The Journal of Open Source Software

libCEED: Fast algebra for high-order element-based discretizations

Jed Brown[1], **Ahmad Abdelfattah**[3], **Valeria Barra**[1], **Natalie Beams**[3], **Jean-Sylvain Camier**[2], **Veselin Dobrev**[2], **Yohann Dudouit**[2], **Leila Ghaffari**[1], **Tzanio Kolev**[2], **David Medina**[4], **Will Pazner**[2], **Thilina Ratnayaka**[5], **Jeremy Thompson**[1], **and Stan Tomov**[3]

1 University of Colorado at Boulder **2** Lawrence Livermore National Laboratory **3** University of Tennessee **4** Occalytics LLC **5** University of Illinois at Urbana-Champaign

DOI: 10.21105/joss.02945

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @thelfer
- @FreddieWitherden,

Submitted: 04 January 2021
Published: 07 July 2021

Summary

Finite element methods are widely used to solve partial differential equations (PDE) in science and engineering, but their standard implementation (Arndt et al., 2020; Kirk et al., 2006; Logg et al., 2012) relies on assembling sparse matrices. Sparse matrix multiplication and triangular operations perform a scalar multiply and add for each nonzero entry, just 2 floating point operations (flops) per scalar that must be loaded from memory (Williams et al., 2009). Modern hardware is capable of nearly 100 flops per scalar streamed from memory (Rupp, 2020) so sparse matrix operations cannot achieve more than about 2% utilization of arithmetic units. Matrix assembly becomes even more problematic when the polynomial degree p of the basis functions is increased, resulting in $O(p^d)$ storage and $O(p^{2d})$ compute per degree of freedom (DoF) in d dimensions. Methods pioneered by the spectral element community (Deville et al., 2002; Orszag, 1980) exploit problem structure to reduce costs to $O(1)$ storage and $O(p)$ compute per DoF, with very high utilization of modern CPUs and GPUs. Unfortunately, high-quality implementations have been relegated to applications and intrusive frameworks that are often difficult to extend to new problems or incorporate into legacy applications, especially when strong preconditioners are required.

`libCEED`, the Code for Efficient Extensible Discretization (Abdelfattah et al., 2021), is a lightweight library that provides a purely algebraic interface for linear and nonlinear operators and preconditioners with element-based discretizations. `libCEED` provides portable performance via run-time selection of implementations optimized for CPUs and GPUs, including support for just-in-time (JIT) compilation. It is designed for convenient use in new and legacy software, and offers interfaces in C99 (International Standards Organisation, 1999), Fortran77 (ANSI, 1978), Python (*Python*, 2021), Julia (Bezanson et al., 2017), and Rust (*Rust*, 2021). Users and library developers can integrate `libCEED` at a low level into existing applications in place of existing matrix-vector products without significant refactoring of their own discretization infrastructure. Alternatively, users can utilize integrated `libCEED` support in MFEM (Anderson et al., 2020; *MFEM*, 2021).

In addition to supporting applications and discretization libraries, `libCEED` provides a platform for performance engineering and co-design, as well as an algebraic interface for solvers research like adaptive p-multigrid, much like how sparse matrix libraries enable development and deployment of algebraic multigrid solvers.

Concepts and interface

Consider finite element discretization of a problem based on a weak form with one weak derivative: find u such that

$$v^T F(u) := \int_\Omega v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) = 0 \quad \forall v,$$

where the functions f_0 and f_1 define the physics and possible stabilization of the problem (Brown, 2010) and the functions u and v live in a suitable space. Integrals in the weak form are evaluated by summing over elements e,

$$F(u) = \sum_e \mathcal{E}_e^T B_e^T W_e f(B_e \mathcal{E}_e u),$$

where $\mathcal{E}_e$ restricts to element e, B_e evaluates solution values and derivatives to quadrature points, f acts independently at quadrature points, and W_e is a (diagonal) weighting at quadrature points. By grouping the operations W_e and f into a point-block diagonal D and stacking the restrictions $\mathcal{E}_e$ and basis actions B_e for each element, we can express the global residual in operator notation (Figure 1), where $\mathcal{P}$ is an optional external operator, such as the parallel restriction in MPI-based (Gropp et al., 2014) solvers. Inhomogeneous Neumann, Robin, and nonlinear boundary conditions can be added in a similar fashion by adding terms integrated over boundary faces while Dirichlet boundary conditions can be added by setting the target values prior to applying the operator representing the weak form. Similar face integral terms can also be used to represent discontinuous Galerkin formulations.

$$A = \mathcal{P}^T \mathcal{E}^T B^T D B \mathcal{E} \mathcal{P}$$

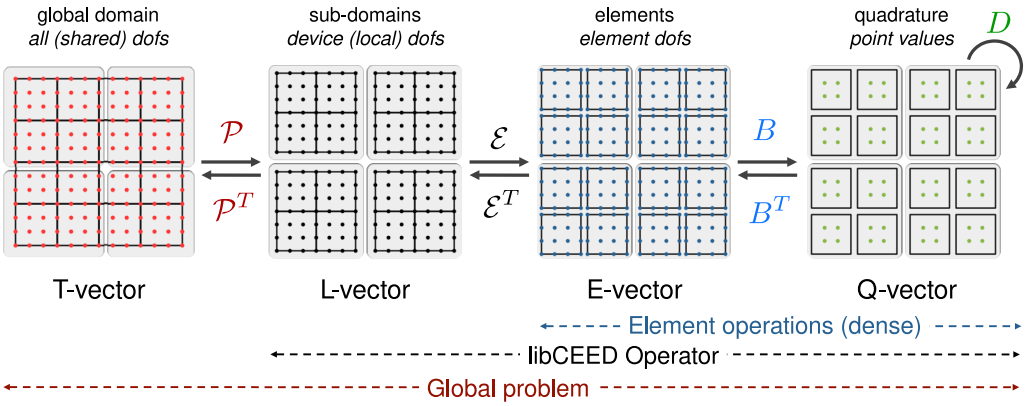

Figure 1: `libCEED` uses a logical decomposition to define element-based discretizations, with optimized implementations of the action and preconditioning ingredients.

`libCEED`'s native C interface is object-oriented, providing data types for each logical object in the decomposition.

Symbol	`libCEED` type	Description
D	`CeedQFunction`	User-defined action at quadrature points
B	`CeedBasis`	Basis evaluation to quadrature (dense/structured)
$\mathcal{E}$	`CeedElemRestriction`	Restriction to each element (sparse/boolean)
A	`CeedOperator`	Linear or nonlinear operator acting on L-vectors

`libCEED` implementations ("backends") are free to reorder and fuse computational steps (including eliding memory to store intermediate representations) so long as the mathematical properties of the operator A are preserved. A `CeedOperator` is composed of one or more operators defined as in Figure 1, and acts on a `CeedVector`, which typically encapsulates zero-copy access to host or device memory provided by the caller. The element restriction $\mathcal{E}$ requires mesh topology and a numbering of DoFs, and may be a no-op when data is already composed by element (such as with discontinuous Galerkin methods). The discrete basis B is the purely algebraic expression of a finite element basis (shape functions) and quadrature; it often possesses structure that is exploited to speed up its action. Some constructors are provided for arbitrary polynomial degree H^1 Lagrange bases with a tensor-product representation due to the computational efficiency of computing solution values and derivatives at quadrature points via tensor contractions. However, the user can define a `CeedBasis` for arbitrary element topology including tetrahedra, prisms, and other realizations of abstract polytopes, by providing quadrature weights and the matrices used to compute solution values and derivatives at quadrature points from the DoFs on the element.

The physics (weak form) is expressed through `CeedQFunction`, which can either be defined by the user or selected from a gallery distributed with `libCEED`. These pointwise functions do not depend on element resolution, topology, or basis degree (see Figure 2), in contrast to systems like FEniCS where UFL forms specify basis degree at compile time. This isolation is valuable for hp-refinement and adaptivity (where h commonly denotes the average element size and p the polynomial degree of the basis functions; see Babuška & Suri (1994)) and p-multigrid solvers; mixed-degree, mixed-topology, and h-nonconforming finite element methods are readily expressed by composition. Additionally, a single source implementation (in vanilla C or C++) for the `CeedQFunctions` can be used on CPUs or GPUs (transparently using the *NVRTC* (2021), HIPRTC, or OCCA (*OCCA Development Site*, 2021) run-time compilation features).

`libCEED` provides computation of the true operator diagonal for preconditioning with Jacobi and Chebyshev as well as direct assembly of sparse matrices (e.g., for coarse operators in multigrid) and construction of p-multigrid prolongation and restriction operators. Preconditioning matrix-free operators is an active area of research; support for domain decomposition methods and inexact subdomain solvers based on the fast diagonalization method (Lottes & Fischer, 2005) are in active development.

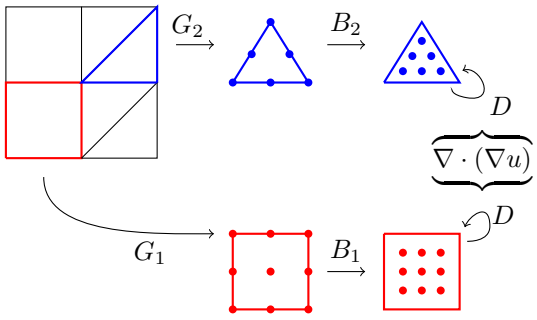

Figure 2: A schematic of element restriction and basis applicator operators for elements with different topology. This sketch shows the independence of Q-functions (in this case representing a Laplacian) on element resolution, topology, and basis degree.

High-level languages

`libCEED` provides high-level interfaces in Python, Julia, and Rust, each of which is maintained and tested as part of the main repository, but distributed through each language's respective package manager.

The Python interface uses CFFI, the C Foreign Function Interface (*C Foreign Function Interface for Python*, 2021). CFFI allows reuse of most C declarations and requires only a minimal adaptation of some of them. The C and Python APIs are mapped in a nearly 1:1 correspondence. For instance, a `CeedVector` object is exposed as `libceed.Vector` in Python, and supports no-copy host and GPU device interperability with Python arrays from the NumPy (Harris et al., 2020) or Numba (Lam et al., 2015) packages. The interested reader can find more details on `libCEED`'s Python interface in Barra et al. (2020).

The Julia interface, referred to as `LibCEED.jl`, provides both a low-level interface, which is generated automatically from `libCEED`'s C header files, and a high-level interface. The high-level interface takes advantage of Julia's metaprogramming and just-in-time compilation capabilities to enable concise definition of Q-functions that work on both CPUs and GPUs, along with their composition into operators as in Figure 1.

The Rust interface also wraps automatically-generated bindings from the `libCEED` C header files, offering increased safety due to Rust ownership and borrow checking, and more convenient definition of Q-functions (e.g., via closures).

Backends

Figure 3 shows a subset of the backend implementations (backends) available in `libCEED`. GPU implementations are available via pure *CUDA* (2021) and pure *HIP* (2021), as well as the OCCA (*OCCA Development Site*, 2021) and MAGMA (*MAGMA development site*, 2021) libraries. CPU implementations are available via pure C and AVX intrinsics as well as the LIBXSMM library (*LIBXSMM development site*, 2021). `libCEED` provides a dynamic interface such that users only need to write a single source (no need for templates/generics) and can select the desired specialized implementation at run time. Moreover, each process or thread can instantiate an arbitrary number of backends on an arbitrary number of devices.

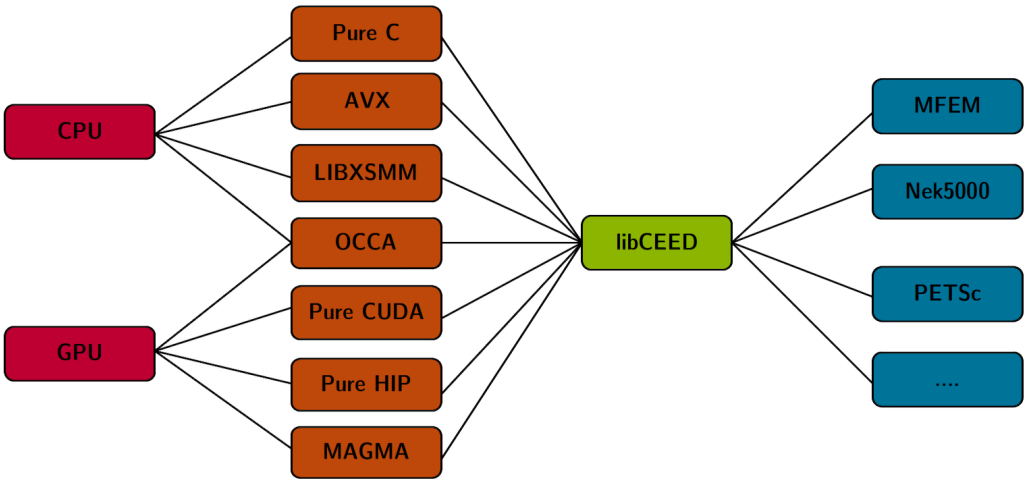

Figure 3: `libCEED` provides the algebraic core for element-based discretizations, with specialized implementations (backends) for heterogeneous architectures.

Performance benchmarks

The Exascale Computing Project (ECP) co-design Center for Efficient Exascale Discretization (*CEED*, 2021) has defined a suite of Benchmark Problems (BPs) to test and compare the

performance of high-order finite element implementations (Fischer et al., 2020; Kolev et al., 2021). Figure 4 compares the performance of `libCEED` solving BP3 (CG iteration on a 3D Poisson problem) or CPU and GPU systems of similar (purchase/operating and energy) cost. These tests use PETSc (Balay et al., 2021) for unstructured mesh management and parallel solvers with GPU-aware communication (Zhang et al., 2021); a similar implementation with comparable performance is available through MFEM.

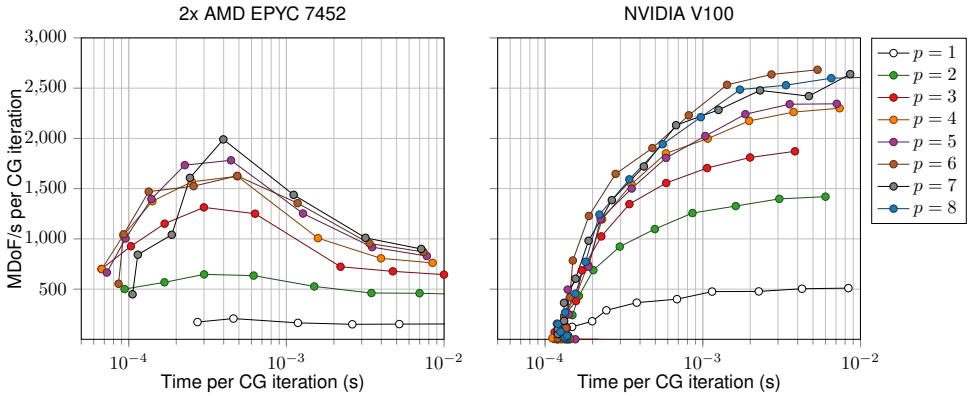

Figure 4: Performance for BP3 using the `xsmm/blocked` backend on a 2-socket AMD EPYC 7452 (32-core, 2.35GHz) and the `cuda/gen` backend on LLNL's Lassen system with NVIDIA V100 GPUs. Each curve represents fixing the basis degree p and varying the number of elements. The CPU enables faster solution of smaller problem sizes (as in strong scaling) while the GPU is more efficient for applications that can afford to wait for larger sizes. Note that the CPU exhibits a performance drop when the working set becomes too large for L3 cache (128 MB/socket) while no such drop exists for the GPU. (This experiment was run with release candidates of PETSc 3.14 and libCEED 0.7 using gcc-10 on EPYC and clang-10/CUDA-10 on Lassen.)

Demo applications and integration

To highlight the ease of library reuse for solver composition and leverage `libCEED`'s full capability for real-world applications, `libCEED` comes with a suite of application examples, including problems of interest to the fluid dynamics and solid mechanics communities. The fluid dynamics example solves the 2D and 3D compressible Navier-Stokes equations using SU/SUPG stabilization and implicit, explicit, or IMEX time integration; Figure 5 shows vortices arising in the "density current" (Straka et al., 1993) when a cold bubble of air reaches the ground. The solid mechanics example solves static linear elasticity and hyperelasticity with load continuation and Newton-Krylov solvers with p-multigrid preconditioners; Figure 6 shows a twisted Neo-Hookean beam. Both of these examples have been developed using PETSc, where `libCEED` provides the matrix-free operator and preconditioner ingredient evaluation and PETSc provides the unstructured mesh management and parallel solvers.

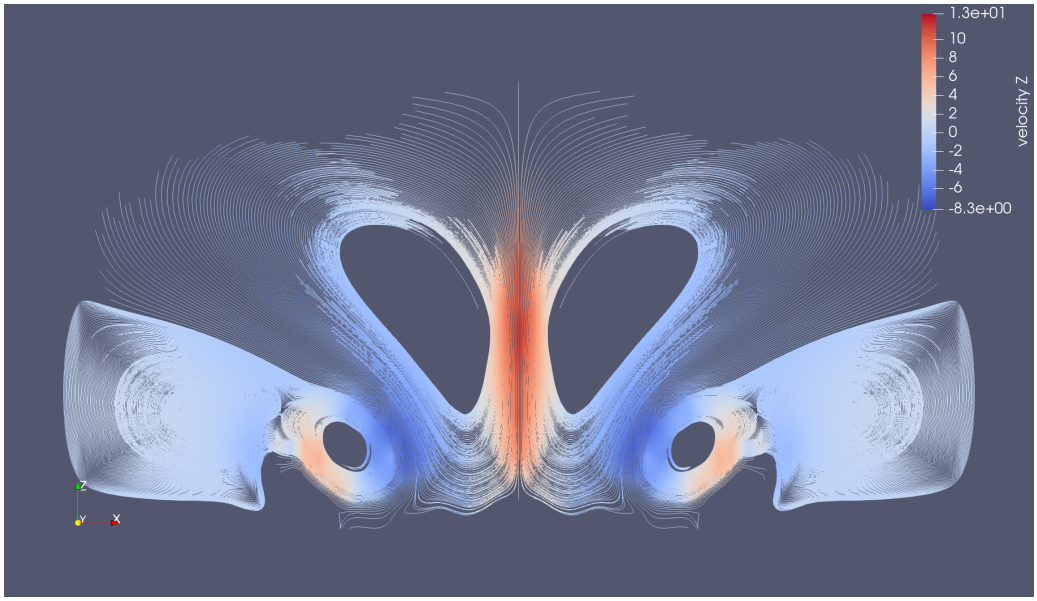

Figure 5: Vortices develop as a cold air bubble drops to the ground.

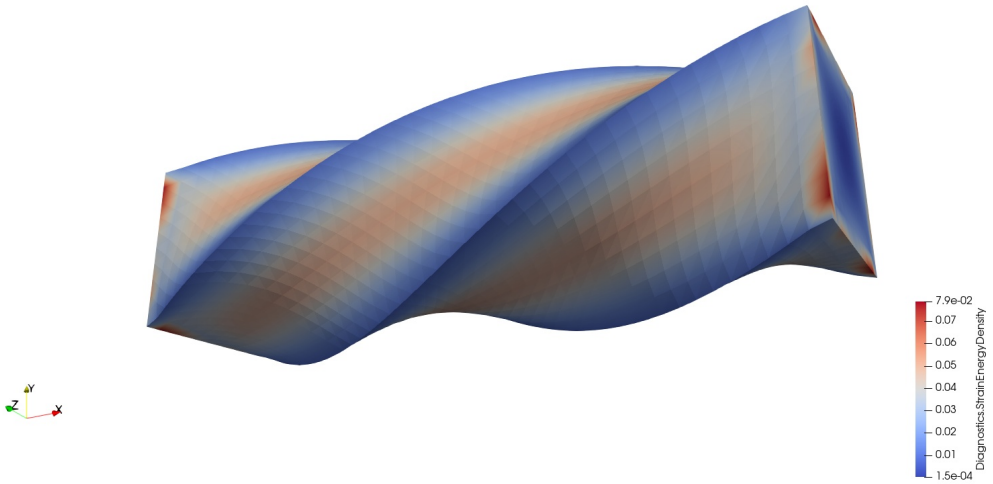

Figure 6: Strain energy density in a twisted Neo-Hookean beam.

`libCEED` also includes additional examples with PETSc, MFEM, and Nek5000 (*Nek5000*, 2021).

If MFEM is built with `libCEED` support, existing MFEM users can pass `-d ceed-cuda:/gpu/cuda/gen` to use a `libCEED` CUDA backend, and similarly for other backends. The `libCEED` implementations, accessed in this way, currently provide MFEM users with the fastest operator action on CPUs and GPUs (CUDA and HIP/ROCm) without writing any `libCEED` Q-functions.

Acknowledgements

This research is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nations exascale computing imperative. We thank Lawrence Livermore National Laboratory for access to the Lassen and Corona machines.

References

Abdelfattah, A., Barra, V., Beams, N., Brown, J., Camier, J.-S., Dobrev, V., Dudouit, Y., Ghaffari, L., Kolev, T., Medina, D., Pazner, W., Rathnayake, T., Thompson, J. L., & Tomov, S. (2021). *libCEED user manual* (Version 0.8). Zenodo. https://doi.org/10.5281/zenodo.4895340

Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Dobrev, J. C. V., Dudouit, Y., Fisher, A., Kolev, Tz., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., & Zampini, S. (2020). MFEM: A modular finite element library. *Computers & Mathematics with Applications*. https://doi.org/10.1016/j.camwa.2020.06.009

ANSI. (1978). Standard X3. 9-1978, programming language Fortran (revision of ANSI X2. 9-1966). In *New York: ANSI*.

Arndt, D., Bangerth, W., Blais, B., Clevenger, T. C., Fehling, M., Grayver, A. V., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Munch, P., Pelteret, J.-P., Rastak, R., Thomas, I., Turcksin, B., Wang, Z., & Wells, D. (2020). The `deal.II` library, version 9.2. *Journal of Numerical Mathematics*, *28*(3), 131–146. https://doi.org/10.1515/jnma-2020-0043

Babuška, I., & Suri, M. (1994). The p and $h-p$ versions of the finite element method, basic principles and properties. *SIAM Review*, *36*(4), 578–632. https://doi.org/10.1137/1036141

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., … Zhang, H. (2021). *PETSc users manual* (ANL-95/11 - Revision 3.15). Argonne National Laboratory.

Barra, V., Brown, J., Thompson, J., & Dudouit, Y. (2020). High-performance operator evaluations with ease of use: LibCEED's Python interface. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, & David Shupe (Eds.), *Proceedings of the 19th Python in Science Conference* (pp. 85–90). https://doi.org/10.25080/Majora-342d178e-00c

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Brown, J. (2010). Efficient nonlinear solvers for nodal high-order finite elements in 3D. *Journal of Scientific Computing*, *45*. https://doi.org/10.1007/s10915-010-9396-8

C foreign function interface for Python. (2021). https://cffi.readthedocs.io

CEED. (2021). https://ceed.exascaleproject.org/

CUDA. (2021). https://developer.nvidia.com/about-cuda

Deville, M. O., Fischer, P. F., & Mund, E. H. (2002). *High-order methods for incompressible fluid flow*. Cambridge University Press. ISBN: 0-521-45309-7

Fischer, P., Min, M., Rathnayake, T., Dutta, S., Kolev, T., Dobrev, V., Camier, J.-S., Kronbichler, M., Warburton, T., Świrydowicz, K., & Brown, J. (2020). Scalability of high-performance PDE solvers. *The International Journal of High Performance Computing Applications*. https://doi.org/10.1177/1094342020915762

Gropp, W., Lusk, E., & Skjellum, A. (2014). *Using MPI: Portable parallel programming with the message-passing interface*. ISBN: 9780262527392

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., R'ıo, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

HIP. (2021). https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html

International Standards Organisation. (1999). *ISO/IEC 9899: 1999 programming languages-c*. American National Standards Institute, New York.

Kirk, B. S., Peterson, J. W., Stogner, R. H., & Carey, G. F. (2006). `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, *22*(3–4), 237–254. https://doi.org/10.1007/s00366-006-0049-3

Kolev, T., Fischer, P., Min, M., Dongarra, J., Brown, J., Dobrev, V., Warburton, T., Tomov, S., Shephard, M. S., Abdelfattah, A., Barra, V., Beams, N., Camier, J.-S., Chalmers, N., Dudouit, Y., Karakus, A., Karlin, I., Kerkemeier, S., Lan, Y.-H., … Tomov, V. (2021). Efficient exascale discretizations: High-order finite element methods. *International Journal of High Performance Computing Applications*. https://doi.org/10.1177/10943420211020803

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based python JIT compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. https://doi.org/10.1145/2833157.2833162

LIBXSMM development site. (2021). http://github.com/hfp/libxsmm

Logg, A., Mardal, K.-A., Wells, G. N., & others. (2012). *Automated solution of differential equations by the finite element method: The FEniCS book* (A. Logg, K.-A. Mardal, & G. N. Wells, Eds.; Vol. 84). Springer. https://doi.org/10.1007/978-3-642-23099-8

Lottes, J. W., & Fischer, P. F. (2005). Hybrid multigrid/Schwarz algorithms for the spectral element method. *Journal of Scientific Computing*, *24*(1), 45–78. https://doi.org/10.1007/s10915-004-4787-3

MAGMA development site. (2021). https://bitbucket.org/icl/magma

MFEM: Modular Finite Element Methods Library. (2021). https://doi.org/10.11578/dc.20171025.1248

Nek5000. (2021). https://nek5000.mcs.anl.gov/

NVRTC. (2021). https://docs.nvidia.com/cuda/nvrtc/index.html

OCCA development site. (2021). http://github.com/libocca/occa

Orszag, S. A. (1980). Spectral methods for problems in complex geometries. *Journal of Computational Physics*, *37*, 70–92. https://doi.org/10.1016/0021-9991(80)90005-4

Python. (2021). https://www.python.org/

Rupp, K. (2020). *CPU-GPU-MIC comparision charts*. https://github.com/karlrupp/cpu-gpu-mic-comparison

Rust. (2021). https://www.rust-lang.org/

Straka, J. M., Wilhelmson, R. B., Wicker, L. J., Anderson, J. R., & Droegemeier, K. K. (1993). Numerical solutions of a non-linear density current: A benchmark solution and comparisons. *International Journal for Numerical Methods in Fluids*, *17*(1), 1–22. https://doi.org/10.1002/fld.1650170103

Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, *52*(4), 65–76. https://doi.org/10.1145/1498765.1498785

Zhang, J., Brown, J., Balay, S., Faibussowitsch, J., Knepley, M., Marin, O., Mills, R. T., Munson, T., Smith, B. F., & Zampini, S. (2021). The PetscSF scalable communication layer. *IEEE Transactions on Parallel and Distributed Systems*. https://doi.org/10.1109/TPDS.2021.3084070