(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}, \tag{1}$$

where the strain energy function W will depend on the chosen material constitutive model. Currently supported choices include models of linear elasticity [3], isotropic hyper-elasticity [5], the Winslow equations [4], and a distortion-measure energy [7]. Here we will use the hyper-elastic strain energy function for a compressible neo-Hookean material [18], where W is given by

$$W = \frac{\mu}{2}(I_1^{\mathbf{C}} - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2, \tag{2}$$

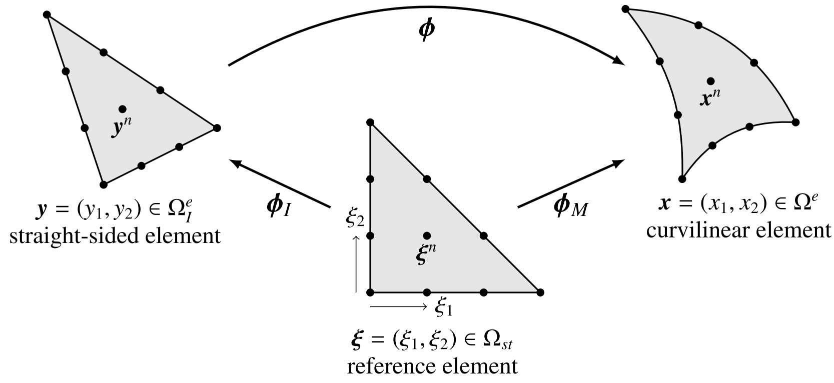


Fig. 1. The mapping relation between the reference, straight-sided, and curvilinear elements.

and $J = \det \nabla\boldsymbol{\phi}$ is the volumetric deformation or Jacobian. For physically admissible deformations without penetration of matter, J has to be positive. λ and μ are the Lamé constants, which we write in terms of the Young's modulus E and Poisson's ratio ν, so that $\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$ and $\mu = \frac{E}{2(1+\nu)}$. Using this formulation it becomes clear that Young's modulus is just a scaling factor and the energy functional only depends on Poisson's ratio, which we set to $\nu = 0.45$. $I_1^{\boldsymbol{C}}$ is the first invariant of the right Cauchy Green tensor, which in our context is given by $I_1^{\boldsymbol{C}} = \mathrm{tr}(\boldsymbol{C}) = \mathrm{tr}(\nabla\boldsymbol{\phi}^T\nabla\boldsymbol{\phi})$.

2.1. Dealing with mesh untangling

The strain energy functions W have a singularity at $J = 0$, so that W asymptotically tends towards infinity. This is a desirable property if the mesh is valid, i.e. all elements fulfil $J > 0$, as it prevents the mesh from tangling, however this is undesirable if the mesh is initially invalid, as it prevents untangling. As is common in these approaches for both linear and high-order meshes, e.g. [19], we apply a regularisation to the volumetric deformation J, that ensures the regularised version J_R remains positive and very small. It follows that the deformation function W no longer exhibits a singularity, and becomes very large for invalid and near-invalid elements driving the optimisation away from such configurations. The proposed regularisation is given as

$$J_R = \frac{1}{2}\left(J + \sqrt{4\delta^2 + J^2}\right), \tag{3}$$

with δ being a small number, set to

$$\delta = \begin{cases} \sqrt{10^{-8} + 0.04(J_{\min})^2}, & \text{if } J_{\min} < 0 \\ 10^{-4}, & \text{otherwise} \end{cases} \tag{4}$$

where $J_{\min}$ is the minimum Jacobian of all mesh elements.

However, all known regularisations destroy the convex property of the energy functional, so that the existence of a minimum of the energy is not theoretically guaranteed.

2.2. Numerical evaluation of the energy functional

Up to this point, the outline of the method refers only to a single element. We now focus on the implementation of optimising the energy functional on a mesh consisting of multiple elements. The energy functional of the whole mesh $\mathcal{E}$ is accordingly calculated as a sum of the elemental contributions

$$\mathcal{E}(\nabla\boldsymbol{\phi}(\boldsymbol{y})) = \sum_{e=1}^{N_{el}} \int_{\Omega_I^e} W(\nabla\boldsymbol{\phi}(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}, \tag{5}$$

with Ω_I^e being an initial undeformed straight-sided element. Finite element shape functions within *NekMesh* are evaluated on a reference element Ω_{st} with coordinates $\boldsymbol{\xi} \in \Omega_{st}$. Hence, another mapping between the reference element Ω_{st} and the straight-sided element Ω_I^e can be defined as $\boldsymbol{\phi}_I : \Omega_{st} \to \Omega_I^e$, as shown in Fig. 1. Applying the coordinate transformation, the energy functional becomes

$$\mathcal{E}(\nabla\boldsymbol{\phi}) = \sum_{e=1}^{N_{el}} \int_{\Omega_{st}} W\left[\nabla\boldsymbol{\phi}_M(\boldsymbol{\xi})\nabla\boldsymbol{\phi}_I^{-1}(\boldsymbol{\phi}_I(\boldsymbol{\xi}))\right] \det(\nabla\boldsymbol{\phi}_I)\,\mathrm{d}\boldsymbol{\xi}. \tag{6}$$

The ideal mapping $\boldsymbol{\phi}_I$ is affine and can be written analytically by combining linear finite element shape functions. It is independent of $\boldsymbol{\xi}$ for tetrahedral elements, so it is computed only once per element and stored. The curvilinear mapping $\boldsymbol{\phi}_M$ is an isoparametric mapping for nodal high-order element discretisations, given by

$$\boldsymbol{\phi}_M(\boldsymbol{\xi}) = \sum_{n=1}^{N} \boldsymbol{x}^n \boldsymbol{\ell}_n(\boldsymbol{\xi}), \tag{7}$$

where N is the number of nodes of the element and $\boldsymbol{\ell}_n$ are the Lagrange polynomial interpolants, which conform with $\boldsymbol{\ell}_m(\boldsymbol{\xi}^n) = \delta_{nm}$.

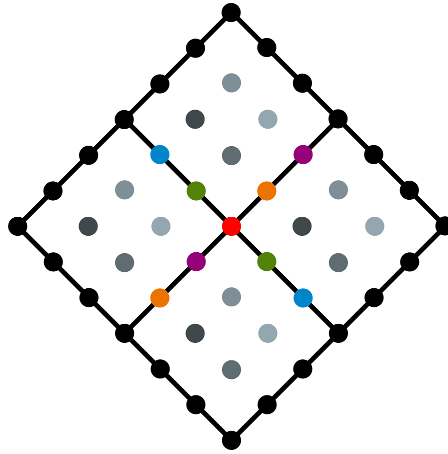


Fig. 2. Node colouring scheme for a domain of four quadrilateral elements. Nodes of the same colour can be processed concurrently. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The above equality relies on the definition of a set of points $\boldsymbol{\xi}^n \in \Omega_{\text{st}}$ that define the isoparametric mapping, for which we use a 3D nodal basis on each tetrahedral element, also known as Hesthaven or α-optimised points [20]. We note that the evaluation of the functional itself, as in Eq. (6), is performed using a different set of points conforming to a quadrature rule proposed by Witherden and Vincent [21]. This quadrature rule has positive weights at all evaluation points thus increasing the robustness of the optimisation procedure. This is advantageous because the use of quadrature rules with negative weights close to the vertices of the element leads to unrealistic displacements of these nodes during the optimisation. This is discussed further in Ref. [10].

2.3. Parallelisation and node-colouring

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, since individual nodes only affect the energy functional of elements they are connected with. A node that is interior to one element will only affect the energy contribution of that one element; inter-elemental boundary nodes will affect all elements they are connected with, which in the case of tetrahedra is two for face-nodes, typically around 4–10 for edge-nodes and around 14–50 for vertex nodes, depending on the connectivity of the mesh. Fig. 2 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.

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

$$\mathcal{E}_i(\nabla\boldsymbol{\phi}) = \sum_{e\ni i} \int_{\Omega_e} W(\nabla\boldsymbol{\phi})\,\mathrm{d}\boldsymbol{y} \tag{8}$$

where $e \ni i$ denotes the set of all elements that own node i. This set of elements hence spans all parts of the mesh that influence the energy functional of node i.

This local approach has the disadvantage that the optimisation might get stuck in local extrema. It is hence the challenge to employ an optimisation strategy that has a high degree of success in finding the global minimum. The main advantage of this strategy, however, is the potential for parallelisation, as each node typically only affects a small fraction of elements of the total mesh. Using node colouring the mesh is split into independent sets that can be processed in parallel. The splitting is subsequently repeated until all free-to-move nodes have been processed once. This makes the strategy very well suited for massively parallel hardware like GPUs.

2.4. Optimisation strategy

The minimisation of the energy functional is performed with 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 with respect to the position of the node undergoing optimisation. These are calculated analytically using the formulae in Appendix B of Ref. [11]. The coordinates of the free-to-move nodes are then updated in the optimisation step k according to

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha\boldsymbol{H}^{-1}\boldsymbol{G}, \tag{9}$$

with α being a step size parameter with $0 < \alpha \leq 1$. A reverse line search using the Wolfe condition is applied to find a value of α that guarantees the node moves towards a smaller energy level. Each free-to-move node is moved every time step (unless no smaller energy level can be found) and the time steps are repeated until a convergence criteria is fulfilled. In addition to the Jacobian regularisation discussed previously, a regularisation is applied to badly conditioned Hessians, to restore the symmetric positive-definite property in the presence of invalid or highly distorted elements, and hence increase the robustness of the method.

2.5. The overall algorithm

Algorithm 1 illustrates how the various evaluations are combined to yield the overall algorithm. The procedure EvaluateFunctional is further broken down in Section 3, where the crucial modifications that have been introduced in order to enable hardware portability are explained.

Algorithm 1 Variational mesh optimisation

```
1:  procedure GlobalOptimise(N)                                  ▷ N is the set of all nodes to be optimised
2:    C ← ColourNodes(N)                                         ▷ divide nodes into coloursets C
3:    call CopyData(N)                   ▷ deep copy from host to device memory when using GPUs
4:    while ||N^{k+1} − N^k||_∞ > ε_conv do                     ▷ using convergence criterion
5:      for all colour-sets c ∈ C do
6:        for all nodes n ∈ c do in parallel
7:          E, G(E), H(E) ← EvaluateFunctional(x_n^k)
                ▷ evaluate functional, derivatives and Hessian, based on node coordinates x
8:          while α > α_ε do                                     ▷ reverse line search, start with α = 1
9:            x^temp ← x_n^k + αH(E)^{-1}G(E)                    ▷ move node in the descent direction
10:           F ← EvaluateFunctional(x^temp)                     ▷ evaluate functional only
11:           if F ≤ E + f_Wolfe then                            ▷ using Wolfe condition
12:             x_n^{k+1} ← x^temp                               ▷ new minimum found
13:             break
14:           end if
15:           α ← (1/2)α
16:         end while
17:         x_n^{k+1} ← x_n^k                                    ▷ unable to optimise, reset node
18:       end for
19:     end for
20:   end while
21: end procedure
```

3. Accelerating the variational mesh optimisation method

This section discusses the acceleration of the variational framework outlined in the previous section using the architecture independent programming model *Kokkos*. We begin with a brief overview of the characteristics of performance portability that we wish to exploit in this work, and discuss some of the potential choices of programming models, as well as our motivation for using the *Kokkos* framework. We then outline the computational work required to port the initial version of the variational framework within *NekMesh* to an efficient *Kokkos* implementation.

3.1. Performance portability

The current coexistence and future uncertainty of different HPC hardware architectures demand an easily maintainable code. This is ideally achieved by supporting a single codebase that allows portability across architectures instead of maintaining specific and separate implementations for each architecture. It is also very desirable to have a codebase that is *performance portable*, meaning it can achieve nearly optimal performance on all architectures. Different options for programming models are the low-level language extensions *CUDA* [22] (specifically for NVIDIA GPUs) and *OpenCL* [23] (which supports a range of compute architectures), or high-level libraries and directives such as *OpenMP* [24], *OpenACC* [25], *SYCL* [26], *RAJA* [27] or *Kokkos* [12]. Low-level paradigms offer optimal performance, but require maintaining multiple codebases or tuning for specific hardware, and have hence not be considered further. Considering high-level libraries, there has been little experience with *SYCL* in the community, however a significant performance overhead compared to its only backend *OpenCL* has been reported. *OpenACC* and *OpenMP* are very widespread heterogeneous programming models, but its compiler directives are somewhat ambiguous and do not guarantee a performance portable memory access pattern. Even though from *OpenMP* version 4.0 onwards directives to offload to GPU architectures are introduced, no open-source compiler support for these capabilities have been available at the time of conducting our research. Similarly, no compiler support for *OpenACC* codebases to be executed on manycore CPU architectures has been available. These two programming models are hence not portable in practise, yet.

A promising model we identified is the *Kokkos* library [12], developed by Sandia National Laboratories as a part of the *Trilinos* suite, that supports a range of backends: *OpenMP* [24] and *Pthreads* [28] for multi-threading on CPUs and accelerators, and *CUDA* to support parallel execution on Nvidia GPUs. *Kokkos* addresses data and vector parallelism with task scheduling algorithms similar to *MPI* [29], *Pthreads* or *Qthreads* [30]. The syntax of *Kokkos* is general and architecture independent, but the backend still leverages architecture specific features, like the utilisation of GPU texture and shared memory. A unique property of *Kokkos* is its support for polymorphic data layouts which allows for optimal memory access and true performance portability. The authors of *Kokkos* present a performance evaluation against native *OpenMP* and *CUDA* implementations in Ref. [12] using a continuous Galerkin finite element algorithm, indicating only a small overhead over lower-level implementations. The *Kokkos* variant with *CUDA* backend runs about 13% faster than the native *CUDA* implementation on a Nvidia Kepler architecture. The *Kokkos-OpenMP* variant is about 10% slower than a native *OpenMP* implementation on the Xeon Phi architecture. This indicates only a small overhead of the library, which we will further assess in the following section.

RAJA is supposed to follow the same spirit as *Kokkos*, however, has been found in a far less mature state than *Kokkos*.

We adopt *Kokkos* as our programming model because the high level of performance it is able to attain and its compatibility with the existing *NekMesh C++* codebase. We only consider shared memory systems in this work, since the use of a single workstation is the most likely scenario for the optimisation of meshes with a few million elements. However, for problems requiring the optimisation of extremely large numbers of elements in the mesh, the work here could be extended to a distributed memory model using *MPI*.

Algorithm 2 Calculating the energy functional of a node with the initial implementation

1: **procedure** EVALUATEFUNCTIONALINITIAL($\boldsymbol{x}_n$)
2: $\mathcal{E}, \boldsymbol{G}(\mathcal{E}), \boldsymbol{H}(\mathcal{E}) \leftarrow 0, \boldsymbol{0}, \boldsymbol{0}$
3: $\boldsymbol{X} \leftarrow \boldsymbol{x} \in \{e \ni n\}$ ▷ coordinates $\boldsymbol{x}$ of all elements e that own node n
4: **for all** coordinate directions d **do**
5: $\nabla \boldsymbol{\phi}_{M_d} \leftarrow \mathcal{D}_d \boldsymbol{X}_d$ ▷ compute deformation tensor in direction d using `dgemm`
6: **end for**
7: **for all** elements $e \ni n$ **do in serial** ▷ all elements e that own node n
8: **for all** quadrature points $i \in e$ **do in serial**
9: $w \leftarrow w_i \det \boldsymbol{\phi}_I(\tilde{\boldsymbol{\xi}}^i)$ ▷ mapping determinant weighted by quadrature
10: $\mathcal{E} \leftarrow \mathcal{E} + W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$
11: **if** calculating gradient and Hessian **then**
12: **for all** coordinate directions j **do**
13: $\boldsymbol{G}(\mathcal{E})[j] \leftarrow \boldsymbol{G}(\mathcal{E})[j] + \partial_{n_j} W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$ ▷ analytic gradient
14: **for all** coordinate directions k **do**
15: $\boldsymbol{H}(\mathcal{E})[j,k] \leftarrow \boldsymbol{H}(\mathcal{E})[j,k] + \partial_{n_j n_k} W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$ ▷ analytic Hessian
16: **end for**
17: **end for**
18: **end if**
19: **end for**
20: **end for**
21: **end procedure**

3.2. The initial Pthreads implementation

With a view to later discussions of the development of the portable *Kokkos* implementation, we start by describing the initial implementation of the variational mesh optimisation algorithm. A short snippet of the code given in Appendix illustrates its characteristics that are discussed in this subsection.

3.2.1. Initial data structures

The initial CPU implementation of the variational meshing optimisation organises the data using shared pointers of *C++* objects. This way the complex associations between the set of nodes and elements can be organised clearly and descriptively, ensuring code maintainability. Each free-to-move node is an instance of the *node* class and each mesh element is an instance of the *element* class. The objects need to be shared because nodes on the surface of elements are part of all its neighbouring elements. Hence, a given *node* object has one or more *element* objects connected to it. At the same time, a given *element* object owns multiple free-to-move *node* objects and other fixed nodes to fill up the ranks. The mapping gradients, $\nabla \boldsymbol{\phi}_M$, for each node are further stored as member variables of each node instance.

3.2.2. Initial algorithm

All nodes within one colour set are processed in parallel, on the outer level of parallelism. Algorithm 1 illustrates how this process fits into the overall mesh optimisation algorithm. The parallelism is enabled by a lightweight dynamic thread-scheduler that is based on *Pthreads*.

Within the optimisation of each node, the energy functional at its initial position is evaluated first, along with its spatial derivatives and its Hessian. These values in turn depend on the contribution of all elements this node is connected with, as given in Eq. (6) and illustrated in Algorithm 2.

Since the whole function ENERGYFUNCTIONALINITIAL is performed using a single thread only, the expensive calculation of the mapping $\nabla \boldsymbol{\phi}_M$ is amalgamated using the level-3 BLAS call `dgemm`. The remaining steps of the function are calculated in a serial loop over all quadrature points of all elements that own the to be optimised node.

3.3. Developing the Kokkos implementation

To make our application portable, the initial implementation is adapted to use the *Kokkos* programming model. Even though the codebase should be changed as little as possible having maintainability in mind, some changes have been required, primarily in order to comply with the new programming model, but also to harvest the low-hanging performance benefits. Most design changes become necessary in order to comply with the *CUDA* backend of *Kokkos*. This is due to the high complexity of achieving maximal performance on GPU architectures, compared with more established CPU architectures. In fact, it is possible to use *Kokkos* for CPUs by only replacing the thread-scheduling. However, more fundamental design changes are required to achieve portability to GPUs. These will also be beneficial for later CPU executions, as they can, for example, instruct parallelism on the vector lanes. This section discusses the design changes in terms of re-factoring data structures and introducing hierarchical parallelism. A short example given in Appendix illustrates the changes introduced in the codebase to accommodate the new programming model.

3.3.1. Developing the Kokkos data structures

The *Kokkos* programming model using the *CUDA* backend requires all data and data dependencies to be expressed in plain arrays. It follows that the associative object-oriented data structures need to be re-factored. The resulting data structures and the rationale for its system is described hereafter.

All nodes of a given element need to be included for many evaluation and processing steps. It has therefore been decided to store all nodal coordinates of one element contiguously. The data is hence stored in three (X, Y, Z) two-dimensional arrays sorted by element ID in the first dimension and local-node ID in the second dimension. This leads to a duplication of stored data, due to the element boundary nodes, but results in a contiguous and hence more efficient memory access. The duplicated storage of node data requires increasing the memory by a factors of 4.2, 3.1 and 2.6 for tetrahedral meshes of orders 3, 4 and 5, respectively. However, even with this duplication of node data, the mapping $\phi_I : \Omega_{st} \mapsto \Omega_I^e$ of each quadrature node occupies most of the memory space, typically by more than 90%.

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 first array containing the number of elements a node is connected with. Each node instance then requires the element ID, as well as the local-node ID determining the position of the node within its element. These data are provided in two additional 3D arrays using the following indexing: the first dimension specifies a colour set, the second specifies a node within the colour set, and the third dimension specifies an element connected with this node. Listing 2 in the Appendix can be consulted for an illustration.

Apart from the node coordinates, utility data for elemental calculations are required and shall be discussed in light of using GPUs. One main group of utility data are the quadrature points and weights for the type and order of elements in the mesh. These are potentially accessed by multiple elemental operations at a time and hence stored in the fast read-only texture memory later. The second main group consists of the array describing the mapping of each node $\phi_I : \Omega_{st} \mapsto \Omega_I^e$ from the reference element to the straight-sided element. A 3×3 matrix and its determinant will be stored for each quadrature node on the global memory. The values are constant throughout the optimisation, as only the mapping ϕ_M between the straight-sided element and the curved element is altered. If the mesh is too big to be stored on the global memory of the GPU, this mapping function could be recalculated each time. This would free the majority of double precision floating-point values stored with each element, at the rather small cost of re-evaluating the mapping each timestep.

Using the *Kokkos* programming model all these arrays are implemented as *Kokkos-Views*. These are multidimensional arrays, that exhibit a polymorphic layout that is specified at compile time. The layout of a 2D array can be either column or row major, depending which memory access pattern is more performant on the underlying hardware. This key feature makes the *Kokkos* programming model a superior candidate in terms of performance portability. *Kokkos-Views* are implemented with a *C++* template class, allowing for a simple syntax. An unconventional feature is the combination of static and dynamic dimensioning of the array. This feature is for example employed to store the mapping-functions, which have a dynamic dimension dependent on the number of nodes, but a fixed dimension with respect to the tensor entries. To manage data in memory on both the CPU and the GPU, a pair of arrays need to be specified, one on the GPU memory and a so called host-mirror on the memory of the CPU. The main function GlobalOptimise first creates all necessary data on the CPU memory, where applicable using the *Nektar++* libraries of the initial implementation. Only after the creation of all *Kokkos-Views* on the host memory, the data arrays are deep-copied onto the memory of the device or GPU using an explicit instruction.

3.3.2. Hierarchical parallelism in the optimisation procedure

Algorithm 3 presents the implementation developed for performance portability using the *Kokkos* programming model. A few changes compared to the initial Algorithm 2 have become necessary to achieve good performance, that are discussed hereafter. Just as in the initial algorithm, all nodes within one colour set are processed in parallel. Additionally though, we introduce a second level of parallelism over the processing of all quadrature points of an element. This is crucial to achieve good performance on GPUs. As will be explained later, due to algorithmic divergences of the optimisation method it is not possible to have a very fine-grained parallelism at the node level. It is hence essential, that a nested level of parallelism is specified to utilise parallelism on the *CUDA*-thread level, that is dictated by the *CUDA*-warp concept.

Kokkos supports three levels of such hierarchical parallelism, denoted as *Team*, *Thread*, and *Vector* parallelism. The parallel computational kernel is conveniently implemented using the so called *lambda functions*: a feature of *C++11*. The outer level of parallelism processing individual nodes utilise the *Kokkos::Team* parallelism. Translated to the *CUDA* backend, this will assign one *CUDA* block to each node of the colourset.

Within this algorithm, all elements connected with a particular node are still processed sequentially, even though they could be processed in parallel. It was found to be the faster option, unless different block sizes per node were to be defined depending on the number of connected elements, which is not possible with the current version of *Kokkos*.

The inner level of parallelism processes the individual quadrature points of one element using the *Kokkos::Thread* parallelism. The contribution of each quadrature point will have to be added up, using the *Kokkos* options of either a parallel reduction or individual atomic operations. Since the number of quadrature points for typical elements is rather small, *Kokkos::atomic_add* operations are utilised. These operations serialise competing accesses of the same data elements, but do not guarantee their order of execution between threads. This serialisation can introduce a performance bottleneck, if there are not sufficient operations that can be processed concurrently. The alternative of storing all contributions and subsequently doing a *Kokkos::parallel_reduce* operation, however, was found to be slower due to the function overhead. The atomic operations can only be performed on *Kokkos::View* arrays, which in turn can only be initialised outside of parallel regions. Hence, we specify a two-dimensional *Kokkos-View* to store the functional, its derivatives and the Hessian of each node of the colourset. The inner level of parallelism translates to *CUDA*-threads when compiling for the *CUDA* backend. On multicore platforms using the *OpenMP* backend it would translate to an explicit instruction to employ vector lanes such as AVX or FMA to achieve parallel executions.

Another difference compared to the initial algorithm is the way $\nabla\phi_M$ is calculated. As multiple threads operate on the inner parallelism, the thread of each quadrature point calculates its corresponding part of the mapping. The calculation repeatedly involves the processing of all nodal coordinates of this element, which are hence loaded into shared memory for faster access using the relevant *Kokkos* functionality. Employing a BLAS or CuBLAS call within this inner level of parallelism for the small matrix–vector multiplications involved here, was not deemed to be beneficial.

Algorithm 3 Calculating the energy functional of a node with the full *Kokkos* implementation

1: **procedure** EVALUATEFUNCTIONAL_KOKKOS($\boldsymbol{x}_n$)
2: $\mathcal{E}, \boldsymbol{G}(\mathcal{E}), \boldsymbol{H}(\mathcal{E}) \leftarrow 0, \boldsymbol{0}, \boldsymbol{0}$
3: **for all** elements $e \ni n$ **do in serial** ▷ all elements e that own node n
4: $\boldsymbol{X} \leftarrow \boldsymbol{x} \in e$ ▷ coordinates $\boldsymbol{x}$ of element e only, load to shared memory
5: **for all** quadrature points $i \in e$ **do in parallel**
6: $\nabla \boldsymbol{\phi}_{M_i} \leftarrow \mathcal{D}_i \boldsymbol{X}_i$
▷ compute deformation tensor for this element and this quadrature point only
7: $w \leftarrow w_i \det \boldsymbol{\phi}_I(\tilde{\boldsymbol{\xi}}^i)$ ▷ mapping determinant weighted by quadrature
8: $\mathcal{E} \leftarrow \mathcal{E} + W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$ ▷ use atomic operation to update $\mathcal{E}$
9: **if** calculating gradient and Hessian **then**
10: **for all** coordinate directions j **do**
11: $\boldsymbol{G}(\mathcal{E})[j] \leftarrow \boldsymbol{G}(\mathcal{E})[j] + \partial_{n_j} W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$
▷ analytic gradient, use atomic operation to update $\boldsymbol{G}(\mathcal{E})$
12: **for all** coordinate directions k **do**
13: $\boldsymbol{H}(\mathcal{E})[j, k] \leftarrow \boldsymbol{H}(\mathcal{E})[j, k] + \partial_{n_j n_k} W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$
▷ analytic Hessian, use atomic operation to update $\boldsymbol{H}(\mathcal{E})$
14: **end for**
15: **end for**
16: **end if**
17: **end for**
18: **end for**
19: **end procedure**

3.3.3. Algorithmic issues when using the CUDA backend

A major challenge for our overall algorithm is that every node might undergo a different path within the optimisation algorithm. As seen in Algorithm 1, the number of iterations required within the reverse line search can vary from node to node thus each node requires independent processing in the current algorithm. This does not result in a noticeable performance penalty using thread parallelism on CPUs. However it has bigger performance implications for GPU executions due to the rather sparse individual node operations. Alternatively, an amalgamation of operations for multiple nodes could result in denser operations on a larger *CUDA* block, which in general is more performant. Such an amalgamation, though, would require a significant rearranging of our algorithms, which is outside the scope of this work.

We thus allocate one *CUDA* block per node, as only they can be executed independently by different streaming multiprocessors of the GPU. A more fine-grained parallelism of allocating one *CUDA*-thread per node would result in excessive code divergence. Since all 32 threads of one *CUDA*-warp (which is also the smallest reasonable *CUDA*-block size) have to execute the same instruction, a vast proportion of threads would be idle at each cycle.

Our algorithm only deals with small matrices and vectors, corresponding to the rather low polynomial orders of the elements, so we specify the smallest reasonable *CUDA* block size of 32 threads. However, if the maximum allowed number of blocks cannot be scheduled, not all theoretically available threads of the GPU can be utilised. The desired maximum number of blocks to be scheduled can be limited by the register size required per thread. As a compromise, we have set the register limit on the compilation for all three GPUs employed for the performance tests to 128 double floats per *CUDA* thread. Our algorithm – with an unrestricted register size – would have assigned 188 double floats per thread (using 4th-order tetrahedra), hence restricting the number of *CUDA* blocks that can be processed concurrently on each streaming multi-processor even stronger. To process the maximum number of blocks, the register limit would need to be 32 double floats, however, this would lead to extensive memory copying from the registers to the cache and vice versa. We have tested different settings of register limits and found that 128 would give the best run times on all three considered GPUs.

3.4. Improved node colouring algorithm

A few aspects concerning the node balancing need to be discussed. Firstly, it is favourable to have a sufficient number of nodes within each colour set to achieve full occupancy of the utilised hardware. In a naive node colouring implementation with a random treatment of the free-to-move nodes, however, the tail of the colour sets consisted only of a few nodes, thus slightly slowing down the overall algorithm.

A second aspect is to construct sets in which each node requires the same amount of processing steps. This can be realised by constructing colour sets of interior nodes only and others for element boundary nodes which are connected with the same or at least similar numbers of elements.

We now implement a new node colouring scheme taking into account the two mentioned aspects, as given in Algorithm 4. All free-to-move vertex and edge nodes are treated in descending order of their number of connected elements. This way nodes connected to similar numbers of elements are grouped together. Further, nodes connected to few elements are treated later, allowing to fill in the gaps of untreated elements in the mesh that have been induced by the nodes connected to many elements. This scheme results in a very small tail of colour sets that do not have a sufficient size. The face and the interior nodes are treated separately each, this way creating colour sets that are equal in size and in the number of connected elements.

Algorithm 4 New node colouring scheme

```
1:  procedure ColourNodes(N)
2:      N_VertexEdge ← all vertex and edge nodes in N
3:      N_VertexEdge ← Sort(N_VertexEdge)        ▷ Sort in descending order of connected elements
4:      N_Face ← all face nodes in N
5:      N_Volume ← all volume nodes in N
6:      C_VertexEdge ← CreateColourSets(N_VertexEdge)
7:      C_Face ← CreateColourSets(N_Face)
8:      C_Volume ← CreateColourSets(N_Volume)
9:      return C = C_VertexEdge ∪ C_Face ∪ C_Volume
10: end procedure
11: procedure CreateColourSets(N)
12:     while N is not empty do
13:         M ← all uncoloured nodes in N
14:         Create a new colour set c
15:         while M is not empty do
16:             Select the next node n ∈ M, include n in c
17:             Let A ← the set of elements connected to node n
18:             Remove n and all nodes in A from M
19:         end while
20:     end while
21:     return all colour sets c
22: end procedure
```

Compared with the naïve node colouring, our sorted node colouring improves runtimes on all considered architectures for typical cases by around 1%–2%. While each sorted colour group is processed more efficiently, the number of colour groups is slightly increased, adding an overhead that compromises the overall efficiency gain.

4. Performance evaluation of the *Kokkos* implementation

To evaluate the performance of the *Kokkos* implementation of the variational framework, we first outline the methodology for the tests, and compare the initial CPU-only implementation of [11] against different variants of the *Kokkos* implementations. Then we expand the tests to consider a wider range of multicore and manycore CPU and GPU architectures, and assess the performance of the implementation at a range of polynomial orders in terms of hardware occupancy and the relative cost per degree of freedom.

4.1. Methodology

Since the number of choices of parameters in this study is potentially very wide, we first consider sensible limitations in order to examine key aspects of the implementation.

The first point to consider is the choice of functional $\mathcal{E}$ in the variational framework, which represents the solid body model to be adopted. We note that there are minor performance differences between the four different methods used within the variational framework, since each functional has its own unique characteristic under the minimisation process, meaning that the number of iterations required to converge to an optimised mesh may vary considerably. In this work, we consider only the hyper-elastic functional, since this was shown in [10] and [11] to consistently yield higher quality meshes. We do however note that for an entirely complete evaluation, the quality of the resulting meshes need to be taken into account. This is however outside the scope of this paper.

Secondly, in this work we only consider meshes consisting of tetrahedral elements and not, for example, hybrid meshes of e.g. tetrahedra and prismatic elements. This choice enables us to remove an aspect of load balancing from our implementation, since different element types require different computational requirements in the calculation of the energy functional. This restriction therefore allows us to consider a well-balanced problem which is more capable of attaining higher peak performance of the hardware. We note that, as shown in [11], in order for the optimisation algorithm to work reliably, the field values on each elemental node need to be evaluated using quadratures of at least four orders higher than the polynomial order of the element, using distributions of points that have positive quadrature weights. As nodal bases and associated quadrature rules satisfying this property for tetrahedral elements are only known up to 9th order, our tests are confined to tetrahedra of up to 5th order.

Finally, we consider only meshes which are initially valid, so that all nodes undergo the same path in the optimisation routine, therefore allowing for a more accurate performance evaluation. We found that nodes of invalid elements typically require up to ten times more operations within the reverse line search than nodes of valid elements. It should be noted that most real cases, however, comprise at least some initially invalid elements. We only evaluate the parallel part of the optimisation algorithm, neglecting the preprocessing and the initial data copying to the device. All timings are taken for 10 optimisation steps and are an average of five runs.

4.2. The different implementations

In this performance evaluation, we compare four different implementations based on the initial and the adapted algorithm. These are three versions to be executed on CPUs and one to be executed on GPUs, where the first two are based on Algorithm 2 and the latter two on Algorithm 3:

Table 1
Selected GPU performance specifications.

	Nvidia Tesla K40	Nvidia GTX 1070	Nvidia Tesla P100
Architecture	Kepler 3.7	Pascal 6.1	Pascal 6.0
Streaming Multiprocessors	15	15	56
FP64 (DP) Cores/SM	64	4	32
FP32 (SP) Cores/SM	192	128	64
GPU Boost Clock	875 MHz	1987 MHz	1480 MHz
Peak FP64 GFLOPs	1680	238.44	5304.32
DRAM Memory	12 GB	8 GB	12 GB
Price (April 2017)	\$3489	\$455	\$4912
Thermal Design Point	235 W	150 W	250 W

1. **Initial native *Pthreads*.** The first implementation is the baseline that achieves multi-threading on a CPU with a native *Pthreads* implementation using a light-weight thread-manager and utilises associative object-oriented data layouts.
2. **Intermediate *Kokkos-OpenMP*.** For the second implementation the direct thread scheduling of *Pthreads* has been replaced with a *Kokkos* parallel for loop using the *OpenMP* backend. Alternatively, a *Pthreads* backend could have been specified, however, was found to perform slightly worse.
3. **Full *Kokkos-OpenMP*.** The third implementation is a full *Kokkos* versions utilising plain data structures and is compiled with the *OpenMP* backend. Again, the *OpenMP* backend gave a superior performance over the *Pthreads* backend and has hence been chosen. Using the same backend for version 2 and 3 further allows a better comparison of the influence of changing the data structures.
4. **Full *Kokkos-CUDA*.** The fourth versions' code-base is identical to version 3, but is compiled for the *CUDA* backend and to be run on the GPU, using the performance portability of *Kokkos*.

4.3. Utilised hardware

For the multicore CPU system an Intel(R) Xeon(R) E5-2670v3 @ 2.30 GHz processors is employed, consisting of 2 sockets of 12 cores each and allowing up to 48 threads using hyper-threading. The two sockets can achieve a maximum theoretical performance of 883 DP-GFLOPs (double precision gigaflop per second), assuming AVX2 (Advanced Vector Instruction 2.0) and FMA (fused multiply–add) operations are used. The operating system of the CPU machine is Debian 8.7, and as the compiler gcc 4.9.2 with the *-O3* optimisation flag has been used.

As a manycore accelerator system we employ an Intel Xeon Phi 7210 of the *Knights Landing* architecture, consisting of 64 cores operating at 1.3 GHz with a boost clock of 1.5 GHz, and using up to 4 hyper-threads per core. We operate the device in the *flat* setting, making use of the 16 GB of fast MCDRAM. The Xeon Phi 7210 has a theoretical peak performance of 3072 DP-GFLOPs and a thermal design point of 215 W.

As GPUs we employ a Nvidia Tesla K40, a Nvidia GTX 1070, and a Nvidia Pascal P100. Some performance specifications are given in Table 1. The executables for the Tesla K40 and the Tesla P100 have been compiled on a *CentOS* Linux 7.3.1611 server, using gcc 4.8.5 with the *-O3* flag. The GTX 1070 machine operates on an *OpenSUSE* Tumbleweed system and the utilised compiler has been gcc 5.4.1 with the *-O3* flag. The numbers of FP64 or double precision (DP) cores per streaming multiprocessor (SM) show that the Tesla cards are intended for DP-intense HPC applications, whereas the GTX card offers mostly single precision cores and is hence intended as a gaming card. The low price tag however can still make it a good option for some applications. On all cards we utilise the DP cores only, as our tests show that the algorithm requires higher accuracy to obtain reasonable results. On all three GPUs we further limit the register size per thread, as discussed in Section 3.3.3, using *-maxregcount 128*.

4.4. General scaling and CPU performance evaluation

The first of the two main points to assess is the performance of the full *Kokkos* CPU implementation (version 3) compared to the initial CPU implementation (version 1). To this end we conduct two standard scaling exercises, a weak and a strong scaling of the three CPU versions. Evaluating the intermediate implementation (version 2) allows to separate the performance influence of the *Kokkos* thread scheduling and the *Kokkos* data structures.

4.4.1. Weak scaling on CPUs

For the weak scaling exercise we construct a set of meshes with a number of elements proportional to the number of concurrent threads they will be processed with. Since the processing effort of our optimisation algorithm depends on the initial mesh deformation, all meshes should ideally be equally deformed for a fair performance evaluation. For the initial mesh deformation we use the simple case of a sphere within a cube: the volume to be meshed is interior to the cube and exterior to the sphere(s). The mesh deformation is then determined by the ratio of the sizes of the sphere and a characteristic element. Different mesh sizes are then created by using stacks of equally sized spheres in a cuboidal domain and equal reference element sizes. All elements are third-order tetrahedra and an over-integration of fourth order is applied. The created mesh sizes are all within a minimum range that ensures the hardware is fully occupied. We use stack lengths of 1–48 spheres for 1–48 threads for the CPU exercise. As an example, Fig. 3 shows the mesh consisting of four spheres to be scheduled with four threads.

The timings obtained for the three CPU versions have been scaled with the exact numbers of elements of each mesh and are given in Fig. 4. The main observation is the superior performance of the new *Kokkos-OpenMP* implementation, which reduces execution time by a factor of two on a single core. The scaling, however, is worse than the initial *Pthreads* implementation, so that 24 threads result in about equal run times. The intermediate *Kokkos* version shows a similar scaling to the full *Kokkos* version, but the run times are in general at least two times slower. Comparing the intermediate to the initial implementation, the run time is slightly faster with one thread, but due to the

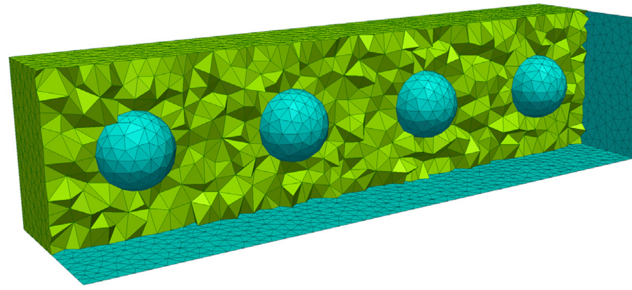


Fig. 3. Stack of four spheres in a cuboid, where the volume is meshed with tetrahedral elements.

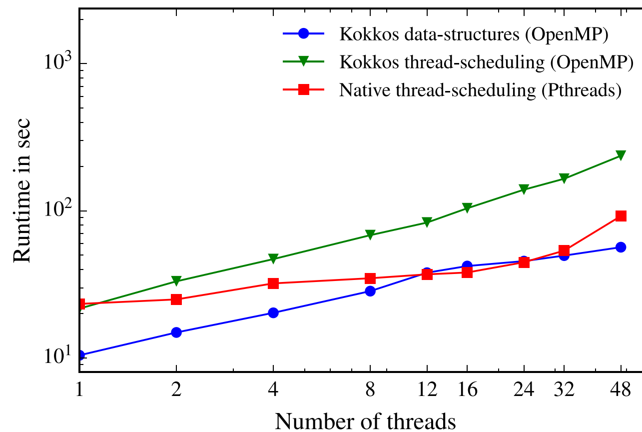


Fig. 4. Weak scaling of the parallel part of different CPU implementations of the variational high-order mesh optimisation method.

inefficiency of the scaling, the performance gap increases as more threads are used. It shows that the thread-scheduling of *Kokkos* performs worse than our native *Pthreads* thread-scheduler. This could be caused by the difference between the dynamic thread scheduling of the native *Pthreads* version and the fork-join thread scheduling of the *Kokkos* versions. The execution times using one thread, however, show that the *Kokkos* thread scheduling overhead is at least on par with *Pthreads*. Hence the *Kokkos* infrastructure allows for similar performance, but the thread-scheduling could be improved. The performance deficit might be much smaller in cases with well balanced loads per thread, but in our case of varying work sizes per node-colour group the effect is significant.

A second important observation is that the new data layout results in faster run times, due to an improved memory efficiency. The improved efficiency can be assigned both to the effects of the polymorphic *Kokkos* view layout, but also to the general re-factoring of the data structures, that has been a precondition for using *Kokkos* views. The change from the associative data layout of shared pointers to flat data arrays will in itself result in faster memory access. Moreover, using 32 or 48 threads and thus in the domain of hyper-threading, the new *Kokkos-OpenMP* version performs far better than the two other versions. Utilising the full number of threads on our CPU machine, the full *Kokkos* version is 1.62 times faster than our initial implementation and 4.18 times faster than the intermediate version.

4.4.2. Strong scaling on CPUs

For the strong scaling exercise we use an initially valid, but non-optimised mesh of 33 K elements and 400 K degrees of freedom, using again 1–48 threads. The elements are third-order tetrahedra and an over-integration of fourth order is applied. The timings for all three CPU versions can be found in Fig. 5.

The overall results of the relative performance between the three versions are similar to the characteristics seen in the weak scaling tests. We clearly observe that the scaling of all versions is roughly linear, but with different slopes. The negative effect of using hyper-threading on the native *Pthreads* version is more pronounced than with the weak scaling. Up to 24 threads (1 thread per core), this version achieves a very good scaling of 64%, but no further speed-up using hyper-threading can be realised. The full *Kokkos* implementation achieves a scaling of 20%. Throughout the range the intermediate *Kokkos* version shows a similar scaling, but 2–2.5 times slower run times than the full *Kokkos* version. Using the full number of threads, the full *Kokkos* version is 1.42 times faster than the initial *Pthreads* implementation and 2.48 times faster than the intermediate version.

For the strong scaling test using the full *Kokkos* version and one thread, 18.32% of the theoretical FLOPs (considering AVX2 and FMA) have been achieved, and still 3.54% using 48 threads. These numbers have been calculated as

$$\%(\text{peak FLOPs}) = \frac{\text{achieved FLOPs}}{\text{peak FLOPs}} \frac{\text{maximum threads}}{\text{utilised threads}}. \tag{10}$$

Again, it becomes clear that the new *Kokkos* data structures are far more efficient than the initial associative ones using arrays of shared pointers. This is an even better result, considering that we used a portable programming model and did not optimise the algorithm or data structures for a specific architecture.

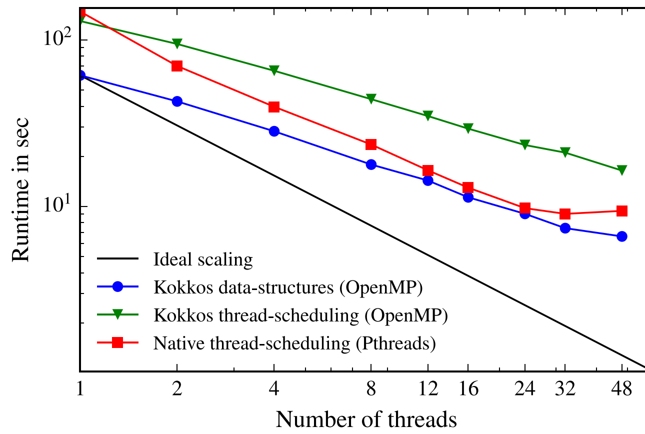


Fig. 5. Strong scaling of the parallel part of different CPU implementations of the variational high-order mesh optimisation method.

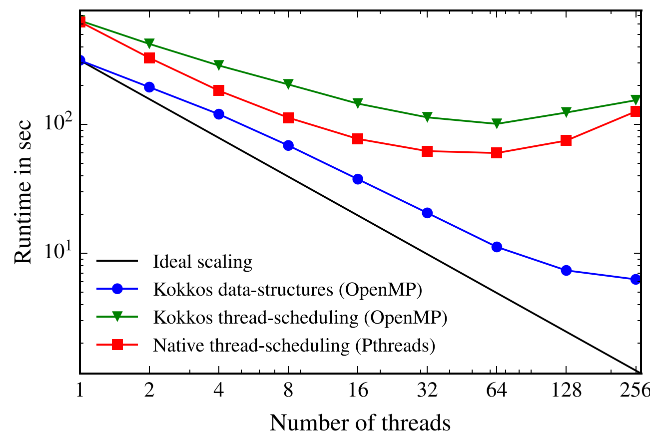


Fig. 6. Strong scaling of the parallel part of different manycore implementations of the variational high-order mesh optimisation method executed on a *Knights Landing* (KNL) accelerator.

4.5. Strong scaling on a manycore accelerator

As a final test, we repeat the strong scaling exercise using the same mesh on the manycore accelerator. The averaged timings of our three CPU implementations are shown in Fig. 6. In this manycore environment, which is additionally hindered by stronger memory bandwidth restrictions, the effect of using simpler data structures can clearly be seen, with the full *Kokkos* implementation being by far the best performing. With the *Kokkos* thread scheduling and data structures a difference of one order of magnitude with a 9.5 times lower runtime than with the native *Pthreads* version is achieved. This indicates the suitability of the *Kokkos* data arrays for highly parallel shared memory systems. Up to 64 threads (1 thread per core) we obtain a very good scaling of 44% for the full *Kokkos* version, compared with 16% for the native version. Further, only the full *Kokkos* version can benefit from hyper-threading using up to 256 threads, whereas it is found counter-productive for the other two versions.

4.6. Performance comparison between architectures

Using the portable *Kokkos* programming model, we already achieved better performance on the CPU, compared to our initial implementation. The second main point to assess is if further performance benefits can be realised running the same algorithm on GPUs. We emphasise that, asides from specifying some arrays to be loaded to texture memory and others to shared memory (which can be specified using the *Kokkos*-syntax), no specific GPU optimisation has been undertaken. This section therefore aims to compare the suitability of CPU vs GPU systems, using the *Kokkos-CUDA* version on three different GPUs and the best performing CPU implementation, the full *Kokkos-OpenMP* version, on the CPU system and an Intel Xeon Phi of the *Knights Landing* architecture.

In this section we consider a set of four different meshes, all using the same geometry and the same number of elements, but using different polynomials orders and therefore different degrees of freedom (DOF), as shown in Table 2. This table also shows the average number of DP-GFLOP (double precision gigaflop) required by our algorithm for 10 optimisation steps, so it contains exactly the number of operations performed within our timing interval. The GFLOP measurements were obtained using the Nvidia Visual Profiler [31].

The averaged run times obtained on our CPU system, the three GPU systems and the KNL accelerator are scaled by the DOF and given in Fig. 7. Fig. 8 additionally shows how much faster the GPU and accelerator run times are compared with the CPU run times. The main observation is that a faster runtime on the GPUs with the latest Pascal architecture is realised. The GTX 1070 is 150%–200% faster, whereas

Table 2
Statistics of the architecture comparison set of meshes; including degrees of freedom (DOF) and double precision FLOP for 10 steps of the mesh optimisation algorithm.

Element order	Elements	DOF	DP-GFLOP	FLOP/DOF
2nd	95982	338 469	163.6	483.5
3rd	95982	1197 165	628.3	524.8
4th	95982	2899 428	1824.8	629.4
5th	95982	5733 204	4943.9	862.3

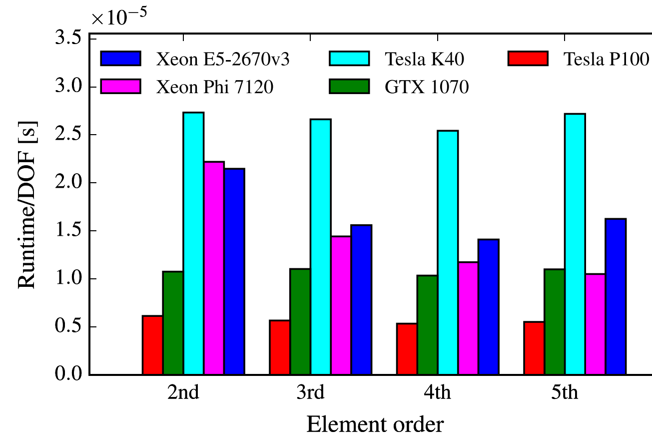


Fig. 7. Run times normalised by DOFs for meshes with varying element order on different systems.

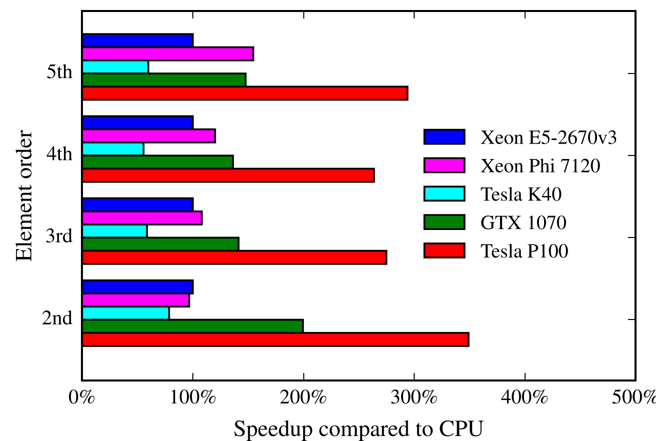


Fig. 8. Speed-ups of different GPU/accelerator systems compared to CPUs for meshes with varying element order.

the Tesla P100 is even 275%–350% faster than the Xeon E5-2970v3 CPU. The performance on the Tesla K40, however, is worse than on the CPU system. The Intel Xeon Phi 7120 accelerator is up to 150% faster than the CPU.

The run times vary for different element polynomial order, which is due to two opposing effects. Firstly, the average number of operations or FLOP required by our algorithm per degree of freedom increases with element polynomial order, as seen in Table 2, which is a common observation for high-order methods. Secondly, the algorithm maps with varying efficiency to the different hardware. As a straightforward measure we consult the percentages of theoretical peak FLOPs that are achieved, given in Fig. 9. The higher the polynomial order, the more efficient the hardware is utilised, across all systems. This is due to more compact data structures of high-order elements, that allow a more efficient memory access. Both effects combined, 4th order elements achieve the optimum run time per DOF for all GPU- and the CPU systems. The Xeon Phi accelerator benefits from even higher polynomial orders.

An important observation is, though, that our architecture-independent algorithm cannot fully utilise the large number of DP cores on the Tesla GPUs, whereas the 4 DP cores per SM on the GTX card can be well utilised. The evaluation of the memory bandwidth below explains this effect, which is most probably caused by the different ratio of DP-compute cores to load-store-units (LSUs).

4.7. *Further performance evaluation of GPU version*

Opposed to CPUs, it is not possible to execute a code only on a specific number of streaming multiprocessors of a GPU. Hence no meaningful scaling exercise for single GPUs can be devised. Yet, there are other meaningful performance metrics that can be evaluated for GPU executions. These are readily available using the *nvprof* profiler, that is part of the *CUDA* package [22]. The equivalent profiling metrics for modern CPUs are very difficult to examine reliably, so we choose to omit those here.

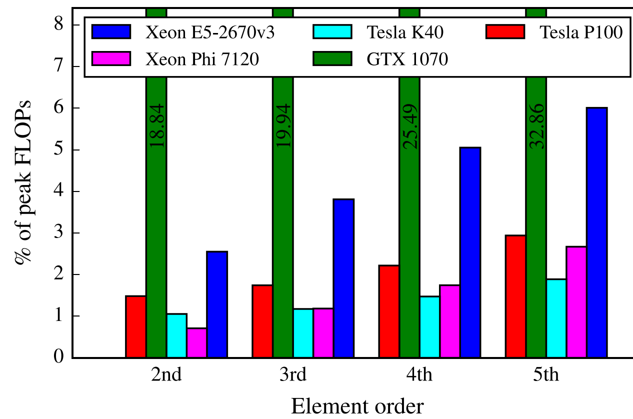


Fig. 9. Achieved percentages of theoretical peak FLOPs (DP) of different systems for meshes with varying element order.

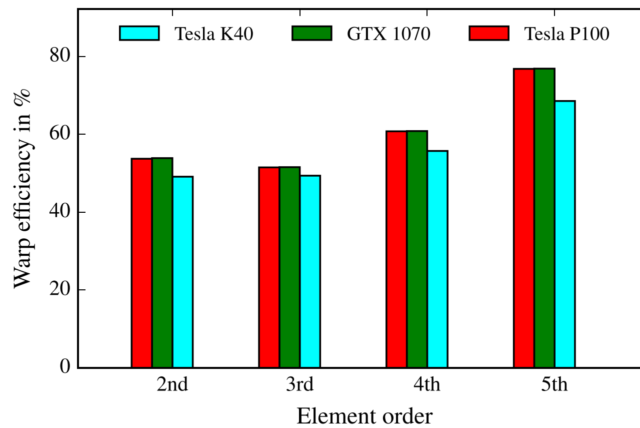


Fig. 10. Warp efficiency of different GPU systems for meshes with varying element order.

Apart from the achieved percentage of theoretical peak FLOPs, we consider the warp efficiency. It is defined as the average percentage of threads in a warp (a *CUDA*-warp always consists of 32 threads) that are performing useful work. This metric is very important as any improvement will directly corresponds to an equally large improvement in percentage of peak FLOPs. The results are shown in Fig. 10. The higher the polynomial order of the mesh elements, the higher is the observed warp efficiency. A considerable amount of the code is spend evaluating the quadrature points of an element, which are assigned a thread each. Higher-order elements have more quadrature points and can hence be distributed better to fill up multiples of 32 threads or one warp. The slightly lower efficiency of the Tesla K40 might be explained by the different *CUDA* compute capability. This trend partly explains why a higher element order results in a higher percentage of theoretical peak FLOPs. The other crucial limiting factor is the memory access.

To this end we evaluate the memory bandwidth of the DRAM on the device. It is a metric to evaluate how well the data structures are mapped to the physical memory in order to allow efficient coalesced memory-access. The percentage of the theoretical peak bandwidth is given in Fig. 11. The important observation is that – independently of the GPU and the polynomial order – the utilisation of the memory bandwidth is rather low. We can infer theoretically from our algorithm that the calculations performed for each node deal with many small individual arrays. We have already discussed that at the same time our algorithm is very memory intense, requiring large register sizes. This will lead to a low degree of coalesced memory accesses and hence result in the low observed memory bandwidth. Further, we also did not undertake any attempts to optimise the memory operations.

We might conclude that our algorithm requires a high ratio of memory load-store-units (LSUs) to compute cores in order to load the required data into the registers. This ratio is high if DP operations on a GTX card are performed, but is far worse with the higher number of DP cores on a Tesla card. This might explain why the percentages of theoretical peak FLOPs on the two Tesla cards are so low.

4.8. Cost comparison

There is no established procedure to compare the overall efficiency between very different architectures, like CPUs and GPUs. The two fundamental constraints for HPC users are time to solution and cost budgets. Time to solution can be compared straightforwardly between all kind of systems. But without taking costs into account it would be too simplistic to compare the hypothetical case of a 1000$ CPU system with a 5000$ GPU system and claim the GPU system performs better on the grounds of achieving half the run time for a given simulation. Provided the applications achieve a good scaling, better run times can most often be gained by spending a higher budget. It follows that the costs to operate a certain hardware system need to be taken into account.

The biggest annualised operating costs contributor for a HPC server is the computing hardware acquisition cost, which has also been considered in [32]. This reference proposes a novel cost metric termed resource utilisation, which is the product of run time and the

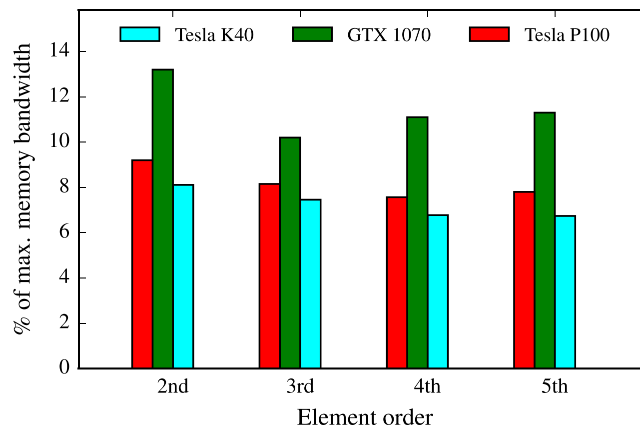


Fig. 11. Achieved percentage of theoretical peak bandwidth of the device memory for different GPU systems and meshes with varying element order.

Table 3
Averaged monthly prices for equivalent systems on bare-metal cloud services, as of April 2017.

System	Monthly price in $
Xeon E5-2670v3	913.53
Xeon Phi 7120 + RAM	774.56
Tesla K40 + host	1001.64
GTX 1070 + host	499.56
Tesla P100 + host	1412.81

hardware acquisition cost, with the units $\$ \times s$. However, this is only a combined time–cost metric when hardware acquisition costs are dominant and not annualised, as in the case of a local workstation. In a HPC server case, where the user pays for a certain length of user-time, however, the product of costs per time and run time has the unit $, and it becomes a pure cost metric. The resource utilisation metric also does not include electricity consumption, maintenance and peripheral costs, which are significant and also need to be carefully considered.

Cloud computing providers take all these costs into account when determining their consumer price. Academic HPC facilities, too, will consider the same cost contributors to meet their annual cost budget, but prices for user-time are not charged as such. We therefore propose a cost metric based on prices of bare-metal cloud-computing providers. Only renting *bare-metal* and not *elastic* cloud services would guarantee the algorithms to be executed on a specific hardware and thus allow for an accurate performance and cost evaluation. We gathered the monthly operating prices from three major providers (Amazon Web Services, Google Cloud Platform, IBM Cloud) for equivalent systems that we used for our performance runs, as of April 2017. For the GPU systems we considered the monthly prices of the GPU itself and added the monthly price of a simple host system consisting of an 8-core Xeon E5-2650v3 processor and 64 GB RAM. For the Xeon Phi accelerator system we considered the device plus 64 GB of RAM. The averaged prices for the equivalent systems are given in Table 3. Based on theoretical performance to price, we would consider a fair charge for a Tesla K40 compared to a Tesla P100 to be lower, however.

The costs of our runs are calculated as the product of run time and monthly price for the system and normalised by the DOFs. The absolute numbers are given in Fig. 12 and the cost improvements compared to the CPU system are given in Fig. 13. The main observation is that both Pascal GPUs achieve a cost advantage over the CPU system of 170%–225% for the Tesla P100 and an even higher 250%–365% for the GTX 1070. The Tesla K40 is not cost efficient, due to both the low hardware utilisation and the currently high price tag. The Xeon Phi accelerator achieves a cost advantage of 115%–180%. Comparing the different polynomial orders of the elements, unsurprisingly, the exact same trends as for the run times can be observed. For all GPU architectures and the CPU the 4th order element is the most efficient, while this effect is only pronounced on the CPU. The cost variation of the Intel Xeon Phi accelerator is the strongest, being hardly competitive using 2nd order elements, but being in the same range as the Pascal GPUs for 5th order elements.

We can compare the two considered constraints costs and run times in a single scatter-plot, as shown in Fig. 14. This conveys the results very effectively; compared with the CPU, the Tesla P100 and the GTX 1070 perform better on both metrics. The choice of which hardware to employ can then be made depending on which constraint is more important for the individual user. The Xeon Phi accelerator performs better than the CPU, too, but is never better than the two Pascal GPUs. The high performance variation of the CPU and the Xeon Phi accelerator depending on the element order is further clearly shown.

5. Conclusion

We have implemented a high-order mesh optimisation algorithm with an architecture independent programming model using the *Kokkos* library. The general mesh optimisation algorithm allows for the correction and optimisation of high-order meshes using a variational energy functional. The implementation has been presented with tetrahedra of second to fifth order and the hyper-elastic energy functional. The new *Kokkos* implementation is based on polymorphic data structures that allow efficient memory access on both CPU and GPU architectures. The implementation further makes use of thread-scheduling with an *OpenMP* backend on CPUs and a *CUDA* backend on GPUs. For the *CUDA* backend we use *Kokkos* functionalities to specify the memory locality of certain data. We emphasise that although we did not attempt to optimise our algorithm for any specific parallel architecture, our results have shown that it is possible to port our

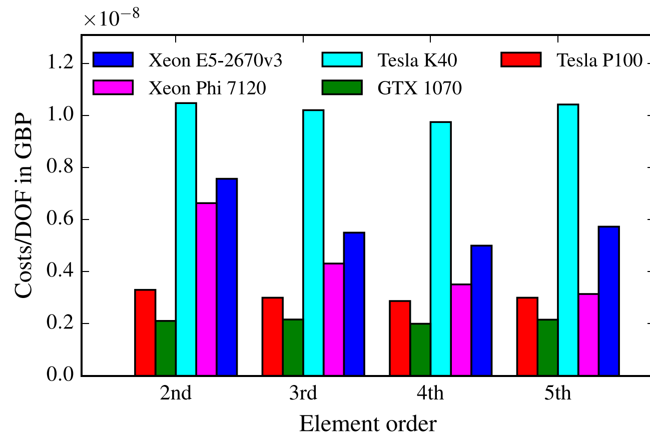


Fig. 12. Bare-metal cloud-computing costs normalised by DOFs for meshes with varying element order on different systems.

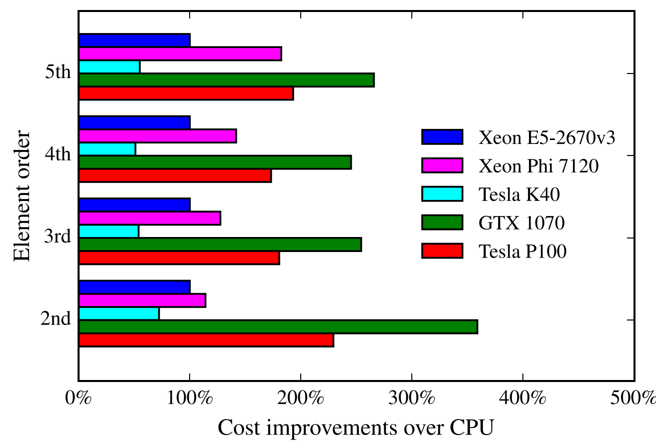


Fig. 13. Bare-metal cloud-computing cost improvements of different GPU/accelerator systems compared to CPUs for meshes with varying element order.

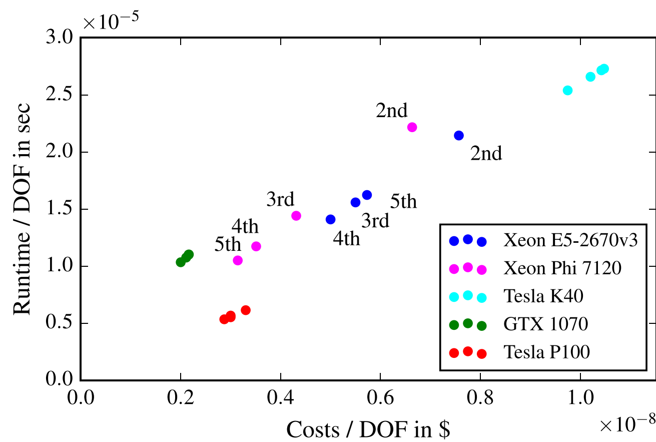


Fig. 14. Comparison of costs and runtime per DOF for different systems and varying element order.

high-order mesh optimisation algorithm to GPUs and Xeon Phi accelerators, and obtain both a time and cost advantage compared to CPU runs.

Our results generally show that *Kokkos* data structures are processed more efficiently on modern hardware than the associative ones used in our initial implementation, which used arrays of shared pointers. This effect was especially pronounced on Xeon Phi *Knights Landing* accelerators with a runtime difference of almost one order of magnitude compared to our optimised *Kokkos* implementation. This highlights the importance of efficient data structures on manycore devices and future hardware systems following this trend. Once a full

Kokkos implementation that works with the *CUDA* backend has been completed, it is deemed to offer good code maintainability. However, we have shown that legacy code relying on heavy object oriented programming features requires substantial refactoring of data structures and associated data management. A programming model that would avoid such a refactoring step would be highly beneficial.

Additionally, we have developed a cost metric that is based on monthly prices for equivalent systems on bare-metal cloud-computing servers. Only with both run-times and a cost metric a fair comparisons between different architectures is possible. Compared with our CPU system, the Tesla P100 is 170%–220% and the GTX1070 250%–360% more cost efficient. The Xeon Phi accelerator's cost efficiency increases from 115% up to 180% with increasing element order.

Kokkos itself has proven to be a useful tool in obtaining high-performance portability of our code. However we do note a few shortcomings of the library in our results. The most notable is the scheduling of threads, which was found to lack some efficiency in our weak- and strong-scaling tests. It is also important to note that some knowledge of the underlying hardware is required in order to tune performance, such as selecting optimal block sizes and register limits, which would ideally be automated in a fully architecture independent programming model. Additionally, the library can only perform as well as the choice of algorithm, as shown by the low efficiency on Tesla GPUs, which is most probably due to the rather sparse operations and an inefficient memory access. To improve this considerably, either detailed code optimisation or a rewriting of the underlying algorithm would be necessary. Both aspects, though, defeat our attempt in porting an initial CPU version to other architectures with minimal effort using an architecture independent programming model. We also found that the investigation of the cause of these problems is hindered by the high-level nature of the library, which means that parts of the parallel codebase are hidden when using profiling tools such as the Nvidia Visual Profiler, making it more complex to realise code optimisation.

Acknowledgements

JE gratefully acknowledges the support by the President's PhD Scholarship of Imperial College London. MG acknowledges support from the PRISM project under EPSRC grant EP/L000407/1. MT acknowledges Airbus and EPSRC for funding under an industrial CASE studentship. DM acknowledges support from the EU Horizon 2020 project ExaFLOW (grant 671571). The Quadro P5000 GPU used for this research was kindly donated by the NVIDIA Corporation.

Appendix. Code snippets

Listing 1: Managing of node coordinates and parallelism in the initial implementation

```
1  // Shared pointer of node objects.
2  vector<vector<boost::shared_ptr<NodeOpti>>> opti_nodes;
3  // Loop over all coloursets in serial.
4  for (int cs = 0; cs < opti_nodes.size(); cs++)
5  {
6      // Get the size of the colourset.
7      const int nodes = opti_nodes[cs].size();
8      // Create pthread jobs to optimise each node
9      vector<Thread::ThreadJob *> jobs(nodes);
10     for (int node = 0; node < nodes; node++)
11     {
12         jobs[node] = opti_nodes[cs][node]->Optimise();
13     }
14     // Queue up parallel jobs and wait for their completion.
15     QueueJobs(jobs);
16     Wait();
17 }
18
19 // The initial node optimisation routine.
20 void NodeOpti::Optimise()
21 {
22     // Get the number of connected elements from a member
23     // variable of the node, m_elmts.
24     const int elmt = m_elmts.size();
25
26     // Loop over all connected elements in serial
27     for (int el = 0; el < elmts; ++el)
28     {
29         // Obtain node coordinates from the node member variables.
30         x = m_node->m_x;
31         y = m_node->m_y;
32         z = m_node->m_z;
33         // Loop over all quadrature points in serial.
34         for (int qp = 0; qp < m_utilities->n_qp; ++qp)
35         {
36             // Calculations per quadrature point are performed
37             // inside this for-loop.
38         }
39     }
40 }
```

Listing 2: Managing of node coordinates and parallelism in the full *Kokkos* implementation

```
// Loop over all coloursets in serial.
for (int cs = 0; cs < css; cs++)
{
    // Get the size of each colourset.
    const int nodes = nodes_array(cs);

    // Use Kokkos syntax to create parallel teams for each node.
    Kokkos::parallel_for (team_policy (nodes, Kokkos::AUTO),
    KOKKOS_LAMBDA (const member_type& teamMember)
    {
        // Get the node to process from rank of thread.
        const int node = teamMember.league_rank();
        // Get the number of connected elements.
        const int elmts = elmts_array(cs, node);

        // Loop over all connected elements in serial
        for (int el = 0; el < elmts; ++el)
        {
            // Get the node indices and location from Kokkos views.
            const int elid = elid_array(cs, node, el);
            const int localnodeid = localnodeid_array(cs, node, el);
            // Get the node coordinates based on the indices.
            x = X(elid, localnodeid);
            y = Y(elid, localnodeid);
            z = Z(elid, localnodeid);
            // Create another layer of parallelism for each quadrature
            // point using Kokkos syntax.
            Kokkos::parallel_for (Kokkos::TeamThreadRange (teamMember, qps),
            [&] (const int qp)
            {
                // Calculations per quadrature point are performed inside
                // this lambda function.
            });
        }
    });
}
```

References

[1] G. Karniadakis, S. Sherwin, Spectral/hp Element Methods for Computational Fluid Dynamics, second ed., Oxford University Press, 2005.
[2] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, D. Mavriplis, CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences, NASA/CR-2014-21878, 2014, pp. 1–73.
[3] Z.Q. Xie, R. Sevilla, O. Hassan, K. Morgan, Comput. Mech. 51 (3) (2013) 361–374. http://dx.doi.org/10.1007/s00466-012-0736-4.
[4] M. Fortunato, P.O. Persson, J. Comput. Phys. 307 (2016) 1–14. http://dx.doi.org/10.1016/j.jcp.2015.11.020.
[5] P.-O. Persson, J. Peraire, Proc. of the 47th AIAA Aerospace Sciences Meeting and Exhibit, January 2009, AIAA-2009-949, 2009, pp. 1–11. http://dx.doi.org/10.2514/6.2009-949.
[6] S.J. Sherwin, J. Peiró, Internat. J. Numer. Methods Engrg. 53 (1) (2002) 207–223. http://dx.doi.org/10.1002/nme.397.
[7] A. Gargallo-Peiró, X. Roca, J. Sarrate, Comput. Mech. 53 (4) (2014) 587–609. http://dx.doi.org/10.1007/s00466-013-0920-1.
[8] Q. Lu, M.S. Shephard, S. Tendulkar, M.W. Beall, Eng. Comput. 30 (2) (2014) 271–286. http://dx.doi.org/10.1007/s00366-013-0329-7.
[9] D. Ibanez, M. Shephard, 25th International Meshing Roundtable, IMR25, 2016.
[10] M. Turner, J. Peiró, D. Moxey, Procedia Eng. 163 (2016) 340–352. http://dx.doi.org/10.1016/j.proeng.2016.11.069.
[11] M. Turner, J. Peiró, D. Moxey, Comput.-Aided Des. (2017). http://dx.doi.org/10.1016/j.cad.2017.10.004.
[12] 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.
[13] G.A. Hansen, P.G. Xavier, S.P. Mish, T.E. Voth, M.W. Heinstein, M.W. Glass, Eng. Comput. 32 (2) (2016) 295–311. http://dx.doi.org/10.1007/s00366-015-0418-x.
[14] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, M. Berzins, Second International Workshop on Extreme Scale Programming Models and Middleware, 2016, pp. 1–4. http://dx.doi.org/10.1109/ESPM2.2016.012.
[15] M. Turner, D. Moxey, S.J. Sherwin, J. Peiró, ECCOMAS Congress 2016 VII European Congress on Computational Methods in Applied Sciences and Engineering, 2016, pp. 5–10.
[16] 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.
[17] C. Geuzaine, Internat. J. Numer. Methods Engrg. 79 (11) (2009) 1309–1331.
[18] J. Bonet, R.D. Wood, Nonlinear Continuum Mechanics for Finite Element Analysis, second ed., Cambridge Univeristy Press, 2008.
[19] V.A. Garanzha, Numer. Linear Algebra Appl. 11 (5–6) (2004) 535–563. http://dx.doi.org/10.1002/nla.363.
[20] J. Hesthaven, T. Warburton, Nodal Discontinuous Galerkin Methods, Springer, 2008.
[21] F.D. Witherden, P.E. Vincent, Comput. Math. Appl. 69 (10) (2015) 1232–1241. http://dx.doi.org/10.1016/j.camwa.2015.03.017.
[22] Nvidia, CUDA8.0 Release Note, 2016. URL http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Release_Notes.pdf.
[23] Khronos Group, The OpenCL Specification, 2015. URL https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf.
[24] OpenMP Architecture Review Board, OpenMP 4.5 Specifications, 2015. URL http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.
[25] OpenACC Organization, OpenACC Programming and Best Practices Guide, 2015. URL http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf.
[26] Khronos Group, SYCL Provisional Specification, 2016. URL https://www.khronos.org/registry/sycl/specs/sycl-2.2.pdf.
[27] R. Hornung, H. Jones, J. Keasler, R. Neely, A. Kunen, O. Pearce, RAJA Overview, Tech. Rep., Lawrence Livermore National Laboratory, 2015.
[28] The Open Group, POSIX threads, 1997. URL http://pubs.opengroup.org/onlinepubs/007908799/xsh/threads.html.
[29] MPI Forum, MPI 3.1 Specifications, 2015. URL http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.
[30] K.B. Wheeler, R.C. Murphy, D. Thain, IEEE Int. Symp. Parallel Distrib. Process. (2008) 1–8. http://dx.doi.org/10.1109/IPDPS.2008.4536359.
[31] Nvidia, CUDA 8.0 Toolkit Documentation, 2016. URL http://docs.nvidia.com/cuda/#.
[32] B.C. Vermeire, F.D. Witherden, P.E. Vincent, J. Comput. Phys. 334 (2017) 497–521. http://dx.doi.org/10.1016/j.jcp.2016.12.049.