



The HighPerMeshes framework for numerical algorithms on unstructured grids

Samer Alhaddad¹ | Jens Förstner¹ | Stefan Groth²  | Daniel Grünwald³ |
 Yevgen Grynko¹ | Frank Hannig²  | Tobias Kenter¹ | Franz-Josef Pfreundt³ |
 Christian Plessl¹ | Merlind Schotte⁴ | Thomas Steinke⁴ | Jürgen Teich² |
 Martin Weiser⁴ | Florian Wende⁴

¹Paderborn Center for Parallel Computing and Department of Computer Science and Department of Electrical Engineering, Paderborn University, Paderborn, Germany

²Hardware/Software Co-Design, Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany

³Fraunhofer Institut für Techno- und Wirtschaftsmathematik, Kaiserslautern, Germany

⁴Zuse Institute, Berlin, Germany

Correspondence

Stefan Groth, Chair of Computer Science 12, Cauerstr. 11, 91058 Erlangen, Germany.
 Email: stefan.groth@fau.de

Summary

Solving partial differential equations (PDEs) on unstructured grids is a cornerstone of engineering and scientific computing. Heterogeneous parallel platforms, including CPUs, GPUs, and FPGAs, enable energy-efficient and computationally demanding simulations. In this article, we introduce the HighPerMeshes C++-embedded domain-specific language (DSL) that bridges the abstraction gap between the mathematical formulation of mesh-based algorithms for PDE problems on the one hand and an increasing number of heterogeneous platforms with their different programming models on the other hand. Thus, the HighPerMeshes DSL aims at higher productivity in the code development process for multiple target platforms. We introduce the concepts as well as the basic structure of the HighPerMeshes DSL, and demonstrate its usage with three examples. The mapping of the abstract algorithmic description onto parallel hardware, including distributed memory compute clusters, is presented. A code generator and a matching back end allow the acceleration of HighPerMeshes code with GPUs. Finally, the achievable performance and scalability are demonstrated for different example problems.

KEYWORDS

code generation, distributed computing, domain-specific languages, numerical algorithms

1 | INTRODUCTION

Simulations of physical systems described by Partial Differential Equations (PDEs) are the cornerstone of computational science and engineering. The ever-increasing number and scale of simulations have led to the rise of different and heterogeneous parallel computing platforms, ranging from multicore CPUs to parallel distributed systems to GPUs and FPGAs. Adapting and implementing complex simulation algorithms on these different architectures is a demanding task requiring in-depth computer science knowledge. Consequently, many large-scale simulation codes address only a narrow and often traditional range of computing environments, missing the performance opportunities offered by new architectures.

In this article, we present the HighPerMeshes embedded Domain-Specific Language (DSL) providing the right abstraction layer to C++ application developers to implement efficient mesh-based algorithms for PDE problems on unstructured grids. The focus of the DSL is on finite element

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.

© 2021 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

(FE) and discontinuous Galerkin (DG) or finite volume (FV) discretizations to address iterative and matrix-free solvers as well as time stepping schemes. Large parts of PDE simulation problems thus can be covered. HighPerMeshes draws heavily on the C++17 standard and template metaprogramming for genericity and extensibility. Additionally, compile-time information through template parameters can benefit the code generation for specific target architectures. Furthermore, we address the acceleration of HighPerMeshes with GPUs. To stay as general as possible, we use OpenCL[†] as a back end, which allows targeting various GPUs and other heterogeneous architectures such as FPGAs. For this purpose, we provide a code generator that produces the necessary OpenCL code from HighPerMeshes code and a back end that allows executing the generated code.

2 | THE HIGHPERMESHERS DOMAIN-SPECIFIC LANGUAGE

Picking the right abstraction level is central for every DSL or library interface targeting mesh-based algorithms for PDEs. On the one hand, it needs to provide idioms for specifying the algorithmic building blocks on an abstraction level that allows an efficient mapping to different computing platforms. On the other hand, it should be detailed enough to allow implementing a wide range of established or yet to be developed discretization schemes and numerical algorithms. The HighPerMeshes DSL aims at providing abstractions on a level that is just high enough to allow for an efficient mapping to sequential and multithreaded CPU execution, distributed memory systems, and accelerators. On this level, the core components of mesh-based PDE algorithms include mesh data structures, the association of Degrees of Freedom (DoFs) to mesh entities such as cells and vertices, and the definition of kernel functions that encapsulate local computations with shape functions defined on single mesh cells or faces.

2.1 | Mesh interface

Computational meshes decompose the computational domain $\Omega \subset \mathbb{R}^d$ into simple shapes such as triangles or tetrahedra by which PDE solutions can be represented. Unstructured meshes do so in an irregular pattern that can be adapted to complex geometries or local solution features in a flexible way. Unlike structured meshes, neighborhood relations between these cells are not implied by the storage arrangement of their constituting vertices, but are usually defined through connectivity lists that specify these neighborhoods. Therefore, the storage efficiency of unstructured meshes can be very low if the specifics of the hardware architecture are not taken into account. Similarly, when accessing or iterating over mesh entities (cells, faces, edges, and vertices for $d = 3$), the memory structuring and arrangement of, for example, geometrically neighboring entities can be critical to performance and present optimization targets on the mesh implementation for different architectures.

The construction of a mesh in the HighPerMeshes DSL starts from a set of vertices $V = \{\mathbf{v}_m \in \mathbb{R}^d\}$ and a set $C = \{i_n \mid n = 0, \dots, \#\text{cells} - 1\}$ of connectivity lists $i_n \subset \{0, \dots, |V| - 1\}$ representing the cells in the mesh. Users can create meshes by providing V and C directly or by using one of the available import parsers for common mesh data files. Each $i \in C$ references into the vertex set V to encode an entity of the cell dimensionality $d_{\text{cell}} \leq d$. Subentities or constituting entities like edges and faces correspond to index sets $j \subset i \in C$ that are deduced according to a particular scheme that is specific to the entity type. All entities are stored in a $(d_{\text{cell}} + 1)$ -dimensional set data-structure using their index sets. In this way, the duplication of subentities is avoided, and each entity can be assigned a unique identifier (ID) so that finding a specific one through its vertices can happen in logarithmic time complexity. In addition, our mesh implementation manages a lookup table which for each entity holds the IDs of all its constituting entities with one dimension lower, and another one with the IDs of all incident super-entities, if present.

2.2 | Buffer types for storing coefficient vectors

PDE solutions are generally discretized using finite-dimensional ansatz spaces and are represented by coefficient vectors with respect to a certain basis. In FE, FV, and DG methods, the basis functions are associated with mesh entities and have a support contained in the union of the cells incident to their entity. The mapping of coefficients, or DoFs, to storage locations and access to them depends on the target architecture and may involve nontrivial communication. Therefore, the DSL provides buffer types for coefficient vector storage to relieve the user from these considerations.

Depending on the ansatz space, a particular number of basis functions is associated with mesh entities of different dimensions. Therefore, the number of coefficients $\eta_{\vec{d}}$ associated with entities of dimension $\vec{d} \in \{0, \dots, d_{\text{cell}}\}$ has to be specified when constructing a buffer. Additionally, global values as coefficients of the constant basis function can be stored, for example,

[†]<https://www.khronos.org/opencl/>.

```

Runtime hpm{..};
auto dofs = MakeDofs<1,1,1,1,2>(); /*  $\eta = \{\bar{\eta}_d, 2\} = \{1, 1, 1, 1, 2\}$  */
auto buffer = hpm.GetBuffer<float>(mesh, dofs);

```

for $d_{\text{cell}} = d = 3$. The buffer holds one value of type `float` for each node, edge, face, and the cell itself. Two additional entries are provided for global values.

DoFs are accessed through a “local-view object” (`local_view` in Listing 1, line 8) inside kernel functions. These local views are a tuple of implementation-specific objects, that are accessible with the `GetDoF` function, that requests DoFs of a certain dimension. This is necessary because access patterns may provide DoFs associated with mesh entities of different dimensions. Given a data access pattern (Section 2.3) and a specific entity—typical program executions loop over all or a subset of the entities in the mesh, one after the other—the corresponding local view makes for a linearly indexable type inside the kernel function, thereby hiding data layout and storage internals.

2.3 | Iterating over the mesh with local kernels

In the PDE solver algorithms that we target, a significant part of PDE computation on meshes involves the evaluation of values, derivatives, or integrals on cells or faces, and is therefore local. This allows for various kinds of parallelization, depending on the target architecture. Typically, these local calculations in space are embedded into time stepping loops or iterative algorithms, which imply dependencies based on the data access patterns of the kernels. With a scheduler that suitably resolves these dependencies, additional parallelism can be exploited by partially overlapping subsequent time steps.

In `HighPerMeshes`, the application developer specifies the calculations as local kernels at entity granularity and invokes a dispatcher to take care of their parallel execution and scheduling. Line 1 of Listing 1 shows the definition of a distributed dispatcher that uses the command line arguments to set up its environment. The advantage of using this dispatcher model is a complete separation of parallelization techniques and kernel definitions. The interface is technology-agnostic, and the user does not need to know the intricate details of parallel and distributed programming.

The dispatcher’s `Execute` method takes a number of kernels to be executed as its arguments. If required, those arguments might be supplemented by a range of time steps as shown in line 4 of Listing 1 in order to iterate the defined sequence of kernels more than once in the specified range. Each kernel must define a range of entities to iterate over. To enable flexible parallelization strategies, the DSL does not guarantee a processing order for these entities. For example, the function call `mesh.GetEntityRange<CellDimension>()` in line 6 specifies that the dispatcher iterates over all cells. `ForEachEntity` in line 5 defines an iteration over all entities in that range. Here, `HighPerMeshes` provides another option: `ForEachIncidence<D>` iterates over all subentities of a certain dimension `D` for the entities in the given range.

The kernel requires a tuple of access definitions, as seen in line 7. Access definitions specify the mode (any of `Read`, `Write`, and `ReadWrite`) and the access pattern for the DoF access. This allows the scheduler to calculate dependencies between kernels, thereby avoiding conflicting DoF accesses in scatter operations despite parallelization. Access patterns determine the DoFs relevant for the calculation by specifying a set of mesh entities incident or adjacent to the local entity. `Cell` in line 7 means that the kernel requires access to the DoFs from the given `buffer` that are associated with the local cell, as frequently used in DG methods. Other common access patterns involve a local cell and all of its incident subentities, usually encountered in FE methods, or the two cells incident to a face for flux computations in DG or FV methods. While `HighPerMeshes` aims at providing all access patterns necessary for common kernel descriptions in FE or DG methods, they can be easily extended by providing the required neighborhood relationship in the mesh interface.

The last argument is a user-defined lambda, that is, an anonymous function (line 8). This lambda defines what is actually computed for each entity in the given range and must be callable with the specified entities, time steps, and a local-view object `local_view` as its arguments. The latter allows access to the requested DoFs.

```

1 DistributedDispatcher dispatcher{argc, argv};
2
3 dispatcher.Execute(
4   Range{100},
5   ForEachEntity(
6     mesh.GetEntityRange<CellDimension>(),
7     tuple(Write(Cell(buffer))),
8     [](const auto& cell, auto step, auto& local_view) { /*kernel body*/ });

```

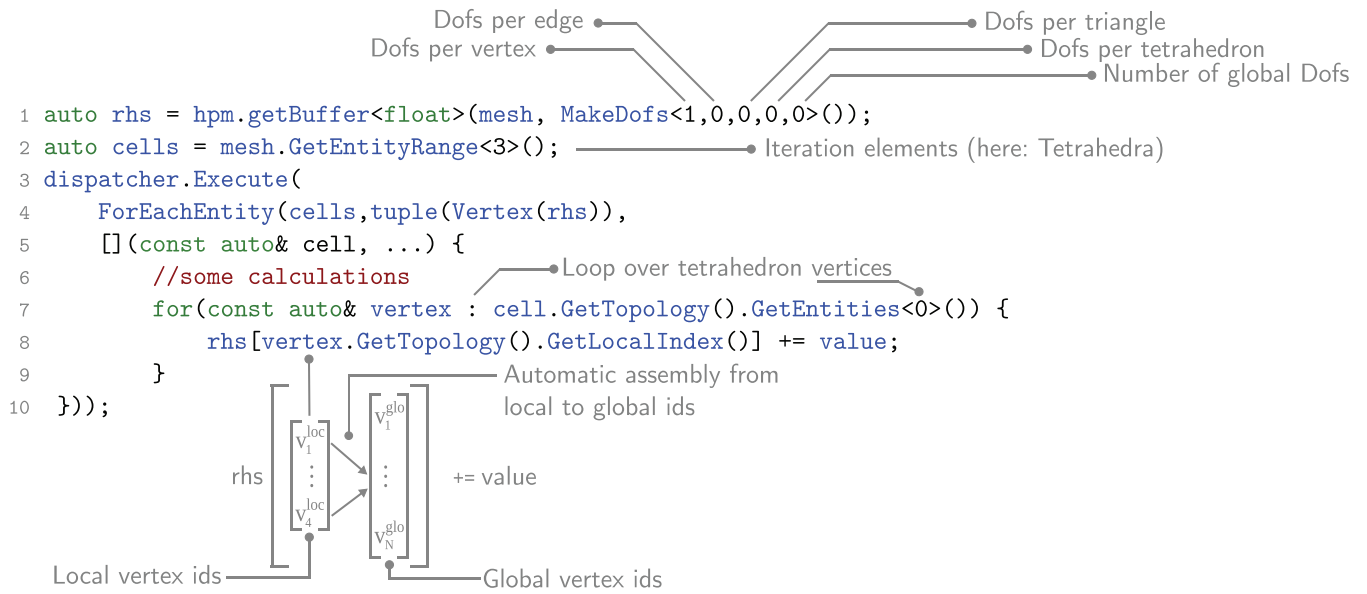
Listing 1: Example of a dispatcher definition and kernel execution

3 | USING THE DSL

In this section, examples and code segments are presented to illustrate the methods described in Section 2 and to explain their use. The examples are elliptic and parabolic differential equations that were numerically solved using the DSL. The grids are mainly irregular 3D simplex meshes, but

TABLE 1 Overview of usage examples

Section	Problem/Method	Local kernels	Solvers
3.1	Poisson, FE method	Matrix-free and rhs assembly	CG method, iterative
3.2	Maxwell's eqs., DG method	Rhs assembly, multiple kernels	RK time stepping
3.3	Monodomain, FE method	Solver assembly	Euler time stepping

**FIGURE 1** Code segment for right-hand side computation

regular 2D simplex meshes are also used. Further information about the algorithms and examples can be found in the public repositories[†]. Table 1 summarizes the features of the presented examples.

3.1 | Matrix-free solver for the Poisson equation

For illustrating the usage of the DSL, the elliptic Poisson problem

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^3, \quad u = 0 \quad \text{on } \Gamma \subset \mathbb{R}^3 \quad (1)$$

with homogeneous Dirichlet boundary conditions is solved by a matrix-free conjugate gradient (CG) method.^{1,2} By discretizing (1) with linear finite elements on a tetrahedralization of Ω , that is, with one DoF per vertex, a system $Ax = b$ of linear equations is obtained.³ Since A is symmetric and positive definite, its solution is the minimizer of the convex minimization problem $F(x) = \frac{1}{2}x^T Ax - b^T x \rightarrow \min$.

In order to solve this linear system of equations, the right-hand side (rhs) b must be assembled. This is done using the buffer datatype and the loop `ForEachEntity`, which iterates over the vertices of each cell (in this case tetrahedra) and stores the corresponding value in the buffer (Figure 1 code line 8).

The homogeneous Dirichlet boundary conditions can be built into the rhs here as well. To solve the system, a matrix-free CG iteration is used. Its main algorithmic building block is the computation of matrix-vector products Ax . Instead of assembling A and performing linear algebra operations, we assemble the product $s = Ax$ directly by evaluating

$$s_j = \sum_j \underbrace{\int_C \nabla \phi_i \nabla \phi_j dC}_{(A_{loc})_{ij}} \cdot x_j \quad (2)$$

[†]<https://github.com/HighPerMeshes/highpermeshes-dsl/examples>, <https://github.com/HighPerMeshes/highpermeshes-drts-gaspi/examples>.

per cell and with ϕ_* as shape functions (see line 5 of Listing 2). To show that the DSL provides a compact syntax, we provide a sketch of an equivalent Matlab implementation in Listing 3 for comparison. Note that the functions `vertexIndicesByCell` and `localStiffnessMatrix` have to be implemented by the user, adding additional programming effort to the Matlab code. We would like to stress that the code, except for some syntax overhead, is very close to the underlying mathematical concepts and completely independent of the target architecture, allowing users without any knowledge of parallel or distributed computing to concentrate on its mathematical structure. Finally, the result can be saved into a file and visualized using, for example, *ParaView*.⁴

```

1 auto AssembleMatrixVecProduct =
2   ForEachEntity(cells, tuple(Vertex(s),Vertex(x)),
3     [](const auto& cell, auto step, auto local_view) {
4       auto& [s, x] = local_view;
5       auto Aloc = localStiffnessMatrix(cell);
6       s += Aloc * x;
7     }
8 );

```

Listing 2: Example of a matrix-free rhs assembly

```

1 function s = AssembleMatrixVecProduct(x)
2   for cell = 1:nCells
3     indices = vertexIndicesByCell(:,cell);
4     Aloc = localStiffnessMatrix(cell);
5     s(indices) = s(indices) + Aloc*x(indices);
6   end

```

Listing 3: Equivalent Matlab sketch of a matrix-free rhs assembly

3.2 | Discontinuous Galerkin time domain (DGTD) Maxwell solver

In the following, we sketch an implementation of a Maxwell solver based on the DGTD numerical scheme.^{5,6} An initial value problem is solved in the time domain in a free space mesh with perfect electric conductor (PEC) boundary conditions. The user can modify the code accordingly if field sources, materials, or absorbing boundaries are needed. The simulation domain is discretized in a triangular or tetrahedral mesh, which is used as an input. Then, DoFs or calculation points are created within the cells, depending on the ansatz order specified by the user. For example, a three-dimensional simulation with third-order accuracy requires 20 DoFs in each cell to represent the unknown fields. The right-hand sides of Maxwell's equations are evaluated during Runge–Kutta time integration at each time step according to the DGTD method formulation

$$\dot{\mathbf{E}} = \mathcal{D} \times \mathbf{H} + (\mathcal{M})^{-1} \mathcal{F} (\Delta \mathbf{E} - \hat{n} \cdot (\hat{n} \cdot \Delta \mathbf{E}) + \hat{n} \times \Delta \mathbf{H}), \quad (3)$$

$$\dot{\mathbf{H}} = -\mathcal{D} \times \mathbf{E} + (\mathcal{M})^{-1} \mathcal{F} (\Delta \mathbf{H} - \hat{n} \cdot (\hat{n} \cdot \Delta \mathbf{H}) + \hat{n} \times \Delta \mathbf{E}). \quad (4)$$

Here $\mathcal{D} \times \mathbf{H}$ and $\mathcal{D} \times \mathbf{E}$ are the curls of the magnetic and electric fields, correspondingly, \mathcal{M} is the mass matrix, \mathcal{F} the face matrix, $\Delta \mathbf{E}$, $\Delta \mathbf{H}$ are field differences between the neighboring cells at the interfaces and \hat{n} the face normal.⁶ The first term (the curls) involves only cell-local DoFs and is therefore called “volume kernel” (see Listing 4).

```

1 auto volumeKernelLoop = ForEachEntity(cells,
2   tuple(Read(Cell(H)), Cell(rhsE), ...),
3   [&](const auto& cell, ..., auto& local_view) {
4     const Mat3D& D = cell.GetGeometry().GetInverseJacobian() * 2.0;
5     ForEach(numVolumeNodes, [&](const int n) {
6       auto& [E, rhsE] = local_view;
7       Mat3D dE;
8       ForEach(numVolumeNodes, [&](const int m) {
9         EH += DyadicProduct(derivative[n][m], E[m]);
10      });
11      rhsE[n] += Curl(D, dE);
12      // code for rhsH: analogue to rhsE
13    });
14  });
15 );

```

Listing 4: Code segment for the Maxwell volume kernel

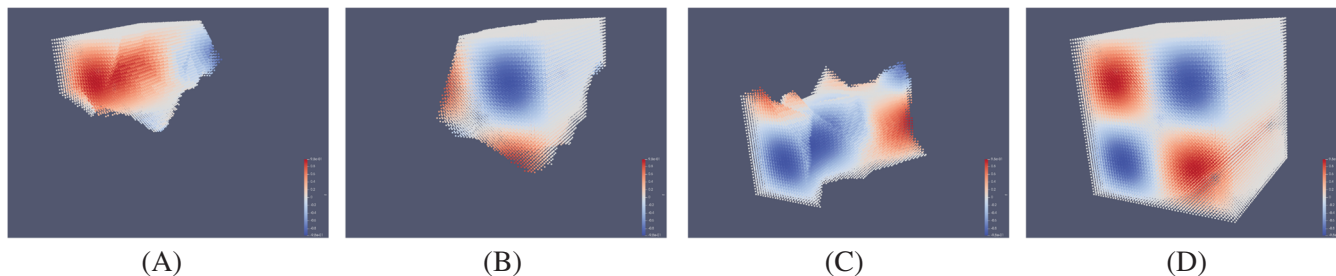


FIGURE 2 The electric field component E_y in the simulation domain. Three parts of the partitioned results can be seen in (A), (B), and (C), while (D) shows the merged result of all four processes

The second term in (3,4), the “surface kernel” (see Listing 5), stems from a surface integral over the cell’s faces. It involves those DoFs from within the two incident cells located on these faces. Calculating the surface kernel requires some operations provided directly by the DSL like `GetNormal()`, `GetAbsJacobianDeterminant()`, ...). The implementation complexity of DG on unstructured meshes comes from the access or mapping to the neighboring cells DoFs in order to calculate fluxes across faces as described in (3) and (4). This access is performed with the data structure `NeighboringNodeMap` (line 15 in Listing 5), which provides the corresponding index for the DoFs in the local view.

In `HighPerMeshes`, the calculated field components in the DoFs can be written completely or selectively to an output file for each time step. The `writeLoop` method comfortably provides this functionality with its specific user defined iteration ranges. Figure 2 shows a visualization of the electric field component E_y in the partitioned simulation domain. For this, the calculated field values in the unstructured DoFs are transformed to arrange them in a structured grid with definable resolution. The simulation domain is a cavity box represented by an unstructured grid discretized spatially into 1585 tetrahedra with PEC (perfect electric conductor) boundary and initial value conditions. The simulation runs on four processes, showing that `HighPerMeshes` can output results in the distributed case that can be easily merged to be viewed with `ParaView`.

```

1 auto surfaceKernelLoop = ForEachIncidence<2>(cells,
2   tuple(
3     Read(ContainingMeshElement(H)),
4     Read(ContainingMeshElement(E)),
5     Read(NeighboringMeshElementOrSelf(H)),
6     Read(NeighboringMeshElementOrSelf(E)),
7     Write(ContainingMeshElement(rhsE))
8   ),
9   [&](const auto& cell, const auto& face, ..., auto& local_view){
10    auto& [H, E, nH, nE, rhsE] = local_view;
11    const auto& NeighboringNodeMap {DgNodeMap.Get(cell, face)};
12    const int faceIndex = face.GetTopology().GetLocalIndex();
13    const auto& faceUnitNormal = face.GetGeometry().GetUnitNormal();
14    const auto edg = (face.GetGeometry().GetNormal()*2.0/
15    cell.GetGeometry().GetAbsJacobianDeterminant()).Norm()*0.5;
16
17    ForEach(numSurfaceNodes, [&](const int m){
18      const auto& dH = edg*Delta(H, nH, m, NeighboringNodeMap);
19      const auto& dE =
20        edg*DirectionalDelta(E, nE, face, m, NeighboringNodeMap);
21      const auto& fluxE = (dE - (dE*faceUnitNormal)*faceUnitNormal + CrossProduct(
22        faceUnitNormal, dH));
23
24      ForEach(numVolumeNodes, [&](const int n){
25        rhsE[n] += LIFT[face_index][m][n]*fluxE;
26      });
27    });
28 );

```

Listing 5: Code segment for the Maxwell surface kernel

3.3 | Finite elements for cardiac electrophysiology

The excitation of cardiac muscle tissue is described by electrophysiology models such as the monodomain model

$$\dot{u} = \nabla \cdot (\sigma \nabla u) + I_{\text{ion}}(u, w),$$

$$\dot{w} = f(u, w), \quad (5)$$

where σ is the conductivity, I_{ion} is the ion current that forms together with the gating dynamics $f(u, w)$ the membrane model. The simplest FitzHugh–Nagumo membrane model defines $I_{\text{ion}}(u, w) = u(1 - u)(u - a) - w$ and $f(u, w) = \epsilon(u - bw)$ with $0 < a, b, \epsilon < 1$.⁷⁻⁹

The method of lines¹⁰ discretizes the monodomain model (5) first in space and then in time. For the discretization of space, we use linear finite elements again, leading to the system

$$\begin{aligned} M\dot{u} &= \sigma Au + M \cdot I_{\text{ion}}(u, w), \\ \dot{w} &= f(u, w) \end{aligned}$$

with mass matrix M and stiffness matrix A . For time discretization, the forward Euler method

$$u_{t+1} = u_t + \tau \underbrace{(M^{-1} \sigma Au_t + I_{\text{ion}}(u_t, w_t))}_{\dot{u} := \dot{u}_d}, \quad w_{t+1} = w_t + \tau f(u_t, w_t) \quad (6)$$

is widely used in cardiac electrophysiology due to its simplicity and its stability for reasonable step sizes τ .¹¹

In order to avoid inverting the globally coupled mass matrix, the row-sum mass lumping technique is applied to M .¹² This yields a diagonal approximation M_l of M and allows for efficient, explicit formation of M_l^{-1} to be used in (6) instead of M^{-1} , and matrix-free storage in vector form. The right-hand side u_t including the matrix–vector product Au_t is assembled directly as in (2) without forming A :

```

1 auto fwEuler = ForEachEntity(
2   mesh.GetEntityRange<0>(),
3   tuple(Vertex(u), Vertex(Read(u_d))),
4   [&](const auto& vertex, .., auto& local_view) {
5     auto& [u, u_d] = local_view;
6     u[0] += tau * u_d[0];
7   }
8 );

```

Listing 6: Code example of an implementation of a first order solver (forward Euler).

4 | MAPPING TO PARALLEL SHARED- AND DISTRIBUTED MEMORY SYSTEMS

In the previous two sections, we introduced the HighPerMeshes DSL from a usage perspective, highlighting in Section 2.3 how a user can implement device- and parallelization-agnostic local kernels. When targeting multi-CPU architectures, the kernels get compiled along with parts of the HighPerMeshes infrastructure, which under the hood allows for parallel execution with a library and runtime system.

HighPerMeshes provides a distributed dispatcher that builds upon the Global Address Space Programming Interface (GASPI)¹³ for scaling over multiple nodes. The latter further uses ACE¹⁴ to accelerate the algorithms by either feeding tasks to ACE's thread pool or by parallelizing the work defined by a task with OpenMP[‡]. For an in-depth explanation of this dispatcher, we refer to.^{15,16}

For the distribution of data and computation to multiple compute nodes and processors, HighPerMeshes manages a hierarchy of global and local mesh partitions. For mesh partitioning, we use the Metis library.¹⁷ Global partitions assign mesh entities to distinct compute nodes, while local partitions can add an additional layer for further work segmentation on each compute node. Each local partition belongs to a unique global partition that determines the actual compute node to which the associated mesh entities belong. Global partitions also store the DoFs corresponding to their owned mesh entities.

To achieve parallelization, we define tasks such that each task applies a kernel given by one loop to each entity in a local partition. The execution of these tasks can then be parallelized. However, the task still requires and produces specific data that other tasks might also access. This data may also be on another physical compute node, thus requiring communication.

The distributed dispatcher creates a dependency graph for all kernels to enforce correct behavior and constructs a dependency between two kernels if they access the same DoFs. For each of these dependencies, the dispatcher calculates the exact intersection of required buffer indices with the help of the specified access pattern. Suppose these index intersections lie on a different global partition. In that case, HighPerMeshes defines a precondition for the receiving task to wait for the data and defines a postcondition for the producing task to send the data to the correct process.

Figure 3 shows an example for two kernels accessing the same buffer, one writing and one reading. In HighPerMeshes, this can be expressed as `dispatcher.Execute(Writer, Reader)`. Here, the ranges to be iterated over are already abstracted as the tasks w_i and r_i with $0 \leq i \leq 7$.

[‡]<https://www.openmp.org/>.

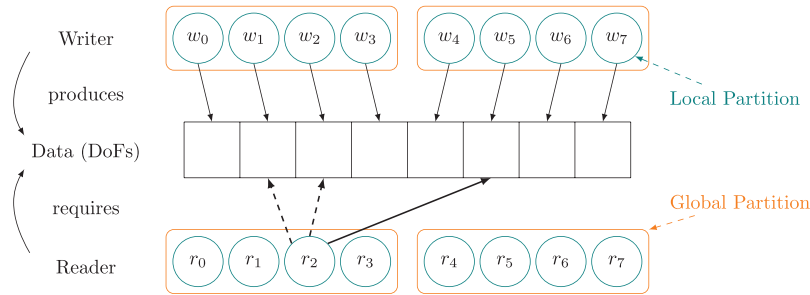


FIGURE 3 Example data accesses for a sending and receiving solver step. The tasks are already assigned to two global partitions (rounded boxes) that are separated into four local partitions (circles)

In this example, the dispatcher must schedule r_2 after w_1 , w_2 , and w_5 , because the writers produce data that the reader requires. Furthermore, because r_2 and w_5 are on different global partitions, that is, their data lies on physically distinct compute nodes, communication is required. The dispatcher can use the calculated intersection of DoFs to define a process that sends data from w_5 to r_2 when w_5 is finished.

Here, HighPerMeshes' access patterns, as described in Section 2.3, show their advantage. A dispatcher implementation can use this abstract data access specification during compile time in order to construct the required procedures to communicate data, transparent to the end user. This would not be possible with random data access.

5 | SOURCE-TO-SOURCE OPENCL CODE GENERATION

When additionally aiming to execute HighPerMeshes kernels on accelerators like GPUs, the embedding of HighPerMeshes into C++ 17 is currently a limitation, as no suitable compilers exist so far. Thus, in order to support also these architectures by HighPerMeshes, we developed a source-to-source code transformation infrastructure that complements the library and runtime system introduced in Section 4. The transformation infrastructure extracts the lambda code given in the kernels described in Section 2.3 into OpenCL.

In addition to the OpenCL kernel code that we generate by source-to-source transformations, OpenCL requires boilerplate code on the host side that either compiles and runs its kernels during runtime, or uses a third-party compiler to compile the kernel code. We chose an approach that reduces the amount of code that actually needs to be generated on the host and instead employ a library-based solution for managing OpenCL kernels on the host side. In this section, we introduce the combination of kernel transformation flow and library-based dispatcher that handles the host side of the OpenCL execution model.

5.1 | Intermediate representation and kernel generation

Clang can expose the abstract syntax tree (AST) of C++ programs and allows writing source-to-source translation tools with LibTooling⁵. To change the resulting source code, LibTooling can extract the location of specific AST nodes in the source code. This range of characters can either be replaced or expanded with a new text. This means it is not possible to modify the AST directly. This is not a maintainable solution for context-sensitive transformations that depend on each other. Furthermore, each transformation must produce a valid C++ program; otherwise, the compiler cannot parse it, which might complicate certain transformations.

Because of this, we translate parts of the AST provided by Clang into a new intermediate representation (IR), where an AST can transform into another AST directly. The AST's node types represent a subset of C++ that allows all the common operations in HighPerMeshes. In order to not have to implement a complete IR of C++, there are certain restrictions to what is allowed in kernels to be transformed with the code generator. Most notably, data structures not provided by HighPerMeshes are not allowed.

5.1.1 | Transformations

The transformation framework is based on the visitor pattern.¹⁸ Each node must provide a `transform` function that applies the visitor's `visit` function to each of its members and creates a new member of its own type. Listing 7 shows the structure for a used variable and its corresponding transform function. This way, if `visit` does not return identity for some member, the latter is transformed into a new node.

⁵<https://clang.llvm.org/docs/LibTooling.html>.


```

1 struct Variable {
2     Type type;
3     Name name;
4 };
5
6 template<typename Visitor>
7 Variable transform
8     (Visitor&& visitor, Variable&& variable) {
9     return {
10         visitor.visit(std::move(variable.type)),
11         visitor.visit(std::move(variable.name))
12     };
13 }

```

Listing 7: Example of the transform function

Visitors in the code generator can have multiple behaviors. They either traverse the AST depth-first or breadth-first and can also implement different stopping conditions. For example, we employ a visitor that only traverses until it encounters a member of a certain type.

Listing 8 shows an example for the initialization of one such visitor. When calling the `transformer`'s `visit` function on a node in the AST, it checks if the given node can be passed to the lambda it received (lines 2–4) and applies it if possible or returns the unmodified node. This way, each transformation creates a new AST from the old one, keeping the nodes that are unchanged and transforming nodes that match the passed lambda. For example, suppose the `transformer`'s `visit` function is applied to the root node of an AST. In that case, it returns a new AST where the function `do_something_with` changes each variable.

```

1 Transformer transformer {
2     [](Variable&& variable) {
3         return do_something_with(std::move(variable));
4     }
5 };

```

Listing 8: Example of the transform function

5.1.2 | Printing

To generate a new source file after all transformations are applied, each type used in the IR must also implement a `print` function that prints their representation in the actual source code. The function recursively prints the complete subtree that makes up the expression or statement. For example, Listing 9 shows this `print` function for the variable type, where only its name is printed. The resulting kernels are printed to a source-file that can be used on the host-side.

```

1 template<typename Stream>
2 void print(Stream& stream, const Variable& variable) {
3     print(stream, variable.name);
4 }

```

Listing 9: Example of the transform function

5.2 | Host side integration

The buffer data types described in Section 2.2 can specify custom allocators similar to the containers provided by the standard template library[¶]. With the C++ bindings of OpenCL 2.0[#], it is possible to use the shared virtual memory capabilities introduced in OpenCL 2.0 using such an allocator. This heavily reduces the code that needs to be generated and the complexity involved in transferring data between host and device. The code generator employs coarse-grained shared virtual memory to allow compatibility with more devices.

[¶]<https://en.cppreference.com/w/cpp/header>.

[#]<https://github.com/khronos.org/OpenCL-CLHPP/>.

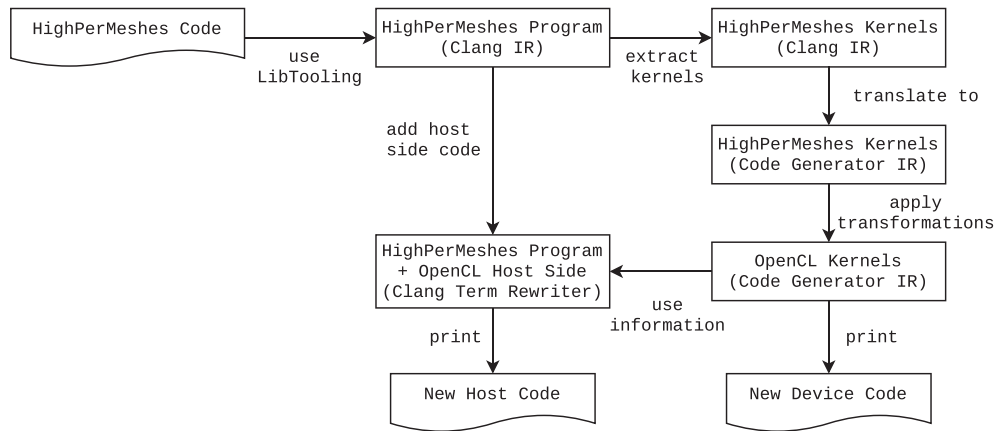


FIGURE 4 Summary of the code generator's workflow

We provide a library implementation for initialization and kernel enqueueing that handles all the boilerplate code associated with compiling the kernels and enqueueing them. A class called `OpenCLHandler` allows reading kernels from either a string or binary. It then provides all the necessary functions to enqueue a kernel. Another class called `OpenCLKernelEnqueuer` is constructed with such an `OpenCLHandler`. It allows directly specifying the arguments of a kernel and execute it. We provide as much code as possible as a library solution because it is less error-prone and more comfortable to test than generating all the necessary code.

In summary, in order to handle the host side integration of OpenCL kernels, the code generator replaces the original dispatcher calls with the `OpenCLKernelEnqueuer` mentioned above, adds the initialization of an `OpenCLHandler`, and applies the correct allocator to all the buffers. For these host-side transformations, the translation into the code generator's IR is not necessary because the code that needs to be generated is far simpler due to the library approach.

5.3 | Code generator workflow

Figure 4 summarizes the entire code generator's workflow with kernel extraction and generation and host side integration.

First, the user provides a HighPerMeshes source-file that can be tested by employing the sequential dispatcher provided by HighPerMeshes. The source-to-source generator finds all `dispatcher.Executecalls` and their corresponding kernels (`ForEachEntity`) based on the IR provided by Clang. We translate these kernels into a new IR that allows transforming the resulting AST into other AST. In this new IR, we can apply all transformations necessary to generate a valid OpenCL kernel from the provided lambdas.

Overall, we have implemented 33 transformers so far. Their most important purpose is to translate the different HighPerMeshes-specific language features such as vectors and matrices into structures that are usable with OpenCL, translate C++ syntax to C syntax, and to create explicit address generation code based on buffer base addresses, view-specific offsets, and offsets based on work-items. After all transformations are applied, the device code is generated by printing all nodes of the final AST.

Furthermore, the infrastructure adapts the given HighPerMeshes source-file to be usable with the new OpenCL kernels by employing LibTooling's usual capabilities of directly modifying the source code. In this step, information from the kernel generation phase is used, such as generated kernel names.

6 | EXPERIMENTS

6.1 | Distributed scalability experiments

In this section, we analyze the distributed scalability of the matrix-vector product (Listing 2), the volume kernel (Listing 4), and the surface kernel (Listing 5). The experiments were performed on a cluster, where each compute node consists of two sockets. Each socket contains an Intel Xeon Gold 6148 "Skylake" CPU, which has 20 cores and a base frequency of 2.4 GHz. Hyperthreading is deactivated. The nodes are connected on a 100 Gb/s Intel Omni Path network. All experiments were executed with 20 threads per socket, as the scalability of our threading approach on a single compute node has already been shown.¹⁵ To show that HighPerMeshes is not implemented for a specific parallelization technology, we analyze two back ends

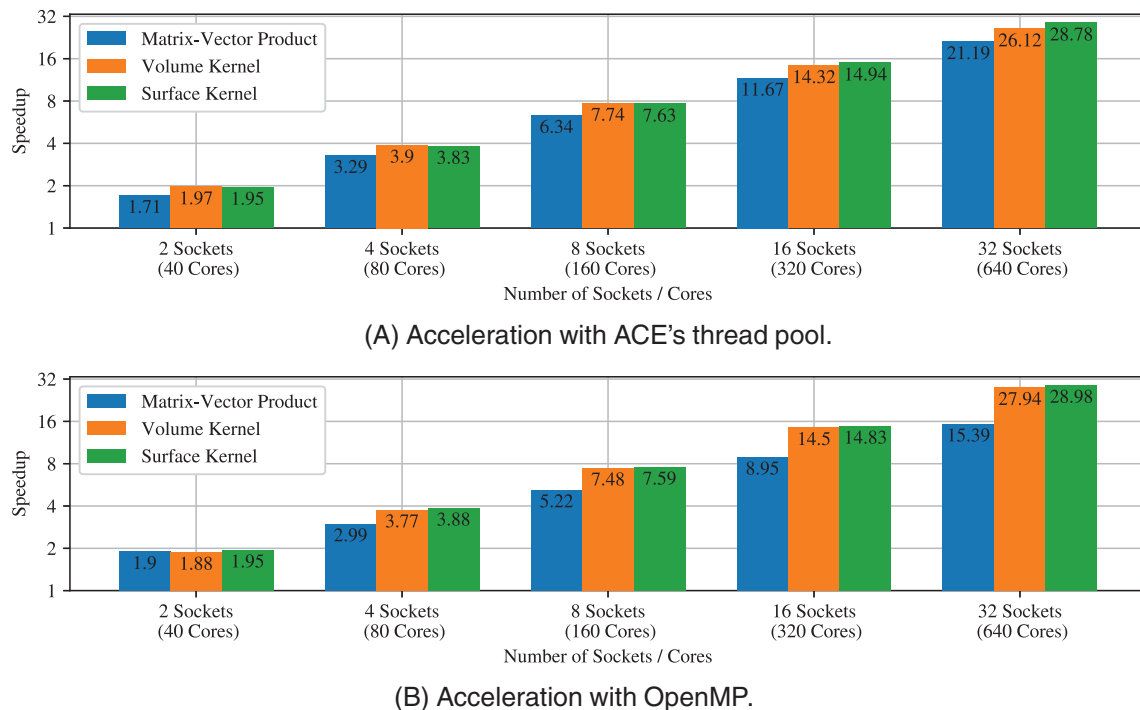


FIGURE 5 Strong scaling speedup for iterating over the specified kernels on a mesh with 400,000 tetrahedra and 1000 time steps on an increasing amount of sockets compared with executing the same kernels on one socket. The evaluated back ends parallelize the programs with ACE's thread pool (A) or by accelerating the scheduled tasks with OpenMP (B)

provided by HighPerMeshes. The first one schedules tasks using ACE's thread pool, while the other accelerates tasks with OpenMP, as described in Section 4.

We conducted strong scaling experiments for 1000 time steps on a synthetic mesh of 400,000 tetrahedra. Such a setup represents a typical problem size targeted by the distributed dispatcher. Figure 5 shows the speedup over a single node for the distributed dispatcher for both back ends and an increasing amount of compute nodes. As a baseline for each experiment, we measured the execution time with both back ends on a single socket, that is, 20 cores, and use the faster one. Thus, the theoretical optimum for each application is a speedup equal to the number of sockets. For 640 cores, the back end feeding threads to ACE's thread pool achieves better speedups for the matrix-vector product with a speedup of 21.19. The volume and surface kernels achieve a better speedup in the case for OpenMP acceleration, with a speedup of 27.94 and 28.98, respectively. Furthermore, the volume and surface kernels scale better than the matrix-vector multiplication because they are more compute-intensive. They iterate over 20 DoFs instead of just one. To achieve this kind of scalability, the dispatcher requires a sufficient workload. The experiment shows that we reach approximately 90% of the optimal speedup for the surface kernel and above 80% for the volume kernel.

For the strong scaling experiment, we also calculated the standard deviation from the mean for the workload on each socket to determine the effectiveness of the load balancing scheme described in Section 4. Here we calculated a maximum standard deviation of 2.27%, showing that the work is evenly distributed between nodes and that no significant bottlenecks are introduced due to task dependencies.

Furthermore, we conducted weak scaling experiments with around 12,500 tetrahedrons per socket used, with minor deviations to the workload per socket depending on the mesh partitioning. Figure 6 shows the weak scaling results for the volume and surface kernel for both back ends. We show the parallel efficiency relative to the respective two-socket variant as a reference. All results are above 90% except for the surface kernel using ACE's thread pool on 32 or 64 threads. On 64 sockets or 1280 cores, even this experiment still reaches an efficiency of 81% compared with the two-socket variant. The same weak scaling experiments were also performed for the matrix-vector product. As this example does not require communication, the measured parallel efficiency was always close to 100%. Therefore we omit these results in the figure to not clutter the graph.

Overall, the strong and weak scaling experiments show that HighPerMeshes allows an efficient and easy distribution of matrix-free algorithms at least to dozens of HPC nodes. They also show that HighPerMeshes provides suitable abstractions for different back ends, as the results for both back ends provide similar results for most of the experiments. Instead, both reference implementations achieve similar speedups, thus showing that the language is portable to different technologies.

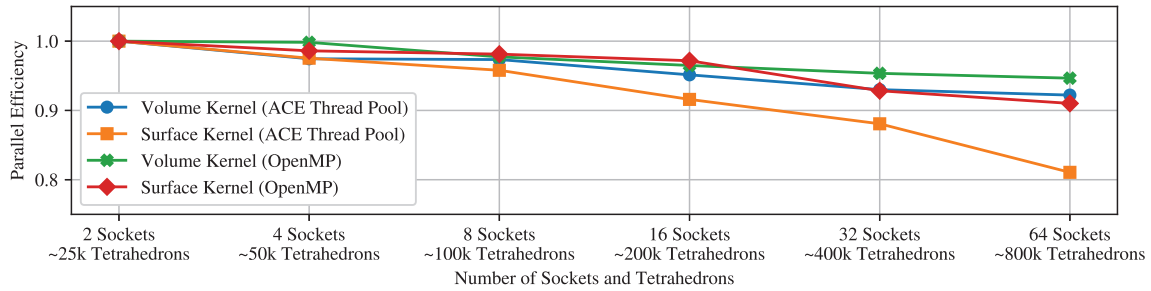


FIGURE 6 Parallel efficiency for the weak scalability experiments. On the basis of a mesh with 400,000 tetrahedrons on 32 sockets, we increase or decrease the number of mesh elements based on the number of sockets used. We show the parallel efficiency of the volume and surface kernel for both back ends. Each kernel is executed for 10,000 time steps

TABLE 2 Speedups of generated GPU kernels over baseline HighPerMeshes CPU execution

Kernel	Speedup GPU over CPU
Forward Euler	4.9
Runge Kutta integration	10.9
Maxwell volume	12

TABLE 3 Speedups of generated GPU kernels over hand-written CPU execution

Kernel	Speedup GPU over CPU
Forward Euler with three-dimensional vectors	2.5
Runge Kutta integration with hand-written host-side code	9.7

6.2 | GPU experiments

In this section, we test the performance of HighPerMeshes' back end using OpenCL as described in Section 5. For this purpose, we calculate the speedup of three kernels compared with our back end using OpenMP. We test the forward Euler method shown in Listing 6, the Runge Kutta integration loop explained in Section 3.2, and the Maxwell volume kernel shown in Listing 4. All experiments are run for a single DoF per entity and use double-precision floating-point types.

Similar to the strong scaling scenario in Section 6.1, we use a mesh of 400,000 tetrahedra and simulate for 1000 time steps. We measured the experiments described in this section on a system with an AMD Ryzen 5 3600X CPU and an AMD Radeon RX 5600 XT GPU, thus comparing in this setup two mid-range consumer devices that a computational scientist might use before moving to an HPC system. Table 2 presents the measured GPU speedups for the three kernels, showing how HighPerMeshes can also exploit the acceleration potential of GPUs for suitable kernels.

In order to critically assess these results further, we investigated if the CPU implementation of our DSL leads to performance deficits compared with a hand-written implementation. Table 3 shows the results for two additional experiments, comparing the same generated GPU kernels with hand-optimized CPU variants of the same kernels. First, we implemented the Runge Kutta Integration loop without statements that are available in HighPerMeshes, instead using standard for-loops and vectors. The loop iterating over all entities in the range is parallelized with OpenMP, similar to the back end provided in HighPerMeshes. Here we can only see a slight difference in performance, only a speedup of 9.7 compared with 10.9, which means that HighPerMeshes introduces some overhead, but does not alter the resulting performance by a significant amount for its high level of abstraction. We also investigated if the influence of vector data structures as provided by HighPerMeshes impact on performance. For this purpose, we measured the speedup for the forward Euler kernel with a three-dimensional vector. Here we only see a speedup of 2.5 compared with 4.9. An explanation is that we now have to consider three vector entries instead of just one, which leads to uncoalesced memory accesses. Here, performance could be improved by using "structures of arrays" instead of using "arrays of structures."

7 | RELATED WORK

There are several other software projects addressing PDE computations on unstructured grids. Traditional library approaches such as deal.II,¹⁹ DUNE,²⁰ or Kaskade 7²¹ focus on application building blocks and usually provide explicit parallelization based on threads or MPI, providing one or a few selected back ends such as PETSc.²² HighPerMeshes provides explicit features that allow implementing new back ends while these toolbox approaches do not grant such extensibility.

High-level DSLs such as FEniCS²³ or FreeFEM²⁴ allow specifying PDE problems in very abstract notation and use code generation techniques to create efficient simulation programs. The scope of HighPerMeshes is more on the side of solver-implementation than abstract mathematical formulations. The projects closest in scope and intention to our work are OP2²⁵/PyOP2²⁶ and Liszt.²⁷ OP2 is an “active library” framework that allows distributing mesh-based compute kernels and accessing data associated with different mesh entities on multicore and many-core architectures. Here, the major difference to HighPerMeshes is that OP2 provides a direct parallelization statement `op_par_loop` instead of using a dispatcher. Liszt is a DSL for PDE solvers embedded in Scala that provides a cross-compiler that analysis FEMs written in their syntax, which is close to Scala. In contrast, we rely on template metaprogramming methods for distributing code while we use code transformation techniques for the OpenCL back end.

There are several DSLs that consider stencil codes on structured grids. In comparison, HighPerMeshes targets the domain of solver codes on unstructured grids. For example, STELLA²⁸ is embedded in C++ and allows parallelization with OpenMP, while Mint²⁹ is embedded in C and uses source-to-source transformation to emit CUDA code. Another example in this domain is ExaSlang,³⁰ a DSL that was developed in the ExaStencils^{31,32} project. It is a multilevel DSL that uses code generation to transform algorithms from more abstract algorithm definitions to concrete solver implementations that are then translated to C++ code.

Regarding code generation, Hipacc^{33,34} is a DSL in the domain of image processing that also uses Clang’s Libtooling to generate code. Compared with their approach, HighPerMeshes’s code generator introduces an additional IR to allow AST-to-AST transformations, while HiPacc only employs term-rewriting.

Alternatives to OpenCL include the CUDA programming model for NVIDIA GPUs and Intel’s recently released oneAPI specification around the more expressive DPC++ language based on SYCL and C++ to target its CPUs, GPUs, and FPGAs. Yet, OpenCL is currently still the most widely supported language for data-parallel architectures, also including AMD GPUs besides the previously mentioned targets. Of the mentioned technologies, HighPerMeshes currently only supports OpenCL, but is designed in such a way, that a new back end can be introduced with a new dispatcher as explained in Section 2.3.

To summarize, HighPerMeshes is a DSL embedded in C++ focused on solver implementations for unstructured grids that completely separates parallelization technology from solver formulation. While HighPerMeshes already provides some back ends for different parallelization technologies, it is designed to easily allow the implementation of other back ends with its dispatcher concept.

8 | CONCLUSION AND FUTURE WORK

HighPerMeshes is an embedded DSL providing high-level abstractions for the design of iterative, matrix-free algorithms on unstructured grids. It is a powerful framework enabling users to run simulations as well as implement their own modifications for complex multiscale problems from a broad range of application domains like optics, photonics, hydrodynamics, gas dynamics, and acoustics.

HighPerMeshes provides data structures and procedures that allow for efficient autparallelization and distribution with the help of GASPI, ACE, OpenMP, and OpenCL. Here, the dispatcher concept allows a clean separation of the parallelization back ends from the rest of the DSL, allowing other technologies, such as SYCL^{||}, in the future. This tackles the problem of not being able to use the DSL if new technologies emerge and makes HighPerMeshes more future-proof compared with other approaches that only provide a fixed number of back end solutions. HighPerMeshes already includes some scalable high-performance back ends. This saves implementation time and effort on one side, and offers flexibility for different computing platforms without the need for code modification on the other side. In our preliminary practical experience, we found that the DSL can indeed be used by numerical analysts ignorant of modern parallel architectures to exploit these to a large extent. Thus, HighPerMeshes enables the user to take advantage of complex parallelization, task scheduling, and data distribution techniques, completely without requiring knowledge about parallelization. Moreover, relying on the HighPerMeshes abstraction relieves the user from adapting the application code to several different target architectures.

The back ends for parallelization and distribution, as described in Section 4 leave room for future work. Alternative back ends can be implemented and used without modification of user code due to the clear decoupling of algorithm and parallelization technology provided by the dispatcher concept. Alternative dispatchers could be based on a hybrid of MPI and OpenMP.³⁵ Another significant next step is to combine the OpenCL back with the GASPI back end, allowing the usage of GPUs on multiple nodes. Another opportunity lies within generating code for

^{||}<https://sycl.tech/>.

an accelerator and distributing the resulting kernels with the distributed dispatcher. While the code generator is functional, more sophisticated optimizations are also in consideration for future work.

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) within the collaborative research project “HighPerMeshes” (01IH16005). The authors gratefully acknowledge the funding of this project by computing time provided by the Paderborn Center for Parallel Computing (PC²).

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Stefan Groth  <https://orcid.org/0000-0002-9043-0746>

Frank Hannig  <https://orcid.org/0000-0003-3663-6484>

REFERENCES

1. Saad Y. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Society for Industrial and Applied Mathematics; 2003.
2. Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. *J Res Natl Bur Stand*. 1952;49(6):409-436.
3. Deuffhard P, Weiser M. *Adaptive Numerical Solution of PDEs*. 1st ed. De Gruyter; 2012.
4. Ahrens JP, Geveci B, Law CC. ParaView: an end-user tool for large-data visualization. *Visualization Handbook*. 2005;717-731. <https://www.sciencedirect.com/book/9780123875822/visualization-handbook#book-info>
5. Hesthaven JS, Warburton T. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. 1st ed. Springer; 2008.
6. Grynko Y, Förstner J. Simulation of second harmonic generation from photonic nanostructures using the discontinuous Galerkin time domain method. In: Agrawal A, Benson T, De La Rue RM, Wurtz GA, eds. *Recent Trends in Computational Photonics. Springer Series in Optical Sciences*. Springer; 2017;204:261-284.
7. Liu F, Zhuang P, Turner IW, Anh V, Burrage K. A semi-alternating direction method for a 2-D fractional FitzHugh-Nagumo monodomain model on an approximate irregular domain. *J Comput Phys*. 2015;293:252-263.
8. Dauby P, Desaive T, Croisier H, Kolh P. Standing waves in the FitzHugh-Nagumo model of cardiac electrical activity. *Phys Rev E*. 2006;73.
9. Wermesant M, Coudière Y, Delingette H, Ayache N, Désidéri J. An electro-mechanical model of the heart for cardiac image analysis. In: Niessen WJ, Viergever MA, eds. *Proceedings of the 4th International Conference Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer; 2001:224-231.
10. Schiesser WE, Griffiths GW. *A Compendium of Partial Differential Equation Models: Method of Lines Analysis with Matlab*. 1st ed. Cambridge University Press; 2009.
11. Puwal S, Roth BJ. Forward Euler stability of the Bidomain model of cardiac tissue. *IEEE Trans Biomed Eng*. 2007;54(5):951-953.
12. Hughes TJ. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. 1st ed. Courier Corporation; 2012.
13. Grünewald D, Simmendinger C. The GASPI API specification and its implementation GPI 2.0. *Proceedings of the 7th International Conference on PGAS Programming Models*. University of Edinburgh; 2013.
14. Grünewald D. Asynchronous constraint execution. Accessed December 14, 2020. <https://github.com/cc-hpc-itwm/ACE>
15. Groth S, Grünewald D, Teich J, Hannig F. A runtime system for finite element methods in a partitioned global address space. In: Palesi M, Palermo G, Graves C, Arima E, eds. *Proceedings of the 17th ACM International Conference on Computing Frontiers (CF)*. ACM; 2020:39-48.
16. Alhaddad S, Förstner J, Groth S, et al. HighPerMeshes – a domain-specific language for numerical algorithms on unstructured grids. In: Balis B, Heras DB, eds. *Proceedings of the 18th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar) in Euro-Par 2020: Parallel Processing Workshops*. Springer; 2020.
17. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput*. 1998;20(1):359-392.
18. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison Wesley; 1994.
19. Alzetta G, Arndt D, Bangerth W, et al. The deal. II library, version 9.0. *J Numer Math*. 2018;26(4):173-183.
20. Bastian P, Blatt M, Dedner A, et al. A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. *Computing*. 2008;82(2-3):121-138.
21. Götschel S, Schiela A, Weiser M. Kaskade 7 – a flexible finite element toolbox. *Comput Math Appl*. 2021;81:444-458.
22. Balay S, Abhyankar S, Adams MF, et al. PETSc web page; 2020. Accessed December 14, 2020. <https://www.mcs.anl.gov/petsc>
23. Logg A, Mardal KA, Wells G. *Automated Solution of Differential Equations by the Finite Element Method*. Vol 84. 1st ed. Springer; 2012.
24. Hecht F. New development in FreeFem++. *J Numer Math*. 2012;20(3-4):251-266.
25. Mudalige GR, Giles MB, Reguly I, Bertolli C, Kelly PHJ. OP2: an active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. *Proceedings of Innovative Parallel Computing (InPar)*. IEEE; 2012:1-12.
26. Rathgeber F, Markall GR, Mitchell L, et al. PyOP2: a high-level framework for performance-portable simulations on unstructured meshes. *SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE Computer Society; 2012:1116-1123.
27. DeVito Z, Joubert N, Palacios F, et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Lathrop SA, Costa J, Kramer W, eds. *Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis (SC)*. ACM; 2011:9:1-9:12.
28. Gysi T, Osuna C, Fuhrer O, Bianco M, Schulthess TC. STELLA: a domain-specific tool for structured grid methods in weather and climate models. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM; 2015:41:1-41:12.
29. Unat D, Cai X, Baden SB. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Lowenthal DK, de Supinski BR, McKee SA, eds. *Proceedings of the 25th International Conference on Supercomputing*. ACM; 2011:214-224.

30. Schmitt C, Kuckuk S, Hannig F, Köstler H, Teich J. ExaSlang: a domain-specific language for highly scalable multigrid solvers. *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE Computer Society; 2014:42-51.
31. Lengauer C, Apel S, Bolten M, et al. ExaStencils: advanced multigrid solver generation. In: Bungartz H, Reiz S, Uekermann B, Neumann P, Nagel W, eds. *Software for Exascale Computing - SPPEXA 2016-2019*. Vol. 136 of *Lecture Notes in Computational Science and Engineering*. Springer; 2020:405-452.
32. Lengauer C, Apel S, Bolten M, et al. ExaStencils: advanced stencil-code engineering. In: Lopes L, Žilinskas J, eds. *Proceedings of Euro-Par 2014: Parallel Processing Workshops*. Vol. 8806 of *Lecture Notes in Computer Science*. Springer; 2014:553-564.
33. Membarth R, Reiche O, Hannig F, Teich J, Körner M, Eckert W. HIPAcc: a domain-specific language and compiler for image processing. *IEEE Trans Parallel Distrib Syst*. 2016;27(1):210-224.
34. Membarth R, Hannig F, Teich J, Körner M, Eckert W. Generating device-specific GPU code for local operators in medical imaging. *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE; 2012:569-581.
35. Rabenseifner R, Hager G, Jost G. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Baz DE, Spies F, Gross T, eds. *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE Computer Society; 2009:427-436.

How to cite this article: Alhaddad S, Förstner J, Groth S, et al. The HighPerMeshes framework for numerical algorithms on unstructured grids. *Concurrency Computat Pract Exper*. 2021:e6616. <https://doi.org/10.1002/cpe.6616>