Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200062

Unified Generation of DG-Kernels for Different HPC Frameworks

Jan HÖNIG [a] Marcel KOCH [b] Ulrich RÜDE [a] Christian ENGWER [b]
Harald KÖSTLER [a]
[a] *Friedrich-Alexander University Erlangen-Nürnberg*
[b] *University of Münster*

Abstract. Code generation specified by a DSL is a popular method to manage maintenance effort and introduce an abstraction layer for higher reusability. In the case of Galerkin methods, the Unified Form Language is a DSL for the weak formulation of a differential equation. In this paper, we present the framework-specific code generation for DUNE and ExaStencils from a problem formulated in the UFL. Moreover, we present optimization strategies, which are applied during the generation process.

Keywords. code generation, dune, exastencils, DG, UFL

1. Introduction

The Finite Element Method (FEM) is widely used for the numerical solution of partial differential equations. Variants like the discontinuous Galerkin (dG) method have drawn much attention within the last two decades. Applications range from hyperbolic problems like shallow water or Maxwell's equations to non-linear degenerated parabolic problems like multi-phase flow in porous media.

To solve these equations, the discretized problems at hand are usually manually expressed in a programming language. This usually involves using some FEM library or framework. Instead of this tedious work, the mathematical problem can be expressed at a higher level, and the code which obtains the numerical solution is generated. The code generation allows for a more flexible interface while remaining the performance compared to the manually written code. To define a given mathematical problem, we use the Unified Form Language (UFL) [1]. UFL is a domain-specific language (DSL) designed for specifying finite element discretizations in variational form and was initiated by the FEniCS Project [2].

We aim at providing code transformation components to support UFL in two different frameworks, DUNE and ExaStencils. DUNE [3,4] is a flexible framework for Multi-Physics- and Multi-Domain-Simulations, and the first option we consider as a backend for our code generation pipeline. ExaStencils [5] is a whole-program code generation framework working on block-structured grids that we use as a second backend for the generation of a dG-kernel. In this case, only certain types of elements, grids, and dG discretizations are supported.

In this paper, we introduce the targeted frameworks, explain the details of the generation pipeline, and compare the two approaches of generating dG-kernels in terms of flexibility and possible optimizations. To showcase the flexibility of our code generation, we consider a simple model problem, the linear transport equation. Within a single code transformation tool, we can handle transformations for different mesh types, two different frameworks, and mesh specific mathematical optimizations. The aim of our code generator is the generation of back end optimized target code, based on a simple problem description in UFL.

1.1. Related Work

UFL is used by both FEniCS [2] and Firedrake [6] frameworks for decleration of FEM of variational form. Recently the Firedrake project made efforts to incorporate loopy's intermediate representation (IR) into their generation framework [7]. The authors use loopy's IR to vectorize computations across multiple elements with promising performance results. The `dune-pdelab` specific code generation of our toolchain with a focus on optimizing dG kernels was first described in [8]. For the ExaStencils framework, a similar approach for quadrature-free shallow water equations was presented in [9].

2. Targeted Frameworks

Our code generation approach targets two different simulation frameworks, DUNE and ExaStencils. Although both aim to provide easy-to-use frameworks for solving partial differential equations (PDEs), the taken approaches differ significantly. DUNE can run its kernels on any mesh, whereas the specific nature of ExaStencils is limited only to regular and cartesian grids. To highlight these different needs and requirements for the code generator, we briefly describe these two frameworks.

2.1. DUNE

DUNE [3,4,10] is a C++ simulation framework for solving PDEs. DUNE relies heavily on generic programming. This allows the compiler to remove most interface-related runtime overhead. Instead of a monolithic codebase, DUNE is composed of several core modules, with a clear separation of concerns.

The `dune-common` module supplies basic dense linear algebra, MPI communications, a build system infrastructure and further basic functionality. `dune-localfunctions` supplies a wide range of finite element basis functions, e.g. (discontinuous) Lagrange functions, Raviart-Thomas basis functions or orthonormal basis functions. Reference element implementations for different geometries and quadrature rules defined on those elements are provided by `dune-geometry` and `dune-istl` offers iterative solvers and preconditioners for sparse matrices with blocking.

The `dune-grid` module defines a hierarchical grid interface, which is implemented by multiple grid managers. The interface is general enough to support grids with a wide range of features, e.g., structured and unstructured grids, conforming and nonconforming refinement, or support for multiple element types. This means that switching from one grid implementation to another usually only requires changing the type of

the grid, and further adjustment of user code is not necessary. Additionally, the interface supports parallelization using MPI.

Discretizations modules, such as `dune-fem` [11] or `dune-pdelab` [12], provide abstractions for finite volume or finite element methods. As an example, `dune-pdelab` introduces C++ classes equivalent to the mathematical notion of grid function spaces or grid operators, which apply element local kernels to every element in the specified grid. Using `dune-pdelab`, the finite volume or finite element assembly is automated to the point where users merely need to supply the element-local kernels and select the right solution scheme. Of course, users are still free to extend functionality by providing their implementations of the defined interfaces.

2.2. ExaStencils

ExaStencils is a whole-program generator [5,13], which provides a multi-layered domain specific language. Its primary focus lies on the generation of highly efficient geometric multigrid solvers for partial differential equations. ExaStencils itself is implemented in Scala. Scala, among other things, provides a powerful pattern matching mechanism, which makes the implementation of the compiler and generator software simpler in terms of development time and maintenance efforts.

The DSL, called ExaSlang [14] offers four different abstraction levels. The continuous specification of the whole simulated problem, i.e., equations, unknowns, boundary conditions, and the computational domain, is described in the first layer. The second layer states the discretized version of the problem, whereas the third layer describes a suitable solver. The combination of the second and third layers results in a complete program specification. Transitioning between these layers themselves is done in a semi-automatic manner under users' guidance. Users decide which layer is most suited for the description of the application. ExaStencils supports the transformation of ExaSlang into C++ and CUDA code, thus targeting different platforms. During this transformation, ExaStencils performs several optimization strategies, e.g., address precalculation, loop transformations including loop blocking, reordering and condition elimination, explicit vectorization, and loop carried common subexpression elimination.

In the case of C++ and a CPU target, ExaStencils can parallelize the generated code. Depending on the settings, it generates code with OpenMP, MPI, or both. For the MPI case, necessary ghost-layers or overlapping of fields can be automatically introduced as well. Given the required parallelization and the patches of fields, i.e., the splitting of the domain onto processes, ExaStencils provides the communication routines between the patches. Although ExaStencils was designed primarily for multigrid methods, it is perfectly capable of handling stencil-only applications as well.

3. Code Generation

Our code generation is based on the domain-specific language UFL [1,15]. Developed by the FEniCS [2] project, UFL describes the weak formulation of a PDE in Python. It is, therefore, best suited for finite element methods. Describing a PDE with UFL is closely related to the theoretical formulation of the PDE.

In the following example, we demonstrate the usage of UFL. Consider the Poisson Equation (1) with its weak formulation: Find $u \in H_0^1$ such that u solves Equation (2).

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{1}$$

$$a(u,v) = F(v) \quad \forall v \in H_0^1, \text{ where}$$
$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{dx} \tag{2}$$
$$F(v) = \int_\Omega f v \, \mathrm{dx}$$

A discretization of this problem can be constructed by choosing a triangulation for Ω and approximating H_0^1 by a space consisting of globally continuous and piecewise linear functions and the corresponding UFL formulation, without boundary treatment, now reads:

```
mesh = # triangulation of Omega
V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = # analytic definition of f
a = inner(grad(u), grad(v)) * dx
F = f * v * dx
```

Listing 1: Example UFL File

As can be seen in Listing 1, operator overloading and supplying appropriate named functions and constants allows for a straight forward translation of the weak formulation into UFL. UFL's representation is an abstract syntax tree (AST) of high-level mathematical objects as well as necessary linear algebra objects. However, representation on this level is insufficient for most optimizations or transformations towards high-performance computing. Therefore, the UFL-AST is transformed into an IR, which is suitable for the optimizations needed in the backend.

Our choice for this IR is loopy [16] together with pymbolic [17]. Pymbolic is a library for precise manipulation of symbolic expressions and is a perfect fit for representing expressions inside a code generation framework. We have chosen pymbolic over other computer algebra systems like sympy [18] because it does not change expressions implicitly and is easily extensible. Loopy's computational kernels are described by loop domains and instructions, and thus loopy is capable of handling statements, their dependencies, loops, and control flow. Additionally, loopy comes with a range of transformations based on the polyhedral model, e.g., loop tiling or loop fusion, which was shown using in a finite element method context [19]. Choosing loopy was evident since it fits perfectly as the IR.

Our code generator realizes the transformation from an UFL-AST into the IR by a tree traversal approach. As can be seen in Figure 1, this approach is used by both frameworks. The generation of the IR, which is post-processed by one of the backends afterward, also depends on the selection of the targeted framework. Thus the output expressed by the IR differs for both backends.

The traversal is realized by a visitor object with type-based function dispatch. We separate the UFL-AST node types into four different categories, geometry evaluations, basis evaluations, quadrature evaluation, and backend agnostic, which are mostly linear algebra nodes. For the node types in the first three categories, the transformation into the IR is backend-specific, for nodes from the last categories, it does not depend on the back

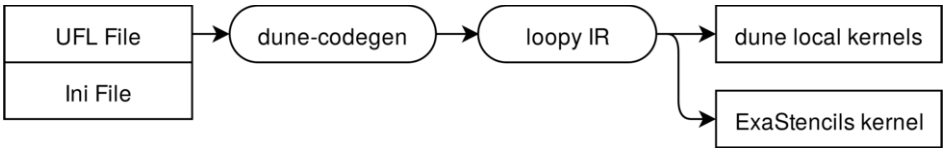


Figure 1. UFL Generation Pipeline

end. For each category, UFL-AST visitor classes are defined, which only handle the node types according to its category. The full visitor type is constructed using mixins with one class from each category. There is one additional back end specific mixin, keeping track of which equation in a system of PDEs is currently handled.

Together with the UFL file, the generation needs additional information, which is contained in the INI file. It may contain information about the selection of optimizations, spatial and temporal discretization, element type, and other settings.

3.1. DUNE Specific Generation

The `dune-pdelab` framework provides many components needed for finite element assembly, as mentioned in Section 2.1. Since these components have been thoroughly tested and used, even in high-performance settings, we rely on these parts and do not generate code replacing them. Instead, we generate local kernels, which compute element local or face local integrals. The local kernels are the most expensive part of the assembly process, except for trivial integrals. Thus, we expect the most performance gain from focusing on generating optimized code for these local kernels.

The `dune-pdelab` code generation is divided into three possible paths. The default path implements the generic `dune-pdelab` implementation of the tree visitor. The other two paths implement back end specific optimizations for high order dG or for low order continuous Galerkin (cG) discretizations. Each implementation has its optimizations for different grid types. We currently distinguish between equidistant, axis-parallel, multilinear, or generic grids. Currently, the optimizing paths of the generation require quadrilateral or hexahedral meshes, whereas the generic path also works with simplices.

In the case of tensor product finite element basis functions and reference elements, sum factorizing reduces the complexity of the local assembly process. This is especially rewarding for higher-order dG discretizations. The article [8] describes several vectorization strategies for sum-factorized kernels realized in our code generator, e.g., batching several sum-factorized sub kernels. The selection of the vectorization strategy can be defined manually or decided automatically either by a cost model or by auto-tuning.

Low order cG discretizations do not profit from sum factorization as much as dG discretizations. In these cases, locally structured meshes are a better approach. Using this optimization, a notable performance gain is achieved by operating on multiple elements in one local kernel. Additionally, this allows for cross element vectorization, which otherwise would be cumbersome to realize in `dune-pdelab`. Users have to request this optimization explicitly since using locally structured meshes increases the number of degrees of freedom similarly to uniform refinement. This needs to be addressed when creating a coarse grid.

Hardware-based optimizations, e.g., vectorization, loop tiling, or loop fusion, are possible in both optimized code generation paths. These kinds of optimizations are real-

ized as transformations of the loopy IR after the UFL form is transformed into the IR. In contrast, optimizations relying on the grid type or basis function type are handled during the transformation from UFL into the loopy IR, since these may induce algorithmic changes. Because of C's lack of standardized vectorization, loopy does not generate vectorized C code by default. We added a custom back end, which uses the wrappers defined in the vector class library [20] for generating vectorized instructions.

It is possible to generate local kernels for both matrix-based and matrix-free computations. Matrix-based computations can rely on a wide range of preconditioners for accelerating the solution of a linear system, but they gain only limited performance from recent hardware developments like vectorization. Matrix free computations, on the other hand, are FLOP bound and thus gain significant performance increases from vectorization, but the access to robust preconditioner is limited. Therefore, our optimizations are most effective for matrix-free computations.

3.2. ExaStencils Specific Generation

In contrast to DUNE, ExaStencils does not provide any components regarding the finite element method. The only data structure it provides is a Field. Because of this limitation, and because ExaStencils is a multigrid and stencil generation tool, we focus only on the particular case of a regular grid. More precisely, the focus lies on a cartesian grid, which has two triangles per square. With this limitation, the ExaStencils generation uses back end specific optimizations.

The generation itself consists of three steps. At first, the given UFL is preprocessed and traversed with an ExaStencils specific visitor, which translates the UFL description into an intermediate representation consisting of Loopy and Pymbolic expressions. During this step, we evaluate quadrature points, weights, basis functions, and the derivatives of basis functions. The evaluation is possible because of the limitations of the mesh and is the essential optimization step.

Secondly, the IR is expressed as ExaStencils code. This includes unrolling of loops, gathering additional information for the ExaStencils generator, and translate the IR as ExaStencils function. For the translation of the loopy IR, we introduced a new back-end for the ExaSlang language. Additional information given to the generation process contains the domain and array size, parallelization strategy.

The generation from the UFL formulation does not create any initialization of fields, visualization, time-stepping loop, nor a main-function. In those cases, we use the information from the INI file and use Jinja [21] templates for a flexible implementation of said accompanying program components.

In the last step of the whole generation process, the generated files are translated into C++ with ExaStencils. During this step, ExaStencils performs the optimization and parallelization, as described in the previous section.

The cartesian grid is visualized in Figure 2. Each cell has one lower and one upper triangle. The coefficients of the triangles are stored in two separate arrays and utilize the regularity of the grid, to create stencil kernels. The generated code consists of three kernels. One handles the integration of the volume of the triangle. The second one handles the integration on the faces and accesses its neighbors in a stencil pattern. The third kernel takes care of the boundaries, and triangles directly adjacent to boundaries.

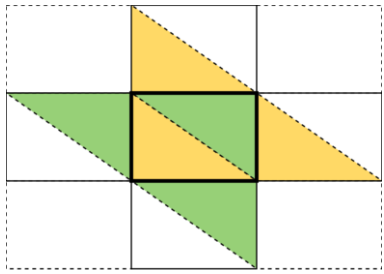


Figure 2. ExaStencil Grids

4. Numerical Evaluation

In this section, we verify our toolchain in terms of correctness and scaling using the linear transport equation,

$$\begin{aligned} \partial_t u + \beta \cdot \nabla u = 0 \quad & \text{in } \Omega \times (0,T) \\ u = 0 \quad & \text{on } \Gamma_D \\ u(\cdot,0) = u_0 \quad & \text{in } \Omega \end{aligned}$$

This equation can be used to model the transport of a concentration through a domain. We choose a discontinuous Galerkin approach with upwinding for the discretization of the problem, leading to the following UFL description in Listing 2.

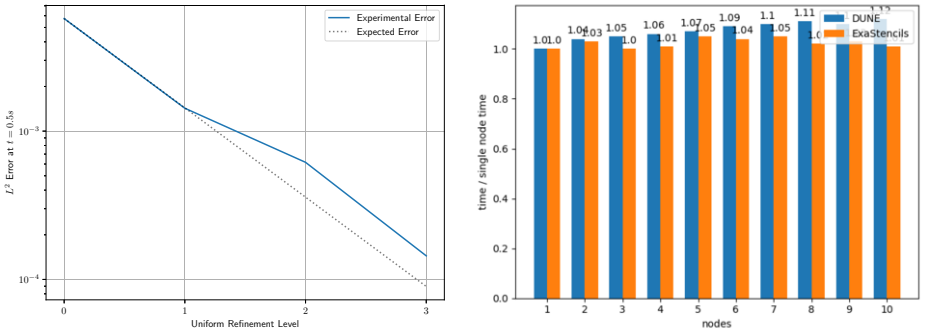
```
def upwinding_flux(normal, inside, outside):
    return (conditional(inner(beta, normal) > 0, inside, outside) *
            inner(beta, normal))
# definition of test and trial functions u and v, beta and initial value
n = FacetNormal(cell)('+')
# mass operator for temporal discretization
mass = u * v * dx
# residual operator r(u,v) = a(u,v) - F(v) for spatial discretization
r = (-1. * u * inner(beta, grad(v)) * dx +
     upwinding_flux(n, u('+'), u('-')) * jump(v) * dS)
```

Listing 2: Linear Transport

In terms of expressiveness, we can compare the lines of code (LOC) of the UFL and INI specification, and the generated code. The specification consists of 76 LOC. The code generation produces 339 LOC for DUNE and 1203 LOC for ExaStencils. In the following, we verify the correctness of our code generator by examining the convergence for a simple configuration. Additionally, we investigate the weak scaling of our generated code.

4.1. Convergence Test

For the convergence test, we use $\Omega = [0,1]^2$ and $T = 0.5$ with an constant advection $\beta = [1\ 1]^T$. The inital condition $u_0(x)$ has a bell shaped concentration of the form



(a) Convergence (b) Parallel Efficiency

Figure 3. Numerical Evaluation

$u_0(x) = cos(\pi||x - x_0||)$ for x in a radius of $r = 0.15$ around $x_0 = [0.25\ 0.25]^T$ and otherwise $u_0 = 0$. The exact solution at time $t = 0.5$ is the same as the initial value, except that the concentration is centered around $x_0 = [0.75\ 0.75]$. During this time frame, the homogenous Dirichlet condition is applicable, since the concentration does not reach the boundary.

Both backends use explicit time-stepping schemes for the temporal discretization, with a timestep size small enough to achieve a stable simulation. Currently, ExaStencils only supports the explicit Euler method, while `dune-pdelab` also supports higher-order Runge-Kutta methods, in this case, a third-order strong stability preserving scheme from [22].

ExaStencils uses a structured simplicial grid, while `dune-pdelab` uses an unstructured simplicial grid with the grid implementation from `dune-uggrid`. The coarse structured grid consists of 1600 elements, while the unstructured grid has 1700 elements on the coarsest level. In both cases, the grid is refined up to three times. Figure 3a shows the the L^2 error of the approximate solution at time $t = 0.5$ for each refinement level. As expected, a convergence order of 2 can be seen.

4.2. Weak Scaling

Next, we investigate the weak scaling of our generated codes. With weak scaling, we can show that the generated kernels still work with the respective framework at hand. Since only element local kernels are generated for the `dune-pdelab` backend, the following results are only influenced by the scalability of the used DUNE components. In [23, 24] the scaling capabilities of the `dune-istl` module are shown and in [25] scaling results using the `dune-pdelab` module can be found. In the case of ExaStencils, the kernels are generated for individual MPI processes, and its performance capabilities were demonstrated in [26]. The communication itself happens outside of the kernels and is entirely handled by the ExaStencils framework.

We consider the same test case as above, with 500 timesteps and a grid of the size 100×100 per core for DUNE and of the size 128×128 per core for ExaStencils. We simulate on the SuperMUC-NG system, which is located at Leibnitz Supercomputing Center (LRZ) in Munich.

From Figure 3b it can be seen that the `dune-pdelab` code achives over 90% parallel efficiency, which is consistent with earlier findings [23,24]. The generated code for ExaStencils achieves over 95% parallel efficiency. This demonstrates the proper usage of our frameworks.

5. Conclusion and Outlook

In this paper, we have shown that we can use one toolchain to generate UFL for different back ends, namely DUNE and ExaStencils. This approach provides us with a general and flexible description of a mathematical problem, which can be numerically solved with code generation to said back ends. In the particular case of the cartesian grid, we can utilize the excellent speed of ExaStencils, while still having the possibility to rely on DUNE's performance for any more general problem. This approach is extensible to other back ends as well, which is already done for FEniCS and Firedrake frameworks. However, each framework can still have a different intention, approach, and specific optimizations. FEniCS's primary focus is on usability and generality, at which it excels, while our work is primarily focused on performance. Firedrake is also geared towards performance, but during their code generations, they use different IRs for algorithmic and hardware optimizations. For future projects, a detailed comparison with Firedrake and its pipeline is inevitable.

The simple example of linear transport shows a promising possibility of generating fast code for a cartesian grid with ExaStencils. In the future, this approach should be expanded to regular grids together with completing all features of the UFL. This includes having a solver for systems of linear equations and condition-based fluxes. In future work, the `dune-pdelab` specific code generation will explore additional optimizations possible within the IR. Furthermore, the generation of an optimized preconditioner will be investigated.

6. Acknowledgments

This research has been funded by the Federal Ministry of Education and Research of Germany (BMBF) through the HPC2SE project. We are grateful to the Leibniz Rechenzentrum Garching for providing computational resources. Dominic Kempf has initialized the code generation project `dune-codegen` from the interdisciplinary center for scientific computing (IWR) at the Heidelberg University. He and René Hess (IWR) are the main contributors of the sum-factorization specific optimizations for the `dune-pdelab` backend. We are grateful to Sebastian Kuckuk, one of the leading developers of ExaStencils, for his support.

References

[1] M. S. Alnæs *et al.*, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *CoRR*, vol. abs/1211.4047, 2012.

[2] M. S. Alnæs *et al.*, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, no. 100, 2015.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework," *Computing*, vol. 82, pp. 103–119, Jul 2008.

[4] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. part ii: Implementation and tests in dune," *Computing*, vol. 82, pp. 121–138, 01 2008.

[5] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, "Exastencils: Advanced stencil-code engineering," in *Euro-Par 2014: Parallel Processing Workshops*, (Cham), pp. 553–564, Springer International Publishing, 2014.

[6] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: automating the finite element method by composing abstractions," *CoRR*, vol. abs/1501.01809, 2015.

[7] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. J. Kelly, "A study of vectorization for matrix-free finite element methods," *CoRR*, vol. abs/1903.08243, 2019.

[8] D. Kempf, R. Heß, S. Müthing, and P. Bastian, "Automatic code generation for high-performance discontinuous galerkin methods on modern architectures," *arXiv preprint arXiv:1812.08075*, 2018.

[9] S. Faghih-Naini, S. Kuckuk, V. Aizinger, D. Zint, R. Grosso, and H. Köstler, "Towards whole program generation of quadrature-free discontinuous galerkin methods for the shallow water equations," *CoRR*, vol. abs/1904.08684, 2019.

[10] "Dune numerics."

[11] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger, "A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module," *Computing*, vol. 90, pp. 165–196, Nov 2010.

[12] P. Bastian, F. Heimann, and S. Marnach, "Generic implementation of finite element methods in the distributed and unified numerics environment (dune)," *Kybernetika*, vol. 46, no. 2, pp. 294–315, 2010.

[13] "Advanced stencil-code engineering." `https://www.exastencils.fau.de`.

[14] C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, H. Köstler, U. Rüde, and C. Lengauer, "Systems of partial differential equations in exaslang," in *Software for Exascale Computing - SPPEXA 2013-2015*, (Cham), pp. 47–67, Springer International Publishing, 2016.

[15] "UFL: Unified form language." `https://fenics.readthedocs.io/projects/ufl`.

[16] A. Klöckner, "Loo.py: transformation-based code generation for GPUs and CPUs," in *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*, (Edinburgh, Scotland.), Association for Computing Machinery, 2014.

[17] "Pymbolic." `https://documen.tician.de/pymbolic`.

[18] A. Meurer *et al.*, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017.

[19] A. Klöckner, L. C. Wilcox, and T. Warburton, "Array program transformation with loo.py by example: High-order finite elements," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pp. 9–16, ACM, 2016.

[20] "C++ vector class library." `https://www.agner.org/optimize/#vectorclass`.

[21] "Jinja." `https://palletsprojects.com/p/jinja`.

[22] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes," *Journal of Computational Physics*, vol. 77, no. 2, pp. 439 – 471, 1988.

[23] O. Ippisch and M. Blatt, "Scalability test of $\mu\phi$ and the parallel algebraic multigrid solver of dune-istl," in *Jülich Blue Gene/P Extreme Scaling Workshop, no. FZJ-JSC-IB-2011-02. Jülich Supercomputing Centre*, 2011.

[24] M. Blatt, O. Ippisch, and P. Bastian, "A massively parallel algebraic multigrid preconditioner based on aggregation for elliptic problems with heterogeneous coefficients," *arXiv preprint arXiv:1209.0960*, 2012.

[25] S. Müthing, M. Piatkowski, and P. Bastian, "High-performance implementation of matrix-free high-order discontinuous galerkin methods," *arXiv preprint arXiv:1711.10885*, 2017.

[26] S. Kuckuk and H. Köstler, "Whole program generation of massively parallel shallow water equation solvers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 78–87, Sep. 2018.