Available online at www.sciencedirect.com

ScienceDirect

Comput. Methods Appl. Mech. Engrg. 310 (2016) 628–645

Computer methods in applied mechanics and engineering

www.elsevier.com/locate/cma

Optimising the performance of the spectral/*hp* element method with collective linear algebra operations

D. Moxey[a,*], C.D. Cantwell[a], R.M. Kirby[b], S.J. Sherwin[a]

[a] *Department of Aeronautics, South Kensington Campus, Imperial College London, SW7 2AZ, UK*
[b] *School of Computing, University of Utah, Salt Lake City, UT 84112, United States*

Received 24 December 2015; received in revised form 18 April 2016; accepted 1 July 2016
Available online 29 July 2016

Abstract

As computing hardware evolves, increasing core counts mean that memory bandwidth is becoming the deciding factor in attaining peak performance of numerical methods. High-order finite element methods, such as those implemented in the spectral/*hp* framework *Nektar*++, are particularly well-suited to this environment. Unlike low-order methods that typically utilise sparse storage, matrices representing high-order operators have greater density and richer structure. In this paper, we show how these qualities can be exploited to increase runtime performance on nodes that comprise a typical high-performance computing system, by amalgamating the action of key operators on multiple elements into a single, memory-efficient block. We investigate different strategies for achieving optimal performance across a range of polynomial orders and element types. As these strategies all depend on external factors such as BLAS implementation and the geometry of interest, we present a technique for automatically selecting the most efficient strategy at runtime.

Keywords: Spectral/*hp* element method; High-order finite elements; Linear algebra optimisation

1. Introduction

The spectral/*hp* element method is now a well-established high-order finite element approach that is gaining popularity within both academia and industry. Traditionally, this method has been confined to the field of computational fluid dynamics [1], with recent advances demonstrating the applicability of the method to increasingly challenging problems within the aeronautics and automotive communities [2]. This has now lead to increased interest in the spectral/*hp* method across a broad range of application areas, including cardiac electrophysiology [3], solid mechanics [4,5], porous media [6] and oceanographic modelling [7]. This can be attributed to the attractive features of the method: it combines the convenient geometric flexibility found in linear finite element methods with spectral-like exponential convergence in the polynomial order p. Software such as *Nektar*++ [8], an open-source framework

[*] Corresponding author.
E-mail address: d.moxey@imperial.ac.uk (D. Moxey).

http://dx.doi.org/10.1016/j.cma.2016.07.001

designed to provide an easy-to-use, efficient and modern environment for the development of high-order finite element methods, has also helped to abate the level of difficulty in using these methods and exposed them to a wider audience.

From a computational perspective, recent evidence suggests that high-order methods are well-placed to take advantage of modern multi-core [9] and many-core [10,11] computing hardware. For large-scale simulations, which are becoming increasingly commonplace in scientific studies, nodes containing many-core processors form the backbone of current-day high-performance computing (HPC) infrastructure. The use of spectral/*hp* element methods is now routine on between $O(10^4)$ and $O(10^5)$ cores in tackling challenging aeronautical flow simulations on large unstructured grids [2], with other codes such as Nek5000 scaling to $O(10^6)$ cores through the use of tensor-product hexahedra [12]. Techniques for optimising the performance of these methods on a single compute node are therefore highly important in the context of reducing execution times.

A significant proportion of the runtime, both in the spectral/*hp* element method and in lower-order methods, is usually spent in the evaluation of a discretised mathematical operator across the mesh of the domain, which takes the form of a matrix–vector multiplication. In the classic linear finite element method, this matrix is usually extremely sparse, leading to the use of an appropriate sparse-matrix format. On the other hand, matrices that represent discretised high-order operators on a single element possess far more structure, particularly at higher polynomial orders. This leads to a wider range of implementation strategies [13–15], in which the elemental matrices need not be assembled to construct the global matrix.

In the context of modern hardware, this distinction of approaches is very important. On a typical node, memory bandwidth is rapidly becoming the deciding factor in attaining peak CPU performance. With core counts increasing and memory hierarchies becoming more complex, data must be managed efficiently, through the use of caching-friendly algorithms, to attain the peak performance of the hardware. Methods that minimise indirection and cache misses can take advantage of this environment, leading to increased efficiency and shorter runtimes. High-order methods are therefore well-suited to this new landscape since they can utilise dense operations that are typically far more memory-compact than sparse operations. Whilst some efforts have been made to develop auto-tuning and automatic compilation to maximise floating point operation counts [16] on different hardware types, thus far, an investigation into the effects of compacting data and minimising data transfer has not been thoroughly investigated in the context of high-order methods.

The aim of this paper is therefore to bridge this gap in current understanding, by utilising the mathematical formulation of the spectral/*hp* element method in order to improve memory efficiency. We present a methodology based on the amalgamation of the action of multiple elemental operators into larger matrix–vector and matrix–matrix multiplications. In particular, we show how the mathematical construction of the method admits several different amalgamation schemes, some (but not all) of which are similar to those described in [13–15], but formulated in a multi-element setting. We demonstrate how the use of these schemes can significantly increase the performance of the method, thereby decreasing the runtime of solvers that are based on this technology. We consider the effects of these schemes on distributed memory hardware architectures, focusing on the performance effects on a single node vs. communication costs between nodes which remains the same regardless of the scheme chosen. The choice of the most optimal scheme is highly dependent not only on the polynomial order, but many other factors such as the underlying hardware, choice of BLAS implementation and the problem to be considered. We therefore present an auto-tuning method designed to rapidly and automatically select the most efficient scheme at runtime on a per-operator basis, overcoming a significant barrier to the usage of these techniques for users. Overall, we show that whilst the effect of amalgamating these operators inevitably leads to a limited increase in the memory footprint in certain cases, significant performance gains can be achieved due to the increase in data transfer efficiency.

1.1. Existing work, novelty and applicability

In the context of finite elements, the idea of grouping together elements of identical structure in order to act on them 'collectively' has been examined previously under the nomenclature of *'worksets'*, as seen for example in [17] and [18]. With the advent of GPU-based methods, similar techniques have been adopted to align with the kernel-based nature of the resulting implementations [10]. However, we emphasise that the purpose and novelty of this paper is not therefore in the presentation of the amalgamation approach, but rather in how the mathematical formulation of the spectral/*hp* element method admits schemes of different implementation (but the same mathematical operation) under the action of amalgamation. We will demonstrate that, across a range of polynomial orders and element types, the performance of the method can differ drastically. Depending on the amalgamation scheme chosen, this difference can be as much as an order of magnitude in terms of the evaluation time. This work therefore notably extends the body of knowledge relating to multi-element operations and their implementation in the high-order context, and is relevant to a number of widely used high-order variants, including element-based continuous Galerkin, discontinuous Galerkin and flux reconstruction. However we do not encompass the global operators which were the focus of previous work.

It is also important to note that the underlying concept of amalgamation of elemental operators has performance benefits for the finite element formulation, be that in the context of either low- or high-order methods. Since the operators we consider lie at the very heart of the finite element formulation, such as taking derivatives and inner products, the methodology we present is applicable to a wide variety of finite element software packages. In terms of high-order methods, which are becoming increasingly prevalent throughout the scientific community, amalgamation schemes are certainly not limited to *Nektar*++. Many other high-order codes are based around the same types of operator evaluations, and so can benefit from the results we present here. However, depending on the precise nature of the formulation being used, not all of the schemes we present may be applicable. We discuss this further in the conclusions section.

1.2. Summary

The paper is structured as follows. In the next section, we give a brief overview of the mathematical formulation of the spectral/*hp* element method and describe the amalgamation schemes. In Section 3, we present initial results demonstrating how these collections can be used to improve performance when applied to a complex three-dimensional mesh. We additionally show the effects of some parameter changes, such as the BLAS implementation, highlighting the need for the auto-tuning strategy that is presented in Section 4. We investigate further detailed hardware metrics such as FLOPS and memory bandwidth attained by the schemes in Section 5. Finally, in Section 6 we apply all of these techniques to a real-world large-scale simulation of compressible flow over an ONERA M6 wing, demonstrating the practical reduction in runtime that can be achieved in using the amalgamation techniques. We give some brief conclusions in Section 7.

2. Collective operations

In this section we outline the operations that will be amalgamated and highlight their role in the underlying method. The starting point for this is a brief overview of the formulation of the spectral/*hp* element method, concentrating on the discretisation of a single element in three dimensions. A more detailed derivation can be found in [1]. We then outline our strategies for amalgamation.

2.1. Spectral/hp formulation

As with any other finite element method, the spectral/*hp* element method begins with the tessellation of a computational domain into a mesh Ω consisting of N non-overlapping elements Ω^e such that $\Omega = \bigcup_{e=1}^{N} \Omega^e$. The first step of the formulation is to consider the expansion of a function u on a standard element Ω_{st}, on which the basis functions and basic operations such as integration and differentiation are defined. In three dimensions we consider either hexahedral, prismatic or tetrahedral elements whose standard regions are defined as

$$\begin{aligned}
\Omega_{\mathrm{st}}^{\mathrm{hex}} &= \{\vec{\xi} \mid -1 \leq \xi_1, \xi_2, \xi_3 \leq 1\},\\
\Omega_{\mathrm{st}}^{\mathrm{pri}} &= \{\vec{\xi} \mid \xi_1, \xi_2, \xi_3 \geq -1; \xi_1 + \xi_3 \leq 0; \xi_2 \leq 1\},\\
\Omega_{\mathrm{st}}^{\mathrm{tet}} &= \{\vec{\xi} \mid \xi_1, \xi_2, \xi_3 \geq -1; \xi_1 + \xi_2 + \xi_3 \leq -1\}.
\end{aligned}$$

A scalar function u is represented on a standard region through a summation of polynomial basis functions ϕ_i as

$$u(\vec{\xi}) = \sum_{i \in \mathcal{I}} \hat{u}_i \phi_i(\vec{\xi}), \tag{1}$$

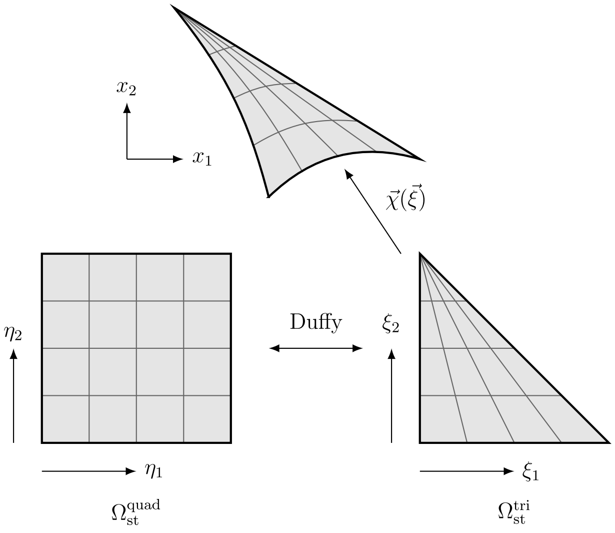

Fig. 1. Duffy transformation from the standard quadrilateral element $\Omega_{\rm st}^{\rm quad}$ with collapsed coordinates (η_1, η_2) to triangular element $\Omega_{\rm st}^{\rm tri}$ with coordinates (ξ_1, ξ_2). We also show how $\Omega_{\rm st}^{\rm tri}$ is taken to a general curvilinear element through the isoparametric mapping $\vec{\chi}$. Interior lines demonstrate how quadrature points, taken to be equally spaced in this example, are transformed under each mapping.

where $\vec{\xi} = (\xi_1, \xi_2, \xi_3) \in \Omega_{\rm st}$ and $\mathcal{I}$ is an indexing set that depends on element type and possibly heterogeneous polynomial orders P_i in each coordinate direction,

$$\begin{aligned}
\mathcal{I}^{\rm hex} &= \{(pqr) \mid 0 \le p \le P_1,\ 0 \le q \le P_2,\ 0 \le r \le P_3\},\\
\mathcal{I}^{\rm pri} &= \{(pqr) \mid 0 \le p \le P_1,\ 0 \le q \le P_2,\ 0 \le p + r \le P_3, P_1 \le P_3\},\\
\mathcal{I}^{\rm tet} &= \{(pqr) \mid 0 \le p \le P_1,\ 0 \le p + q \le P_2,\ 0 \le p + q + r \le P_3,\ P_1 \le P_2 \le P_3\}.
\end{aligned}$$

Each element therefore has a number of elemental degrees of freedom $N_{\rm dof} = |\mathcal{I}|$. The key part of the spectral/*hp* element method is that, unlike other high-order finite element methods, the basis on each element is expressed as a tensorial expansion of one-dimensional functions, even for elements such as the prism and tetrahedron. In the hexahedral element this is a straightforward tensor product, leading to a decomposition of the form

$$\phi_{pqr}(\vec{\xi}) = \psi_p(\xi_1)\psi_q(\xi_2)\psi_r(\xi_3).$$

A function on the standard hexahedron is therefore expressed through the summation

$$u(\vec{\xi}) = \sum_{p=0}^{P_1}\sum_{q=0}^{P_2}\sum_{r=0}^{P_3} \hat{u}_{pqr}\psi_p(\xi_1)\psi_q(\xi_2)\psi_r(\xi_3). \tag{2}$$

In tetrahedral and prismatic elements, we use a collapsed coordinate system $(\eta_1, \eta_2, \eta_3) \in [-1, 1]^3$, obtained through one or more applications of the square-to-triangle Duffy transformation [19], to obtain a similar tensorial decomposition. Fig. 1 demonstrates the Duffy transformation in a two-dimensional setting through the mapping

$$\eta_1 = 2\frac{1+\xi_1}{1-\xi_2} - 1, \qquad \eta_2 = \xi_2.$$

Consequently, the summation over each mode i as a function of p, q and r, then becomes more complex, as indicated by the indexing sets above. For example, a prismatic element has the general expansion

$$u(\vec{\xi}) = \sum_{p=0}^{P_1}\sum_{q=0}^{P_2}\sum_{r=0}^{P_3-p} \hat{u}_{pqr}\psi_p^a(\eta_1)\psi_q^a(\eta_2)\psi_{pr}^b(\eta_3), \tag{3}$$

where $\eta_i \in [-1, 1]$ is the collapsed coordinate and, additionally, ψ^b depends on both p and r. We note that although the number of modes has been reduced, we still maintain a complete polynomial space on $\Omega_{\rm st}$. The basis functions ψ^a

and ψ^b are the tensor-product modified hierarchical basis functions defined in [1], which combine a set of orthogonal Jacobi polynomials with linear finite element basis functions to achieve a separation of boundary and interior modes.

We must additionally discuss how a given PDE will be discretised. For the problems we consider here, the continuous or discontinuous Galerkin methods will be used, where the PDE is transformed into a weak form involving the L^2 inner product of functions lying in polynomial test and trial spaces. We therefore equip each element type with a distribution of N_Q quadrature points $(\xi_{1j}, \xi_{2j}, \xi_{3j})$ and weights w_j to numerically calculate the resulting integrals. In the hexahedron, the tensor-product expansion means our choice of quadrature is straightforward: we select Q_i Gauss–Lobatto–Legendre points, with $Q_i = P_i + 2$ being large enough to exactly evaluate the integral of polynomial products in the ith coordinate direction for straight-sided elements. Owing to the use of the Duffy transformation in the other elements, a singularity arises at the collapsed vertex of triangles. We therefore select $Q_i = P_i + 1$ Gauss–Radau points in these directions to avoid this issue. However, we note that for *all* element types, a tensor product of quadrature points is used so that $N_Q = Q_1 Q_2 Q_3$, as can be seen in Fig. 1.

Finally, the standard region must be mapped to a general element $\Omega^e \subset \Omega$. The reference element co-ordinates $\vec{\xi} \in \Omega_{\text{st}}$ are mapped to the Cartesian coordinates $\vec{x} = (x_1, x_2, x_3) \in \Omega^e$ through a bijection $\vec{\chi} = (\chi_1, \chi_2, \chi_3)$ with $x_i = \chi_i^e(\vec{\xi})$. Each component χ_i is isoparametric, so that it is written as an expansion (1) in terms of the modified basis functions ψ. We may then determine geometric factors such as the Jacobian $J_\xi \chi$ and its determinant, which are used to calculate, for example, integrals and derivatives on Ω^e through an application of the chain rule. In general, elements may be straight-sided, in which case $J_\xi \chi$ is constant, or curvilinear so that $J_\xi \chi$ is a function of ξ. Fig. 1 demonstrates a mapping from a Ω_{st} into a curvilinear element.

2.2. *Operator evaluation strategies*

In previous work in two [13] and three dimensions [14,15], various evaluation strategies were devised for key discretised mathematical operators. To illustrate this, we examine the most straightforward operation: the *backward transformation* as shown in Eq. (2), which recovers the physical solution u given the coefficients $\hat{u}_{pqr}$. In the case of a nodal basis, where ψ_p is equipped with a set of (possibly non-tensorial) quadrature points $\{\vec{\zeta}_q\}$ and obeys the Lagrange interpolant property that $\psi_p(\vec{\zeta}_q) = \delta_{pq}$, this is equivalent to an interpolation operator. For simplicity we consider the hexahedral element type.

A naive implementation of this triple summation at each of the quadrature points leads to an evaluation involving $O(P^6)$ floating point operations, where we assume that P_i and Q_i are all of the same order P. In the *local* evaluation strategy, we reformulate the summation as a matrix–vector product $\mathbf{B}\hat{\mathbf{u}}$, where $\mathbf{B} = [\psi_j(\xi_i)]_{ij}$ is a $N_Q \times N_{\text{dof}}$ matrix that stores the evaluation of the basis functions at each quadrature point and $\hat{\mathbf{u}}$ is a vector of the coefficients $\hat{u}_{pqr}$, ordered so that p runs fastest, followed by q and then r. This permits the use of an efficient linear algebra library to evaluate the matrix–vector product, potentially increasing performance over a naive implementation, as these packages tend to be highly optimised over a range of computational hardware.

An alternative approach is to apply the technique of *sum-factorisation* [20]. In this evaluation strategy, the tensor-product expansion of the basis functions ϕ_i is leveraged to decrease the number of operations and thus the order of complexity of the evaluation. In the hexahedron, this is achieved by rewriting (2) as

$$u(\vec{\xi}) = \sum_{p=0}^{P_1} \psi_p(\xi_1) \left[\sum_{q=0}^{P_2} \psi_q(\xi_2) \left[\sum_{r=0}^{P_3} \hat{u}_{pqr} \psi_r(\xi_3) \right] \right].$$

We see that if the bracketed summations are stored in memory, the application of sum-factorisation leads to a reduction to $O(P^4)$ floating point operations. A similar technique can be used for the tetrahedron and prism, although it is typically less efficient than the hexahedron due to the inter-dependency of the p, q and r indices. With a little more work, we can again use linear algebra packages by rewriting the summation as a series of matrix–matrix operations,

$$\left((\hat{\mathbf{U}}_{[P_1]}^\top \mathbf{B}_1^\top)_{[Q_2]}^\top \mathbf{B}_2^\top \right)_{[Q_3]}^\top \mathbf{B}_3^\top, \tag{4}$$

where $\hat{\mathbf{U}}_{[P_1]}$, for example, denotes the reinterpretation of the vector $\hat{\mathbf{u}}$ as a $P_1 \times P_2 P_3$ matrix stored in column-major format. The parentheses also highlight where temporary storage is required to store intermediate steps. Whilst intuition may point towards sum-factorisation being the quickest way to evaluate these operators due to the reduction in operator count, our previous work demonstrates that the fastest technique depends heavily on polynomial order, element type and the type of operator under consideration. This points towards there being the need for a number of different amalgamation schemes in order to attain optimal performance.

2.3. Amalgamation schemes

Our earlier studies applied the strategies of the previous section by iterating over each element, evaluating the operator and measuring the total execution time for the entire mesh. However, in the context of memory efficiency and using the CPU cache effectively, this approach may not prove to be the most optimal if matrices are not stored contiguously in memory. Additionally, since local matrices must be generated for each element, there is a large cost incurred in moving them from main memory to the processor.

The purpose of this work is therefore to reformulate these strategies in the context of multiple elements. We aim to remove local matrices wherever possible, thereby reducing data movement and increasing data locality. We will leverage both the tensor-product decomposition of the spectral/hp element method and the use of a standard region, on which we can define an operator for many elements simultaneously. Then, through grouping local elemental storage of the coefficient and physical spaces, we aim to apply standard level-3 BLAS operations such as `dgemm` for matrix multiplication wherever possible. These routines, owing to their widespread use, are highly optimised. Indeed, many HPC facilities and vendors supply custom implementations with processor-specific code that can achieve a significant percentage of peak floating-point performance.

The first step is to consider the mathematical principles behind the amalgamation procedure and decide precisely how to group elemental operations together. The defining characteristics of the action of an operator on an element Ω^e are the element type, the triple (P_1, P_2, P_3) defining its polynomial order and the geometric mapping χ^e with its derivatives and Jacobian determinant J^e. These properties should then dictate the grouping of elements within a given mesh.

However, for the purposes of amalgamating elements, we note that the geometric information can be ignored. Since each element is mapped to a standard element Ω_{st}, we may choose to perform our computationally expensive operations on Ω_{st} and then apply the geometric information as a secondary step. To demonstrate how this works, we must investigate an alternative to the backwards transformation (1), which does not use the geometric terms. We therefore consider another fundamental operation: calculating the L^2 inner product of a function u with respect to the basis functions $\phi_i^e(\vec{x}) = \phi_i(\chi^e(\vec{\xi}))$ on each element Ω^e. On a single element, this is defined as

$$A_i = \int_{\Omega^e} u\phi_i^e \, d\vec{x} = \int_{\Omega_{\text{st}}} u\phi_i \, J^e d\vec{\xi} \tag{5}$$

yielding a vector of solutions $\mathbf{A} = [A_i]_{i=1}^{N_{\text{dof}}}$. If we sample $u^e = u|_{\Omega^e}$ at each quadrature point, creating a vector $\mathbf{u}^e = [u^e(\xi_i)]_{i=1}^{N_Q}$, and do the same for J^e, we may construct a vector $\mathbf{w}^e(\mathbf{u}^e) = [J^e(\xi_i)w_i u^e(\xi_i)]_{i=1}^{N_Q}$, where w_i are the quadrature weights on the standard element. Then (5) can be expressed as the matrix–vector product $\mathbf{A} = \mathbf{B}^\top \mathbf{w}^e$. We may therefore naturally extend this into a multi-element setting by combining the geometric terms and quadrature weights into a single vector $\mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_\text{N}]$ that spans a group of elements $1 \leq e \leq N$.

Our strategy for amalgamation therefore begins by grouping elements which are of the same type and have the same polynomial order. For N_{el} elements in any given group, the elemental coefficients $\hat{\mathbf{u}}^e$ and physical solution at the quadrature points $\mathbf{u}_j^e = u(\xi_{1j}, \xi_{2j}, \xi_{3j})|_{\Omega^e}$ are stored in a contiguous vector, which we denote by $\hat{\mathbf{U}} = [\hat{\mathbf{u}}^1, \dots, \hat{\mathbf{u}}^{N_{\text{el}}}]$ and $\mathbf{U} = [\mathbf{u}^1, \dots, \mathbf{u}^{N_{\text{el}}}]$ respectively. We then consider four different schemes, which are depicted graphically in Fig. 2.

2.3.1. Local sum factorisation

LocalSumFac represents the baseline existing implementation that we will compare against. We iterate over each expansion, applying the sum-factorised version of a given operator. All operations are considered locally and without any amalgamation. Therefore the matrix representation of an operator, as well as the Jacobian determinant J^e and related geometric factors, are not guaranteed to be contiguous in memory, depending on the implementation choices that have been adopted within the code. However, the use of a per-element geometric information allows us to benefit from a memory improvement, since planar-sided elements have a constant Jacobian determinant.

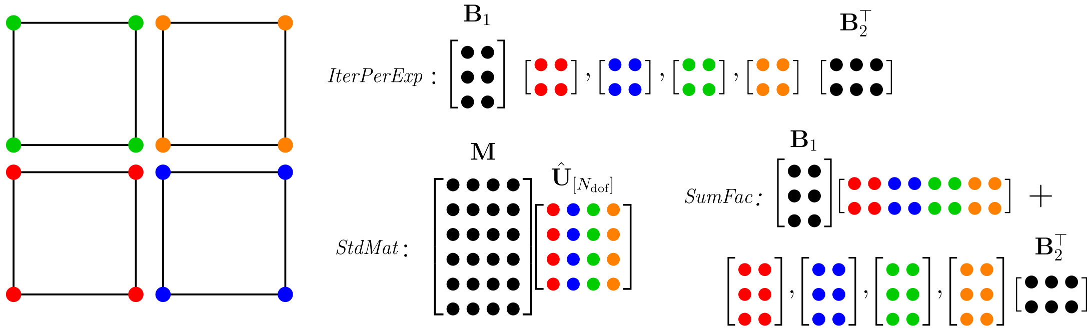

Fig. 2. Diagrammatic representation of each amalgamation scheme. Four quadrilateral elements with $P_1 = P_2 = 1$ and $Q_1 = Q_2 = 3$ are considered for the backward transform operator.

2.3.2. Iteration over elements with sum-factorisation

In the *IterPerExp* scheme, we use the local sum-factorisation technique of Eq. (4) in a similar fashion to *LocalSumFac*. However we now combine this with the geometry amalgamation technique outlined above, so that the geometric information is stored contiguously in memory. The standard basis matrices $\mathbf{B}_i$ are shared between each element; however no particular attention is paid to keep them contiguous in memory. In regard to the geometric terms, we note that on planar-sided elements, J^e is usually constant as a function of ξ. However, to ensure memory is streamed efficiently and to avoid unnecessary branching, we store the J^e terms as a contiguous vector of size $N_\mathrm{el} \times N_Q$.

2.3.3. Standard matrix approach

The aim of the *StdMat* scheme is to utilise the standard element, by first forming a matrix $\mathbf{M}$ on Ω_st that represents our operator. Considering, for example, the inner product, $\mathbf{M}$ has dimensions $N_Q \times N_\mathrm{dof}$. To calculate the inner product, we first perform a pointwise multiplication (i.e. Hadamard product) of the function u at the quadrature points, $\mathbf{U}$ with the vector $\mathbf{W}$ that contains the combined Jacobian and quadrature weights, to obtain a vector $\mathbf{V}$. We then calculate the action of $\mathbf{M}$ on multiple elements through a matrix–matrix multiplication

$$\hat{\mathbf{U}}_{[N_\mathrm{dof}]} = \mathbf{M}\mathbf{V}_{[N_Q]},$$

where, for example, $\hat{\mathbf{U}}_{[N_\mathrm{dof}]}$ is the interpretation of $\hat{\mathbf{U}}$ as an $N_\mathrm{dof} \times N_\mathrm{el}$ matrix stored in column-major format, so that each column is the vector of elemental coefficients $\hat{\mathbf{u}}^e$. $\mathbf{V}_{N_Q}$ is interpreted similarly as an $N_Q \times N_{el}$ matrix. The resulting $N_\mathrm{dof} \times N_\mathrm{el}$ matrix $\hat{\mathbf{U}}_{[N_\mathrm{dof}]}$ has each column being the resulting elemental inner product with respect to the basis function that has this column as its index. In this case, we can evaluate the inner product with a single call to the BLAS routine `dgemm` (matrix–matrix multiplication), taking full advantage of the optimised BLAS functionality. We note that one advantage of this method is that its implementation is precisely the same for each element type, making the scheme straightforward to implement. We also eliminate the need for local elemental matrices. However, we do not explicitly use the tensor-product composition property of the elements. This means that, in comparison with approaches that use sum-factorisation, the effective operation count is larger and thus the *StdMat* scheme may prove more expensive.

2.3.4. Collective sum-factorisation

In the *SumFac* scheme, we extend the sum-factorisation technique of Eq. (4) to multiple elements. As in the previous scheme, the goal is to extend the formulation of Eq. (4) through the use of `dgemm`. However, due to the way that elemental data is laid out in memory, this is not always possible without memory-intensive transposition of the data. We therefore opt to iterate over the elements for certain substeps. For example, the backward transformation on the hexahedron is calculated in four steps:

- preallocate a vector $\mathbf{U}_1$ of size $P_1 P_2 Q_3$;
- for each element, calculate $\mathbf{u}_1^e = \mathbf{B}_3(\hat{\mathbf{u}}^e_{[P_3]})^\top$, placing the result in the correct position inside $\mathbf{U}_1$, which amalgamates the contributions from each element;

- calculate $\mathbf{U}_2 = \mathbf{B}_2(\mathbf{U}_1)^{\top}_{[P_2]}$; and
- finally, compute $\mathbf{U} = \mathbf{B}_1(\mathbf{U}_2)^{\top}_{[P_1]}$.

Whilst the iteration over elements in the first step cannot be avoided, since the $\mathbf{B}_i$ matrices are re-used, data locality can be ensured, giving the best possible chance for caching to be used effectively. Similar formulations can be made for all of the operators and element types under consideration.

2.4. Implementation of amalgamations

To evaluate the action of an operator on the entire mesh, upon setup we ensure that the memory for each collection is arranged contiguously. Then, when an operator evaluation is required as part of a solver, we iterate over each collection group and apply the selected amalgamation scheme. The implementation inside *Nektar++* allows for the selection of different schemes for different collection groups. As we shall see in the following section, this is important since each group may have vastly different performance metrics. As well as the backward transform and inner product operators already introduced, we will consider a further two key operators:

- the L^2 inner product with respect to the derivatives in each direction of the basis functions: this is used to construct elemental Laplacian (stiffness) matrices and in other discretisations such as the discontinuous Galerkin method;
- the derivative $\partial u/\partial x_i$ of a function u in each of the coordinate directions x_i, calculated at each of the quadrature points.

Together, these four operators form the building blocks of a large variety of solvers, one of which we shall investigate in Section 6. However, in the following section, we present results from timing each of schemes when applied to a three-dimensional example, demonstrating how amalgamation can be used to improve runtime performance.

3. Results

In this section, we apply the amalgamation schemes described previously to an example mesh containing tetrahedral and prismatic elements. We demonstrate how each scheme exhibits different performance characteristics, which, when combined appropriately lead to an overall improvement in execution times across a range of polynomial orders. We examine whether the number of elements in an amalgamation group significantly impacts on the observed performance. Finally, we highlight how the choice of BLAS implementation can affect the observed performance, which motivates the introduction of an auto-tuning strategy in the following section.

3.1. General performance characteristics

In our first series of tests, we aim to determine a general picture of the performance characteristics of each amalgamation scheme. We therefore examine a mesh of a rabbit aorta section, which contains two sets of intercostal artery pairs, as shown in Fig. 3. This is a representative example of a typical biological flow configuration. As the inset shows, the mesh comprises a boundary layer of curvilinear prismatic elements, with tetrahedral elements being used to fill the interior of the volume. The mesh contains 21,564 prismatic elements and 40,877 tetrahedral elements in total.

Our initial tests are performed across a range of polynomial orders between $1 \leq P \leq 8$, with two collection groups being created: one for the prismatic elements and another for the tetrahedral elements respectively. For each group, we record the average execution time, measured across a number of samples, for the evaluation of each operator and amalgamation scheme outlined in the previous section.

The test hardware used is a single core of a 2.7 GHz Intel Xeon E5-2697v2 CPU with 30 MB of L3 cache and 192 GB of RAM. A standard Debian Linux installation is used, containing gcc v4.7.2 with default `-O3` optimisation flags and, for our initial tests, the standard Netlib BLAS implementation [21]. The system was kept idle at the time of testing to the best of our ability to limit the effects of other processes on timings.

Figs. 4 and 5 show the relative timings of each amalgamation scheme, normalised by the *IterPerExp* scheme. We see that, in general, significant performance improvements can be observed, particularly at lower polynomial orders where large speedups of up to a factor of 8 are seen. Depending on the element and operator type, the *StdMat* scheme generally performs best at linear and quadratic orders. However as the polynomial order is increased and

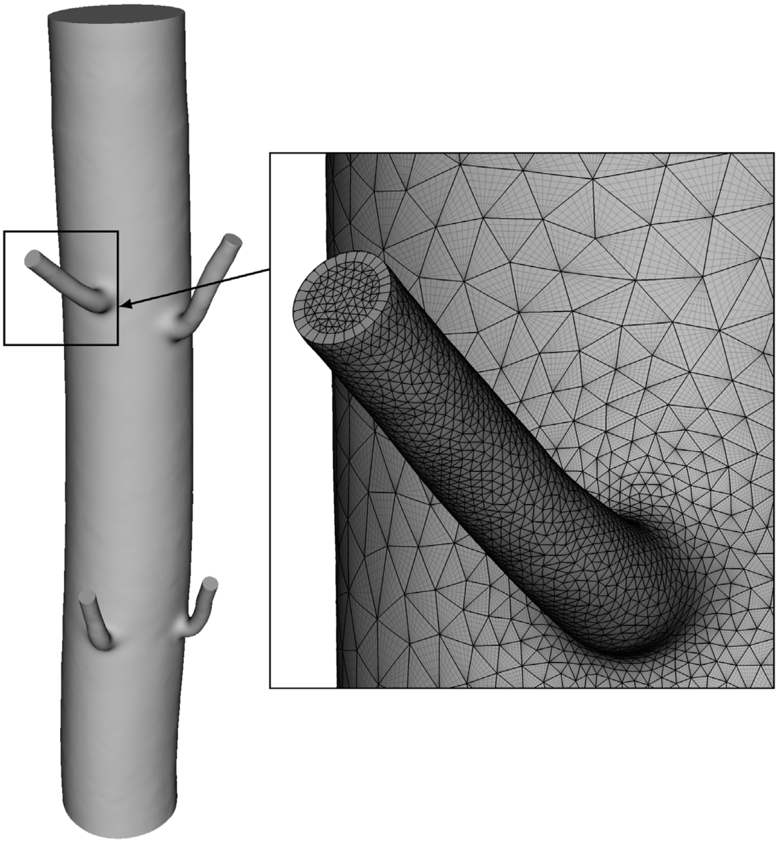

Fig. 3. Mesh used in performance study of two rabbit intercostal pairs. Inset shows the mesh detail near each branch, with thin lines indicating collapsed coordinates inside triangles.

the standard matrix becomes larger, this scheme rapidly becomes prohibitively expensive. As the polynomial order increases further, it is clear that the sum-factorised variants of either *SumFac*, *IterPerExp* or the baseline *LocalSumFac* are generally the correct scheme to select. We note that, broadly speaking, the *IterPerExp* and *LocalSumFac* methods appear to perform at similar levels. This indicates that the additional expense that is incurred by storing elemental Jacobian determinants for planar-sided elements is not substantial in comparison to the cost of the operator. Here, we posit that the local sum-factorised matrix sizes are large enough that, regardless of whether matrices are allocated contiguously in memory or not, a local approach to sum-factorisation still provides a highly-efficient implementation.

3.2. *Effect of group size*

An additional factor to consider is the number of elements in the amalgamation group. Since BLAS is known to utilise less of the peak CPU performance for smaller matrix ranks, increasing the number of elements in an amalgamation group will lead to larger matrix sizes and thus may increase performance. To this end, we repeat the previous procedure, examining the inner product operator with the *IterPerExp* scheme for tetrahedral elements, as they have the smallest local matrix sizes. The number of elements in the amalgamation is limited to powers of 10 until the entire mesh of tetrahedral elements is recovered. The measured execution time is then scaled to determine the equivalent time required to evaluate the action of the operator on the whole mesh. The results, depicted in Fig. 6, show that with some minor exceptions, the runtime is broadly equal across the range of element sizes. Other tests performed with different BLAS implementations also did not show a significant difference in performance. However, we note that this factor may be more important when considering two-dimensional elements, which will generally have smaller matrix sizes. For the purposes of the rest of this paper however, this parameter is not considered further.

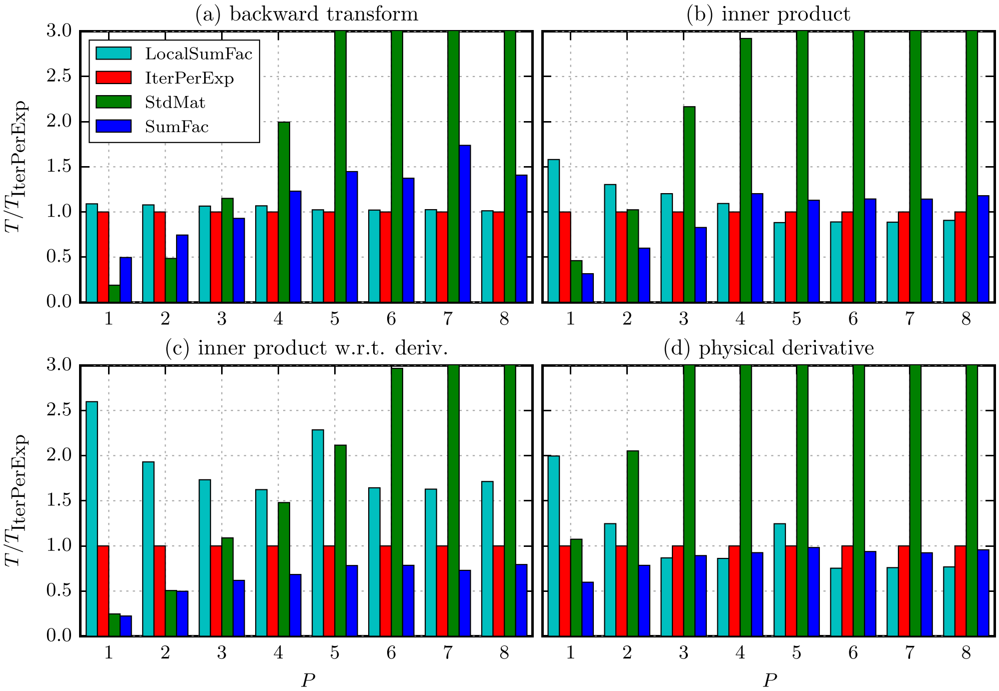

Fig. 4. Initial performance characteristics for the four key operators with prismatic elements using the reference BLAS implementation from NETLIB.

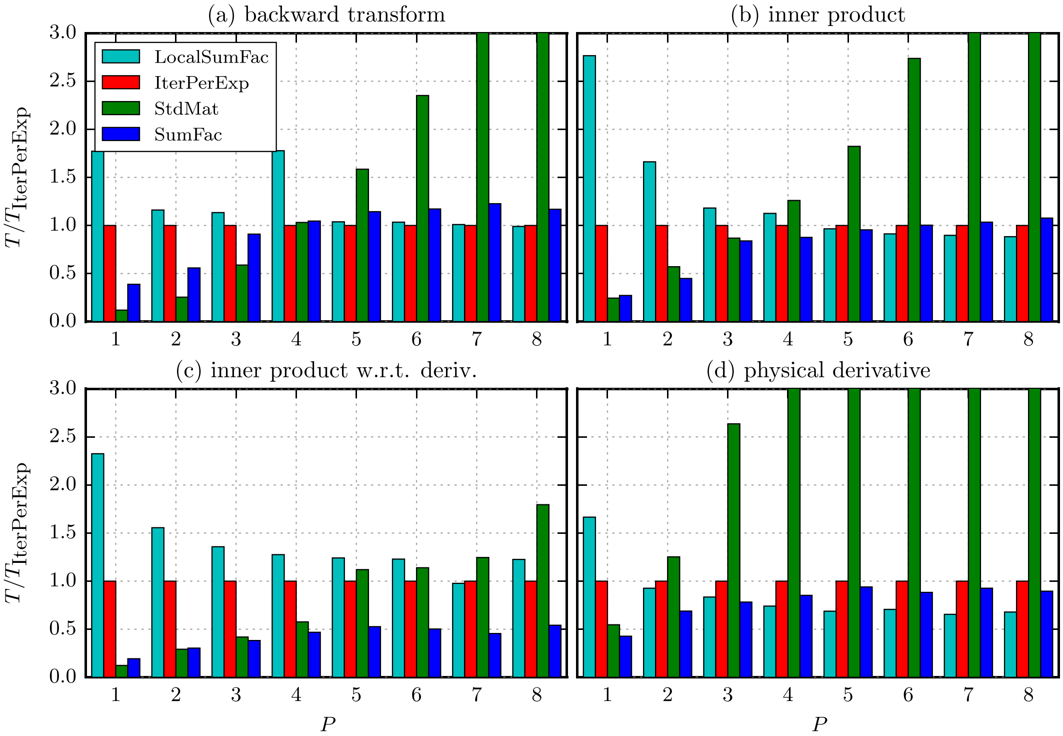

Fig. 5. Initial performance characteristics for the four key operators with tetrahedral elements using the reference BLAS implementation from NETLIB.

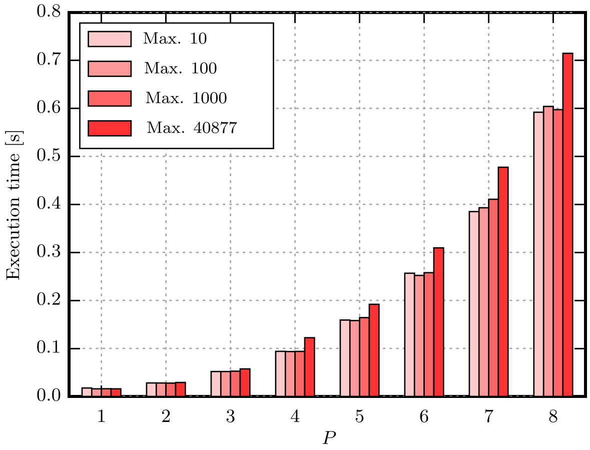

Fig. 6. Effect of amalgamation sizes on the observed evaluation time for the inner product operator on tetrahedral elements.

4. Auto-tuning strategy

Whilst from the previous section we see that the performance improvements that can be obtained with amalgamation schemes are significant, it is clear that the choice of scheme in an *a priori* fashion is highly nontrivial. As we have seen in both this and previous work, operator count calculations alone are not sufficient to give an accurate estimation of the most efficient scheme. In reality, the number of variables involved in analysing the efficiency of each scheme is highly problem- and hardware-dependent.

For example, let us consider the effects of switching from the Netlib BLAS implementation to the OpenBLAS library [22], widely regarded to be highly efficient on modern hardware. To obtain a general idea of the changes in performance characteristics, we repeat the experiment of Fig. 5 on tetrahedral elements, but this time using OpenBLAS, which is configured to only use a single thread. The results, which can be seen in Fig. 7, demonstrate remarkably different properties. For example, the *StdMat* scheme is now far more efficient across the entire range of polynomial orders, with even greater speedups observed at low polynomial orders. The *SumFac* scheme also exhibits greater performance in the mid- to high-polynomial order. This may be due to the relatively poor performance of OpenBLAS on small matrices, such as those used in the other methods we are comparing *StdMat* and *SumFac* to. However, we see that the baseline *LocalSumFac* method is now widely outperformed. Changing a single parameter in our computational setup has therefore drastically altered the amalgamation schemes that would yield an optimal simulation.

The choice of BLAS implementation is far from the only parameter that can affect performance. From the perspective of hardware, the choice of CPU, its cache hierarchy, bus speed and peak memory bandwidth are all important in each scheme's performance. Different software choices such as the choice of compiler, operating system and optimisation settings also impact runtime behaviour. Although we may be able to observe general trends in the relative performance of each scheme, important factors such as the polynomial order crossing point, whether this extends more generally to other CPU models, and the further effects of multi-core simulations, all point towards the need for a simple method to automatically select the fastest scheme at runtime for a given problem.

To mitigate these factors, we have developed a simple and effective auto-tuning strategy which can be used at runtime to automatically select the fastest amalgamation scheme. At the start of a simulation, *Nektar++* creates a list of elements that lie in the three-dimensional problem mesh, as well as lists of quadrilateral and triangular elements that represent the two-dimensional boundary conditions. These are then used to construct the collection groups, as defined in Section 2. We then iterate over each collection group and apply the following procedure:

1. Query a lookup table for existing performance metrics, based on the group's polynomial order, element type and number of elements in the group. All groups with 100 elements or more are identified to have the same metrics, based on the results of Fig. 6.
2. If no existing metrics are found, then:
   - Execute each amalgamation scheme once, recording the execution time in order to estimate the number of operator evaluations that can be evaluated in one second.
   - Run this number of tests against random input data.

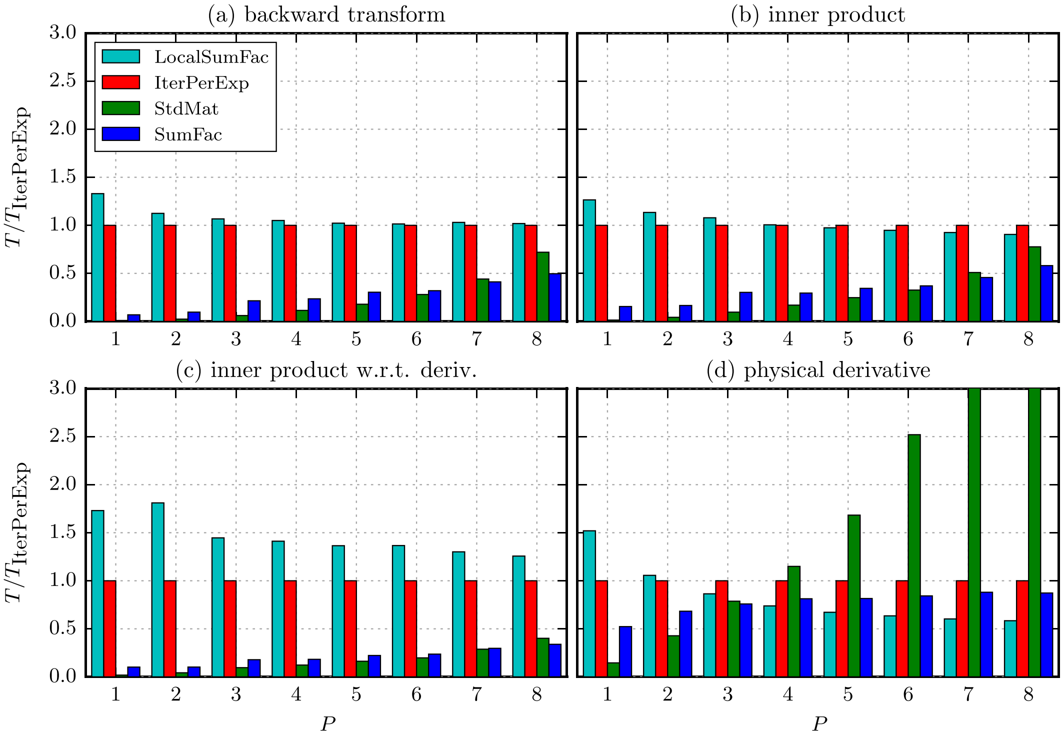

Fig. 7. Performance characteristics for the four key operators with prismatic elements under OpenBLAS.

- The scheme to select is the one with the fastest average time.
- Record this metric in the lookup table.

3. Otherwise, use the obtained metric to set the correct scheme.
4. Go to the next group and apply step 1 until all groups are processed.

The use of a lookup table, as well as an equivalence relation between groups having greater than 100 elements, means that in a typical simulation, this auto-tuning method will require less than one minute to run. This is true even for large problems that run in parallel on a HPC cluster. In the worst case, the mesh will be sufficiently large that the initial operator evaluation will be greater than one second, meaning that no averaging procedure will be performed.

We additionally note that occasionally, operating system kernel task management will abruptly skew observed execution times, and so a lack of averaging could lead to an incorrect indication of the optimal performance metric. However, we posit that this is unlikely in all but the most extreme cases. For scheme selection to be significantly influenced by kernel operations requires a large kernel interruption of the order of one second. In general, we regard such cases as 'extreme' as we would expect our hardware to be HPC-based: that is, composed of empty, dedicated compute nodes with very few processes running at time of execution. As a representative example, in Fig. 6, the recorded runtime for each amalgamation scheme is less than 0.7 s for a mesh of 40,000 elements. However, for a realistic flow simulation, in order to obtain results in a timely fashion we may use around $O(10^2)$ processors depending on the polynomial order. In this case then, assuming optimal scaling of this communication-free routine, the execution time for each scheme will be less than 0.007 s, allowing for 140 operator executions to be measured.

5. Hardware performance analysis

In order to better understand the performance observations of the different amalgamation schemes with respect to the underlying low-level hardware, we give a brief overview of metrics obtained through examination of CPU hardware counters, which give a more detailed picture of the operations being performed on the CPU. To reduce the parameter space, we now consider 1000 prismatic elements under the actions of the physical derivative operator, which admits a simple tensor product structure, and the inner product operator, in which a more complex procedure must be adopted to account for the dependency in the tensor product indices. These tests are performed using OpenBLAS and a single core of an Intel i7-5960K 6-core Haswell-architecture CPU, on a machine with 32 GB RAM and the Ubuntu 15.10 Linux distribution and kernel version 4.1.12, using standard `-O3` compiler optimisations with the `gcc 5.2.1` C++ compiler.

In addition to the runtime, we examine some of the key system characteristics, including the memory bandwidth attained in streaming data from main memory and the cache hit ratio for the highest-level L3 cache. These metrics are obtained using the Intel Performance Hardware Counter library at appropriate points within the test program, which is executed in the same manner as the previous sections. We additionally calculate the number of floating point operations per second (FLOPS) achieved for the *SumFac* and *StdMat* schemes by examining the matrix–matrix multiplications in each scheme and taking the standard approach of estimating the floating-point operation count of a $M \times K$ and $K \times N$ matrix as $2MNK$ [23]. The *IterPerExp* and *LocalSumFac* schemes are omitted as they share nearly identical operation counts as *SumFac*.

Fig. 8 shows the results of this study. We see that this architecture again yields different set of runtime performance characteristics, which further emphasises the need for autotuning as part of the practical use of these schemes in real-world applications. For the most part, the fastest schemes for the inner product operator follow a somewhat similar trend, with *StdMat* being fastest at lower polynomial orders and *SumFac* fastest towards higher polynomial orders. We see that the three main factors in this performance are a combination of the operator count of the algorithm, the memory bandwidth attained and the FLOPS achieved. At lower polynomial orders, *StdMat* and *SumFac* share reasonably similar order of complexity, but *StdMat* is able to achieve a higher FLOPS count, peaking at around 20 GFLOPS, which represents around 30% of the peak performance of one core of this processor. As the polynomial order increases, even though *StdMat* achieves consistently higher FLOPS counts, the lower operator count of *SumFac* and higher memory bandwidth become more important in the smaller-sized matrices that *SumFac* uses, leading to reduced runtimes. We also observe that the *IterPerExp* scheme, which is designed to optimise memory throughput, shares a similar improvement in memory bandwidth over the *LocalSumFac* scheme and allows it to achieve better runtime performance. L3 cache performance is reasonably good across all of the schemes, with improvements over *LocalSumFac* in a number of polynomial orders, but in this case does not prove to be a limiting factor. We see generally similar trends in the physical derivative operator, although this time the *SumFac* scheme shows less of a performance increase over the other two sum-factorisation-based schemes.

These results align with the findings of other multi-core performance models, such as the roofline performance model [24] as seen in other high-order performance analysis [25]. In this model, the arithmetic intensity (i.e. number of FLOPS/byte) dictates whether performance is ultimately limited by the maximum FLOPS attainable by the processor or the memory bandwidth of the system. Our amalgamation schemes all use matrix–matrix or matrix–vector multiplications that are of a reasonably high arithmetic intensity. However, the smaller matrices arising from sum-factorisation will result in a lower arithmetic intensity. The development of multiple amalgamation schemes is therefore important, as this allows us to investigate a broader extent of the arithmetic intensity parameter space. This may therefore lead to better memory bandwidth utilisation and lower runtimes, even though the number of FLOPS does not achieve as high a level as other schemes, which we broadly see in the results of Fig. 8.

6. Application to a real example

To demonstrate the benefits of the amalgamation scheme in combination with our auto-tuning method, we now examine a real-world example in a HPC environment. We consider the compressible Euler equations,

$$\partial_t \vec{u} + \nabla \cdot \mathbf{F}(\vec{u}) = \mathbf{0}$$

where $\vec{u} = [\rho, \rho u, \rho v, \rho w, E]^\top$ is a vector of conserved variables in terms of the velocity (u, v, w), density ρ and specific total energy E, with the flux tensor

$$\mathbf{F}(\vec{u}) = \begin{bmatrix} \rho u & \rho v & \rho w \\ p + \rho u^2 & \rho u v & \rho u w \\ \rho u v & \rho v^2 + p & \rho v w \\ \rho u w & \rho v w & \rho w^2 + p \\ u(E+p) & u(E+p) & v(E+p) \end{bmatrix},$$

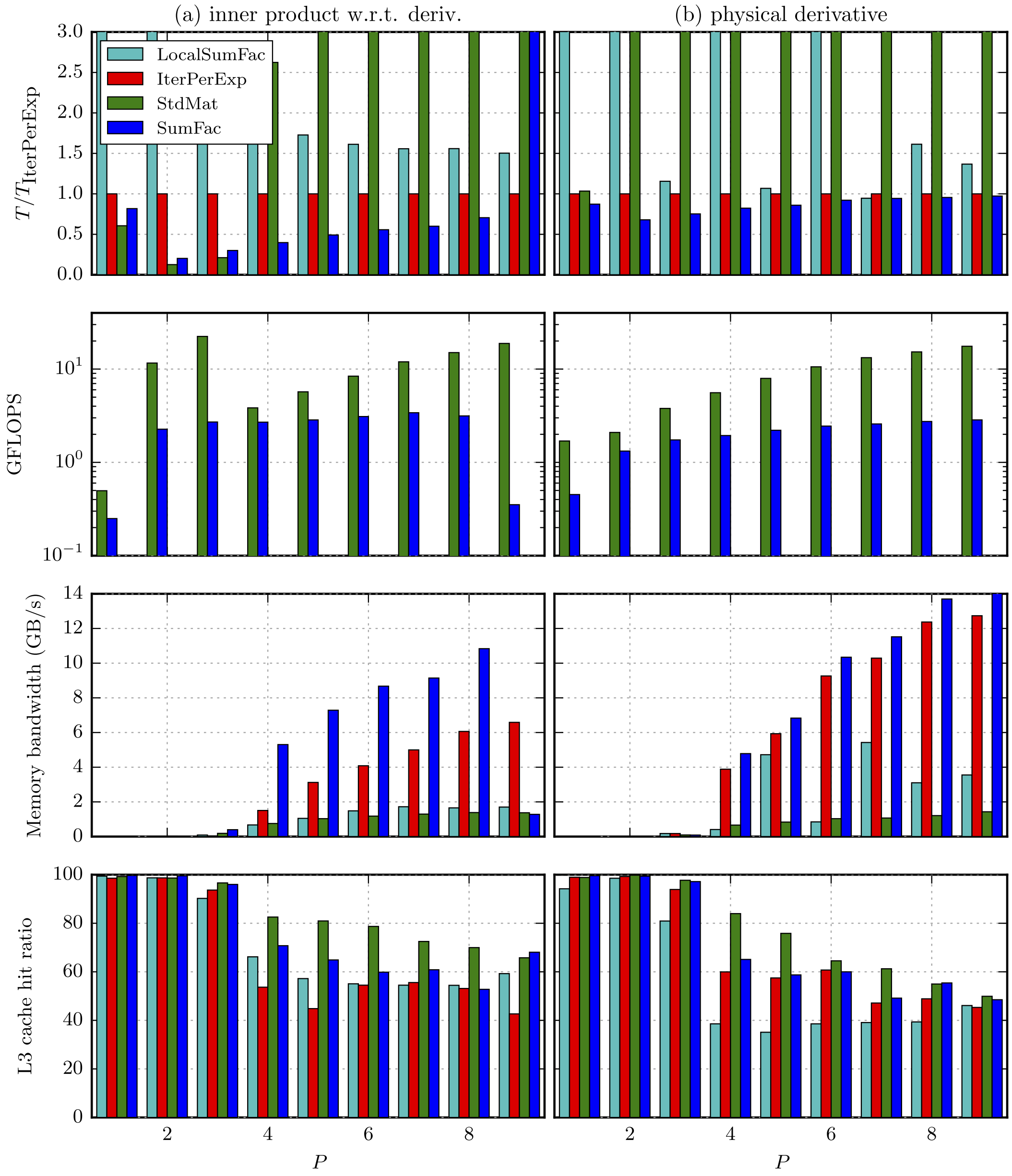

Fig. 8. Measurements of runtime, GFLOPS, memory bandwidth and L3 cache hit ratios for prismatic elements under OpenBLAS on an Intel Core i7-5930K.

and p being the pressure. An equation of state is needed to close the system. In this case, we use the ideal gas law $p = \rho RT$ where T is the temperature and R is the gas constant.

To discretise this hyperbolic system, we adopt a discontinuous Galerkin method, which is well-suited in this setting due to its local conservation properties. Specific details on the discretisation are available in [26] and on its implementation in *Nektar++* in [8,27]. However, we note that the discretisation makes heavy use of the inner product operator with respect to the derivatives of the basis functions, since the weak form involves the calculation of the volume flux term

$$\int_{\Omega^e} \nabla \vec{v} \cdot \mathbf{F}(\vec{u})\, d\vec{x}$$

for a test function $\vec{v}$ composed of polynomial functions.

Table 1
Results of auto-tuning method from the root node for the ONERA M6 wing test case for `cx2` and `ARCHER`. For each operator, the fastest scheme is highlighted in bold text.

Machine	Operator	Scheme timings [s]			
		LocalSumFac	*IterPerExp*	*StdMat*	*SumFac*
`cx2`	BwdTrans	0.00213393	0.00209944	**0.000202192**	0.000534608
	IProductWRTBase	0.00245141	0.00200234	**0.000233064**	0.000521411
	IProductWRTDerivBase	0.0266448	0.017248	**0.00201284**	0.00298702
	PhysDeriv	0.00485056	0.00492247	0.00389733	**0.00319892**
`ARCHER`	BwdTrans	0.000643393	0.000638955	**2.36882e−05**	4.74285e−05
	IProductWRTBase	0.000754697	0.000712303	**2.78743e−05**	0.000150587
	IProductWRTDerivBase	0.00827777	0.00530682	**0.00019947**	0.000643919
	PhysDeriv	0.00075556	0.000595179	**0.000287773**	0.000318533

Table 2
Measured run-times and improvements for default *LocalSumFac* method vs. auto-tuned collections for the ONERA M6 example.

Machine	Wall-time per timestep [s]		Improvement
	LocalSumFac	Auto-tuned collections	
`ARCHER`	1.308	0.744	43%
`cx2`	0.356	0.135	62%

We consider the flow over an ONERA M6 swept wing, a common aeronautics test case [28], at a freestream Mach number $M_\infty = 0.4$. The distribution of the Mach number across the wing surface and symmetry plane, as the flow is evolved to a steady state, is visualised in Fig. 9. A curvilinear tetrahedral mesh of 147,805 elements represents the underlying geometry, generated through the use of an elastic analogy described in [4]. The flow is evaluated using a fully-explicit four-stage 4th-order Runge–Kutta scheme with a timestep of $\Delta t = 10^{-7}$ at a uniform polynomial order of $P = 2$. A HLLC Riemann solver is used to solve the one-dimensional Riemann problem arising at elemental interfaces.

To run this simulation, containing around 1.5m degrees of freedom per conserved variable, requires the use of larger scale computing resources. We consider two machines of varying capabilities. The first, `cx2`, is located within Imperial College HPC facilities. We use 16 nodes containing two 2.93 GHz, 6-core Intel Xeon X5670 CPUs with 12 MB L3 cache each and 24 GB of RAM, connected over an Infiniband interconnect, for a total of 192 cores. The second is `ARCHER`, the UK national supercomputer located at EPCC at the University of Edinburgh. This is a Cray XC30 MPP machine, comprising of nodes containing two 2.7 GHz, 12-core Intel Xeon E5-2697 v2 (Ivy Bridge) processors with 64 GB of RAM and connected via a Cray Aries interconnect. We use 20 nodes for our experiment, giving a total of 960 computing cores. We note that the same mesh and polynomial order is used for each system.

Table 1 shows the timings obtained from the auto-tuning method on the root process, which executed at solver setup in the span of around 20 s. We note that each processor runs its own sets of timings and this should only therefore be seen as a representative example. The autotuning method is run across all nodes simultaneously, with each processor selecting its own fastest observed time. We deliberately choose this approach, as opposed to, for example, the root process dictating the scheme, since mixed element meshes will lead to partitions requiring different scheme selection. The table shows the timings for each method, with the fastest highlighted in bold. We see that typically the *StdMat* operator is preferred, which corresponds to the general trends seen in Section 3 for this polynomial order. Indeed, over the *LocalSumFac* baseline scheme, we see that generally the amalgamation schemes perform extremely well, being almost an order of magnitude faster in the case of `cx2`.

To see the effect of each scheme on the overall runtime, we now measure the average wall-time needed per timestep over 5000 samples. This timing therefore incorporates all other aspects of the compressible Euler solver, such as the Riemann solver and communication costs. Table 2 shows the measured wall-time for each of the machines. The use of our amalgamation schemes, in combination with the auto-tuning method, results in a reduction in overall runtime

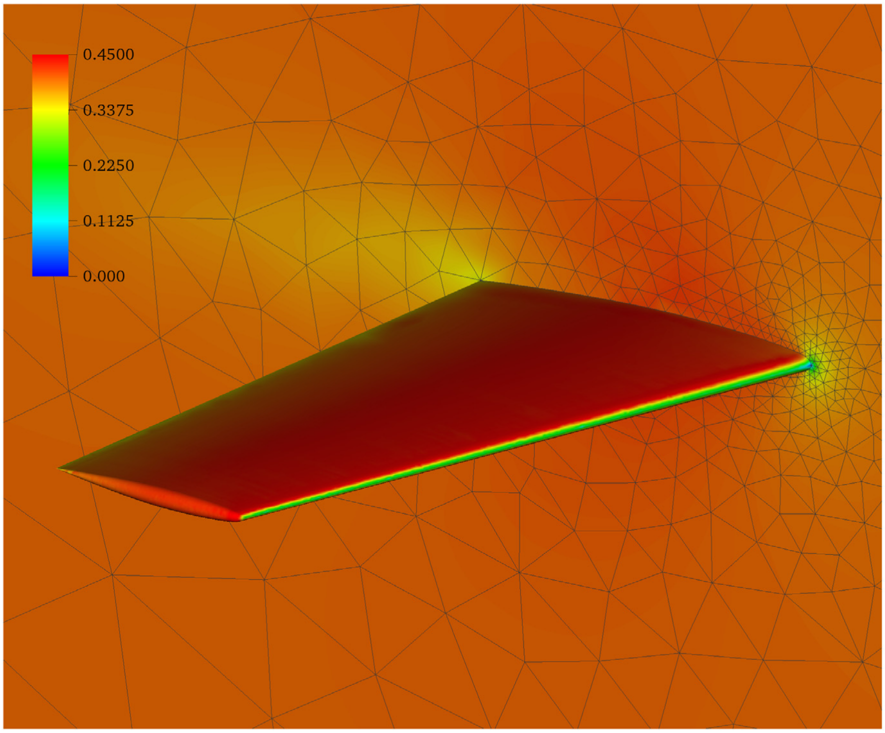

Fig. 9. Distribution of the Mach number $0 < M \leq 0.45$ across the surface of the ONERA M6 wing as the flow evolves to a steady state.

of around 40% for cx2 and 60% for ARCHER. This is a considerable improvement in execution time and shows the potential advantages that can be achieved when using amalgamation.

7. Conclusions

In this work, we have presented a methodology for amalgamating the action of various key finite element operators across a range of elements. The resulting amalgamation schemes demonstrate improved performance due to their more efficient use of data locality and reduction in data transfer across the memory bus, enabling increased performance through exploiting optimised BLAS routines and the CPU cache structure. An auto-tuning method was presented, enabling the automatic selection of the most efficient scheme at runtime. We have shown how these schemes can be leveraged to improve runtimes, both by examining the schemes individually and by applying them to a large-scale simulation of the compressible Euler equations. The results clearly demonstrate the importance and benefits of streaming data from memory efficiently.

As alluded to in the introduction, we stress that the results shown here are generally not specific to the spectral/*hp* element method, due to the fundamental nature of the operators being used. Other high-order schemes, such as the popular nodal discontinuous Galerkin method [26], rely on the evaluation of the same types of operators, which in turn have similar matrix formulations. However, we note that the *SumFac* scheme may not be applicable, depending on the choice of basis functions used in the local expansion of each element. As we describe in Section 2, sum-factorisation relies on the ability to write local expansion modes as the tensor product of one-dimensional functions. In the nodal DG scheme, hybrid elements such as prisms and tetrahedra typically use Lagrange interpolants together with a set of suitable solution points, such as Fekete or electrostatic point distributions. This choice of basis functions is inherently non-tensor-product based, and so the *SumFac* schemes we consider here cannot therefore be utilised for these element types. However, the *IterPerExp* and *StdMat* schemes are both equally applicable in this setting.

Possible routes for further work could focus around improvements to the autotuning method, which may not be well-suited for a fully *hp*-adaptive simulation. Other kernel implementations that are specifically designed to optimise performance tailored to particular operators, such as the approach investigated in [25] that generate matrix-specific code based on the sparsity of the standard matrix, could also prove useful in furthering the performance of this work.

We also note that in general, the performance gains presented in Section 6 may not be achievable in other solvers. In particular, where the problem is either fully implicit or semi-implicit, a large proportion of the overall runtime is usually spent inverting the global matrix system. In parallel this is done iteratively, with the action of the global matrix represented through a block-diagonal elemental matrix and an appropriate gather–scatter operation. Future studies on this topic should therefore focus on making this process more memory efficient in combination with our amalgamation schemes.

Acknowledgements

We thank D. Ekelschot and M. Turner for their assistance in generating the mesh and parameters for the simulation of Section 6. We also thank F. Witherden for initial discussions motivating this study. This work was funded in part by support from the libHPC II EPSRC project under grant EP/K038788/1. DM additionally acknowledges support under the Laminar Flow Control Centre funded by Airbus/EADS and EPSRC under grant EP/I037946. SJS acknowledges Royal Academy of Engineering support under their research chair scheme. We thank the Imperial College High Performance Computing Service for computing time used to calculate the results seen in Section 6. We additionally acknowledge access to ARCHER with support from the UK Turbulence Consortium under EPSRC grant EP/L000261/1.

References

[1] G. Karniadakis, S. Sherwin, Spectral/hp Element Methods for Computational Fluid Dynamics, second ed., Oxford University Press, 2005.

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

[3] C.D. Cantwell, S. Yakovlev, R.M. Kirby, N.S. Peters, S.J. Sherwin, High-order spectral/hp element discretisation for reaction–diffusion problems on surfaces: Application to cardiac electrophysiology, J. Comput. Phys. 257 (2014) 813–829.

[4] D. Moxey, D. Ekelschot, U. Keskin, S.J. Sherwin, J. Peiró, A thermo-elastic analogy for high-order curvilinear meshing with control of mesh validity and quality, Procedia Eng. 82 (2014) 127–135.

[5] A.C. Nogueira Jr, M.L. Bittencourt, Spectral/hp finite elements applied to linear and non-linear structural elastic problems, Lat. Am. J. Solids Struct. 4 (2007) 61–85.

[6] A. Comerford, K. Chooi, M. Nowak, P. Weinberg, S. Sherwin, A combined numerical and experimental framework for determining permeability properties of the arterial media, Biomech. Model. Mech. Biol. (2014) 1–17.

[7] C. Eskilsson, S. Sherwin, A triangular spectral/hp discontinuous Galerkin method for modelling 2D shallow water equations, Internat. J. Numer. Methods Fluids 45 (2004) 605–623.

[8] C. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. de Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, et al., Nektar++: An open-source spectral/hp element framework, Comput. Phys. Comm. (2015).

[9] S. Yakovlev, D. Moxey, S.J. Sherwin, R.M. Kirby, To CG or to HDG: a comparative study in 3D, J. Sci. Comp. 67 (1) (2016) 192–220. http://dx.doi.org/10.1007/s10915-015-0076-6.

[10] F.D. Witherden, A.M. Farrington, P.E. Vincent, PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach, Comput. Phys. Comm. 185 (2014) 3028–3040.

[11] J. King, S. Yakovlev, Z. Fu, R.M. Kirby, S.J. Sherwin, Exploiting batch processing on streaming architectures to solve 2D elliptic finite element problems: A hybridized discontinuous galerkin (HDG) case study, J. Sci. Comput. 60 (2014) 457–482.

[12] S. Kerkemeier, S. Parker, PFF, 2010. Scalability of the NEK5000 spectral element code. Jülich Blue Gene/P Extreme Scaling Workshop 2010, Tech. Rep.

[13] P.E. Vos, S.J. Sherwin, R.M. Kirby, From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low-and high-order discretisations, J. Comput. Phys. 229 (2010) 5161–5181.

[14] C. Cantwell, S. Sherwin, R. Kirby, P. Kelly, From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements, Comput. & Fluids 43 (2011) 23–28.

[15] C. Cantwell, S. Sherwin, R. Kirby, P. Kelly, From h to p efficiently: selecting the optimal spectral/hp discretisation in three dimensions, Math. Model. Nat. Phenom. 6 (2011) 84–96.

[16] J. Shin, M.W. Hall, J. Chame, C. Chen, P.F. Fischer, P.D. Hovland, Speeding up nek5000 with autotuning and specialization, in: Proceedings of the 24th ACM International Conference on Supercomputing, ACM, 2010, pp. 253–262.

[17] J.R. Stewart, H.C. Edwards, The sierra framework for developing advanced parallel mechanics applications, in: Large-Scale PDE-Constrained Optimization, Springer, 2003, pp. 301–315.

[18] R.P. Pawlowski, E.T. Phipps, A.G. Salinger, S.J. Owen, C.M. Siefert, M.L. Staten, Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, part ii: Application to partial differential equations, Sci. Program. 20 (2012) 327–345.

[19] M.G. Duffy, Quadrature over a pyramid or cube of integrands with a singularity at a vertex, SIAM J. Numer. Anal. 19 (1982) 1260–1262.

[20] S.A. Orszag, Spectral methods for problems in complex geometries, J. Comput. Phys. 37 (1980) 70–92.

[21] Netlib BLAS website. http://www.netlib.org/blas/.

[22] OpenBLAS website. http://www.openblas.net/.

[23] F.G. Gustavson, High-performance linear algebra algorithms using new generalized data structures for matrices, IBM J. Res. Dev. 47 (2003) 31–55.

[24] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (2009) 65–76.

[25] B.D. Wozniak, F.D. Witherden, F.P. Russell, P.E. Vincent, P.H. Kelly, GiMMiK—Generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics, Comput. Phys. Comm. (2016).

[26] J.S. Hesthaven, T. Warburton, Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications, vol. 54, Springer, 2007.

[27] D. de Grazia, G. Mengaldo, D. Moxey, P.E. Vincent, S.J. Sherwin, Connections between the discontinuous Galerkin method and high-order flux reconstruction schemes, Internat. J. Numer. Methods Fluids 75 (2014) 860–877.

[28] V. Schmitt, F. Charpin, Pressure distributions on the ONERA M6 wing at transonic mach numbers Experimental database for computer program assessment, 4, 1979.