Parallel Computing 108 (2021) 102841

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

GPU algorithms for Efficient Exascale Discretizations

Ahmad Abdelfattah [e], Valeria Barra [f], Natalie Beams [e], Ryan Bleile [i], Jed Brown [f], Jean-Sylvain Camier [a], Robert Carson [j], Noel Chalmers [h], Veselin Dobrev [a], Yohann Dudouit [a], Paul Fischer [b,c,d], Ali Karakus [m], Stefan Kerkemeier [b], Tzanio Kolev [a,*], Yu-Hsiang Lan [b], Elia Merzari [b,k], Misun Min [b], Malachi Phillips [c], Thilina Rathnayake [c], Robert Rieben [i], Thomas Stitt [i], Ananias Tomboulides [b,l], Stanimire Tomov [e], Vladimir Tomov [a], Arturo Vargas [i], Tim Warburton [g], Kenneth Weiss [i]

[a] *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550, United States of America*
[b] *Mathematics and Computer Science, Argonne National Laboratory, Lemont, IL 60439, United States of America*
[c] *Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States of America*
[d] *Department of Mechanical Science and Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States of America*
[e] *Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, United States of America*
[f] *Department of Computer Science, University of Colorado, Boulder, CO 80309, United States of America*
[g] *Department of Mathematics, Virginia Tech, Blacksburg, VA 24061, United States of America*
[h] *AMD Research, Advanced Micro Devices Inc., Austin, TX 78735, United States of America*
[i] *Weapons and Complex Integration, Lawrence Livermore National Laboratory, Livermore, CA 94550, United States of America*
[j] *Computational Engineering Division, Lawrence Livermore National Laboratory, Livermore, CA 94550, United States of America*
[k] *Department of Nuclear Engineering, Penn State, PA 16802, United States of America*
[l] *Department of Mechanical Engineering, Aristotle University of Thessaloniki, 54124, Greece*
[m] *Mechanical Engineering Department, Middle East Technical University, Ankara, 06800, Turkey*

A R T I C L E   I N F O

Keywords:
High-performance computing
GPU acceleration
High-order discretizations
Finite element methods
Exascale applications

A B S T R A C T

In this paper we describe the research and development activities in the Center for Efficient Exascale Discretization within the US Exascale Computing Project, targeting state-of-the-art high-order finite-element algorithms for high-order applications on GPU-accelerated platforms. We discuss the GPU developments in several components of the CEED software stack, including the libCEED, MAGMA, MFEM, libParanumal, and Nek projects. We report performance and capability improvements in several CEED-enabled applications on both NVIDIA and AMD GPU systems.

1. Introduction

Exascale computing will provide scientists and engineers with an advanced tool to explore physical phenomena over a large range of scales and in complex domains. To maximize this potential, the simulation codes must be efficient in their use of data movement, both in terms of implementation and algorithmic complexity. The DOE Center for Efficient Exascale Discretizations (CEED) [1,2] seeks to meet both of these goals by providing highly performant libraries for high-order discretizations on GPU-based compute nodes that form the basis for current- and next-generation HPC platforms.

Central to CEED is the use of matrix-free high-order finite element discretizations, which require only $O(n)$ data movement and yield exponential convergence rates, $O(h^p)$, for pth-order approximations to solutions having sufficient regularity. With the number of degrees of freedom scaling as $n = O\left((p/h)^d\right)$, in d dimensions, convergence through increased p offers clear advantages over simple reductions in the grid spacing h. Kreiss and Oliger [3] noted early on the particular relevance of increased approximation order in controlling cumulative dispersion errors for large-scale transport problems where propagated feature sizes λ are much smaller than the domain length, L, which is clearly in the scope of problems that are enabled by exascale architectures. Although exponential convergence is lost in many practical applications that lack regularity, high-order methods still provide favorable error constants with respect to norms and insidious sources of error such as numerical dispersion, and use of hp methods can sometimes restore exponential convergence [4]. Efficient implementation of methods of all orders, with a particular emphasis on high-order, is the principal objective of the CEED efforts.

* Corresponding author.
E-mail address: tzanio@llnl.gov (T. Kolev).

https://doi.org/10.1016/j.parco.2021.102841
Received 23 November 2020; Received in revised form 23 August 2021; Accepted 3 September 2021
Available online 25 September 2021

Practical application of spectral methods for complex domains was first considered by Orszag [5], who laid out several essential elements for performant implementations. The principal feature was the use of pth-order tensor-product polynomial approximations in a d-dimensional reference domain, $\mathbf{r} \in \hat{\Omega} = [-1,1]^d$, transformed to a complex domain, Ω, through an invertible map, $\mathbf{x}(\mathbf{r})$. Unlike Fourier bases, stable polynomial bases[1] can yield exponential convergence for non-periodic boundary conditions, provided the solution has sufficient regularity [6]. Orszag [5] noted that, although separability was lost in the transformed domain, the forward operator evaluation could still be effected efficiently using tensor-product sum factorization. He thus suggested using conjugate gradient iteration, preconditioned with spectrally-equivalent low-order (i.e., sparse) operators, to yield high-order accuracy at low-order costs. All three of these ideas—orthogonal-polynomial based bases, tensor-product sum factorization, and low-order preconditioners, are common elements of modern high-order finite element codes (e.g., [7–12]), although the low-order preconditioners are commonly supplanted with p-multigrid (e.g., [13,14]) and/or Schwarz-overlapping methods [15].

Let Q_p denote the Lagrange polynomial bases on Gauss–Lobatto quadrature points. In the case of tensor product elements, for p^d degrees-of-freedom per element, the use of tensor-product sum-factorization reduces operator evaluation costs from $O(p^{2d})$ to (near-optimal) $O(p^{d+1})$ and memory/storage costs from $O(p^{2d})$ to (optimal) $O(p^d)$. Matrix-free Q_p bases form the foundation for much of the CEED software stack, including MFEM, Nek5000/RS, libCEED, MAGMA, and libParanumal. We are also interested in high-order discretization on (non-tensor) triangular and tetrahedral P_p elements.

Throughout the paper we use the CEED bake-off problems (BPs), introduced in [16], in order to test and compare the performance of high-order codes. The CEED BPs are community benchmarks for matrix-free operator evaluation of mass (BP1), stiffness (BP3) or collocated stiffness matrix (BP5). They include a mixture of compute-intensive kernels, nearest-neighbor communication and vector reductions that is representative of high-order applications. See [16] for details.

Both the *tensor* and *non-tensor* cases pose unique challenges for high-order algorithms on GPU platforms. In the following Section 2 we describe the GPU-specific developments that are addressing these challenges in each of the CEED open-source libraries. We follow this by a discussion in Section 3, results from several CEED-enabled applications in Section 4, and conclusions in Section 5.

2. GPU developments in the center for efficient exascale discretizations

The major *work intensive* operations for pth order elements are local interpolation and differentiation, which involve tensor contractions for Q_p and DGEMMs for P_p elements. For example, for tensor elements in 3D derivatives at nodal points $(r_i, s_j, t_k) \in \hat{\Omega}$ take the form $u_r^e(r_i, s_j, t_k) = \sum_m \hat{D}_{im} u^e_{mjk} = D_r \underline{u}^e$, where the u^e_{ijk} are the local basis coefficients for $u^e(\mathbf{r})$ on Ω^e and $\hat{D}$ is a dense $(p+1) \times (p+1)$ derivative matrix.

On a CPU, performance in the *tensor* Q_p case relies primarily on casting the tensor contractions, which comprise 90% of the flops, as optimized matrix–matrix products. On GPUs, vectorization must be expressed over the entire set of E elements local to a given GPU in order to leverage the node-parallel architecture and amortize kernel launch overhead. Kernels thus tend to be expressed at the operator level, such as the discrete Laplacian, $\underline{w}_L = A_L \underline{u}_L$, where A_L=block-diag(A^e), with $A^e = \mathbf{D}^T \mathbf{G}^e \mathbf{D}$ the matrix-free form of the stiffness matrix. For deformed elements A^e is dense, with $(p+1)^6$ entries. The factored form, however, involves only the tensor-product derivatives, D_r, D_s, and D_t, and six diagonal matrices, $G^e_{rr}, G^e_{rs}, \ldots, G^e_{tt}$ in the symmetric tensor $\mathbf{G}^e$.

[1] Stable bases include orthogonal polynomials or Lagrange polynomials based on Gauss-type quadrature points.

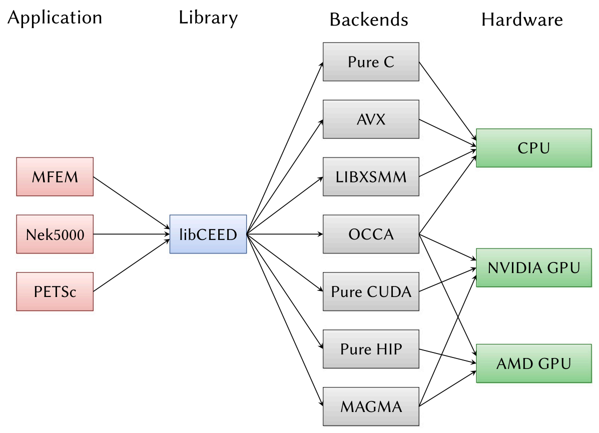

Fig. 1. libCEED allows different applications to share highly optimized discretization kernels.

Including $\underline{u}^e$, the number of reads is thus $7(p+1)^3$ per element, with a corresponding flop count of $12(p+1)^4 + 15(p+1)^3$ (double the first term for non-collocated quadrature). The $O(p)$ arithmetic intensity (flop-to-byte ratio) with low-memory data structures leads to high performance on GPU and CPU architectures that have sufficient registers/cache to exhibit memory locality through the sequence of dependent operations.

In the rest of this section, we describe the developments in each of the CEED packages that address the multitude of issues that arise when implementing operations of the form $A_L \underline{u}_L$ on GPUs. For detailed description of these operations see [16].

2.1. libCEED

libCEED [17,18] is a new library that offers a purely algebraic interface for matrix-free operator evaluation and supports run-time selection of implementations tuned for a variety of computational device types, including CPUs and GPUs. libCEED's purely algebraic interface can unobtrusively be integrated in new and legacy software to provide performance portable interfaces. While the focus is on high-order finite elements, the approach is algebraic and thus applicable to other discretizations in factored form. libCEED's role, as a lightweight portable library that allows a wide variety of applications to share highly optimized discretization kernels, is illustrated in Fig. 1, where a non-exhaustive list of specialized implementations (backends) is listed. libCEED provides a low-level Application Programming Interface (API) for user codes so that applications with their own discretization infrastructure (e.g., those in PETSc, MFEM and Nek5000) can evaluate and use the core operations enabled by libCEED. GPU backends are available via pure CUDA or HIP implementations, as well as the OCCA and MAGMA libraries. libCEED provides a unified interface, so that users only need to write a single source code and can select the desired specialized implementation at run time. Moreover, each process or thread can instantiate an arbitrary number of backends.

Since matrix-free finite element algorithms move potentially $O(p^d)$ times less data than algorithms using sparse matrices, where p is the polynomial order, and d the dimension, $O(p^d)$ speedup can theoretically be achieved on architectures where the matrix-free algorithms are limited in performance by the data movements, i.e. are memory bound.

The matrix-free algorithms to apply *tensor* finite element operators are completely memory bound on GPU due to their low arithmetic intensities. $O(p^d)$ speedup can be achieved in practice for tensor finite element when compared to a standard approach using a sparse matrix, see Fig. 2, because the GPU kernels are solely memory bound.

However, applying *non-tensor* finite element operators does not necessarily result in memory bound GPU kernels. The libCEED interface allows the caller to provide arrays representing the evaluation and

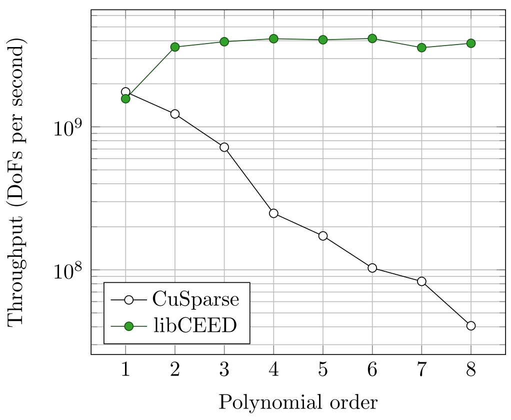

Fig. 2. Comparison of the performance in the saturated regime for the CEED benchmark problem BP1 [16] on a NVIDIA V100 using a sparse matrix (wit `code CuSparse`) against the `cuda-gen` backend of libCEED for different polynomial orders p.

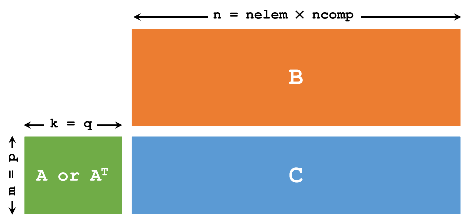

Fig. 3. Shape of the DGEMM operation for the non-tensor basis action in libCEED.

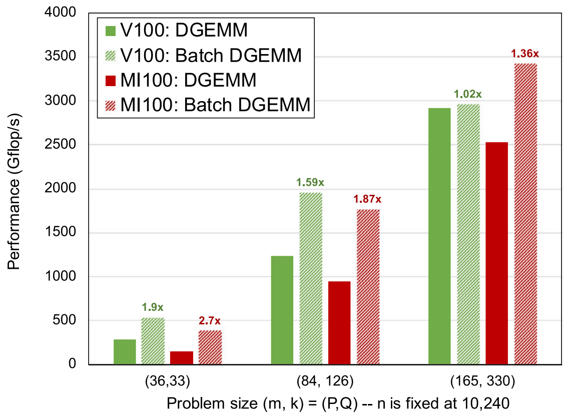

Fig. 4. Performance of different DGEMM configurations using hipBLAS and cuBLAS. Results are for different (P, Q) pairs on a Nvidia V100 (CUDA 11.2) and an AMD Instinct™ MI100 (ROCm 4.2) GPUs.

gradient of the basis functions along with the quadrature weights to be used. With the arithmetic intensity of the matrix-free algorithms increasing in $O(p^d)$ for general non-tensor elements – compared to $O(p)$ in the tensor element case – GPU kernels quickly become computationally bound for high p orders on sufficiently many elements, especially in 3D. The higher number of basis functions and quadrature points for non-tensor elements also results in operations that are more difficult to cache efficiently on the very limited memories of GPU caches and registers. Therefore, achieving peak performance for non-tensor finite elements is more challenging, and the theoretical gain is not as interesting as for tensor finite elements. We note that there exist specialized sum factorization schemes for collapsed elements (e.g. [19,20], with recent SIMD vectorization targeting CPU performance [10]) and other fast evaluation methods such as [21,22]; adding such methods to libCEED would require addition of a public interface and dedicated backend support.

On NVIDIA GPUs the libCEED library achieves close to peak performance for operators using tensor finite elements. The performance on the CEED benchmark problems is comparable to state-of-the-art specialized hand tuned kernels [23].

The CUDA and HIP backends provide native support for non-tensor finite elements, while the OCCA and MAGMA backends, which depend on their eponymous libraries [24,25], also support non-tensor finite elements. The MAGMA backend achieves the highest performance of all libCEED GPU backends for non-tensor finite elements.

2.2. MAGMA

MAGMA [25] is a high-performance linear algebra library that includes LAPACK for GPUs, BLAS, sparse iterative solvers, and many other general matrix computation kernels. The batched computations provided by MAGMA can be generalized to provide highly efficient tensor computations [26]. While the libCEED MAGMA backend contains specialized tensor basis kernels separate from the MAGMA library itself, the library's batched GEMM capabilities are used directly to optimize non-tensor basis computations, with a goal of hardware portability [27]. In contrast to the recent CPU-based work of Sun et al. [11], which applies code generation and transformations toward SIMD vectorization across batches of (tensor and non-tensor) element kernels, the MAGMA backend's batched computations leverage standard library BLAS routines for the non-tensor basis actions within libCEED's algebraic framework.

As the non-tensor basis computations in libCEED are basis- and quadrature-rule-agnostic, the full interpolation or gradient matrices must be applied for an input vector for every element, rather than performing a series of small tensor contractions. If we consider all elements local to the process at the same time (`nelem`), we can reshape the input vector to be a matrix of size $d \times P \times$ `nelem` $\times$ `ncomp`, with each column corresponding to one component for one element, and P and Q representing the total number of basis and quadrature points in the element, respectively. Now the application of the interpolation or gradient matrix action for all the elements is easily represented by one standard general matrix–matrix multiplication, $C = AB$, as represented in Fig. 3 for input B, output C, and basis matrix A.

Fig. 3 shows a typical shape for the GEMM call in a libCEED non-tensor basis computation, with dimensions (m, n, k). Here m and k are relatively small, as they are tied to the number of basis nodes or quadrature points in one element; n can potentially be orders of magnitude larger, as it depends on the number of elements in the local operator. Since vendor-provided GEMM routines are generally designed to achieve maximum performance for square matrices, these unbalanced dimensions may prevent the GEMM operation from reaching the GPU peak performance. Therefore, to make the best possible use of the available BLAS libraries, we also consider performing the GEMM operation in Fig. 3 as a batched GEMM operation, split across the n dimension, so that each batched operation has the same A matrix, but uses submatrices $\hat{B}$ and $\hat{C}$ of B and C, with η columns each. The additional parameter of batch size η increases the space in which we can search for the best possible parameter set, given dimensions m and k, with the goal of creating a more balanced workload for the GPU. Transforming the GEMM in Fig. 3 into a batched GEMM does not require setting up pointer arrays that may impact the performance, and thus does not add any overheads. Both cuBLAS and MAGMA provide stride-based batched GEMM kernels.

Fig. 4 shows example performance numbers for the GEMM versus the batched GEMM for typical sizes encountered in the MFEM-libCEED BP3 [16] benchmark for triangle or tetrahedron non-tensor elements. The figure considers an NVIDIA V100 GPU and initial experience with an AMD MI100 GPU. The best-performing routine of MAGMA and the vendor-provided BLAS is shown for each GPU. We see that batching the

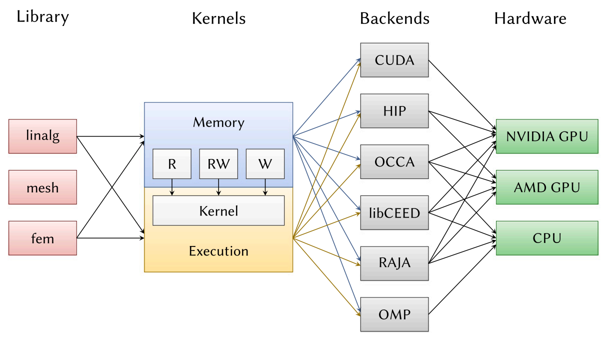

Fig. 5. Diagram of MFEM's modular design for accelerator support, combining flexible memory management with runtime-selectable backends for executing key finite element and linear algebra kernels.

GEMM operation across the n dimension achieves a better performance than launching a single GEMM operation for these sizes, with the speedup relative to the single GEMM indicated above each batched GEMM bar.

To determine the best possible combination of routine (vendor BLAS or MAGMA library) and batch size η (with a standard, non-batch call corresponding to $\eta = n$), we use data from offline benchmark parameter sweeps to construct a lightweight abstraction layer. This layer automatically selects the best choice for the non-tensor GEMM operations. In [27], we show that for higher orders of basis functions, the benefit of the optimized GEMM formulation is clear in comparison to the pure CUDA kernels in the "cuda-ref" backend (up to 10× speedup for the MAGMA formulation on the V100 GPU).

2.3. MFEM

MFEM [12] is a general-purpose finite element library that since version 4.0 supports hardware accelerators, such as GPUs, as well as programming models and libraries, such as CUDA, HIP, OCCA [24], libCEED [17], RAJA [28] and OpenMP. The goal of the MFEM developments in CEED is to provide state-of-the-art optimized performance high-order kernels to applications in an ease of use, flexible form.

The MFEM performance portability approach is based on a system of backends and kernels working seamlessly with a lightweight memory spaces manager. A distinctive feature of this approach is the ability to select the backends at runtime. For instance, different MPI ranks can choose different backends (like CPU or GPU), allowing applications to take full advantage of heterogeneous architectures. Another important aspect of MFEM's approach is the ability to easily mix CPU-only code with code that utilizes the new backends, thus allowing for selective gradual transition of existing capabilities. Most of the kernels are based on a single source, while still offering good performance. For performance-critical kernels, where a single source does not provide good performance, the implementation introduces dispatch points based on the selected backend and, in some cases, on kernel parameters such as the finite element order.

Fig. 5 illustrates the main components of MFEM's modular design for accelerator support. The *Library* side of MFEM (on the left) represents the software components where new kernels have been added. *Kernels* and *memory management* are the two ingredients most programming models have to deal with when providing such an abstraction to address code portability for HPC platforms. Similarly to Fig. 1 the MFEM design allows for a variety of runtime-selectable backends that can execute its kernels on both CPU and GPU hardware. Unlike the libCEED figure though, Fig. 5 includes the kernel abstraction for a much broader range of meshing, finite element and linear algebra features that a general finite element library like MFEM needs to support. For more details, see [12,16] and Section 3.

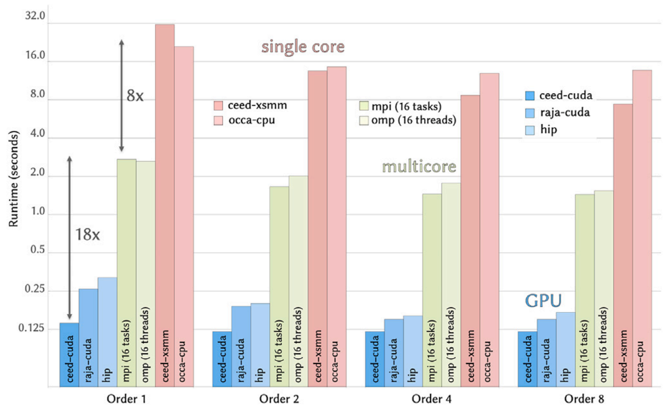

Fig. 6. Performance results with selection of the backend available in MFEM v4.2: 2D Poisson problem with 1.3 million degrees of freedom solved using 200 unpreconditioned CG iterations, using Intel Xeon Gold 6130@2.1 GHz CPU plus NVIDIA GV100 (ceed-raja, raja-cuda, CUDA 10.1) and AMD Radeon Instinct^{TM} MI60 GPUs (hip, ROCm 3.8). Switching from serial to parallel execution on a desktop workstation leads to an order of magnitude performance improvement (note: y-axis is logarithmic). Using the desktop GPU results in another order of magnitude performance. Note that these results are representative for the state of the MFEM backends as of version 4.2. We do not claim that they represent a fair comparison between CPUs and GPUs because not all backends are fully optimized. (For example much better CPU results are reported in [13] and [14].)

MFEM's GPU acceleration has demonstrated excellent performance in a number of single-GPU and multi-GPU benchmarks. The high-order algorithms in MFEM are particularly well suited for GPUs as shown by the results in Fig. 6 which report performance results from LLNL's Corona machine and a Linux configuration similar to the compute nodes of LLNL's Sierra supercomputer. We note that hand-tuning is still required for good performance and the AMD results are preliminary.

2.4. libParanumal

The Paranumal project [29] started at Virginia Tech in 2017 as a new GPU effort targeting 90% of the capabilities of the CPU version of Nek5000. It was not originally designed as a user facing library but rather as a collection of high-order finite element streaming benchmarks (streamParanumal), CEED benchmarks (benchParanumal), and self contained mini-apps (libParanumal). These were all developed ab initio using the Open Concurrent Computing Abstraction (OCCA) [24] and kernel language (OKL) [30]. Although OKL is a generic portable kernel programming language the initial kernels were optimized for NVIDIA P100 and V100 GPUs, see [23]. By design the libParanumal OKL kernels use intrinsic C types without recourse to structs and classes facilitating their use in other projects.

Many finite element operations are heavily memory bound including Krylov updates, finite element gather and scatter operations that admit high throughput hardware agnostic implementations for both the NVIDIA and AMD GPUs [31]. Implementations of these kernels have been released in the streamParanumal standalone benchmark suite [32].

Portable implementations of the CEED BP benchmarks [16] are available in the benchParanumal library. Figs. 7 and 8 show the performance of the benchParanumal implementations for the CEED BP1 and BP5 benchmark respectively on a single AMD MI100 (ROCm 3.9) and NVIDIA V100 SXM2 (CUDA 10.1). The benchmarks achieved sustained average memory bandwidth depending on polynomial degree and mesh size of up to 950 GB/s (AMD MI100) and 800 GB/s (NVIDIA V100 SXM2) despite involving sequences of tensor contractions for each element in the matrix–vector operations.

The libParanumal library is a self contained high-order finite element library that uses the same highly optimized OKL kernels as

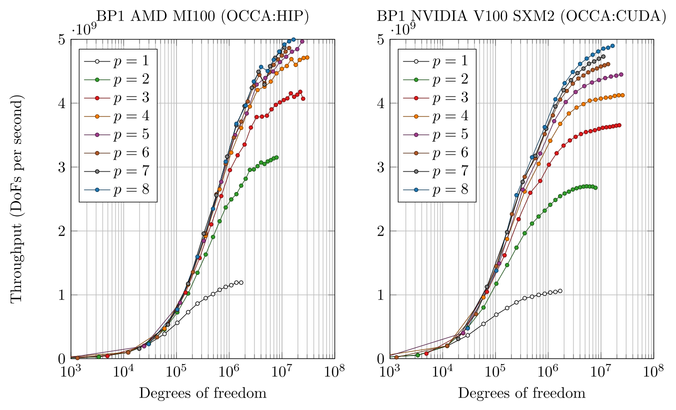

Fig. 7. Performance of the benchParanumal version of the CEED benchmark problem BP1 on a single GPU of HPE/Tulip AMS Instinct™ MI100 (left) and NVIDIA V100 SXM2 (right).

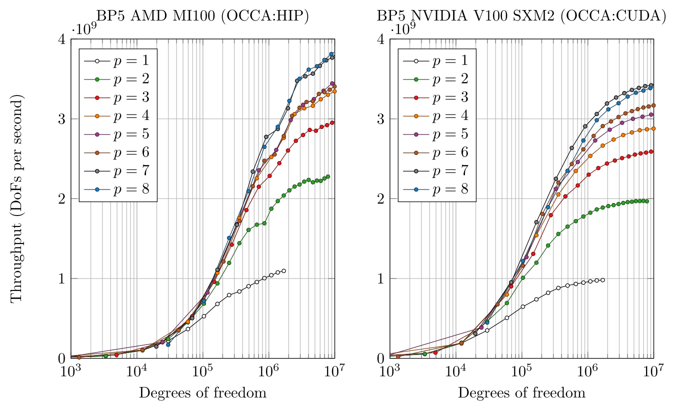

Fig. 8. Performance of the benchParanumal version of the CEED benchmark problem BP5 on a single GPU of HPE/Tulip AMD Instinct™ MI100 (left) and NVIDIA V100 SXM2 (right).

the streamParanumal and benchParnaumal benchmark suites. It also includes sub-libraries for dense linear algebra, Krylov solvers, parallel mesh handling and polynomial approximation, p-type and algebraic multigrid, time stepping, gather-scatter operations and halo exchanges, and core miscellaneous operations. The libParanumal sub-libraries support meshes consisting of triangles, quadrilaterals, tetrahedra, or hexes. The libParanumal project also includes mini-apps demonstrating GPU accelerated PDE solvers for linearized acoustics, scalar advection, Galerkin–Boltzmann finite moment gas-dynamics, compressible Navier–Stokes, elliptic equations, Fokker–Planck, and incompressible Navier–Stokes. Each solver supports multi-GPU simulation via Nek's gslib for efficient MPI based gather-scatter operations and halo exchanges [33].

The libParanumal library is highly modular and early fork of the project has been used with modifications as a platform for the custom physics requirements of the Nek5000 user community [34] as described in the next section albeit without the most recent developments in portable streaming and operator kernels. The algorithms and design principles of libParanumal kernels have also influenced the design of kernels produced by the libCEED `cuda-gen` backend (see Section 2.1). libParanumal is also beginning to be used in non-CEED projects as for example in high fidelity time-dependent room acoustic modeling [35].

2.5. Nek5000/RS

Nek5000 [36] is a spectral element code that is used for a wide range of thermal-fluids applications. A companion code, NekCEM [37], is used for computational electromagnetics. These codes have scaled to millions of MPI ranks using the Nek-based *gsLib* communication library to handle all near-neighbor and other stencil type communications (e.g., for algebraic multigrid) [38]. On CPUs, tensor contractions constitute the principal computational kernel (typically > 90% of the flops). These can be cast as small dense matrix–matrix products resulting in high performance with a minor amount of tuning [39].

For GPU-based platforms, node-level parallelism requires kernels written at a higher level than simple tensor contractions. Early GPU ports started with NekCEM [40], using OpenACC and running on OLCF/Titan up to 16,384 NVIDIA K20X GPUs. The OpenACC-based implementation and performance studies were extended to Nek5000 [41, 42].

For portability and performance reasons, we decided to develop a new version of Nek5000, called NekRS, which is written in C++/OCCA [24]. The NekRS kernels started as a fork from libParanumal in late 2018 and were tailored and expanded to meet the specific requirements of large-scale turbulent flow simulations in complex domains (e.g., as illustrated in Fig. 10). It retains access to the standard Nek5000 interface, which allows users to leverage existing user-specific source code such as statistical analysis tools for turbulence.

Several recent developments in NekRS have led to significant performance gains on Summit. These include (i) An accelerator-oriented variant of gslib [33] that selects from several communication strategies, including pack/unpack on the host or device, and GPU-direct or host-based communication. Runtime-adaptation picks the fastest strategy. (ii) Chebyshev-accelerated additive Schwarz smoothing. This approach combines the standard Nek5000 additive Schwarz method with the Chebyshev-accelerated Jacobi smoothing provided in libParanumal (and, e.g., deal.ii [14]). Local Schwarz solves are performed with fast-diagonalization implemented with tensor contractions that have a complexity that is on par with operator evaluation. (iii) Projection-based initial guesses to avoid redundant iteration work in successive timesteps [43,44]. The performance impact of these developments are described in detail in [34].

The advantage of basing NekRS on OCCA is clear from the results of Table 1, which demonstrates full Navier–Stokes performance results for NVIDIA and AMD GPUs. The table shows a single-GPU comparison of the averaged-walltime per timestep for the MI60, MI100, and A100, compared to a single V100 on Summit. We remark that extensive tuning has been applied for the V100, which has been the primary development platform for NekRS. Despite this, the performance on the other GPUs is within the scope of what we would expect for these nodes. The A100 performs remarkably well, with speedup 1.5× of the performance of the V100 and at a near-strong-scale-limit value of $n = 2.22$M gridpoints on a single GPU. Despite the fact that the AMD GPU code has seen less tuning than the NVIDIA code the fact that the performance is on par illustrates the portability provided by the OCCA base.

Table 1
NekRS Navier–Stokes performance on a single GPU of HPE/Tulip AMD Instinct™ MI100, AMD Radeon Instinct™ MI60, NVIDIA V100 PCIe and ALCF/Theta-GPU NVIDIA A100 SXM4, compared to OLCF/Summit NVIDIA V100 SXM2, for turbulent pipe flow simulation with $Re = 19,000$, $E = 6840$, $p = 6$, and $n = 2,346,120$. Time per step in seconds (t_{step}) is averaged over 100 steps. R is the ratio of t_{step} on Summit V100 to that on other systems.

System	Device	Backend	$t_{step}(s)$	R
Summit	V100	CUDA	8.51e−02	1
Tulip	MI100	HIP	9.96e−02	0.85
Tulip	MI60	HIP	1.41e−01	0.60
Tulip	V100	CUDA	8.85e−02	0.96
Theta-GPU	A100	CUDA	5.59e−02	1.52

3. Discussion

In this section we share some of the porting experiences on the CEED project, and discuss some of the GPU lessons we have learned.

Overall, we have found that porting to GPU architectures is a disruptive process, similar to the transition from serial to MPI parallel programming. As such, we recommend to start a new code for the GPU port, if possible, (as with NekRS and libParanumal) as opposed to incrementally porting an existing code (as with MFEM). We also recommend taking advantage of GPU-accelerated libraries, such as libCEED, when applicable. For low-order applications, the CEED software also provides access to new classes of algorithms (high-order methods) that can take better advantage of GPU hardware compared to traditional low-order approaches [16].

For new codes, we have found the use of OCCA and its OKL language a pragmatic choice that has allowed us to make quick progress in capturing e.g. the capabilities of Nek5000 in libParanumal without choosing a specific manufacturer's GPU programming model since OCCA translates OKL code into CUDA, HIP, OpenCL, or OpenMP at runtime for native just-in-time (JIT) compilation. OCCA has also been instrumental in the exploration of the high-order algorithmic space, as different versions of the CEED kernels can be easily implemented, modified and tested with it. Examples are provided with the OCCA distribution that demonstrate the simplicity of the API and kernel language [45].

For the porting of existing codes, we have found that integration of kernels at the *for-loop* level, as with Kokkos and RAJA, has several important benefits. For example, in MFEM, the original code was transformed to use a new *for-loop* abstraction defined as a set of code `MFEM_FORALL` macros, in order to take advantage of various *backends* supported via the new macros. This approach allows for gradual code transformations that are not too disruptive for both MFEM developers and users. Existing applications based on MFEM are able to continue to work as before with easy transition to accelerated kernels. This approach also allows interoperability with other software components and external libraries that can be used in conjunction with MFEM (e.g., *hypre*, PETSc, SUNDIALS). The main challenge in this transition to *kernel-centric* implementation is the need to transform existing algorithms to take full advantage of the increased levels of parallelism in the accelerators while maintaining good performances on standard CPU architectures.

An important aspect of GPU programming is the need to manage memory allocation and transfers between the CPU (host) and the accelerator (device). This can be a frequent source of bugs and inefficiencies in complex applications. For example in MFEM, a special code `Memory` class was introduced to manage a pair of host and device pointers and provides a simple interface for copying or moving the data when needed. An important feature of this `Memory` class is the ability to work with externally allocated host and/or device pointers which is essential for interoperability with other libraries. The code `Memory` has also been useful in the porting of MFEM-based applications, see Section 4.4.

Finally, the optimization of GPU kernels really requires understanding of the GPU hardware and its multi-level memory hierarchy as well as advanced techniques such as code generation and JIT compilation. To illustrate these points, we describe in the rest of the section the sequence of developments that led to the `cuda-gen` backend of libCEED, which achieves close to peak performance for high-order operator evaluation with tensor finite elements on NVIDIA GPUs.

The first libCEED CUDA backend was the reference backend `cuda-ref`, which established a blueprint for GPU porting in libCEED. The main difficulty at this point was to handle all the runtime aspects of libCEED. Being able to produce efficient GPU kernels relies on the compiler knowing as much as possible during compilation, which conflicts with the generality of the libCEED approach where users are free to specify e.g., polynomial order and number of quadrature points used at runtime. For this reason, it was critical to use JIT compilation to generate on the fly GPU kernels with as much information as we could provide at runtime. In general, we believe that JIT compilation will play an important role in HPC in the future, and is essentially a requirement for high-order applications with runtime order selection.

The second libCEED CUDA backend, `cuda-shared`, focused on optimizing each GPU kernel individually to achieve peak performance. The tensor kernels are highly memory bound, so the challenge was to use the different memory bandwidths efficiently. Typically, the bottlenecks are the local/shared memory bandwidths and the memory access patterns to the data. If we compare the `cuda-ref` and `cuda-shared` backends in Fig. 9, we see that for low orders (1 to 3) the performance of the `cuda-ref` and `cuda-shared` backends are similar, the `cuda-ref` kernels do not yet saturate the local/shared memory bandwidth. However, for orders higher to 4, we observe that the `cuda-ref` backend performance deteriorates with the order. This is due to local/shared memory bandwidth getting more and more saturated. On the other hand, the `cuda-shared` manages by careful memory accesses and unrolling loops to continue saturating the global memory bandwidth and thus achieves high performance for each GPU kernel.

The final, and best performing backend, `cuda-gen` uses a code generation approach, based on the `cuda-shared` backend, to generate at runtime (with JIT compilation) a unique optimized GPU kernel representing the whole operator. Since the `cuda-ref`, `cuda-shared` are decomposing the matrix free operators in a sequence of GPU kernels, they require the storage, in global memory, of unnecessary temporary results in between each kernel launch. Fusing GPU kernels prevents these unnecessary data storage and movements between kernel launches resulting in a 2-3 time speedup over the `cuda-shared` backend, see Fig. 9, and around 5 time speedup over the reference backend `cuda-ref` on the CEED benchmark problem BP3.

4. Applications

The CEED effort includes the development of algorithms, software, simulation and modeling, performance analysis and optimization for CEED-engaged applications. While we are focused on exascale applications in the ECP, CEED is also extending its contribution to a broader range of engineering and science application areas, such as nuclear energy, wind energy, fusion, solid mechanics, additive manufacturing, internal combustion, and recent extension to weather modeling and aerosol transport research. In this section, we demonstrate the impact of the CEED-developed open source codes, Nek5000/RS and MFEM with full simulation capability, scaling on various acceleration architectures (including the full scaling performance on Summit GPUs), in DOE's ExaSMR, ExaWind, NEAMS, MARBL, and ExaAM projects.

4.1. ExaSMR

ExaSMR's target geometry is a small modular reactor assembly comprising 37 bundles, each having a 17×17 array of rods, which totals to ∼10,000 long communicating channels. For development, we consider two geometries: a very long single 17×17 bundle, and a collection of 37 such bundles that are shorter in length. We analyzed NekRS performance behaviors for both geometries. Detailed performance for various geometries is discussed in [34]. Here we present the baseline performance of the long 17×17 bundle, illustrated in Fig. 10(a), having the ratio between the characteristic length L and the rod diameter D as $L/D \approx 288$.

Table 2 demonstrates strong- and weak-scaling runs out to 175 million elements on Summit—roughly twelve times larger than 15M-element that were "hero calculations" on Mira as recently as 2020. We measured the average-walltime per step in seconds, t_{step}, using 101–200 steps for simulations with $Re_D = 5000$. For the strong scaling, we used $E = 175{,}618{,}000$ and $N = 7$, totaling 60 billion grid points. We observe the 17×17 rod-bundle case continues to scale well to all of Summit,

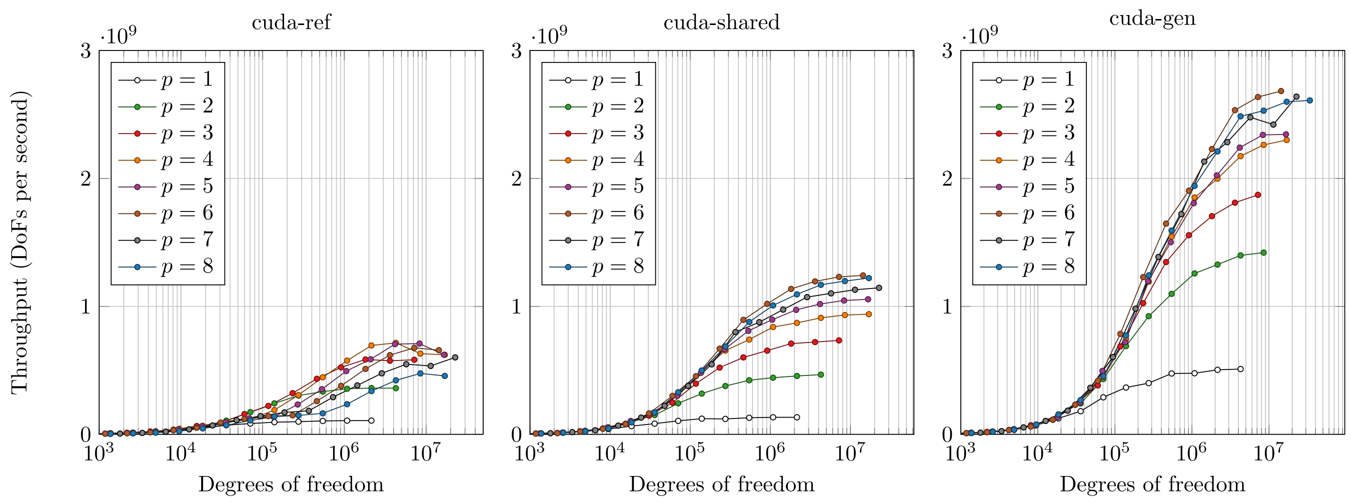

Fig. 9. Performance of the `cuda-ref` (left) `cuda-shared` (center) and `cuda-gen` (right) backends of libCEED for the CEED benchmark problem BP3 on an NVIDIA V100 GPU.

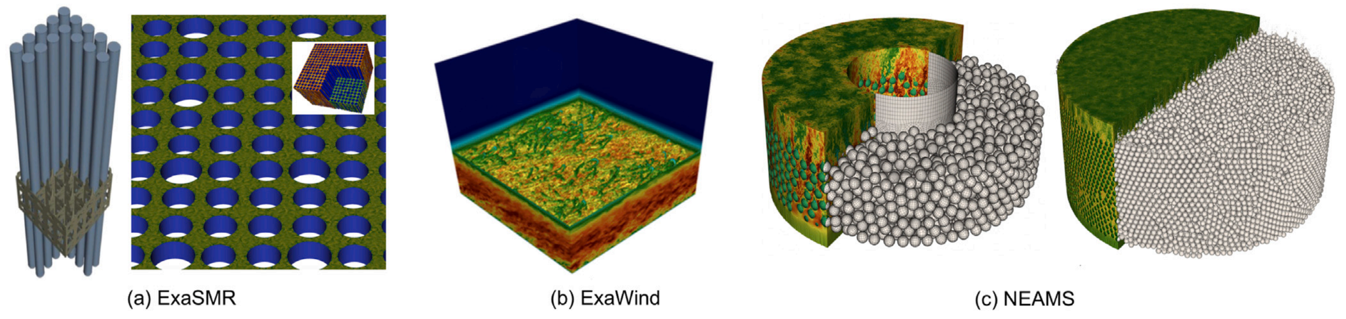

Fig. 10. Nek5000/RS applications: (a) ExaSMR's 17×17 fuel rod configuration and turbulent flows profile, (b) ExaWind's atmospheric boundary layer modeling and analysis, (c) NEAMS's pebble-bed reactor configurations with simulation demonstrating turbulent flows past 3344 pebbles in an annulus and 44257 pebbles in a cylinder.

Table 2
ExaSMR: NekRS strong and weak scaling performed on Summit, using 6 GPUs per node, for simulating turbulent flow in the 17 × 17 rod-bundle of Fig. 10(a), right, with $Re_D = 5000$. Time per step in seconds (t_{step}), velocity iteration count (v_i), and pressure iteration count (p_i), are all averaged over 100 steps. R is the ratio of t_{step} of 1810 nodes to that of others for strong scaling and t_{step} of 87 nodes to that of others for weak scaling, provided with the ideal ratio, $\mathrm{R}_{\mathrm{ideal}}$ and the parallel efficiency, $\mathrm{P}_{\mathrm{eff}}$.

ExaSMR application performance: 17 × 17 fuel rods simulation												
Case	Node	gpu	E	N	E/gpu	n/gpu	v_i	p_i	$t_{step}(s)$	R	$\mathrm{R}_{\mathrm{ideal}}$	$\mathrm{P}_{\mathrm{eff}}$ (%)
Strong	1810	10860	175618000	7	16171	5.5M	4	2	1.855e−01	1.00	1.00	100
	2536	15216	175618000	7	11542	3.9M	4	2	1.517e−01	1.22	1.40	87
	3620	21720	175618000	7	8085	2.7M	4	2	1.120e−01	1.65	2.00	82
	4180	25080	175618000	7	7002	2.4M	4	2	1.128e−01	1.64	2.30	71
	4608	27648	175618000	7	6351	2.1M	4	2	1.038e−01	1.78	2.54	70
Case	Node	gpu	E	N	E/gpu	n/gpu	v_i	p_i	$t_{step}(s)$	R	$\mathrm{R}_{\mathrm{ideal}}$	$\mathrm{P}_{\mathrm{eff}}$ (%)
Weak	87	522	3324000	7	6367	2.1M	4	2	8.57e−02	1.00	1.00	100
	320	1920	12188000	7	6347	2.1M	4	2	8.67e−02	0.98	1.00	98
	800	4800	30470000	7	6347	2.1M	4	2	9.11e−02	0.94	1.00	94
	1600	9600	60940000	7	6347	2.1M	4	2	9.33e−02	0.91	1.00	91
	3200	19200	121880000	7	6347	2.1M	4	2	9.71e−02	0.88	1.00	88
	4608	27648	175618000	7	6351	2.1M	4	2	1.03e−01	0.83	1.00	83

using $n/P = 2.1M$ with 70% parallel efficiency from the base of 1810 nodes using $n/P = 5.5M$, where P is the number of V100s. We see 80% efficiency is sustained for $n/P = 2.6M$. The weak scaling uses the meshes increased by 120, 440, 1100, 2200, 4400, and 6340 layers in the streamwise (z) direction, extruded from a two-dimensional 17 × 17 mesh having $E =$ 27,700 spectral elements. Weak-scaling this problem from 271 to 4608 nodes (1626 to 27648 GPUs) sustains more than 80% parallel efficiency throughout, using $2.1M$ grid points per GPU.

We note that the pressure iteration counts, p_i, are relatively very low for the 17 × 17 bundle compared to the pebble cases, which have $p_i \sim 8$ for the same timestepper and preconditioner. The geometric complexity of the 17 × 17 rod-bundle is relatively mild compared to the pebble case and also the synthetic initial condition does not quickly transition to full turbulence. We expect higher pressure iteration counts (e.g., $p_i \sim$ 4–8) once fully turbulent flow is established for this case.

4.2. ExaWind

Efficient simulation of atmospheric boundary layer flows (ABL) is important for the study of wind farms, urban canyons, and basic weather modeling. In collaboration with the ECP ExaWind team, we identified a well-documented test case, the Global Energy and Water Cycle Experiment Atmospheric Boundary Layer Study (GABLS), to demonstrate the suitability of high-order methods for large eddy simulations (LES) of the ABL. Initial convergence results of an LES study with Nek5000 are shown in Fig. 10(b). We have initiated a performance study for this problem on Theta-GPU (A100s) and Summit (V100s) for an $E = 32768$ spectral element mesh with $N = 7$ (i.e., n=11.2M). Our simulations represent turbulent flows on the physical domain [400 m × 400 m × 400 m] with geostrophic wind speed of 8 m/s in x-direction and reference potential temperature of 263.5 K with no-slip boundary as well as wall functions based on log-law at the lower wall, otherwise periodic boundary conditions. Single-node scaling shows the 80% strong-scale limit to be 1.8M points/GPU for both the V100 and A100, with the A100 running at .055s/step and 1.55 times faster than the V100.

4.3. NEAMS

Fig. 10(c) demonstrates turbulent flows past 3344 pebbles in an annulus (left) and 44257 pebbles in a cylinder (right) that were computed with NekRS on Summit using 840 GPUs and 1788 GPUs, respectively. The cylinder case has 13M elements of order $N = 7$, for a total of 4.4B grid points. The annulus configuration is a prototype for pebble-bed reactor configurations that are being studied by the DOE's Nuclear Energy Advanced Modeling and Simulation project. We have developed a novel meshing strategy for generating high-quality hexahedral element meshes that ensure accurate representation of densely packed spheres for these geometries [46]. The meshing algorithm includes Voronoi tessellation, edge collapse, facet projection onto the spheres, and mesh smoothing with quality measurements. These simulations strong-scale well and the NEAMS target configuration of an annulus with 300,000 pebbles will require about 30B grid points, which is well within the current performance envelope on Summit.

4.4. MARBL

MARBL is a next-gen multi-physics simulation code being developed at LLNL. The code provides multi-material radiation-magneto-hydrodynamics with applications in inertial confinement fusion (ICF), pulsed power and equation of state/material strength experiments as part of the NNSA ATDM program. One of the central components of MARBL is the BLAST package [47], which uses an ALE formulation to simulate conservation laws of mass, momentum, and energy in a moving material frame. The BLAST package utilizes high-order finite element discretizations of physical processes on a high-order (curved) moving mesh. BLAST's finite element discretization infrastructure is entirely based on the MFEM library. Therefore, the GPU port of BLAST makes extensive use of on the matrix-free approach and GPU support via MFEM. In this section we provide specifics about the major GPU kernels in BLAST, and the impact of the CEED project in these GPU development efforts.

Memory management. Since MARBL/BLAST is based on MFEM, it directly uses the high-level memory management interface for reading and writing device data. In addition, the MARBL team has enhanced the MFEM's memory manager capabilities by introducing the Umpire [48] memory manager providing access to memory pools. This approach enables the following benefits: substantially reduces slowdowns caused by `code cudaMalloc` performance; sharing of device memory buffers inside MARBL to reduce the total device usage; and sharing overall temporary memory between other external packages in MARBL that use Umpire.

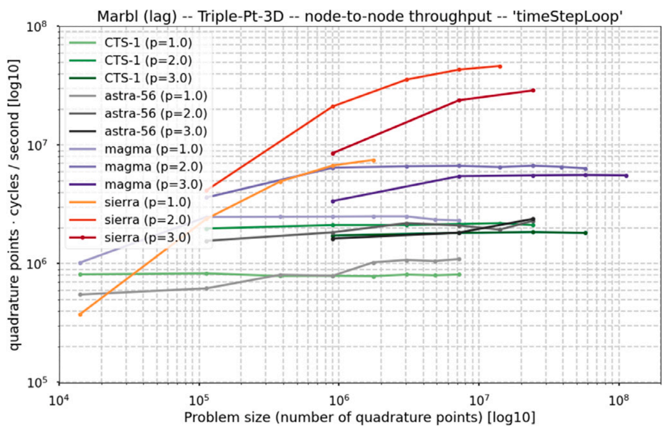

Fig. 11. 3D Triple-point problem throughput test in MARBL. Comparison of 3 CPU-based systems versus the NVIDIA V100 GPU-based Sierra.

Lagrangian phase. In this phase the multi-material compressible Euler equations are solved on a moving curved mesh [49,50]. The optimization of the needed mass and force operators has been aided by the matrix-free methods that were introduced by the CEED-developed Laghos miniapp [51], which models the main computational kernels of Lagrangian hydrodynamics. The GPU kernels for these methods were implemented by the MARBL team and reside in the BLAST code. The latest Laghos GPU implementations of these kernels give an alternative that might be used in the future, based on performance tests. A key CEED benefit provided to MARBL is the ability to drop in replacements for these expensive kernels as they become available. Physics-specific quadrature point computations were implemented by the MARBL team, making use of the RAJA nested parallel loop abstractions combined with MFEM's GPU capabilities, including GPU-friendly data structures, small dense matrix kernels, and use of shared memory. This phase also requires the computation of a *hyperviscosity* [52] coefficient, which involves consecutive applications of a Laplacian operator. This procedure has been ported on the GPU by applying directly the MFEM's optimized diffusion kernels.

Remesh phase. The mesh optimization phase of BLAST is based on the Target-Matrix Optimization Paradigm (TMOP), where the mesh optimization problem is posed as a variational minimization of a nonlinear functional [53,54]. The development of the GPU port was performed in MFEM's mesh optimization miniapp, and then directly ported to MARBL, as both codes use the same core TMOP algorithms.

Remap phase. The remap algorithm in BLAST has two main components, namely, velocity remap, which is solved by a continuous Galerkin advection discretization, and remap of other fields, which is modeled by flux-limited discontinuous Galerkin (DG) advection [55,56]. Using the MFEM infrastructure, the MARBL developers have developed custom GPU code for matrix-free DG advection remap. It is expected that this approach will be improved significantly by the future work in the CEED-developed Remhos miniapp, as it contains novel matrix-free DG remap methods [57]. The continuous Galerkin advection solve is also fully GPU ported. Similarly to the CG mass matrix inversion in the Lagrangian phase, the remap GPU code is implemented inside MARBL, and the alternative to switching to the optimized MFEM kernels will be explored. In Fig. 11 we present a recent study of MARBL that compares node-to-node throughput of several CPU machines at LLNL (a Commodity Technology System (*CTS*), *Astra* and *Magma*) versus the LLNL *Sierra* machine, showing clear advantage of the GPU executions. Table 3 shows timings of the three main phases of the application, along with the final speedup of the matrix-free CPU vs GPU kernels.

Table 3
CPU and GPU timings on 3 nodes of the LLNL's *rzgenie* (36 tasks per node) and *rzansel* (4 GPUs per node) machines. FA is traditional full assembly, while PA is matrix-free partial assembly. This is a full 3D high-order ALE simulation with 224,160 elements.

Phase	FA CPU	PA CPU	PA GPU	Speedup
Time loop	3854.16	2866.54	221.03	12.9
Lagrange	1773.68	1098.42	69.73	15.7
Remesh	557.98	366.24	42.67	8.5
Remap	1513.99	1393.34	100.95	13.8

4.5. ExaConstit

ExaConstit [58] is a general implicit quasi-static non-linear solid mechanics velocity-based finite element application built on the MFEM framework [12]. This code is being developed at LLNL for the ExaAM project in the ECP, with the goals of connecting local additive manufactured microstructures to local macroscopic properties by means of crystal plasticity finite element methods. As part of a larger workflow of the ExaAM workflow to simulate the additive manufacturing process, ExaConstit and its constitutive library, ExaCMech [59], needed be refactored to run on the GPU. This refactoring was required to run the hundreds to thousands of high-fidelity simulations on exascale hardware in a timely manner for the larger workflow. ExaCMech was ported over to the GPU using RAJA `code forall` loops wrapped around the entire large constitutive kernel. Within ExaConstit, the dominant computational cost lies within the linearized system solve within a Newton–Raphson scheme. Therefore, the primary focus had been transitioning from a traditional full assembly method over to partial and element assembly methods. This transition required the physics/constitutive calculations to be completely separated from the assembly method and called in a separate setup phase. The setup phase is now responsible for calculating the updated stress and material tangent stiffness matrix. Afterwords, these values can be incorporated into any of the runtime selected linear assembly methods. Initial partial and element assembly formulations are based on [60] and [61], respectively. In order to keep compute kernels backend agnostic, MFEM's `code forall` abstraction macros, which make use of the RAJA backends, and memory management capabilities were leveraged within ExaConstit for a vast majority of the compute kernels. Within ExaConstit, a few reduction operations were also converted to RAJA reduction policies to take advantage of the GPU as these are not available within the MFEM API. The end result of this refactoring is a ∼14.5x speedup when using the GPU element assembly over the CPU full assembly on Summit. For an ExaAM challenge problem sized linear hexahedron mesh of 6.7 million elements that undergoes 5% monotonic strain, the GPU port and assembly improvements result in a runtime decrease of roughly 35 node-hours down to 2.5 node-hours on 8 nodes of Summit for just the ExaConstit stage. The results of these simulations will then be used determine the properties being used in the part-scale simulation of the larger ExaAM workflow being run on exascale hardware. Finally from a physics point of view, these improvements are also enabling the ExaAM team to study the highly complex deformation processes that occur within additively manufactured microstructures, such as those shown in Fig. 12, at an unprecedented level of fidelity of typical crystal plasticity methods.

5. Conclusions

In this paper we described the development of GPU-oriented algorithms for high-order finite element discretizations in the ECP CEED projects. We presented the current GPU capabilities of several CEED components, including libCEED, MAGMA, MFEM, libParanumal and Nek, which can now run efficiently on both NVIDIA and AMD GPUs. We also discussed some of the challenges of porting to exascale GPU architectures and presented application results that use the CEED-developed GPU technologies.

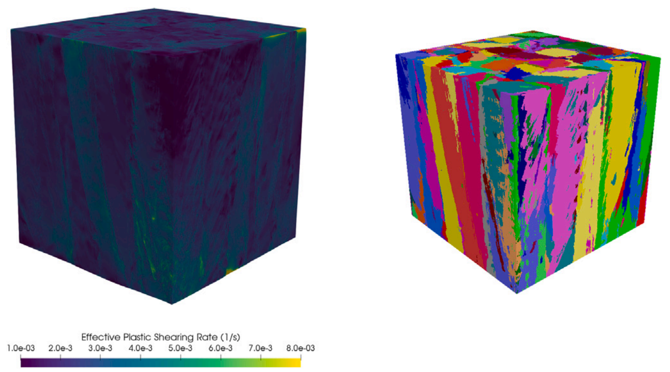

Fig. 12. A representative microstructure within an additive manufactured part over a 500 μm volume cube and discretized into 27 million linear hexahedron elements. The highly heterogeneous effective plastic shearing rate is plotted along with the microstructure where each crystal is represented as a different color. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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 nation's exascale computing imperative.

The research used resources of the Argonne Leadership Computing Facility, which is supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. This research also used resources of the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract DE-AC05-00OR22725. Work performed under the auspices of the U.S. Department of Energy under Contract DE-AC52-07NA27344 (LLNL-JRNL-816034).

References

[1] Center for Efficient Exascale Discretizations, Exascale Computing Project, DOE, ceed.exascaleproject.org.
[2] T. Kolev, P. Fischer, M. Min, J. Dongarra, J. Brown, V. Dobrev, T. Warburton, S. Tomov, M.S. Shephard, A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, N. Chalmers, Y. Dudouit, A. Karakus, I. Karlin, S. Kerkemeier, Y.-H. Lan, D. Medina, E. Merzari, A. Obabko, W. Pazner, T. Rathnayake, C.W. Smith, L. Spies, K. Swirydowicz, J. Thompson, A. Tomboulides, V. Tomov, Efficient exascale discretizations: High-order finite element methods, Int. J. HPC App. (2021) 1–26, http://dx.doi.org/10.1177/10943420211020803.
[3] H.O. Kreiss, J. Oliger, Comparison of accurate methods for the integration of hyperbolic problems, Tellus 24 (1972) 199–215, http://dx.doi.org/10.3402/tellusa.v24i3.10634.
[4] I. Babuška, M. Suri, The p and $h-p$ versions of the finite element method, basic principles and properties, SIAM Rev. 36 (4) (1994) 578–632, http://dx.doi.org/10.1137/1036141.
[5] S.A. Orszag, Spectral methods for problems in complex geometry, J. Comput. Phys. 37 (1980) 70–92, http://dx.doi.org/10.1016/0021-9991(80)90005-4.
[6] D. Gottlieb, S.A. Orszag, Numerical Analysis of Spectral Methods: Theory and Applications, SIAM-CBMS, Philadelphia, 1977.
[7] D. Arndt, N. Fehn, G. Kanschat, K. Kormann, M. Kronbichler, P. Munch, W.A. Wall, J. Witte, Exadg: High-order discontinuous Galerkin for the exa-scale, in: H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, W.E. Nagel (Eds.), Software for Exascale Computing - SPPEXA 2016-2019, Springer International Publishing, Cham, 2020, pp. 189–224.

[8] P.D. Bello-Maldonado, P.F. Fischer, Scalable low-order finite element preconditioners for high-order spectral element Poisson solvers, SIAM J. Sci. Comput. 41 (5) (2019) S2–S18, http://dx.doi.org/10.1137/18M1194997.

[9] C. Canuto, P. Gervasio, A. Quarteroni, Finite-element preconditioning of g-NI spectral methods, SIAM J. Sci. Comput. 31 (6) (2010) 4422–4451, http://dx.doi.org/10.1137/090746367.

[10] D. Moxey, R. Amici, M. Kirby, Efficient matrix-free high-order finite element evaluation for simplicial elements, SIAM J. Sci. Comput. 42 (3) (2020) C97–C123, http://dx.doi.org/10.1137/19M1246523.

[11] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D.A. Ham, P.H. Kelly, A study of vectorization for matrix-free finite element methods, Int. J. High Perform. Comput. Appl. 34 (6) (2020) 629–644, http://dx.doi.org/10.1177/1094342020945005.

[12] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J.C.V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, S. Zampini, MFEM: A modular finite element library, Comput. Math. Appl. (2020) http://dx.doi.org/10.1016/j.camwa.2020.06.009.

[13] M. Kronbichler, W.A. Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, SIAM J. Sci. Comput. 40 (5) (2018) A3423–A3448, http://dx.doi.org/10.1137/16M110455X.

[14] M. Kronbichler, K. Ljungkvist, Multigrid for matrix-free high-order finite element computations on graphics processors, ACM Trans. Parallel Comput. 6 (1) (2019) 1–32, http://dx.doi.org/10.1145/3322813.

[15] J.W. Lottes, P.F. Fischer, Hybrid multigrid/Schwarz algorithms for the spectral element method, J. Sci. Comput. 24 (2005) 45–78.

[16] P. Fischer, M. Min, T. Rathnayake, S. Dutta, T. Kolev, V. Dobrev, J.-S. Camier, M. Kronbichler, T. Warburton, K. Swirydowicz, J. Brown, Scalability of high-performance PDE solvers, Int. J. HPC App. 34 (5) (2020) 562–586, http://dx.doi.org/10.1177/1094342020915762.

[17] J. Brown, A. Abdelfattah, V. Barra, N. Beams, J.S. Camier, V. Dobrev, Y. Dudouit, L. Ghaffari, T. Kolev, D. Medina, W. Pazner, T. Ratnayaka, J. Thompson, S. Tomov, libCEED: Fast algebra for high-order element-based discretizations, J. Open Source Softw. 6 (63) (2021) 2945, http://dx.doi.org/10.21105/joss.02945.

[18] A. Abdelfattah, V. Barra, N. Beams, J. Brown, J.-S. Camier, V. Dobrev, Y. Dudouit, L. Ghaffari, T. Kolev, D. Medina, W. Pazner, T. Ratnayaka, J.L. Thompson, S. Tomov, libCEED User manual, zenodo, 2021, http://dx.doi.org/10.5281/zenodo.5077489.

[19] G. Karniadakis, S. Sherwin, Spectral/Hp Element Methods for Computational Fluid Dynamics, Oxford University Press, Oxford, 2005.

[20] P.E. Vos, S.J. Sherwin, R.M. Kirby, From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations, J. Comput. Phys. 229 (13) (2010) 5161–5181, http://dx.doi.org/10.1016/j.jcp.2010.03.031.

[21] M. Ainsworth, G. Andriamaro, O. Davydov, Bernstein-Bézier Finite elements of arbitrary order and optimal assembly procedures, SIAM J. Sci. Comput. 33 (6) (2011) 3087–3109, http://dx.doi.org/10.1137/11082539X.

[22] R.C. Kirby, Fast simplicial finite element algorithms using Bernstein polynomials, Numer. Math. 117 (4) (2011) 631–652, http://dx.doi.org/10.1007/s00211-010-0327-2.

[23] K. Swirydowicz, N. Chalmers, A. Karakus, T. Warburton, Acceleration of tensor-product operations for high-order finite element methods, Int. J. High Perform. Comput. Appl. 33 (4) (2019) 735–757, http://dx.doi.org/10.1177/1094342018816368.

[24] D.S. Medina, A. St-Cyr, T. Warburton, OCCA: A unified approach to multi-threading languages, 2014, arXiv preprint arXiv:1403.0968.

[25] MAGMA: Matrix Algebra on GPU and Multicore Architectures, icl.utk.edu/magma.

[26] A. Abdelfattah, M. Baboulin, V. Dobrev, J.J. Dongarra, C.W. Earl, J. Falcou, A. Haidar, I. Karlin, T.V. Kolev, I. Masliah, S. Tomov, High-performance tensor contractions for GPUs, in: International Conference on Computational Science, ICCS, 6-8 June 2016, San Diego, California, USA, 2016, pp. 108–118, http://dx.doi.org/10.1016/j.procs.2016.05.302.

[27] N. Beams, A. Abdelfattah, S. Tomov, J. Dongarra, T. Kolev, Y. Dudouit, High-Order Finite Element Method using Standard and Device-Level Batch GEMM on GPUs, in: 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, Proceedings. To Appear, 2020.

[28] R.D. Hornung, J.A. Keasler, The RAJA Portability Layer: Overview and Status, LLNL-TR-661403, LLNL, 2014.

[29] N. Chalmers, A. Karakus, A.P. Austin, K. Swirydowicz, T. Warburton, libParanumal: a performance portable high-order finite element library, 2020, http://dx.doi.org/10.5281/zenodo.4004744. URL github.com/paranumal/libparanumal.

[30] D. Medina, OKL: a unified language for parallel architectures, Ph.D. thesis, Rice University, 2015.

[31] N. Chalmers, T. Warburton, Portable high-order finite element kernels I: Streaming operations, 2020, arXiv preprint arXiv:2009.10917.

[32] N. Chalmers, T. Warburton, streamParanumal: Streaming Microbenchmarks for High-order Finite Element Methods, URL github.com/paranumal/streamparanumal.

[33] Gslib: Gather-scatter library, 2020, URL github.com/Nek5000/gslib.

[34] P. Fischer, S. Kerkemeier, M. Min, Y. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, T. Warburton, NekRS, a GPU-accelerated spectral element Navier-Stokes Solver, CoRR, 2021, arXiv:2104.05829.

[35] A. Melander, E. Strøm, F. Pind, A. Engsig-Karup, C.-H. Jeong, T. Warburton, N. Chalmers, J.S. Hesthaven, Massive parallel nodal discontinuous Galerkin finite element method simulator for room acoustics, Tech. rep, 2020.

[36] Nek: Open source, highly scalable and portable spectral element code, 2020, URL nek5000.mcs.anl.gov.

[37] NekCEM: Scalable high-order computational electromagnetic code, 2020, URL github.com/NekCEM/NekCEM.

[38] P.F. Fischer, K. Heisey, M. Min, Scaling limits for PDE-based simulation, in: 22nd AIAA Computational Fluid Dynamics Conference, 2015, p. 3049.

[39] M.O. Deville, P.F. Fischer, E.H. Mund, High-Order Methods for Incompressible Fluid Flow, Cambridge University Press, Cambridge, 2002.

[40] M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, M. Min, An MPI/OpenACC implementation of a high order electromagnetics solver with GPUDirect communication, Int. J. High Perform. Comput. Appl. 30 (3) (2016) 320–334, http://dx.doi.org/10.1177/1094342015626584.

[41] J. Gong, S. Markidis, E. Laure, M. Otten, P. Fischer, M. Min, Nekbone performance on GPUs with OpenACC and CUDA fortran implementations, special issue on sustainability on ultrascale computing systems and applications, J. Supercomput. 72 (11) (2016) 4160–4180, http://dx.doi.org/10.1007/s11227-016-1744-5.

[42] E. Otero, J. Gong, M. Min, P. Fischer, P. Schlatter, E. Laure, OpenACC Acceleration for the $P_N - P_{N-2}$ algorithm in Nek5000, J. Parallel Distrib. Comput. 132 (2019) 69–78, http://dx.doi.org/10.1016/j.jpdc.2019.05.010.

[43] P.F. Fischer, Projection techniques for iterative solution of $A\underline{x} = \underline{b}$ with successive right-hand sides, Comput. Methods Appl. Mech. Engrg. 163 (1998) 193–204, http://dx.doi.org/10.1016/S0045-7825(98)00012-7.

[44] A.P. Austin, N. Chalmers, T. Warburton, Initial guesses for sequences of linear systems in a GPU-accelerated incompressible flow solver, 2020, arXiv preprint arXiv:2009.10863.

[45] OCCA: Lightweight performance portability library, 2020, URL libocca.org.

[46] Y.-H. Lan, P. Fischer, E. Merzari, M. Min, All-hex meshing strategies for densely packed spheres, in: The 29th International Meshing Roundtable, 2021.

[47] R.W. Anderson, V.A. Dobrev, T.V. Kolev, R.N. Rieben, V.Z. Tomov, High-order multi-material ALE hydrodynamics, SIAM J. Sci. Comput. 40 (1) (2018) B32–B58, http://dx.doi.org/10.1137/17M1116453.

[48] D. Beckingsale, M. McFadden, J. Dahm, R. Pankajakshan, R. Hornung, Umpire: Application-focused management and coordination of complex hierarchical memory, IBM J. Res. Dev. (2019) 1–10, http://dx.doi.org/10.1147/JRD.2019.2954403.

[49] V.A. Dobrev, T.V. Kolev, R.N. Rieben, High-order curvilinear finite element methods for Lagrangian hydrodynamics, SIAM J. Sci. Comput. 34 (5) (2012) B606–B641, http://dx.doi.org/10.1137/120864672.

[50] V.A. Dobrev, T.V. Kolev, R.N. Rieben, V.Z. Tomov, Multi-material closure model for high-order finite element Lagrangian hydrodynamics, Internat. J. Numer. Methods Engrg. 82 (10) (2016) 689–706, http://dx.doi.org/10.1002/fld.4236.

[51] Laghos: High-order Lagrangian hydrodynamics miniapp, 2020, URL github.com/ceed/Laghos.

[52] P.D. Bello-Maldonado, T.V. Kolev, R.N. Rieben, V.Z. Tomov, A matrix-free hyperviscosity formulation for high-order ALE hydrodynamics, Comput. Fluids (2020) http://dx.doi.org/10.1016/j.compfluid.2020.104577.

[53] V. Dobrev, P. Knupp, T. Kolev, K. Mittal, V. Tomov, The target-matrix optimization paradigm for high-order meshes, SIAM J. Sci. Comput. 41 (1) (2019) B50–B68, http://dx.doi.org/10.1137/18M1167206.

[54] V.A. Dobrev, P. Knupp, T.V. Kolev, K. Mittal, R.N. Rieben, V.Z. Tomov, Simulation-driven optimization of high-order meshes in ALE hydrodynamics, Comput. Fluids 208 (2020) http://dx.doi.org/10.1016/j.compfluid.2020.104602.

[55] R.W. Anderson, V.A. Dobrev, T.V. Kolev, R.N. Rieben, Monotonicity in high-order curvilinear finite element arbitrary Lagrangian–Eulerian remap, Internat. J. Numer. Methods Engrg. 77 (5) (2015) 249–273, http://dx.doi.org/10.1002/fld.3965.

[56] R.W. Anderson, V.A. Dobrev, T.V. Kolev, D. Kuzmin, M.Q. de Luna, R.N. Rieben, V.Z. Tomov, High-order local maximum principle preserving (MPP) discontinuous Galerkin finite element method for the transport equation, J. Comput. Phys. 334 (2017) 102–124, http://dx.doi.org/10.1016/j.jcp.2016.12.031.

[57] H. Hajduk, D. Kuzmin, T.V. Kolev, R. Abgrall, Matrix-free subcell residual distribution for Bernstein finite element discretizations of linear advection equations, Comput. Methods Appl. Mech. Engrg. 359 (2020) http://dx.doi.org/10.1016/j.cma.2019.112658.

[58] R.A. Carson, S.R. Wopschall, J.A. Bramwell, ExaConstit, 2019, http://dx.doi.org/10.11578/dc.20191024.2, URL github.com/LLNL/ExaConstit.

[59] N.R. Barton, R.A. Carson, S.R. Wopschall, U.N.N.S. Administration, Ecmech, 2018, http://dx.doi.org/10.11578/dc.20190809.2, URL github.com/LLNL/ExaCMech.

[60] A.K. Gupta, B. Mohraz, A method of computing numerically integrated stiffness matrices, Internat. J. Numer. Methods Engrg. 5 (1) (1972) 83–89, http://dx.doi.org/10.1002/nme.1620050108.

[61] A.K. Gupta, Efficient numerical integration of element stiffness matrices, Internat. J. Numer. Methods Engrg. 19 (9) (1983) 1410–1413, http://dx.doi.org/10.1002/nme.1620190910.