Finite and Spectral Element Methods on Unstructured Grids for Flow and Wave Propagation Problems

Dominik Göddeke, Dimitri Komatitsch and Matthias Möller

Abstract Finite element methods are one of the most prominent discretisation techniques for the solution of partial differential equations. They provide high geometric flexibility, accuracy and robustness, and a rich body of theory exists. In this Chapter, we summarise the main principles of Galerkin finite element methods, and identify and discuss avenues for their parallelisation. We develop guidelines that lead to efficient implementations, however, we prefer generic ideas and principles over utmost performance tuning.

1 Introduction

Many relevant processes and phenomena from a wide range of scientific areas and application domains can be described by mathematical models comprising (a system of) partial differential equations (PDEs). A simple example is the Poisson equation

$$-\Delta u = f\,, \tag{1}$$

which is fulfilled by the scalar quantity u that represents the state of minimal energy subject to load f and appropriate boundary conditions. As an illustration, consider the deformation due to loading of an elastic membrane that is fixed on a frame.

Dominik Göddeke
Department of Mathematics, Institute of Applied Mathematics, TU Dortmund, Germany, e-mail: dominik.goeddeke@math.tu-dortmund.de

Dimitri Komatitsch
Laboratory of Mechanics and Acoustics, CNRS UPR 7051, Aix-Marseille University, Centrale Marseille, France, e-mail: komatitsch@lma.cnrs-mrs.fr

Matthias Möller
Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands, e-mail: m.moller@tudelft.nl

A large class of model problems can be written as first-order systems of the form

$$\partial_t U + \nabla \cdot \mathbf{F}(U) = 0\,, \tag{2}$$

where $\mathbf{F} = [F^1, \ldots, F^n]$ represents an n-dimensional flux function that depends on the solution U but not on its derivatives. As an example, consider the flow of air around an aeroplane at high speeds, which can be modelled by the equations of gas dynamics.

A third example of an important real-world phenomenon that is modelled by a time-dependent PDE is the propagation of waves, for instance the propagation of seismic waves in the Earth to calculate the effects of earthquakes, or the propagation of ultrasonic acoustic waves in ocean acoustics or in non-destructive testing. The appropriate mathematical model is the elastodynamics equation

$$\rho \partial_t^2 \mathbf{u} - \nabla \cdot \mathbf{T}(\mathbf{u}, \nabla \mathbf{u}) = \mathbf{f}\,, \tag{3}$$

where the stress tensor $\mathbf{T}$ depends on the multi-dimensional displacement field $\mathbf{u}$ and/or its (transposed) gradient, which yields a second-order system.

The numerical treatment of such PDE problems typically involves two aspects, the *discretisation* that maps the continuous model to a formulation suitable for computers, and the *solver* that computes actual solution approximations. Both from an engineering and mathematical point of view, the finite element method (FEM) is often well suited for the discretisation: Finite elements offer a high degree of geometric flexibility since they can be naturally formulated on unstructured grids. Furthermore, they can deliver high (guaranteed) accuracy and robustness when enhanced with *h*/*hp*-adaptation and a-posteriori error estimation techniques, for which solid theoretical foundations exist in the FEM framework. In combination with powerful and robust iterative solvers for the resulting linear or non-linear systems of equations, finite elements form the underlying fabric of many modern simulation tools.

2 Finite Element Analysis in a Nutshell

It is far beyond the scope of this Chapter to give a comprehensive introduction to the finite element method including all its variants and theoretical aspects. Thus, we only outline the basic concepts of the *continuous* Galerkin finite element method and refer the interested reader to, e.g., [8, 10] for an introduction to practical finite element analysis, and to the more theoretical textbooks [2, 4]. A good overview of *discontinuous* Galerkin methods, which are not considered in this Chapter but feature amenable properties that can be helpful for designing highly efficient parallelised codes, can be found for instance in [14] and the references therein.

2.1 Variational Formulation

Let the scalar quantity u be governed by the generic PDE model problem

$$Lu = f$$

within the d-dimensional domain $\Omega \subset \mathbb{R}^n$, $n = 1,2,3$, where L represents a linear spatial differential operator. Moreover, u has to fulfil certain conditions at the boundary $\Gamma := \partial\Omega$, e.g., a combination of Dirichlet and Neumann conditions:

$$\begin{aligned} u &= u_D \quad \text{on} \quad \Gamma_D \subseteq \Gamma \ , \\ \partial_n u &= g \quad \text{on} \quad \Gamma_N = \Gamma \setminus \Gamma_D \ . \end{aligned}$$

The first step in deriving the finite element method is to translate the problem at hand into its variational form, which amounts to integrating the weighted residual over the domain Ω and forcing the result to vanish. Thus, the solution u is sought in some suitable function space V, referred to as trial space, such that

$$\int_\Omega w(Lu - f)\,\mathrm{d}\mathbf{x} = 0 \qquad \forall w \in W \ , \tag{4}$$

where W denotes the space of test functions w. Both spaces have to comply with the demands of the differential operator L and the Dirichlet boundary conditions.

In the second step, the infinite dimensional function spaces V and W are approximated by finite dimensional ones, denoted by V_h and W_h.

2.2 Galerkin Discretisation

To simplify the presentation, let us adopt the same set of basis functions $\{\varphi_i\}_{i=1}^N$ for the discrete test and trial spaces W_h and V_h, respectively. The approximate solution and its derivatives can then be represented as:

$$u(\mathbf{x}) \approx u_h(\mathbf{x}) = \sum_{j=1}^N \varphi_j(\mathbf{x}) u_j, \qquad Lu(\mathbf{x}) \approx Lu_h(\mathbf{x}) = \sum_{j=1}^N L\varphi_j(\mathbf{x}) u_j$$

Substituting them into the weak form (4) and replacing the weighting function w by all possible φ_i yields a linear system of equations for the vector of unknowns $u = [u_1, \ldots u_N]^T$:

$$\sum_{j=1}^N \left[\int_\Omega \varphi_i L\varphi_j \,\mathrm{d}\mathbf{x} \right] u_j = \int_\Omega \varphi_i f \,\mathrm{d}\mathbf{x}, \qquad i = 1,2,\ldots,N$$

As an example, consider Poisson's equation (1), which corresponds to defining the differential operator according to $L[\cdot] := -\Delta[\cdot] = -\nabla\cdot\nabla[\cdot]$. Performing integration

by parts results in seeking $u \in V := \{u \in \mathscr{H}^1(\Omega) : u = u_D \text{ on } \Gamma_D\}$ such that

$$\int_\Omega \nabla w \cdot \nabla u \,\mathrm{d}\mathbf{x} = \int_{\Gamma_N} wg \,\mathrm{d}s + \int_\Omega wf \,\mathrm{d}\mathbf{x}$$

for all test functions $w \in W := \{w \in \mathscr{H}^1(\Omega) : w = 0 \text{ on } \Gamma_D\}$ that vanish on the Dirichlet boundary part. Here and below $\mathscr{H}^1(\Omega)$ denotes the space of square integrable functions with square integrable first weak derivatives. The discrete counterpart of the problem at hand can be expressed in compact matrix form as

$$Su = b \quad \text{with} \quad s_{ij} = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j \,\mathrm{d}\mathbf{x} = s_{ji} \quad \text{and} \quad b_i = \int_{\Gamma_N} \varphi_i g \,\mathrm{d}s + \int_\Omega \varphi_i f \,\mathrm{d}\mathbf{x}\,. \tag{5}$$

Note that $s_{ij} \neq 0$ if and only if φ_i and φ_j have overlapping supports, and that basis functions are typically constructed so that they fulfil a local support property. For practical applications, the system (5) may thus be very large but will remain sparse.

2.3 Element-Based Assembly

In all finite element methods, the domain Ω is covered by non-overlapping simple geometric objects, e.g., triangles and/or quadrilaterals in two space dimensions, and tetrahedra, hexahedra etc. in three space dimensions. This (fully unstructured) partition $\mathscr{T}_h = \{T_1, \ldots, T_M\}$ is referred to as a mesh, or triangulation, of Ω. It is common practice to associate the degrees of freedom with entities of this mesh such as element vertices or midpoints of edges/faces. Depending on the choice of basis functions, the unknowns u_j may represent nodal solution values, i.e. $u_j = u_h(\mathbf{x}_j)$, integral mean values of the solution, or they can be related to solution derivatives.

The global integral terms are then assembled by summing over individual element contributions that may either be computed exactly or approximated by some cubature rule (quadrature formula), e.g., Gaussian quadrature. The weighting coefficients $\hat{\omega}_k$ and cubature points $\hat{\mathbf{x}}_k$ are typically tabulated for some reference element $\hat{T}$ with a regular shape. That is,

$$\int_{\hat{T}} I(\hat{\mathbf{x}}) \,\mathrm{d}\hat{\mathbf{x}} \approx \sum_{k=1}^{N_{\text{cub}}} \hat{\omega}_k I(\hat{\mathbf{x}}_k) \tag{6}$$

for a generic integrand $I(\cdot)$. As a result the numerical evaluation procedure reads

$$\int_{\Phi_T(\hat{T})} \varphi_i(\mathbf{x}) L\varphi_j(\mathbf{x}) \,\mathrm{d}\mathbf{x} = \int_{\hat{T}} \varphi_i(\Phi_T(\hat{\mathbf{x}})) L\varphi_j(\Phi_T(\hat{\mathbf{x}})) \,|\det J(\hat{\mathbf{x}})| \,\mathrm{d}\hat{\mathbf{x}}\,. \tag{7}$$

That is, it involves a change of variables, where $\Phi_T : \hat{T} \mapsto T$ is the mapping from the reference mesh element to the physical one so that coordinates are related by $\mathbf{x} = \Phi_T(\hat{\mathbf{x}})$, and $J = D\Phi_T$ denotes the Jacobian matrix of the transformation.

Let the basis $\{\varphi_i\}_{i=1}^N$ be given by the definition of shape functions on individual elements. For instance, using linear polynomials that equal unity at one vertex of a triangle/tetrahedron and vanish at all other vertices leads to Lagrange finite elements of degree $p = 1$. The choice of shape/basis functions determines the order of the final approximation. In so-called p-adaptive schemes the shape functions, and hence, the approximation order may therefore differ from one element to the other.

In parametric finite elements, the reference element is also used to define the shape functions in terms of referential coordinates, i.e. $\hat{\varphi}_i(\hat{\mathbf{x}})$. Substituting the relation $\varphi_i(\Phi_T(\hat{\mathbf{x}})) = \hat{\varphi}_i(\hat{\mathbf{x}})$ into expression (7) yields the final integration formula (6) to be implemented into a finite element code. This approach makes it possible to, say, adopt higher-order basis functions but still use a mapping of low order, called sub-parametric approach. On the other hand, using the same order for both components (iso-parametric) or even increasing the order of the mapping (super-parametric) may be beneficial if curved boundaries need to be approximated with high accuracy.

In summary, the assembly procedure for the global stiffness matrix S of the Poisson problem reduces to evaluating all non-vanishing matrix coefficients

$$s_{ij} = \sum_{T\in\mathscr{T}_h} \int_{\hat{T}} \underbrace{\left(J^{-T}(\hat{\mathbf{x}})\hat{\nabla}\hat{\varphi}_i(\hat{\mathbf{x}})\right)\cdot\left(J^{-T}(\hat{\mathbf{x}})\hat{\nabla}\hat{\varphi}_j(\hat{\mathbf{x}})\right)|\det J(\hat{\mathbf{x}})|}_{=:I(\hat{\mathbf{x}},i,j)}\,\mathrm{d}\hat{\mathbf{x}} \tag{8}$$

after applying the chain rule and the theorem of local inverses.

The assembly procedure of the volumetric part of the right-hand side vector

$$b_i = \sum_{T\in\mathscr{T}_h} \int_{\hat{T}} \underbrace{\hat{\varphi}_i(\hat{\mathbf{x}}) f(\Phi_T(\hat{\mathbf{x}}))\,|\det J(\hat{\mathbf{x}})|}_{=:I(\hat{\mathbf{x}},i)}\,\mathrm{d}\hat{\mathbf{x}} \tag{9}$$

is sketched in Fig. 1. Note that numerical cubature rules of higher order are typically adopted to integrate the function f with sufficient accuracy.

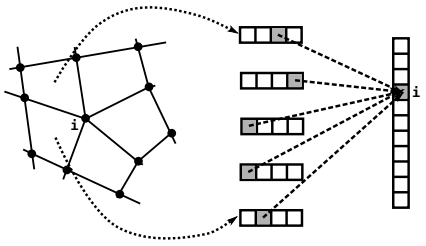

Fig. 1 Assembly of (9): The local element integrals for all five elements contributing to node i are computed independently and assembled into the global vector.

The same approach can be applied to the entries of the bilinear form if, say, the differential operator L is replaced by a non-linear one as it is the case for the elastodynamics equation (3). The above strategy is not restricted to linear and bilinear forms, it can be naturally extended to multi-linear forms such as the non-linear convection term $\mathbf{u}\cdot\nabla\mathbf{u}$ that plays a central role in the Navier-Stokes equations. The tensor contraction approach [13] is an alternative concept for the assembly of arbitrary

multi-linear forms not addressed here due to space constraints. For the extension of the tensor contraction approach to GPUs, the interested reader is referred to [15].

2.4 Group Finite Element Formulation

An alternative approach that is commonly used in finite elements for conservation laws such as the first-order system (2) is the *group finite element formulation* developed by Fletcher [9]. Instead of evaluating the non-linear flux function $\mathbf{F}(U) \approx \mathbf{F}(U_h)$ based on the interpolated solution $U_h(\mathbf{x},t) = \sum_{j=1}^{N} \varphi_j(\mathbf{x}) \mathrm{U}_j(t)$, the same basis is adopted to interpolate the fluxes

$$\mathbf{F}(U) \approx \mathbf{F}_h(\mathbf{x},t) = \sum_{j=1}^{N} \varphi_j(\mathbf{x}) \mathbf{F}_j(t), \qquad \mathbf{F}_j(t) = \mathbf{F}(\mathrm{U}_j(t)), \qquad \mathrm{U}_j(t) = U(\mathbf{x}_j,t)\,.$$

The resulting semi-discretised variational formulation of system (2) reads

$$\sum_{j=1}^{N} \left[\mathrm{M}_{ij} \frac{\mathrm{d}\mathrm{U}_j(t)}{\mathrm{d}t} + \mathbf{c}_{ij} \cdot \mathbf{F}_j(t) \right] = 0\,, \tag{10}$$

where $M = \{\mathrm{M}_{ij}\}$ is the consistent mass matrix with $N_\mathrm{D} \times N_\mathrm{D}$ blocks defined by $\mathrm{M}_{ij} = m_{ij}\mathbb{I}$, where $\mathbb{I}$ stands for the identity tensor. The coefficients of the mass matrix $M = \{m_{ij}\}$ and those of the discrete gradient operator $\mathbf{C} = \{\mathbf{c}_{ij}\}$ are given by

$$m_{ij} = \int_\Omega \varphi_i(\mathbf{x}) \varphi_j(\mathbf{x})\,\mathrm{d}\mathbf{x}, \qquad \mathbf{c}_{ij} = \int_\Omega \varphi_i(\mathbf{x}) \nabla\varphi_j(\mathbf{x})\,\mathrm{d}\mathbf{x}\,. \tag{11}$$

These coefficients remain constant unless the mesh is changed, and therefore, they can be computed a priori and stored (maybe adopting an optimized renumbering strategy) for later use. It goes without saying that the element-based assembly is readily applicable to assemble M and $\mathbf{C}$ in the preprocessing step.

As an example, consider the linear advection equation $\partial_t u + \nabla \cdot \mathbf{f}(u) = 0$ with flux function $\mathbf{f}(u) = \mathbf{v}u$, whereby the externally given velocity field $\mathbf{v} = \mathbf{v}(\mathbf{x},t)$ may vary both in time and space. Adopting the group finite element formulation, the convective operator $K = \{k_{ij}\}$ can be updated very efficiently letting $k_{ij} = \mathbf{c}_{ij} \cdot \mathbf{v}_j(t)$ without the need to perform costly numerical integration in every time step.

2.5 Edge-Based Assembly

For finite elements featuring the partition-of-unity property $\sum_{j=1}^{N} \varphi_j(\mathbf{x}) = 1$, which holds for instance for Lagrangian or B-spline basis functions, the matrix of auxiliary coefficients has zero row sums, i.e., $\sum_{j=1}^{N} \mathbf{c}_{ij} = 0\ \forall i$. This property makes it possible

to cast equation (10) into the following edge-based form [20]

$$\sum_{j=1}^{N} \mathrm{M}_{ij} \frac{\mathrm{d}\mathrm{U}_j(t)}{\mathrm{d}t} + \sum_{j \in \mathscr{S}_i} \mathbf{c}_{ij} \cdot [\mathbf{F}_j(t) - \mathbf{F}_i(t)] = 0 \,, \tag{12}$$

where the index set $\mathscr{S}_i = \{1 \leq i \neq j \leq N : \operatorname{supp} \varphi_i \cap \operatorname{supp} \varphi_j \neq \emptyset\}$ extends over all neighbouring degrees of freedom j that share a common edge with i. In [19], an alternative flux decomposition has been developed that amounts to performing integration by parts in the spatial discretisation of the divergence terms, yielding

$$\mathbf{c}_{ij} = -\mathbf{c}_{ji} + \mathbf{s}_{ij}, \qquad \mathbf{s}_{ij} = \int_{\Gamma} \varphi_i(\mathbf{x}) \varphi_j(\mathbf{x}) \mathbf{n} \, \mathrm{d}s \,,$$

whereby the symmetric boundary term $\mathbf{s}_{ij}$ vanishes in the interior. Finally, equation (12) can be recast into the equivalent edge-based formulation [19]

$$\sum_{j=1}^{N} \left[\mathrm{M}_{ij} \frac{\mathrm{d}\mathrm{U}_j(t)}{\mathrm{d}t} + \mathbf{s}_{ij} \cdot \mathbf{F}_j(t) \right] + \sum_{j \in \mathscr{S}_i} \mathrm{G}_{ij}(t) = 0 \tag{13}$$

with skew-symmetric Galerkin fluxes $\mathrm{G}_{ij}(t) = \mathbf{c}_{ij} \cdot \mathbf{F}_i(t) - \mathbf{c}_{ji} \cdot \mathbf{F}_j(t) = -\mathrm{G}_{ji}(t)$.

3 Implementation and Parallelisation Strategies

At a sufficiently high level of abstraction, element-based FEM as depicted, e.g., in (8), reduces to a double-nested loop over matrix positions (i, j) and elements $T \in \mathscr{T}_h$, respectively. Within each element, another triple-nested loop over all cubature points for all pairs of test and trial functions is executed to compute the integral(s). The edge-based approach shares many features with the element-based one, so we can focus on elements. Depending on how one arranges the loops, one obtains different assembly algorithms. We first discuss advantages and disadvantages of choosing the ij- or T-loop as the outermost one, and then consider the order of the per-element loops. We emphasise two aspects that are clear from Sect. 2: First, only the resulting matrix entries s_{ij} are written to memory, all other operations either read data from memory or perform arithmetics. This simple observation is crucial for extracting parallelism in finite element methods. Second, due to the unstructured nature of the mesh $\mathscr{T}_h$, memory indirections cannot be avoided.

3.1 Choice of the Outermost Loop

Nonzeros-first The *nonzeros-first* approach can be seen as the direct implementation of expression (8). The outermost loop iterates over all non-vanishing matrix

entries s_{ij}, and for each entry, all elemental contributions are *gathered* by computing each integral in the sum for which φ_i and φ_j have overlapping support. In the example depicted in Fig. 1 for a right hand side assembly, we need to compute five per-element contributions for node i, corresponding to the vector entry i. Based on the observation above, the method is intrinsically parallel and free of synchronisation requirements because all computations can be performed independently for all non-vanishing matrix entries. This constitutes a major conceptual advantage of this algorithm, but immediately implies its potential disadvantage: All per-element or per-edge integrations that contribute to the ij sum are performed redundantly, e.g., a given element is computed first for node i and again for node j if both nodes are in its support. This method is called 'assembly-by-nonzeros' in [3].

Elements-first To avoid this redundancy, one can alternatively iterate over all elements in the outermost loop, and *scatter* the contributions stemming from each element to the matrix entries representing the support of the element. We mostly focus on this technique throughout this Chapter. Subsets of the generic approach presented here have been considered before, for instance 'assembly-by-elements' in [3], our own work (to our knowledge the first high-order FEM-GPU implementation [16]), and the 'add-to' algorithm [23]. The generic algorithm proceeds as follows:

Loop over all elements $T \in \mathscr{T}_h$
  Determine N_{test}, N_{trial} and N_{cub}, $\{\hat{\omega}\}_{\text{cub}}$, $\{\hat{\mathbf{x}}\}_{\text{cub}}$
  Loop over all $i_{\text{test}} = 1,\ldots,N_{\text{test}}$ test functions
    Loop over all $j_{\text{trial}} = 1,\ldots,N_{\text{trial}}$ trial functions
      Loop over all $k_{\text{cub}} = 1,\ldots,N_{\text{cub}}$ cubature points
        Compute the integrand $I(\hat{\mathbf{x}}_{k_{\text{cub}}}, i_{\text{test}}, j_{\text{trial}})$
        Scatter the result: $s_{G(i_{\text{test}},j_{\text{trial}})} += \hat{\omega}_{k_{\text{cub}}} I(\hat{\mathbf{x}}_{k_{\text{cub}}}, i_{\text{test}}, j_{\text{trial}})$

Here, $I(\hat{\mathbf{x}}_{k_{\text{cub}}}, i_{\text{test}}, j_{\text{trial}})$ denotes the evaluation of the integrand in Eq. (8) for the given cubature point and pair of test/trial functions, and $\hat{\omega}_{k_{\text{cub}}}$ is the cubature weight that depends on the cubature formula. G is the mapping from local degrees of freedom i_{test} and j_{trial} to their global counterparts i and j, depicted in Fig. 1 for the vector-assembly case. This way of casting the assembly into a loop ordering removes all redundant computations. However, it comes at a cost because the inherent parallelism of the 'nonzeros-first' approach is partially lost. To see this, consider again Fig. 1, and assume that all five elements are computed simultaneously via some parallelisation of the outermost loop in the algorithm sketched above. Due to the first observation on input and output data on the previous Page, this is not an issue for the bulk of computations in the assembly process: As only local computations and read operations for global data are performed per element, the local portion of the computation is still trivially parallel. However, once all five parallel threads reach the accumulation operation (the += statement), a *race condition* occurs because five threads update the i-th memory location in the target array (or matrix) simultaneously and thus there is no guarantee about the final sum: In many

cases it will be partial and thus incorrect. In practice, the result can even vary over several executions of the program on the same machine.

To ensure correctness in the accumulation step, the increments need to be made *mutually exclusive*, i.e., one needs to ensure that during the 'read-modify-write' sequence performed by one thread, no other threads interfere. There are essentially two different solutions to this problem, leading to different assembly algorithms.

Synchronisation Most parallel programming environments for multicores and GPUs provide built-in mechanisms to synchronise on certain sequences of operations and/or on memory locations. To use them, code statements that may cause data races must be labelled with specific keywords, or dedicated function calls can be used. The general idea is that the hardware and/or the runtime can serialise the sequence of operations automatically, because they have been made aware of the condition. For instance, both NVIDIA CUDA and OpenMP provide so-called *atomic memory updates* for, e.g., increment operations. Using them ensures that as many operations as possible remain parallel, because only the actual increment operation becomes protected through an automatic serialisation. In longer sequences of operations, exclusive access to resources (e.g., array entries) must be ensured by other synchronisation techniques. In OpenMP for instance, so-called 'critical sections' can be used, while in CUDA, cheap barrier synchronisation between blocks of threads is available.

Decoupling To resolve the conflicting writes without resorting to synchronisation, the computation can also be rearranged so that the race condition can never occur. The basic idea is to take advantage of the fact that the support for each degree of freedom constitutes, in typical large meshes, just a few neighbouring elements. Any two elements may safely increment 'their' memory locations if the supports do not overlap, i.e., if the memory locations they write to are independent. In the decoupling technique, entities (e.g., elements or edges) that can lead to problems are a priori partitioned accordingly into disjoint sets. The parallel loop over all entities is then replaced with a sequential loop over all such sets, while all entities within one set can still be treated independently in parallel. Such independent sets are typically computed through some kind of *colouring* algorithm [7].

Matrix-free methods In this Chapter, we focus on methods that yield actual matrices and vectors, in some standard format such as CSR. An alternative approach are 'matrix-free' methods. One example that has been successfully demonstrated on GPUs in [23] is the 'local matrix approach': All local element matrices are computed independently (e.g., as in the elements-first algorithms we describe below) and stored for later use, but the global system is never actually assembled. The complete 'proto-assembly' is thus free of both synchronisation and redundancy. The idea is then to modify all operations that would normally make use of the matrix (or vectors), e.g., sparse matrix vector multiplies. This method is most beneficial if the assembly rather than the solver dominates the complete simulation.

3.2 Per-Element Loops

Both the nonzeros-first and the elements-first approach assume for correctness that the inner triple-nested loop is executed sequentially. For large meshes with a high number of elements, this is a valid assumption on CPU-type architectures. On GPUs however, it can be advantageous to expose parallelism in these loops as well, in particular in higher-order finite element methods. More parallelism can often increase overall throughput substantially owing to less granularity effects and less resource contention. The corresponding code transformations typically lead to data structures that are also beneficial on CPU-type architectures, e.g., through better exploitation of the vectorisation capabilities (SIMD units). We return to this issue in Sect. 4, because it is highly dependent on the actual finite element method.

3.3 An Improved Blocked Version

The loop structure given in the previous Section reveals that large parts of the algorithm depend on the actual element type and discrete operator(s). Let us emphasise a few important examples:

- The determinant of the Jacobian is computed differently in the transformation to different reference elements (e.g., Cartesian vs. barycentric coordinates), or is even the same for all elements.
- Cubature formulae vary, per se and also depending on the element type.
- Coefficients of the various derivatives of the operator(s) must be evaluated at each cubature point.

The naive solution to this problem is to include large amounts of nested conditional statements and callback functions. In terms of efficiency, it is generally a bad idea to do so in the innermost loops, due to the comparatively high function call overhead, and the resulting branch divergence that prevents SIMD/SIMT execution. C++ template metaprogramming or creative use of the preprocessor can partially alleviate the issue at compile time, although this is no longer possible when runtime decisions are needed, e.g., when mixed-element methods and/or p-adaptivity are employed.

As a remedy, the elements can be reorganised into 'distributions'

$$\mathscr{T}_h = D_1 \cup D_2 \cup \ldots \cup D_{N_D}$$

of elements featuring identical or similar properties (e.g., polynomial degree, shape). This additional level of partitioning allows one to 'hose' all conditionals out of the innermost loops, and to drastically reduce the amount of callback functions. Note that the colouring approach can be incorporated into this scheme in a natural way via an additional outermost sequential loop, and if element types are not mixed, colour groups coincide with distributions. Finally, the distribution loop may be executed in

parallel or sequentially, depending on the (relative) size of the various distributions $D_l \subset \mathscr{T}_h$.

The next pseudocode snippet illustrates the loop ordering that stems from this distribution-based approach, including the mandatory synchronisation points. We introduce an additional blocking layer that can be used to adapt the computation to the hardware at hand. We refer to this algorithmic template as 'sets-first'.

```
For all distributions D_l ⊂ T_h sequentially or in parallel
  Preallocate work memory for a block of B elements of type D_l
  —Barrier synchronisation if outer loop is parallel—
  Compute static, common data for element type D_l, store in work memory
  —Barrier synchronisation if outer loop is parallel—
  For all sets S = {T_{e_1}, ..., T_{e_B}} ⊂ D_l of size B in parallel
    For all elements T ∈ S in parallel
      Loop over the test and trial functions, compute the integral(s)
      using the precomputed data in work memory, and scatter the result
```

3.4 Implementation on Multicore CPUs

On CPU-type architectures, it is not necessary to exploit all algorithmically available degrees of parallelism. Instead, the entire body of the distribution loop can be executed in parallel, equidistributed among all available threads. Load balancing is needed if the cardinality of each distribution is not large enough to keep all available threads busy. The additional blocking layer should be chosen with respect to cache sizes to increase locality and improve efficiency. It is sometimes beneficial to split the loop over all elements: The local element matrices in each set can first be computed independently into work memory, followed by a second nested loop that performs the actual scattering, protected by a 'critical section'.

3.5 Implementation on GPUs

Throughout this Chapter, we use CUDA terminology, but emphasise that the implementation guidelines are equally valid in other programming environments. One important general recommendation is to implement one kernel for each type of distribution, to facilitate clean and reusable code. The individual kernels can additionally be equipped with C++ template metaprogramming and/or preprocessor statements to reduce boilerplate overhead by, e.g., treating all Lagrangian elements on triangles by one meta-kernel. The loop over all distributions and/or colour groups then naturally translates to separate kernel launches.

Within each kernel, the main concern towards an efficient implementation is to translate algorithmic algorithmic to GPU concepts. Examples include the mapping of loop nesting levels to CUDA entities (blocks, warps, single threads), and the mapping to various memory spaces (global, shared, constant, registers). The following characteristics of the CUDA programming model must be taken into consideration: (1) Global synchronisation is only possible at the kernel launch granularity. (2) Threads in a block may synchronise inexpensively, and have access to small, but fast shared memory. (3) Global memory is orders of magnitude slower than registers or shared memory. (4) Instruction divergence within a warp should be avoided. (5) The latter is particularly true for memory instructions, i.e., addresses touched by a single load or store instruction must meet certain alignment criteria so that the hardware can coalesce memory accesses by a single warp into ideally a single memory transaction. (6) Atomic memory updates are not equally efficient on all hardware generations and for all data types.

4 Examples and Applications

In this Section, we provide guidelines for devising generically applicable yet competitive (with respect to performance) mappings of algorithms to the hardware, under the constraints outlined in Sec. 3.5. Our description is based on representative examples: Low-order Lagrangian discretisations are discussed quite generically, high-order (spectral) elements are presented in the scope of a linear wave propagation application, and the edge-based approach is presented for a gas dynamics application. We do not specifically cover the case of Discontinuous Galerkin methods, although its high-order forms are close to the high-order spectral-element method, and refer to [14] instead.

4.1 Low-Order Lagrangian-Type Elements

Low-order methods are widely used in practice. In this Section, no explicit assumptions are made on the problem at hand (2D vs. 3D, time-dependent, (non)linear, scalar vs. multiple fields), but we do assume a single distribution D_l in the loop structure on p. 11, i.e., elements of the exact same type. In the following, we focus on summarising the main ideas to explore the optimisation space. We explicitly do not aim at describing implementations that perform optimally for a specific finite element method on a specific hardware generation for a specific problem only. Further information can be found in [3, 11, 25, 23].

A naive implementation of the ‘sets-first’ approach developed in Sect. 3.3 is straightforward: We simply use the data structures for the (typically, existing) CPU implementation, associate one fixed-size set of elements with a CUDA thread block, and let each thread compute the triple per-element loop. For the final accumulation

into the global matrix, either colouring or atomic memory updates can be used. This implementation generally performs poorly because it does not make good use of the available bandwidth, which, due to the low arithmetic intensity of low-order methods, is the main bottleneck: Memory accesses are unstructured, the effective bandwidth is low because coalescing into a minimal amount of memory transactions per warp may be poor, and the comparatively small cache does not enable automatic reuse of, e.g., nodal data that are shared due to common support.

Improved data structures The first optimisation we highlight aims at improving memory access patterns only. We arrange all input data into a column-major 2D array, indexed by elements. In each row of this structure, we store nodal data associated with one element: The first few entries contain static data such as per-element nodal coordinates, the next few entries are associated with target indices in the matrix/vector data structure to be assembled into (plus eventually some padding), and the final optional entries are associated with dynamic data. It depends on the problem at hand whether dynamic fields evaluations (transport directions depending on coordinates in convection-diffusion-equations, the $\mathbf{u} \cdot \nabla \mathbf{u}$ term in the non-linear Navier-Stokes equations, cf. Sect. 1) are best copied here, or instead read from their original locations. This data structure can be built from information readily available in the finite element framework.

For the actual computation, we can now associate each thread with one element. The threads execute 'their' triple-nested loop to compute an element matrix each, by iterating over this data structure. All data required by consecutive threads are stored contiguously in memory by construction, and the column-major layout ensures that all memory accesses are fully coalesced, i.e., no bandwidth is wasted. This approach implies certain redundancies, as actual data instead of just indices are stored several times.

The final step is to scatter the element matrices to the target matrix/vector in global memory. There are several ways to implement this: The easiest one is to use atomic memory updates for all data, but such an approach may be slow, especially on older (for NVIDIA, pre-Kepler) hardware generations. Alternatively, a two-pass strategy can be used. All computed element matrices are first written to global memory in a contiguous way, and a second kernel is invoked that performs the actual assembly. In this kernel, each thread is responsible for one target entry in the matrix, gathers all required data from the global array of element matrices, and performs a sequential reduction to compute the final entry. In this case, there is no need to store target indices in the 2D data structure. Instead, an additional 'reduction list' is precomputed, which for each nonzero entry contains the indices in the array of element matrices that need to be gathered. To be more precise, this list typically starts with the target index, and the number of subsequent source indices depends on the connectivity of the mesh. Padding up to a fixed maximum size (amount of elements influencing one node) with 'negative indices' is a standard way to ensure coalesced memory accesses into this list and SIMT computation during the reduction. Fig. 2 (left) illustrates the data flow of such a reduction.

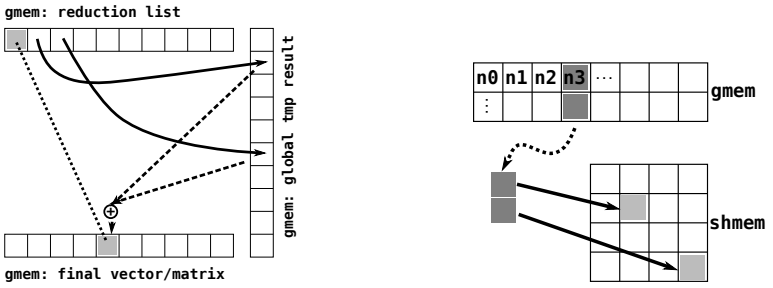

Fig. 2 Left: Data flow through reduction lists: While index lookups (including the highlighted target index) are perfectly coalesced, the gathering step is not. Right: Shared memory implementation, data flow of the lookup of the last node of the first element (2D bilinear quadrilaterals) treated by each warp (only two threads shown). The lookup into the map is perfectly coalesced, only accesses to shared memory are irregular.

This approach is appealing due to its simplicity and versatility: In essence, all GPU-related challenges (regarding input data) are resolved by introducing redundancy in the underlying data structure, which in turn can be easily precomputed in standard finite element programs. The additional storage is well-invested, in particular for schemes where the assembly is invoked several times. Also, indices require the same storage as actual values in single precision. However, the approach does not exploit the fact that several neighbouring elements share nodal data due to the local support property.

Improved data sharing The following description of a (more involved) approach that uses shared memory to facilitate data reuse builds upon a proposal by Cecka et al. [3]. The basic aim is to preserve generality while increasing performance.

In a preprocessing phase, we partition the elements into sets of approximately equal size such that (1) the total number of element neighbours is minimised across partitions and (2) all required data (defined below) for one partition fit into shared memory. A tunable parameter is the 'size' of shared memory: Data reuse and the number of resident warps to improve latency hiding need to be balanced. A small safety factor should be added to the partition size because CUDA uses a small amount of shared memory for itself, e.g., for kernel parameters. This constrained optimisation problem is commonly encountered in distributed memory parallelisation techniques for unstructured meshes, and standard graph partitioning software such as METIS [12] can be used to compute feasible approximations. Nodal data associated with each partition is then stored contiguously in memory.

We associate one thread block with one partition, and choose a number of threads that evenly divides the number of elements to be computed (associated with the partition), rounded up to the next multiple of the warp size. The first phase of the kernel uses all threads to load all nodal data of the entire partition into shared memory. Since the nodal data are contiguous in global memory, it is easy to find a mapping of threads to memory locations that ensures a fully coalesced access pattern. We

now need to find a mapping of global node numbers to their indices in shared memory. To this end, we precompute an auxiliary 2D array per block, quite similar to the data structure described above. In it, we store the local indices for each element in a column-major fashion. This data structure can be stored in global or in shared memory; the former is more advisable because (1) data accesses to it are always fully coalesced and (2) the more actual nodal data we can store in shared memory, the higher the benefits from data sharing.

The second phase of the kernel then iterates over this data structure: Each thread looks up an index, accesses the data in shared memory, and as soon as all nodal data for the computation of one element matrix are available, it is computed (e.g., after every fourth memory access for the Laplace operator and bilinear elements on quadrilaterals). All operations per warp are executed in lock-step, and irregular memory accesses are limited to shared memory (where they constitute less of a problem), as shown in Fig. 2 (right).

For the actual assembly into the global matrix/vector, we can again employ the two strategies outlined above, or a third one that avoids the overhead of a two-pass solution: We again exploit the local support property. For all 'inner' elements of the partition, we know that no other element from any other partition will influence a nonzero entry. Therefore, we can also perform the reduction directly in shared memory in the same kernel without resorting to a two-pass strategy, and for all other elements, we can use atomic memory updates. It depends on the hardware and on the finite element method which implementation performs best.

We conclude this Section by referring to Cecka et al. [3, Sect. 4.3], who propose an approach that uses the nonzeros-first strategy within each set of elements. This technique is quite involved, and it tends to require quite problem-specific implementation adjustments to achieve performance.

4.2 High-Order Spectral Element Discretisations for Wave Propagation

Let us now discuss the case of high-order methods by focusing on the spectral element method, which is a variant of the FEM that is well-suited for instance for wave propagation modelling in heterogeneous media [29], e.g., seismic waves in the Earth [24], ultrasonic acoustic waves in ocean acoustics [5] or non-destructive testing [30]. Let us consider a linear anisotropic elastic rheology for a solid model, in which case the differential form of the acoustic wave equation can be written as

$$\begin{aligned} \rho\ddot{\mathbf{u}} &= \nabla\cdot\sigma+\mathbf{f}\,, \\ \sigma &= \mathbf{C}:\varepsilon\,, \\ \varepsilon &= \tfrac{1}{2}[\nabla\mathbf{u}+(\nabla\mathbf{u})^T]\,, \end{aligned} \tag{14}$$

where $\mathbf{u}$ is the displacement vector, σ the symmetric, second-order stress tensor, ε the symmetric, second-order strain tensor, $\mathbf{C}$ the fourth-order stiffness tensor, ρ the

density, and $\mathbf{f}$ an external force representing the acoustic or seismic source. Denoting the physical domain of the model and its boundary by Ω and Γ respectively, we can write the weak form of this equation by dotting it with an arbitrary test function $\mathbf{w}$ and integrating by parts over the whole domain:

$$\int_\Omega \rho\, \mathbf{w}\cdot\ddot{\mathbf{u}}\,\mathrm{d}\Omega + \int_\Omega \nabla\mathbf{w} : \mathbf{C} : \nabla\mathbf{u}\,\mathrm{d}\Omega = \int_\Omega \mathbf{w}\cdot\mathbf{f}\,\mathrm{d}\Omega + \int_\Gamma (\boldsymbol{\sigma}\cdot\hat{\mathbf{n}})\cdot\mathbf{w}\,\mathrm{d}\Gamma \qquad (15)$$

The contour integral of the last term vanishes because of the free surface, i.e., traction-free boundary condition: the traction vector $\boldsymbol{\tau} = \boldsymbol{\sigma}\cdot\hat{\mathbf{n}}$ is zero at the surface.

The physical domain is subdivided into hexahedral mesh cells within which variables are approximated by high order interpolants. The SEM resorts to Lagrange polynomials of degree $n = 4$ to 8 to interpolate functions such as the unknown displacement field [6, 27]. The anchor points are most of the time chosen as the $n+1$ Gauss-Lobatto-Legendre (GLL) points because the mass matrix then becomes perfectly diagonal, which in turn leads to the use of fully explicit time schemes [29], e.g., a second-order Newmark or a fourth-order Runge-Kutta scheme. Consequently, the method is by design very efficient on large parallel computers [24, 26]. Numerical integration over the elements is performed using a GLL integration rule, and thus each spectral element contains $(n+1)^3$ such GLL integration points. The final matrix system to solve is

$$\mathrm{M}\ddot{\mathrm{U}} + \mathrm{K}\mathrm{U} = \mathrm{F}\,, \qquad (16)$$

where U is the unknown displacement vector that needs to be computed, M is the diagonal mass matrix, K is the stiffness matrix, and F is the source term, whose detailed expressions can be found for instance in [29] and [24].

In almost all wave propagation applications a large number of time steps is performed, and thus in the SEM algorithm the total cost is dominated by the contents of the serial time loop. In addition, since the mesh is static and the algorithm is fully explicit, all the time steps have identical cost, which facilitates optimisation. The computations performed at each time step consist of two very different kinds of operations: global vector updates, whose goal is to march the global vector of unknowns in time, and local matrix-matrix products inside each spectral element followed by an 'assembly' phase, whose goal is to perform the local elastic force calculations and sum them into the global elastic force vector to be able to compute the acceleration vector at the next time step.

Operations of the first kind are of the typical type $\mathbf{u}^{\mathrm{new}} = \mathbf{u}^{\mathrm{old}} + \Delta t\,\dot{\mathbf{u}} + \frac{\Delta t}{2}\,\ddot{\mathbf{u}}$, where $\mathbf{u}$, $\dot{\mathbf{u}}$ and $\ddot{\mathbf{u}}$ are the global displacement, velocity and acceleration vectors, respectively, and Δt is the time step. They are all trivially parallel, and thus a simple CUDA implementation with a thread per degree of freedom to update is sufficient because it contains no dependencies. The second step is by far the more complex one and consists mostly of local matrix products performed inside each element to compute its contribution to the stiffness matrix term KU of (16) according to (15). The global displacement vector is first copied into each spectral element using a local-to-global mesh numbering mapping that has been precomputed and stored before the beginning of the time loop. Small matrix products are then performed between

a derivative matrix, whose components are the derivatives of the Lagrange polynomials at the GLL points, and the displacement vector $\mathbf{u}$ in 2D cut planes along the three local topological directions (i, j, k) of the spectral element. The computed local values are then summed at global mesh points using the local-to-global mesh numbering mapping in order to compute the acceleration vector $\ddot{\mathbf{u}}$. This 'assembly process' must in principle imply an atomic sum because different elements add to the same memory location of a global array.

This can be analysed more precisely by recalling that in each of the three spatial directions the Lagrange interpolants, defined on $[-1,1]$, are built from the GLL points, which include the boundary points -1 and +1. For polynomial basis functions of degree n, there are $n+1$ GLL quadrature points, and thus there are $n-1$ interior points in addition to +1 and -1. In three dimensions, out of the $(n+1)^3$ GLL points that each spectral element comprises, there are thus $(n-1)^3$ interior points that are not shared with neighbouring elements in the mesh, and $(n+1)^3-(n-1)^3$ that may be shared (and will very often be shared in practice) with neighbouring elements of the unstructured mesh through a common face, edge or corner. In acoustic wave propagation we use a polynomial degree $n=4$ because it has been shown to provide an optimal trade-off between accuracy and cost [6, 27]. Thus, out of the 125 GLL points of each spectral element, only 27 are interior and not shared, and 98, i.e. a vast majority, are shared with other elements.

Since the contributions to the elastic force vector are calculated locally and independently inside each spectral element before being summed at the potentially shared points, we decide to assign a different thread to each of the 125 points of each spectral element. We thus handle a spectral element with a block of 128 threads (4 warps) because using a multiple of the 32-thread warp size is best, and use one thread per GLL quadrature point. Therefore, 125 out of 128 threads perform actual work, while three are purposely unused and idle. We first copy the global displacement vector corresponding to each element into shared memory using the global-to-local mapping. The derivative matrix of the Lagrange polynomials is stored in constant memory, and the CUDA kernel then multiplies it with the coefficients in shared memory, at the GLL points.

The derivative matrices have size $(n+1)\times(n+1)$, i.e., 5×5. We inline these small matrix products manually (they are too small to be efficiently handled by BLAS3 calls), and store them in constant memory to take advantage of its faster access times and cache mechanism: It is as fast as registers if all threads access the same item simultaneously.

In order to maximise efficiency, we also apply a number of optimizations that are specific to CUDA: we arrange data so that accesses to local data stored in global memory can be coalesced into large memory transactions, and we try to avoid bank conflicts in shared memory. However, there are two important limitations in this crucial and dominant part of spectral element calculations: first, it is memory-bound because it performs a relatively large number of memory accesses compared to a relatively small number of calculations, owing to the small size of the matrices involved. Second, indirect local-to-global addressing is required because of the unstructured nature of the mesh, which unavoidably leads to some uncoalesced memory access

patterns. On recent hardware, we nonetheless measure very good throughput of this calculation kernel because coalesced memory accesses are an issue that is far less critical than in the past. To further improve performance we perform these global accesses through the texture cache, but the gain is small, as expected.

The final key issue is to decide how to best handle the summation of all the elastic forces, local to each spectral element, into the global vector of elastic forces, in which many global points are shared between adjacent elements as seen above. In principle, this sum could simply be atomic, the only requirement in order to get correct results being to ensure that different warps never update the same shared location simultaneously. In this application, mesh colouring (cf. Sect. 3.1) has been shown to be more efficient. To do so, we partition the mesh elements into a finite number of disjoint subsets, with the property that any two elements in a given subset do not share any global mesh nodes. Data at these nodes can therefore be added to their corresponding global location without any possibility of access conflict, thus removing the need for an atomic locking mechanism. Mesh colouring is performed once and for all on the host in a preprocessing stage during the meshing step by pre-computing maximally independent sets of mesh elements. Adding an outer serial loop over the mesh colours, each colour is then simply handled through a call to the CUDA calculation kernel, resulting in one kernel call per colour. This is acceptable if (and only if) the total number of colours for a given mesh remains reasonable, which is always the case in practice. Tests not shown here show that for unstructured finite element meshes we typically need 10–30 colours.

This approach is implemented in the SPECFEM3D software, and the full source code is available at www.geodynamics.org.

4.3 Group FEM for Gas Dynamics

The final example deals with the gas dynamic equations, modelled by a first-order hyperbolic system of non-linear coupled equations that can be written in the divergence form (2). In particular, it expresses the conservation laws for the mass, momentum, and energy of an inviscid compressible fluid. That is,

$$U = \begin{bmatrix} \rho \\ \rho\mathbf{v} \\ \rho E \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\otimes\mathbf{v} + p\mathbb{I} \\ \rho E\mathbf{v} + p\mathbf{v} \end{bmatrix} = \mathbf{v}U + \begin{bmatrix} 0 \\ \mathbb{I} \\ \mathbf{v} \end{bmatrix} p\,, \tag{17}$$

where ρ is the density, $\mathbf{v} = (v^1, v^2, v^3)$ is the three-dimensional velocity vector, E is the total energy and $\mathbb{I}$ is the 3×3 identity tensor. The equation of state

$$p = (\gamma - 1)\left(\rho E - 0.5\rho\|\mathbf{v}\|^2\right)$$

for an ideal polytropic gas with, e.g., $\gamma = 1.4$ for air, is used to relate the pressure p to the conserved quantities.

A general class of high-resolution methods for the compressible Euler equations was introduced in [20] and refined in a series of publications. The interested reader is referred to [17, 18] and the references therein for a detailed description of the state of the art of so-called *algebraic flux correction* (AFC) schemes for hyperbolic systems. Here, the focus lies on their efficient implementation on GPUs. The following presentation is partly based on algorithms implemented in the open-source software package Featflow2 (http://www.featflow.de).

Let system (17) be discretised in space by Fletcher's group finite element formulation [9] as outlined in Sect. 2.4. The resulting semi-discrete problem reads

$$M\frac{\mathrm{d}\mathrm{U}}{\mathrm{d}t} = R(\mathrm{U}) \tag{18}$$

where the entries of the right-hand side $R = \{\mathrm{R}_i\}$ according to (10) are given by

$$\mathrm{R}_i = -\sum_{j=1}^{N} \mathbf{c}_{ij} \cdot \mathbf{F}_j \,. \tag{19}$$

It serves as a base scheme for many high-resolution methods, but it gives rise to non-physical undershoots and overshoots in the vicinity of discontinuities such as shock waves, and hence, it is not applicable per se. A common stabilisation strategy consists in adding artificial viscosity to prevent the creation of wiggles, and ideally, to ensure that physical quantities such as the density and pressure variables remain positive. In the framework of AFC-schemes [20, 17, 18] this is achieved by replacing the consistent mass matrix with its row-sum lumped counterpart

$$M_L = \mathrm{diag}\{m_i\mathbb{I}\}, \qquad m_i = \sum_{j=1}^{N} m_{ij}$$

and augmenting the right-hand side by artificial diffusion and limited antidiffusion

$$\mathrm{R}_i := \mathrm{R}_i + \sum_{j\in\mathscr{S}_i} \mathrm{D}_{ij}(\mathrm{U}_j - \mathrm{U}_i) + \alpha_{ij}\mathrm{F}_{ij} \,. \tag{20}$$

The choices for $D = \{\mathrm{D}_{ij}\}$ given in [20, 17, 18] differ in the arithmetic intensity but use the same input data. Hence, we consider a generic discrete diffusion operator which is defined as a symmetric matrix with zero row- and column sums [21]

$$\mathrm{D}_{ii} := -\sum_{j\in\mathscr{S}_i} \mathrm{D}_{ij}, \qquad \mathrm{D}_{ij} = \mathrm{D}_{ij}(\mathbf{c}_{ij}, \mathbf{c}_{ji}, \mathrm{U}_i, \mathrm{U}_j) = \mathrm{D}_{ji} \,. \tag{21}$$

In the scope of this Chapter, it also suffices to consider the skew-symmetric antidiffusive fluxes F_{ij} and the symmetric limiting coefficients $\alpha_{ij} \in [0,1]$ to be functions that depend on the precomputed coefficients $m_{ij}, \mathbf{c}_{ij}$ and $\mathbf{c}_{ji}$, and on the dynamically changing data $\mathrm{U}_i, \mathrm{U}_j$, $\mathrm{d}\mathrm{U}_i/\mathrm{d}t$ and $\mathrm{d}\mathrm{U}_j/\mathrm{d}t$, and their edge-neighbouring values. All algorithmic details can be found in [17, 18]. The treatment of boundary conditions is a non-trivial task that cannot be addressed here due to space constraints. Thus,

one should bear in mind that additional terms need to be computed at the boundary, which, however, consumes only a negligible fraction of the overall computing time even in case the most naive implementation is adopted.

Integrating the semi-discrete form (18) in time by the two-level θ-scheme yields

$$M\frac{\mathrm{U}^{n+1}-\mathrm{U}^{n}}{\Delta t}=\theta R(\mathrm{U}^{n+1})+(1-\theta)R(\mathrm{U}^{n}) \tag{22}$$

which is non-linear for $\theta\in(0,1]$, and hence, needs to be computed iteratively, e.g., by successive approximations [20, 17], or linearised based on a Taylor series expansion [18, 28]. In any case, the left-hand side has the form $[M/\Delta t-\theta P]\mathrm{U}$, where

$$P=\{\mathrm{P}_{ij}\},\qquad \mathrm{P}_{ij}=\mathrm{K}_{ij}+\mathrm{D}_{ij}+\text{boundary contributions} \tag{23}$$

is an approximation such that $P(\mathrm{U})\mathrm{U}\approx R(\mathrm{U})$ or to its Jacobian, i.e. $P(\mathrm{U})\approx\frac{\partial R(\mathrm{U})}{\partial \mathrm{U}}$. The Galerkin part $\mathrm{K}_{ij}=-\mathbf{c}_{ij}\cdot\mathbf{A}_j$ exploits the so-called homogeneity property of the Euler equations, i.e. $\mathbf{F}_j=\mathbf{A}_j\mathrm{U}_j$, where $\mathbf{A}_j=\frac{\partial\mathbf{F}}{\partial U}(\mathrm{U}_j)$ is the nodal value of the flux Jacobians. In summary, the core components to be implemented on GPUs are:

- Vector assembly procedures for $R(\mathrm{U})$ based on (19) or (20)
- Matrix assembly procedures for operator (23) for the Galerkin scheme ($D\equiv 0$) and in the presence of artificial viscosities ($D\neq 0$)

Vector assembly The assembly of the right-hand side vector (19) for the Galerkin discretisation is straightforward because implementation techniques from sparse matrix vector multiplication (SpMV, [1]) can be readily adapted. Let the coefficient matrices $\mathbf{C}=(C^1,C^2,C^3)$ be stored in a GPU-friendly matrix format in global memory, then the multiply-add operation in the standard (scalar) SpMV-kernel is replaced by $\mathrm{+=}\sum_{d=1}^{3}c_{ij}^{d}F^{d}(\mathrm{U}_j)$, where $F^{d}(\cdot)$ stands for arithmetic operations according to (17). It should be noted that the computation of a single field variable of the target vector R, say total energy, requires all five field variables from the input vector U. It is therefore advisable to invest some amount of shared memory to store the relevant parts of U. After the synchronisation of all threads of the CUDA thread block, each entry in the destination vector is then computed by one thread.

The right-hand side (20) is assembled differently by resorting to one of the edge-based formulations (12) or (13) to maximise data reuse of the input solution vector. Without loss of generality, let us present an edge-by-edge assembly based on the second variant. In particular, the contribution of edge ij needs to be computed only once and can then be scattered to positions i and j as follows:

$$\mathrm{R}_i := \mathrm{R}_i+\mathrm{G}_{ij}+\mathrm{D}_{ij}(\mathrm{U}_j-\mathrm{U}_i)+\alpha_{ij}\mathrm{F}_{ij} \tag{24}$$

$$\mathrm{R}_j := \mathrm{R}_j-\mathrm{G}_{ij}-\mathrm{D}_{ij}(\mathrm{U}_j-\mathrm{U}_i)-\alpha_{ij}\mathrm{F}_{ij} \tag{25}$$

The strategy from Sect. 4.1 to optimise the memory access pattern of the element-based assembly can be easily adapted to the edge-based procedure. The column-major 2D array, which remains unchanged for fixed meshes, is indexed by the edge number ij and contains the precomputed coefficients m_{ij}, $\mathbf{c}_{ij}$ and $\mathbf{c}_{ji}$ and the integer

values i and j which serve both as source and target indices in the solution and right-hand side vector, respectively. A fixed-sized set of edges is then associated with a CUDA thread block that implements (24)–(25) using either atomic memory updates or the suggested two-pass strategy with 'edges' instead of 'elements'. In our current implementation, the solution values are directly read from their original location instead of copying them into the 2D data structure.

As an alternative, the static data structure can be reordered and partitioned based on an edge-colouring algorithm (cf. Sect. 3.1), accompanied by a permutation of the edge numbering. This yields a contiguous storage of exactly those edges which have no common start and end points, and that can therefore be processed independently without synchronisation. Following Sects. 3.5 and 4.1, one sequentially-launched CUDA kernel is used for all edges of the same colour. One potential drawback of this colouring strategy is that data reuse is precluded by construction. Even worse, the jump $|j-i|$ between the two indices of the edge ij but also the jump between the index pairs i and i' as well as j and j' of the succeeding edge $i'j'$ may become large and lead to extremely unstructured memory accesses. A reordering algorithm that reduces the jumps in memory access is developed in [22]. The remaining terms of the right-hand side of (22) can be handled efficiently by standard SpMV kernels and its extension to interleaved matrices, respectively.

Matrix assembly As for the vector assembly, the assembly of the Galerkin part of the operator (23) is straightforward. Moreover, neither atomic memory updates nor colouring is required to augment the off-diagonal blocks with the artificial viscosity tensor D_{ij} if either the static 2D data structure or the reduction list employed in the multi-pass strategy is extended by the target positions (i,j) and (j,i) in the global matrix. It is only the diagonal entries that give rise to concurrent updates (21).

We therefore propose the following strategy combining the static data structure and the reduction list to maximise data reuse and minimise latency due to atomic memory updates. As before, a fixed-sized set of edges is associated with a CUDA thread block running a single kernel. In the first phase, solution data that is required to process the edges under consideration are gathered into shared memory. Next, *all* local blocks K_{ij} for the Galerkin part and the artificial viscosities D_{ij} with $j \neq i$ are computed and stored in shared memory. In the following phase, $\mathrm{K}_{ij} + \mathrm{D}_{ij}$ is scattered to the non-contentious off-diagonal position (i,j) in global memory based on the static 2D data structure. An additional 'reduction list' is used to calculate the diagonal entries $\mathrm{K}_{ii} - \sum_{j \in \mathscr{S}_i} \mathrm{D}_{ij}$ from the previously computed local data and store the result in global memory. Between each of the different phases, the threads of the CUDA thread block are synchronised. However, it should be noted that this approach has a relatively high demand on shared memory so that a careful tuning of size-parameters is necessary. This is particularly true for compute intensive artificial viscosities such as Roe's approximate Riemann solver which essentially requires two 5×5 matrix-matrix multiplications per edge. If shared memory becomes the limiting factor then this multi-phase approach can be replaced by a multi-pass strategy similar to the one adopted in Sect. 4.1 for the element assembly. That is, the

different tasks are implemented in individual kernels launched for the same fixed-sized set of edges, whereby intermediate results are written to global memory.

Acknowledgements This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 'Software for Exascale Computing' (SPPEXA), through DFG SFB 708 '3D Surface Engineering of Tools for the Sheet Metal Forming—Manufacturing, Modelling, Machining—', by the European 'Mont-Blanc: European scalable and power efficient HPC platform based on low-power embedded technology' #288777 project of call FP7-ICT-2011-7, and by the G8 and French ANR 'Interdisciplinary Program on Application Software towards Exascale Computing for Global Scale Issues' (SEISMIC IMAGING project, ANR-10-G8EX-002). This work was granted access to the high-performance computing resources of the French supercomputing centre CCRT under allocation #2012-046351 awarded by GENCI (Grand Equipement National de Calcul Intensif).

References

1. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing, pp. 18:1–18:11 (2009)
2. Brenner, S., Scott, L.: The Mathematical Theory of Finite Element Methods. No. 15 in Texts in Applied Mathematics. Springer (1994)
3. Cecka, C., Lew, A.J., Darve, E.: Assembly of finite element methods on graphics processors. Int. J. Numer. Methods Engineering **85**(5), 640–669 (2011)
4. Ciarlet, P.: The Finite Element Methods for Elliptic Problems. North-Holland (1978)
5. Cristini, P., Komatitsch, D.: Some illustrative examples of the use of a spectral-element method in ocean acoustics. J. Acoust. Soc. Am. **131**(3), EL229–EL235 (2012)
6. De Basabe, J.D., Sen, M.K.: Grid dispersion and stability criteria of some common finite-element methods for acoustic and elastic wave equations. Geophysics **72**(6), T81–T95 (2007)
7. Diestel, R.: Graph Theory, *Graduate Texts in Mathematics*, vol. 173, 4th edn. Springer (2010)
8. Donea, J., Huerta, A.: Finite Element Methods for Flow Problems. John Wiley & Sons (2003)
9. Fletcher, C.: The group finite element formulation. Comp. Methods Applied Mechanics Engineering **37**, 225–243 (1983)
10. Hughes, T.J.R.: The finite element method, linear static and dynamic finite element analysis. Prentice-Hall International (1987)
11. Huthwaite, P.: Accelerated finite element elastodynamic simulations using the GPU. J. Comput. Phys. **257**, 687–707 (2014)
12. Karypis, G., Kumar, V.: A fast and high-quality multilevel scheme for partitioning irregular graphs. SIAM J. Scientific Computing **20**(1), 359–392 (1998)
13. Kirby, R.C., Logg, A.: A compiler for variational forms. ACM Trans. Math. Software **32**(3), 417–444 (2006)
14. Klöckner, A., Warburton, T., Bridge, J., Hesthaven, J.S.: Nodal discontinuous Galerkin methods on graphics processors. J. Comput. Phys. **228**(21), 7863–7882 (2009)
15. Knepley, M.G., Terrel, A.R.: Finite element integration on GPUs. ACM Trans. Math. Software **39**(2) (2013)
16. Komatitsch, D., Michéa, D., Erlebacher, G.: Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. J. Parallel Distributed Computing **69**(5), 451–460 (2009)
17. Kuzmin, D., Möller, M.: Flux-Corrected Transport: Principles, Algorithms, and Applications, 1st edn., chap. Algebraic Flux Correction II: Compressible Flow Problems, pp. 207–250. Scientific Computation. Springer (2005)

18. Kuzmin, D., Möller, M., Gurris, M.: Flux-Corrected Transport: Principles, Algorithms, and Applications, 2nd edn., chap. Algebraic Flux Correction II: Compressible Flow Problems, pp. 193–238. Scientific Computation. Springer (2012)
19. Kuzmin, D., Möller, M., Turek, S.: Multidimensional FEM-FCT schemes for arbitrary time stepping. Int. J. Numer. Methods Fluids **42**(3), 265–295 (2003)
20. Kuzmin, D., Möller, M., Turek, S.: High-resolution FEM-FCT schemes for multidimensional conservation laws. Comp. Methods Applied Mechanics Engineering **193**, 4915–4946 (2004)
21. Kuzmin, D., Turek, S.: Flux correction tools for finite elements. J. Comput. Phys. **175**, 525–558 (2002)
22. Löhner, R.: Cache-efficient renumbering for vectorization. Int. J. Numer. Methods Biomedical Engineering **26**, 628–636 (2008)
23. Markall, G., Slemmer, A., Ham, D., Kelly, P., Cantwell, C., Sherwin, S.: Finite element assembly strategies on multi-core and many-core architectures. Int. J. Numer. Methods Fluids **71**(1), 80–97 (2013)
24. Peter, D., Komatitsch, D., Luo, Y., Martin, R., Le Goff, N., Casarotti, E., Le Loher, P., Magnoni, F., Liu, Q., Blitz, C., Nissen-Meyer, T., Basini, P., Tromp, J.: Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes. Geophys. J. Int. **186**(2), 721–739 (2011)
25. Plaszewski, P., Banas, K., Maciol, P.: Higher order FEM numerical integration on GPUs with OpenCL. In: International Multiconference on Computer Science and Information Technology, pp. 337–342 (2010)
26. Rietmann, M., Messmer, P., Nissen-Meyer, T., Peter, D., Basini, P., Komatitsch, D., Schenk, O., Tromp, J., Boschi, L., Giardini, D.: Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. In: Proceedings of the SC'12 ACM/IEEE conference on Supercomputing, pp. 38:1–38:11 (2012)
27. Seriani, G., Oliveira, S.P.: Dispersion analysis of spectral-element methods for elastic wave propagation. Wave Motion **45**, 729–744 (2008)
28. Trépanier, J.Y., Reggio, M., Ait-Ali-Yahia, D.: An implicit flux-difference splitting method for solving the Euler equations on adaptive triangular grids. Int. J. Numer. Methods Heat Fluid Flows **3**, 63–77 (1993)
29. Tromp, J., Komatitsch, D., Liu, Q.: Spectral-element and adjoint methods in seismology. Comm. Comput. Phys. **3**(1), 1–32 (2008)
30. van Wijk, K., Komatitsch, D., Scales, J.A., Tromp, J.: Analysis of strong scattering at the micro-scale. J. Acoust. Soc. Am. **115**(3), 1006–1011 (2004)