ECP Milestone Report

Port and optimize the CEED software stack to Aurora / Frontier EA Systems

WBS 2.2.6.06, Milestone CEED-MS37

Tzanio Kolev
Paul Fischer
Natalie Beams
Jed Brown
Jean-Sylvain Camier
Noel Chalmers
Veselin Dobrev
Yohann Dudouit
Stefan Kerkemeier
Yu-Hsiang Lan
Yimin Lin
Neil Lindquist
Damon McDougall
David Medina
Elia Merzari
Misun Min
Scott Moe
Will Pazner
Malachi Phillips
Thilina Ratnayaka
Kris Rowe
Mark S. Shephard
Cameron W. Smith
Stanimire Tomov
Tim Warburton

September 30, 2021

ECP Milestone Report
Port and optimize the CEED software stack to Aurora / Frontier EA Systems
WBS 2.2.6.06, Milestone CEED-MS37

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

September 30, 2021

ECP Milestone Report
Port and optimize the CEED software stack to Aurora / Frontier EA Systems
WBS 2.2.6.06, Milestone CEED-MS37

Approvals

Submitted by:

Tzanio Kolev, LLNL CEED PI	Date

Approval:

Andrew R. Siegel, Argonne National Laboratory Director, Applications Development Exascale Computing Project	Date

Revision Log

Version	Creation Date	Description	Approval Date
1.0	September 30, 2021	Original	

EXECUTIVE SUMMARY

The goal of this milestone was to port the CEED software stack, including Nek, MFEM and libCEED to the Frontier and Aurora early access hardware, work on optimizing the performance on AMD and Intel GPUs, and demonstrate impact in CEED-enabled ECP applications.

As part of this milestone, we also performed a number of other activities, including continued optimizations for CEED applications at scale on Summit, performance evaluation on non-ECP hardware, including the Fugaku's A64FX chip and the NVIDIA A100 GPU architecture, exploring the use of just-in-time (JIT) compilation in applications at scale, the potential of mixed precision optimizations in the CEED discretization and solver algorithms, and more. During the milestone period, the CEED team released new versions of six of its packages, including major releases of MFEM, NekRS and OCCA. We also organized the fifth CEED Annual meeting (CEED5AM) which included nearly 100 researchers from national labs, universities and industry.

The specific tasks addressed in this milestone were as follows.

- CEED-T13 (ADCD04-81): Multi-node scaling on Summit/Sierra. Demonstrate the use of JIT in applications at scale.
- CEED-T14 (ADCD04-82): Explore the potential of mixed precision optimizations in the CEED discretization and solver algorithms.
- CEED-T15 (ADCD04-83): Port Nek, MFEM and libCEED to Frontier/Aurora EA and demonstrate impact in applications.
- CEED-T16 (ADCD04-84): CEED Annual meeting (CEED5AM).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1. INTRODUCTION

The goal of this milestone was to port the CEED software stack, including Nek, MFEM and libCEED to the Frontier and Aurora early access hardware, work on optimizing the performance on AMD and Intel GPUs, and demonstrate impact in CEED-enabled ECP applications.

As part of this milestone, we also performed a number of other activities, including continued optimizations for CEED applications at scale on Summit, performance evaluation on non-ECP hardware, including the Fugaku's A64FX chip and the NVIDIA A100 GPU architecture, exploring the use of just-in-time compilations in applications at scale and potential of mixed precision optimizations in the CEED discretization and solver algorithms, and more. During the milestone period, the CEED team released new versions of 6 of its packages, including major releases of MFEM, NekRS and OCCA. We also organized the fifth CEED Annual meeting (CEED5AM) which included nearly 100 researchers from national labs, universities and industry.

The specific tasks addressed in this milestone were as follows. CEED-T13 (ADCD04-81): Multi-node scaling on Summit/Sierra. Demonstrate the use of JIT in applications at scale; CEED-T14 (ADCD04-82): Explore the potential of mixed precision optimizations in the CEED discretization and solver algorithms; CEED-T15 (ADCD04-83): Port Nek, MFEM and libCEED to Frontier/Aurora EA and demonstrate impact in applications; and CEED-T16 (ADCD04-84): CEED Annual meeting (CEED5AM).

2. CEED APPLICATIONS AND PERFORMANCE ON SUMMIT

In this section we review several recent activities in improving the performance of CEED components on NVIDIA V100 GPUs and multi-node application scaling on Summit.

2.1 Non-Deterministic (*fast*) Kernels for V100 GPUs in MFEM

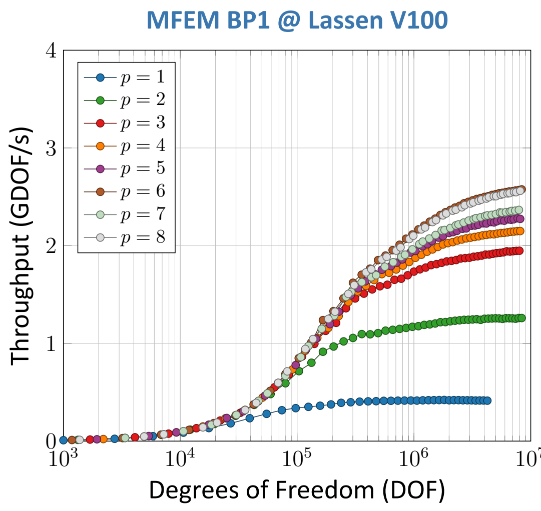
The MFEM team has been working on adding new non-deterministic (also called *fast*) GPU kernels to the library. A main distinctive feature of these kernels is to avoid the cost of two kernel launches: the restriction's `Mult` and `MultTranspose` operations, which were used to scatter and gather the data to the benchmark core kernels. These prefix and postfix kernels were also useful in order to avoid *atomic* instructions and to ensure *determinism*, which is sometimes expected by MFEM applications. Such kernels are available by simply adding on the command line the *fast* option keyword when specifying the device used at runtime. Figures 1 and 2 demonstrate the speedup of using these new *fast* kernels on V100 GPUs.

2.2 Breakthrough: MFEM Generated Kernels with Reduced $N_{0.8}$ on NVIDIA GPUs

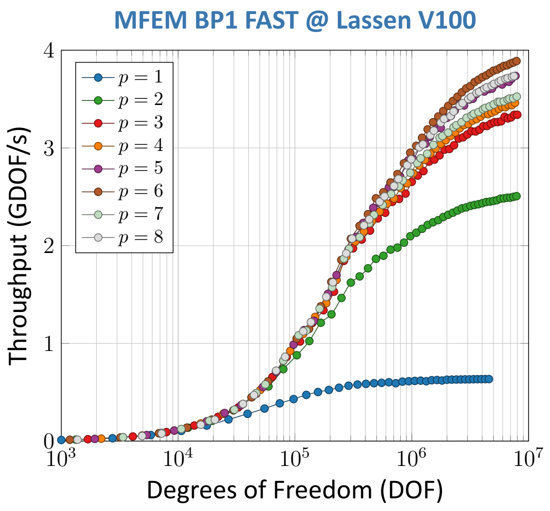
The MFEM team has been performing research in generating optimized kernels from general mathematical description, matching the performance of hand-tuned ones. This approach is capable to offer a sustainable way to generate finite element kernels, as well as adapting the knowledge capitalized during the CEED project. The starting point is the Unified Form Language (UFL) which has been formally described in 2012 and is well used in the finite element Python community through multiple projects (FEniCS, Firedrake). The motivation is to offer the best integrated finite element kernels for all finite-element-based applications, by compiling idiomatic mathematical variational formulations into low-level C++ kernels, reducing execution time by specializing to known tensor shapes.

The underlying idea is to augment traditional compiler representations with a data-aware graph-based structure: it allows the MFEM team to represent kernels as graph nodes with their data dependencies exposed. New performance optimizations can then be realized through graph transformations that preserve the behavioral equivalence of the calculations being performed, ensuring application correctness. Enabling this graph-based representation provides significant potential for optimization both in terms of kernel launches and memory bandwidth usage. For example, in many applications kernels can be fused together and memory can be reused which reduce both kernel launches and memory bandwidth load.

This way of generating optimized high-order finite element kernels is called *XFL*, for *Acc*elerated *F*orm *L*anguage. An example of what the input code for the CEED BP3 benchmark looks like in the UFL specification is presented in the listing in Figure 3.

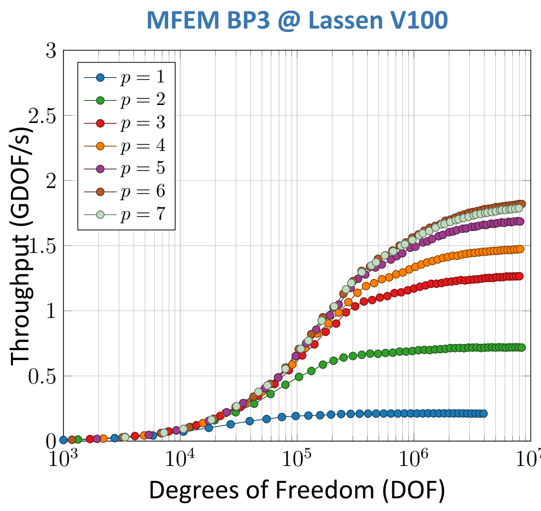


(a) Deterministic kernels

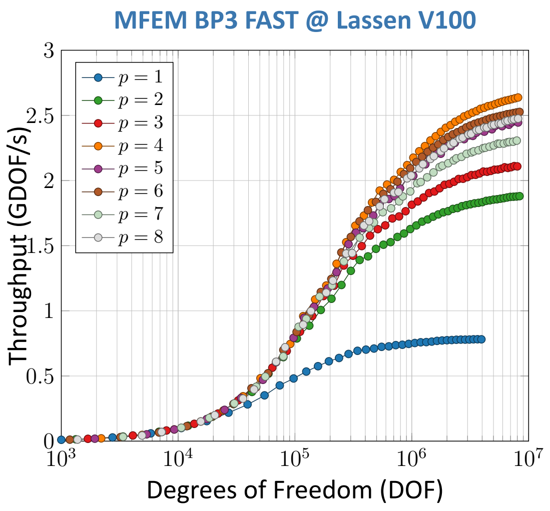


(b) *fast* non-deterministic kernels

Figure 1: MFEM results for the CEED BP1 benchmark on a single NVIDIA Volta V100 SXM2 GPU on Lassen using the deterministic (*a*) and *fast* non-deterministic (*b*) kernels.



(a) Deterministic kernels



(b) *fast* non-deterministic kernels

Figure 2: MFEM results for the CEED BP3 benchmark on a single NVIDIA Volta V100 SXM2 GPU on Lassen using the deterministic (*a*) and *fast* non-deterministic (*b*) kernels.

For CEED applications, the rate of work is measured in billions-DOFs-per-second (GDOF/s, or *gigadofs*). Two of the principal metrics of interest are the peak rate of work per unit resource ($r_{\max}$) and the local problem size on the node required to realize 80% percent of the peak rate of work per unit resource ($N_{0.8}$). As explained in [4], users are typically interested in reduced time-to-solution by increasing the number of nodes until parallel efficiency reaches an intolerable level (about 80%). Improving the $N_{0.8}$ directly allows to reduce the time-to-solution: the smaller the value of $N_{0.8}$, the more processors that can be used and the

```
# CEED 3D Bake-off BP3
N = 32
order = 6

# Define a finite element space on the mesh
fe = FiniteElement("Lagrange", hexahedron, order)
fes = FunctionSpace(UnitHexMesh(N,N,N), fe)

# Define the boundary conditions
bc = DirichletBC(fes)

# Define the solution vector x as a finite element grid function
x = Function(fes)
x = 0.0

# Define the trial and test functions that will be used in the forms
u = TrialFunction(fes)
v = TestFunction(fes)

# Set up the linear form b(.)
b = v*dx

# Set up the parallel bilinear form a(.,.)
a = inner(grad(u), grad(v))*dx

# Benchmark and solve the linear system a x = b
benchmark(a == b, x, bc)
```

Figure 3: CEED BP3 benchmark written in UFL and used to generate the *XFL* MFEM kernels from Figures 4 and 5.

faster the calculation will run.

Therefore the results with the new MFEM XFL kernels presented in Figures 4 and 5, are of particular interest due to the improved $r_{\max}$ and $N_{0.8}$ values for both the BP1 and BP3 CEED benchmarks. While the $r_{\max}$ for BP1 at lower orders present an interesting speedup, the $N_{0.8}$ are more noticeably reduced by a factor of about 10 for BP1.

2.3 ExaWind Atmospheric Boundary Layer Flow Modeling, Convergence and Performance

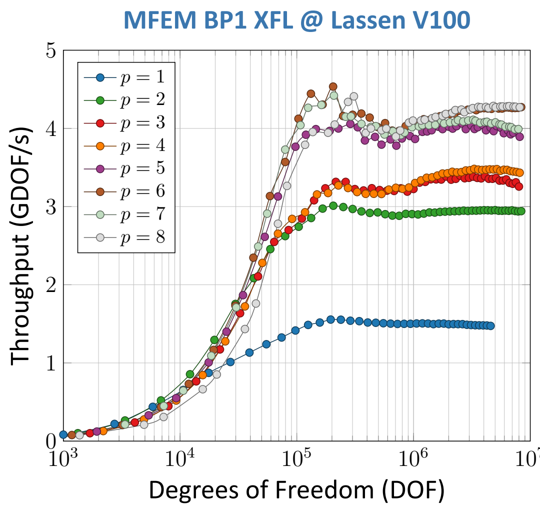
In collaboration with the ECP ExaWind team at NREL, we have studied atmospheric boundary layer (ABL) flows relevant to wind-farm modeling on DOE's pre-exascale platforms and NREL's Eagle computer. Through this collaboration we are conducting cross-verification and validation of Nek5000/RS and AMRWind. Our focus is the GABLS1 benchmark problem defined on a fixed computational domain [400m $\times$ 400m $\times$ 400m].

The objective of the code comparison is to reduce scaling bottlenecks in each code and to identify turbulence modeling effects that impact important statistics such as the boundary-layer height and turbulence intensity. We have made extensive performance comparisons between Nek5000/RS and AMRWind and continue to optimize their performance. A series of strong- and weak-scaling studies on advanced computing platforms monitors average time-per-step and the ratio of simulation time to real time. We also contrast the results as a function of resolution, for a variety LES modeling approaches, wall models, and boundary conditions. Low-order statistics, spectral analysis, and turbulent structure analysis give insight to the importance of these parameters. The results will be submitted as a journal paper in next quarter.

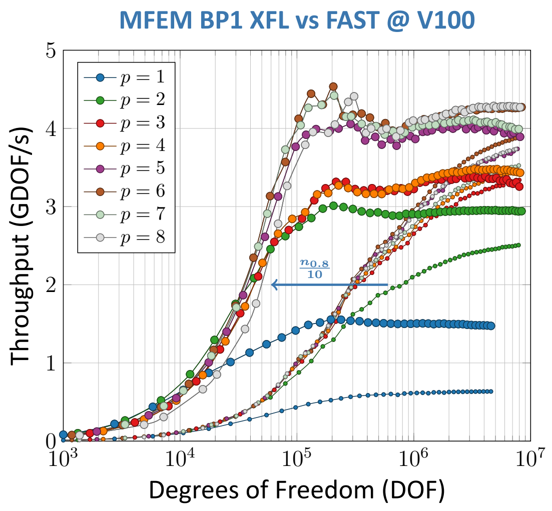
Figures 6 and 7 demonstrate convergence results of Nek5000/RS at varying grid resolutions in the range of 1.56m–0.19m for a fixed computational domain [400m $\times$ 400m $\times$ 400m]. We include in the convergence studies comparisons to existing data from NCAR, IMUK, and MO, where the finest grid resolution is 0.39m.

2.4 Rapid Turn-Around Full-Core Reactor Simulations on Summit

In this section we summarize several developments related to NEAMS reactor simulations on Summit, including performance optimization for full-core pebble-bed models, GPU multi-mesh simulations and new

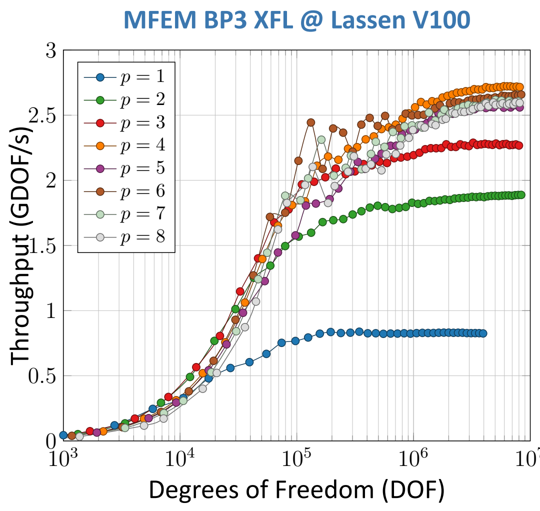


(a) Reduced $N_{0.8}$ with *XFL* kernels

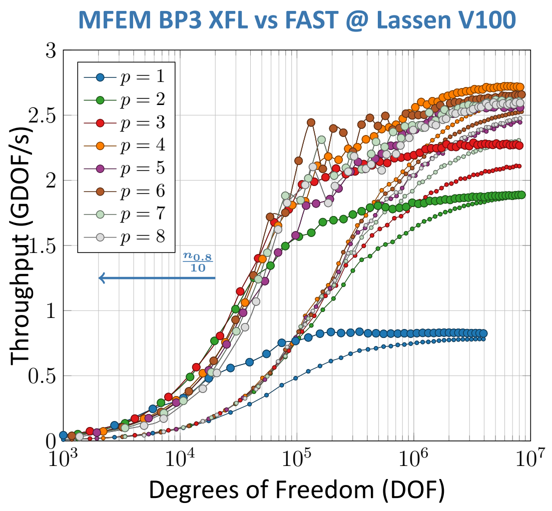


(b) *XFL* vs *fast* kernels

Figure 4: Reduced $N_{0.8}$ obtained with the *XFL* (a) and *XFL* vs *fast* (b) kernels on CEED BP1 benchmark on a single NVIDIA V100 SXM2 GPU on Lassen.



(a) Reduced $N_{0.8}$ with *XFL* kernels

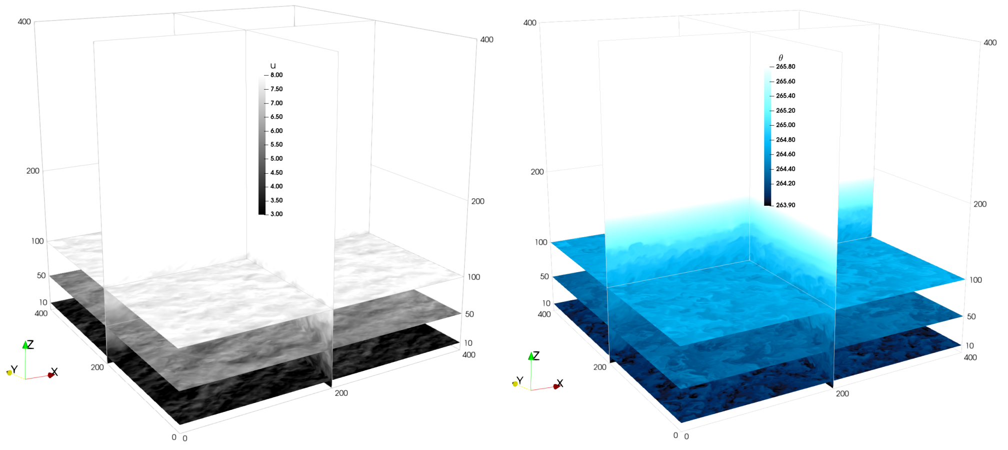


(b) *XFL* vs *fast* kernels

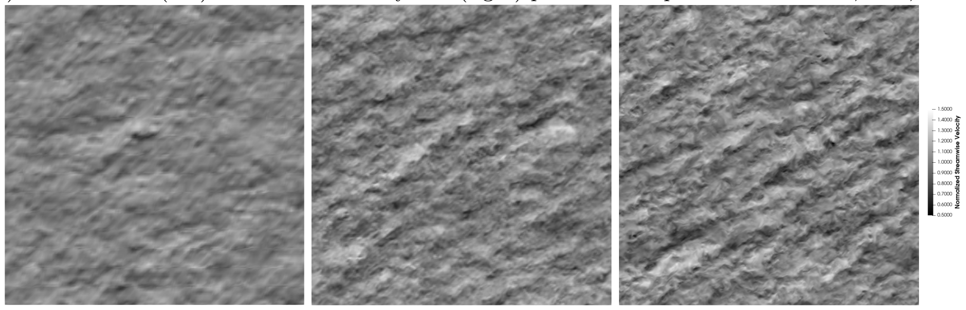
Figure 5: Reduced $N_{0.8}$ obtained with the *XFL* (a) and *XFL* vs *fast* (b) kernels on CEED BP3 benchmark on a single NVIDIA V100 SXM2 GPU on Lassen.

solver Development for molten salt reactors.

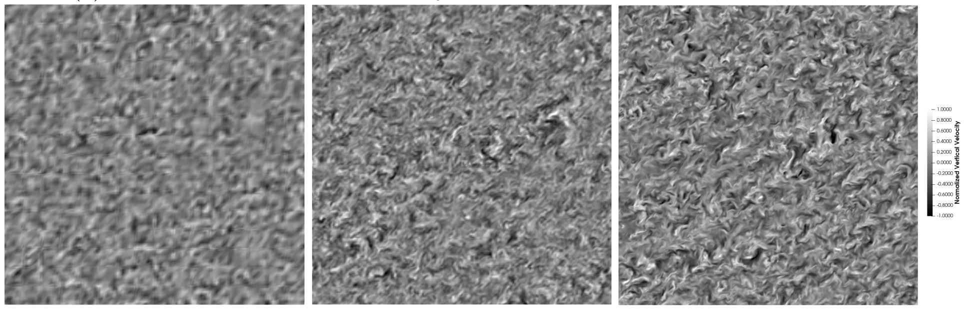
NEAMS Full-Core Pebble-Bed Performance Optimization. The main target of our NEAMS study is the full core for the pebble bed reactor (Figure 8, left), which has 352,625 spherical pebbles and a fluid mesh comprising $E = 98,782,067$ elements of order $N = 8$ ($n \approx 51$B). In this case, we consider the characteristics-based timestepping with $\Delta t = 4.$e-4 or 8.e-4, corresponding to respective Courant numbers of $CFL \approx 2$ and 4. Table 1 lists the battery of tests considered for this problem, starting with the single-sweep Chebyshev-Additive Schwarz (1-Cheb-ASM) pMG smoother, which is the default choice for smaller (easier)



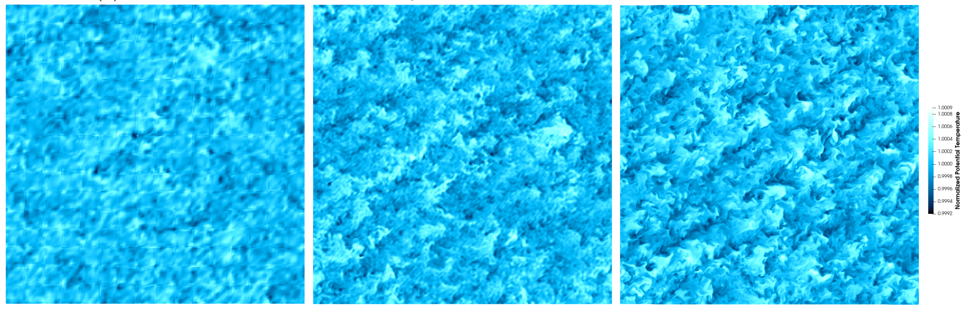
(a) NekRS ABL: (left) streamwise velocity and (right) potential temperature at z=10m, 50m, 100m.



(b) NekRS ABL: streamewise velocity u, z=10m, at resolution n=128^3, 256^3, 512^3



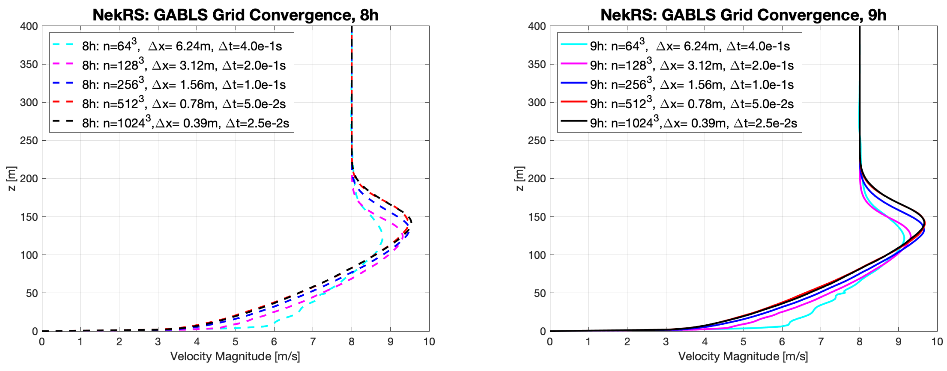
(c) NekRS ABL: vertical velocity w, z=10m, at resolution n=128^3, 256^3, 512^3



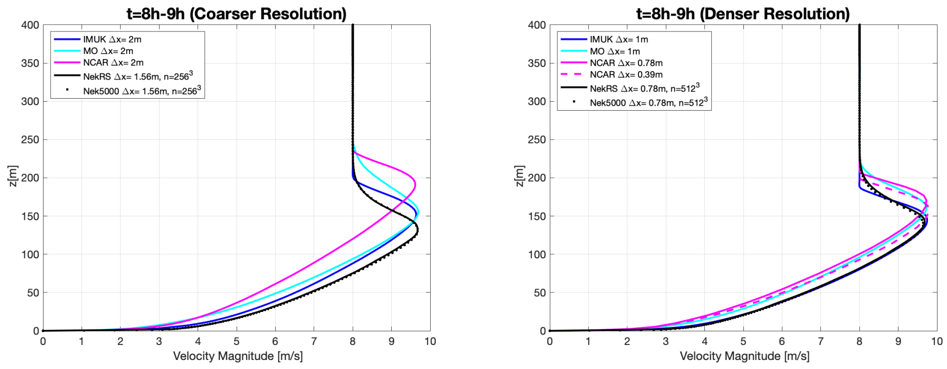
(d) NekRS ABL: potential temperature θ, z=10m, at resolution n=128^3, 256^3, 512^3

Figure 6: ExaWind ABL flows: NekRS mesh convergence using varying spatial resolutions of 3.12m (n=128^3), 1.56m (n=256^3), and 0.78m ($n = 512^3$) on the domain [400m × 400m × 400m].

problems. Unfortunately, this choice and the two-smoothings Chebyshev-Jacobi (2-Cheb-Jac) option yield very high coarse-grid solve costs because of the relative frequency in which the full V-cycle must be executed. (For



(a) NekRS grid convergence for mean velocity magnitude at t=8h and 9h.



(b) NekRS time-space mean velocity magnitude for t=7h–8h and 8h–9h.

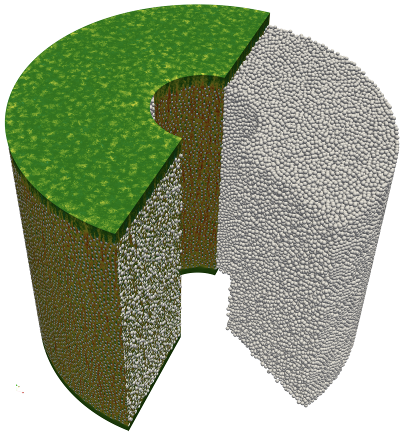
Figure 7: ExaWind ABL flows: NekRS grid convergence and validation and verification in comparison to other codes (IMUK, MO, NCAR) for 8h–9h on coarser and denser resolutions.

this problem, the coarse-grid system has ≈ 98M/27K ≈ 3600 degrees-of-freedom per V100.) Analysis of the standard NekRS output suggested that more smoothings at the finer levels would alleviate the communication burden incurred by the coarse-grid solves for problems at this scale.

The first step in optimization was thus to increase the number of smoothings (2-Cheb-ASM) and to increase the number of pMG levels to four, with polynomial orders N=8, 6, 4, 1 (where 1 is the coarse grid). These steps yielded a 1.6× speed-up over the starting point. Subsequently, we boosted the number of prior solutions to use as an approximation space for the pressure initial guess from $L = 8$ to 30, which yielded an additional factor of 1.7, as indicated in Figure 8.

Given the success of projection and additional smoothing, which lowered the FlexCG iteration counts to <6, it seemed clear that GMRES would be viable. A downside of GMRES is that the memory footprint scales as K, the maximum number of iterations and the work (and potentially, communication) scales as K^2. With K bounded by 6, these complexities are not onerous and one need not worry about losing the projective property of GMRES by having to use a restarted variant. Moreover, with so few vectors, the potential of losing orthogonality of the Arnoldi vectors is diminished, which means that classical Gram-Schmidt can be used and one thus safely use a *communication minimal* single all-reduce on a vector of length $< K$ to complete the Arnoldi orthogonalization step.

The next optimizations focused on the advection term. First, we reduced the number of quadrature points from $N_q = 13$ to 11 (in each direction). Elevated quadrature is necessary for *stability*, but not for accuracy. While one can prove stability for $N_q \geq 3N/2$ [15], it is not mandatory and, when using the characteristics method, which visits the advection operator at least four times per timestep, it can pay to reduce N_q as long as the flow remains stable. Second, we *increased* Δt by a factor of 2, which requires 2 subcycles to advance the hyperbolic advection operator (i.e., doubling its cost), but does not double the number of velocity and



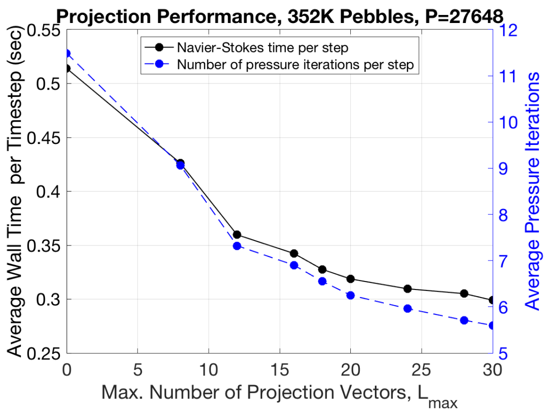

Projection Performance, 352K Pebbles, P=27648
Navier-Stokes time per step
Number of pressure iterations per step
Average Wall Time per Timestep (sec)
Average Pressure Iterations
Max. Number of Projection Vectors, L_max
0.55
0.5
0.45
0.4
0.35
0.3
0.25
12
11
10
9
8
7
6
5
0
5
10
15
20
25
30

Figure 8: Full-core pebble-bed reactor results: (left) velocity distribution in an all-hex mesh comprising 98M elements of order $N = 8$ for 352K pebbles, run on all of Summit (27648 NVIDIA V100s); (right) average time per step and pressure iteration count as a function of projection-space dimension, $L_{\max}$.

pressure iterations. Case (f) in Table 1 shows that the effective cost (based on the original Δt) is $t_{\text{step}} = 0.188$ s. In case (g) we arrive almost at $t_{\text{step}} = 0.18$s by turning off all I/O to stdout for all timesteps modulo 1000. The net gain is **a factor of 3.8** over the starting (default) point. Most importantly, with these optimizations, it is now possible to simulate a flow-through time of **a full reactor core in six hours** of wall-clock time on Summit.

Table 2 shows three strong scaling studies for the pebble bed at different levels of optimization, with the final one corresponding to Case (f) of Table 1. The limited memory on the GPUs means that we can only scale from $P = 9216$ to 27648 for these cases and in fact cannot support $L = 30$ at $P = 9216$, which is why that value is absent from the table. As noted earlier, a fair comparison for the last set of entries would be to run the $P = 9216$ case with a smaller value of L—it would perform worse, which would give a scaling advantage to the $L = 30$ case. This advantage is legitimate, because $L = 30$ is an improved algorithm over (say) $L = 8$, which leverages the increase memory resources that come with increasing P.

Major Algorithmic Variations, 352K Pebbles, P=27648								
Case	Solver	Smoother	L	N_q	Δt	v_i	p_i	t_{step}
(a)	FlexCG	1-Cheb-ASM:851	8	13	4e-4	3.6	22.8	.68
(b)	FlexCG	2-Cheb-Jac:851	8	13	4e-4	3.6	17.5	.557
(c)	”	2-Cheb-ASM:851	8	13	4e-4	3.6	12.8	.468
(d.8)	”	2-Cheb-ASM:8641	8	13	4e-4	3.6	9.1	.426
(d.L)	”	2-Cheb-ASM:8641	0–30	13	4e-4	3.6	5.6	.299
(e)	GMRES	”	30	13	4e-4	3.5	4.6	.240
(f)	”	”	30	11	8e-4	5.7	7.2	.376
(g)	”	”	30	11	8e-4	5.7	7.2	.361 (no I/O)

Table 1: Progression of algorithmic trials. See Figure 8 for Cases (d.L), L=0:30.

Rapid Turn-Around. Remarkably, the performance results presented here imply that it is now possible to simulate a single flow-through time for a full core in less than six hours when using all of Summit for these simulations. The analysis proceeds as follows. In the nondimensional units of this problem, the domain height is $L_z = 130$ units, while the mean flow speed in the pebble region is $U \approx 1/\phi$, where $\phi \approx 0.36$ is the void fraction in the packed bed. The nondimensional flow-through time is thus $L_z/U \approx 130 \times 0.36 = 46.6$ (modulo a reduction of the flow speed in the upper and lower plena). The timestep size from the final two rows of Table 5 is 8×10^{-4}, corresponding to 58250 steps for a single flow-through time, which, at 0.361 seconds per step, corresponds to 5.84 hours of wall-clock time per flow-through time.

NEAMS NekNek Multimesh on GPUs towards Kairos Power.

Nek5000/RS has support for overset grids that can greatly simplify meshing requirements, particularly

NekRS Strong Scaling: 352K pebbles, E=98M, n=50B						
$N=8$, $N_q=13$, $\Delta t=$ **4.e-4**, $L=8$, **1-Cheb-Jac:851**						
Node	GPU	n/P	v_i	p_i	t_{step}	Eff
1536	9216	5.4M	3.6	17.3	.97	1.00
2304	13824	3.6M	3.6	18.0	.84	76.9
3072	18432	2.7M	3.6	16.6	.75	64.6
3840	23040	2.1M	3.6	19.6	.67	57.9
4608	27648	1.8M	3.6	17.5	.55	58.7
$N=8$, $N_q=13$, $\Delta t=$ **4.e-4**, $L=8$, **1-Cheb-ASM:851**						
Node	GPU	n/P	v_i	p_i	t_{step}	Eff
1536	9216	5.4M	3.6	11.6	.81	100
2304	13824	3.6M	3.6	12.3	.65	83.0
3072	18432	2.7M	3.6	12.3	.71	57.0
3840	23040	2.1M	3.6	13.5	.54	60.0
4608	27648	1.8M	3.6	12.8	.46	58.6
$N=8$, $N_q=11$, $\Delta t=$ **8.e-4**, $L=30$, **2-Cheb-ASM:8641**						
Node	GPU	n/P	v_i	p_i	t_{step}	Eff
1536	9216	5.4M	-	-	-	-
2304	13824	3.6M	5.7	7.2	.55	100
3072	18432	2.7M	5.7	7.2	.56	73.6
3840	23040	2.1M	5.7	7.2	.39	84.6
4608	27648	1.8M	5.7	7.2	.36	76.3

Table 2: NekRS Strong Scaling using BDF2 with characteristic.

for cases involving rotating geometries or domains having widely disparate resolution requirements in close proximity. Another common use case arises when one wants to merge meshes from two or more subregions that have complex domain features, each of which are readily meshed individually. Overset grids obviate the need to make the combined meshed conforming. This latter use case is quite common in reactor thermal-hydraulics modeling. For example, if one wants a detailed mesh that resolves a complex spacer grid to merge with a larger mesh to resolve turbulence within fuel rod bundles downstream of the spacer grid.

The overset grid support requires resolution of several technical issues relating to mass conservation, timestepping, and parallelism which have been addressed for the SEM in CEED-supported work [17, 18] and in earlier efforts by Merrill *et al.* [16]. The most critical technology, however, is to have fast and robust general interpolation, in parallel. Nek's `gslib` library has provided this support for over a decade and has demonstrated scalability to millions of ranks. For interpolation, there are two parts: `findpts()`, and `findpts_eval()`. The former identifies the processor, element, and local element coordinates $(p^*, E^*, \mathbf{r}^*)$ corresponding to any requested point $\mathbf{x}^* \in \Omega$. The latter uses these computational coordinates to evaluate $u^* := u(\mathbf{x}^*)$ for any field, u. In the case of nonmoving grids, `findpts()` is called only once, whereas *findpts_eval()* is called on

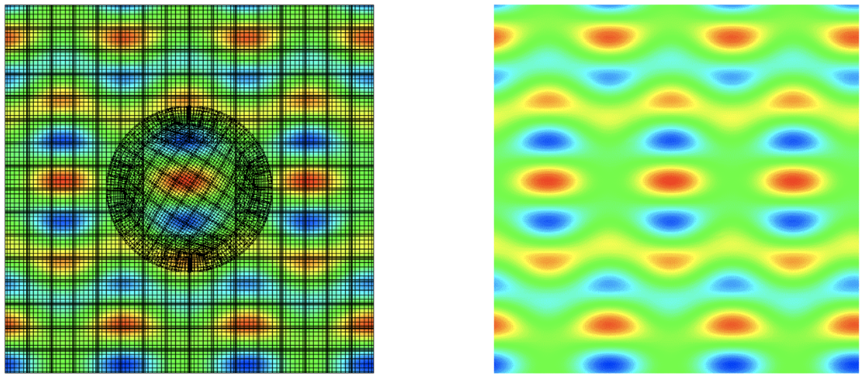


Figure 9: 3D NekRS neknek simulation with two meshes, a rotating cylinder and square annulus shown with (left) and without (right) the spectral element grids. No sign of mesh imprinting is visible in the figure on the right.

every timestep to set the boundary values along the subdomain interface in the overlapping regions.

ANL summer student Neil Lindquist (UTK) has updated both `findpts()`, and `findpts_eval()` to run on GPUs when called within NekRS. The principal advantage of this development is that the volumetric data does not need to be moved to the host in order to do the interpolation. Neil has also extended NekRS to have basic support for the overset grids that are required by our industrial partners. Figure 9 illustrates the results for a 3D test problem with a rotating cylindrical mesh embedded in a fixed outer mesh.

NekRS and Nek5000 are being used by Kairos Power to develop high fidelity datasets for pebble bed configurations with liquid salts. The data will be used to benchmark faster running models, including RANS models and porous media models. Recently, the Nek5000 overset grid variant (neknek) has been used by Kairos Power for investigate complex pebble bed configurations that involve not just a pebble bed but also flow bypass through graphite blocks. This use case is particularly important, as bypass flow is a critical concern not only for liquid salt reactors but also for traditional gas reactors. Support for overset grids with NekRS will be of significant value in future high-fidelity thermal-hydraulics simulations, enabling transition to larger models that are more representative of the configurations of interest to Kairos Power.

NEAMS Molten Salt Reactor Solver Development. With ANL summer student Yimin Lin (Rice University) and in collaboration with the Argonne NEAMS team, we are developing Navier-Stokes (NS) solver coupled Poisson-Nernst-Planck (PNP) solver for the study of molten salt reactor systems. We implemented electroneutrality constraints into NekCEM's existing PNP solver and translated this into Nek5000 for a new NS-PNP solver. We validated the results with NekCEM for the PNP part.

We consider fluid interacting with ions governed by the coupled NS-PNP solver written as

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} - \nu\nabla^2\mathbf{u} + \nabla p &= q\mathbf{E}, & (1)\\ \nabla\cdot\mathbf{u} &= 0, & (2)\\ \frac{\partial c_i}{\partial t} + \nabla\cdot(\mathbf{F_i} + c_i\mathbf{u}) &= 0, & (3)\\ -\nabla\cdot\left(\sum_{i=1}^{M}\mu_i z_i c_i \nabla\phi\right) &= \nabla\cdot\left(\sum_{i=1}^{M} z_i D_i \nabla c_i\right), & (4)\end{aligned}$$

where ν is the fluid viscosity and the diffusion flux is defined by

$$\mathbf{F}_i = -D_i\left(\nabla c_i + z_i c_i \frac{e}{k_B T}\nabla\phi\right). \qquad (5)$$

Figure 10 demonstrates the simulations for four species with electroneutrality. Currently we are developing improved implementation for the electroneutrality that will be consistent with the boundary conditions for multiple species.

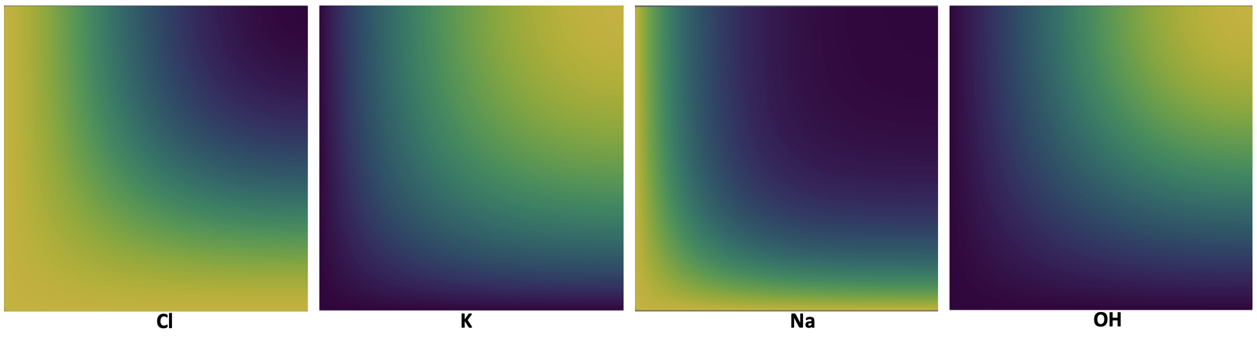


Figure 10: Nek5000 simulation based on Navier-Stokes solver coupled with Poisson-Nernst-Planck solver: molar concentrations of four species with electroneutrality constraints.

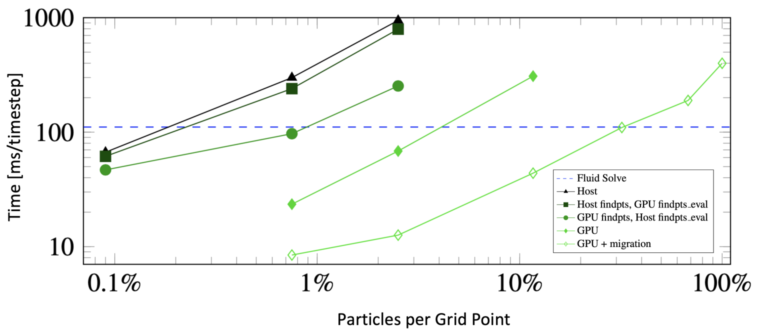


Figure 11: NekRS Lagrangian particle tracking: comparison of the implemented optimizations with $64^3 \times 8^3 \approx 134 \times 10^6$ grid points distributed over 60 GPUs. Solid lines show time to simulate particles in addition to the cost to simulate the fluid.

2.5 Lagrangian Particle Tracking for COVID-19 Research

Multiple groups are using Nek5000/RS to track aerosol dispersion in flow situations related the spread of COVID-19 and other airborne viruses (e.g., [3, 8, 14]). Such dispersion can be assessed by monitoring scalar field evolution [3, 8] or by directly tracking Lagrangian particles [14]. Nek5000 currently supports state-of-the-art one-, two-, and four-way coupling for particle tracking through the ppiclF library developed by David Zwick at the University of Florida [22]. The library makes extensive use of the parallel `findpts` and `findpts_eval` routines from Nek's `gslib` library. ANL summer student Neil Lindquist (UTK) has ported these utilities to NekRS, which allows these interpolation utilities to work with the data directly on the device, rather than moving data to the host for interpolation.

Neil has extended the interpolation capability to support efficient particle tracking in NekRS (currently, using a simple Lagrangian model, $\dot{\mathbf{x}}^* = \mathbf{u}(\mathbf{x}^*)$). From an initial "host-only" implementation, the optimized version is almost two orders of magnitude faster for a 512^3 NekRS simulation running 3.3M particles, as shown in Figure 11. Among the many developments, one of the most important is to migrate particles to the GPU where the data resides; in the case of GPUs the subdomains are relatively large such that particles spend many timesteps between migrations and the associated overhead is thus readily amortized. We see in the case of Figure 11 that this strategy yields a cost per particle update that is only four times greater than the (fast, Eulerian) Navier-Stokes update when the number of particles equals the number of grid points. Figure 12 illustrates particle dispersion in three other applications, namely, a turbulent atmospheric boundary

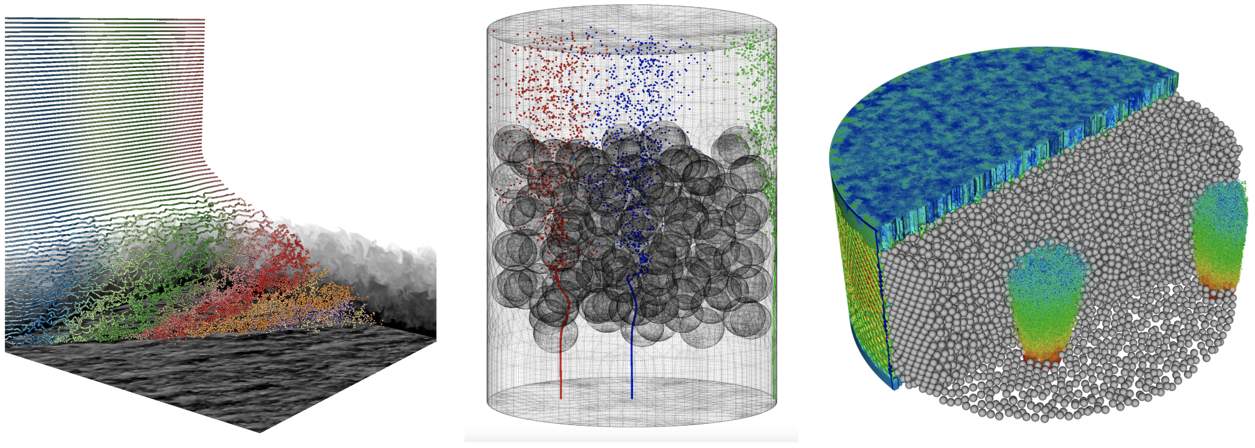


Figure 12: NekRS Lagrangian particle tracking simulations for atmospheric boundary layer flow (GABLS) and flow dispersion in pebble bed reactor model problems.

layer flow and pebble-bed reactor model having 146 and 44257 spherical pebbles.

3. CEED RESULTS ON FRONTIER AND AURORA EA SYSTEMS

In this section we report recent porting and performance result from our work on early-access (EA) systems for Frontier and Aurora, using AMD and Intel GPUs respectively.

3.1 ExaSMR 17×17 Rod-Bundle Strong Scaling Performance on Spock

With a new release of NekRS 21.1, we performed strong-scaling studies of a 17×17 pin bundle of the type. The mesh comprises 27700 elements in the x-y plane and is extruded in the axial (z) direction with 3 layers. In this study, we do not use characteristics-based timestepping, but instead use a more conventional semi-implicit scheme that requires CFL 0.5 because of the explicit treatment of the nonlinear advection term. Table 3 presents strong scaling results on both Summit and Spock for a case with $E = 83100$ and $N = 7$ ($n = 28$M) for n/P ranging from 5.7M down to 1.7M. We see that 68% efficiency using $n/P = 2.3$M on Spock while 82% efficiency using $n/P = 2.0$M on Summit.

ExaSMR results on Summit (NVIDIA V100 GPUs), 17×17 Rod-Bundle						
Node	GPU	n/P	v_i	p_i	t_{step}	Eff
1	5	5.7007e+06	3	1	1.1652e-01	100
1	6	4.7506e+06	3	1	9.7226e-02	99
2	8	3.5629e+06	3	1	7.7673e-02	93
2	10	2.8503e+06	3	1	6.7980e-02	86
3	14	2.0360e+06	3	1	5.0815e-02	82
3	16	1.7815e+06	3	1	4.8664e-02	75
3	18	1.5835e+06	3	1	5.9795e-02	54

ExaSMR results on Spock (AMD MI100 GPUs), 17×17 Rod-Bundle						
Node	GPU	n/P	v_i	p_i	t_{step}	Eff
2	5	5.7007e+06	3	1	1.5523e-01	100
2	6	4.7506e+06	3	1	1.3193e-01	98
2	8	3.5629e+06	3	1	1.0567e-01	92
3	12	2.3753e+06	3	1	9.5255e-02	68
4	16	1.7815e+06	3	1	7.6610e-02	63

Table 3: NekRS Strong Scaling Performance on Summit (top) vs. Spock (bottom). 17×17 rod-bundle with three layers in z with E=83100, n=28M.

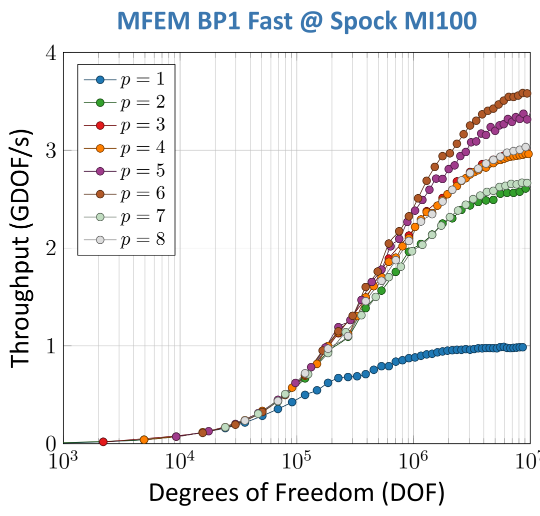
3.2 MFEM *fast* Kernels on AMD MI100 GPUs

Figure 13 present the performance of the new MFEM *fast* kernels on a single AMD MI100 GPU hosted on the Spock Frontier EA system for the BP1 and BP3 CEED benchmarks.

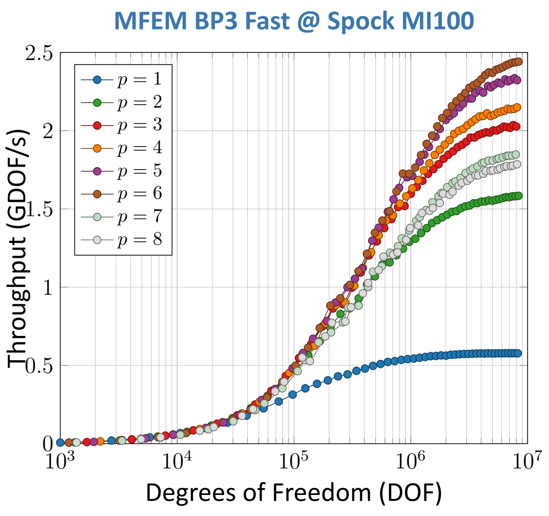
3.3 libParanumal Performance on AMD MI100 GPUs

The *libParanumal* package developed under CEED includes implementations for the CEED BP and BPS bake-off problems consisting of solving L^2 mass projection or the Poisson equation posed on a cube domain with Dirichlet boundary conditions and discretized with hexahedral finite elements.

For this phase of preparing for the Frontier exascale system we targeted test platforms include an AMD MI100 GPU hosted on the Spock Frontier EA system and we used a system equipped with the much more widely available NVIDIA V100 PCI-E variant as a baseline. The first component of this activity was to calibrate our expectations. On the one hand we anticipate that the superior memory bandwidth of the MI100 which has a peak theoretical HBM bandwidth of 1200GB/s with sustained achievable streaming in excess of 1000GB/s will yield higher throughput for kernels with low arithmetic intensity than the V100 which has peak theoretical HBM bandwidth of 900GB/s and achievable streaming throughput of over 800GB/s (see our technical report [6] for a detailed study of the attainable streaming throughputs on V100 and MI100 GPUs). On the other hand the aggregated on-core shared memory capacity of the MI100 is approximately



(a) BP1 kernels



(b) BP3 kernels

Figure 13: CEED BP1 and BP3 benchmark on a single AMD MI100 GPU for the non-deterministic (fast) MFEM kernels.

the same as the V100 GPU which can be the main performance limiting factor when running kernels with highly collaborative threads communicating via the local data share. The matrix-vector kernels at the heart of the BP5 and BPS5 benchmark implementations rely critically on shared memory for thread-block level collaboration [20], thus although the matrix-vector operations are nominally HBM memory bound the actual achieved performance may be lower than the theoretical peak of the MI100 may suggest.

The CEED BP benchmarks are designed to test throughput performance and portability of core finite element kernels on both CPU and GPU. In this work we have focused on maintaining appropriate relatively high performance of the *libParanumal* BP solvers when transitioning from the NVIDIA V100 to the AMD MI100 GPU. The CEED team collaborated closely with AMD Research to optimize the *libParanumal* compute kernels so that our kernels which were previously optimized for NVIDIA GPUs can also run efficiently on AMD MI GPUs with minimal architecture specific tweaks. In Figure 14 we show a comparison of the throughput on these two GPUs for the CEED BP1 and BP3 problems[1] We see that both the BP1 and BP3 throughputs for both GPUs is similar with some minor variations at each order related to the specifics of the way that the compilers generate code and the different device memory throughputs and on-core shared memory capabilities. The experiments confirmed our model predictions based on critical performance limiters for both GPUs.

The CEED BPS benchmarks are made more challenging than the BP problems by the inclusion of a Kershaw coordinate transform [12] that results in geometrically distorted elements and also the requirement to use preconditioners to accelerate convergence of the iterative solver. A sample mesh subject to the piece-wise linear Kershaw map is illustrated in Figure 15.

Initial experiments by the CEED team revealed that depending on the severity of the distortion it could take hundreds of preconditioned conjugate gradient iteration to converge. Over the reporting period we have worked to improve the preconditioning capabilities of *libParanumal* to specifically address this type of situation. For instance we enhanced the existing heterogeneous multigrid by re-engineering the p-multigrid and algebraic multigrid implementations including the addition of highly customized sparse approximate inverse based smoothers. These improvements resulted in substantial reductions in both iteration counts and in computational time for solving the Kershaw mapped BPS problems. Considerable additional speed ups were achieved when combining the new smoothers with reduced precision for the multigrid cycle (see Section

[1]The CEED BP1 and BP3 were configured with number of quadrature points in each hexahedral axis to be the polynomial degree plus two.

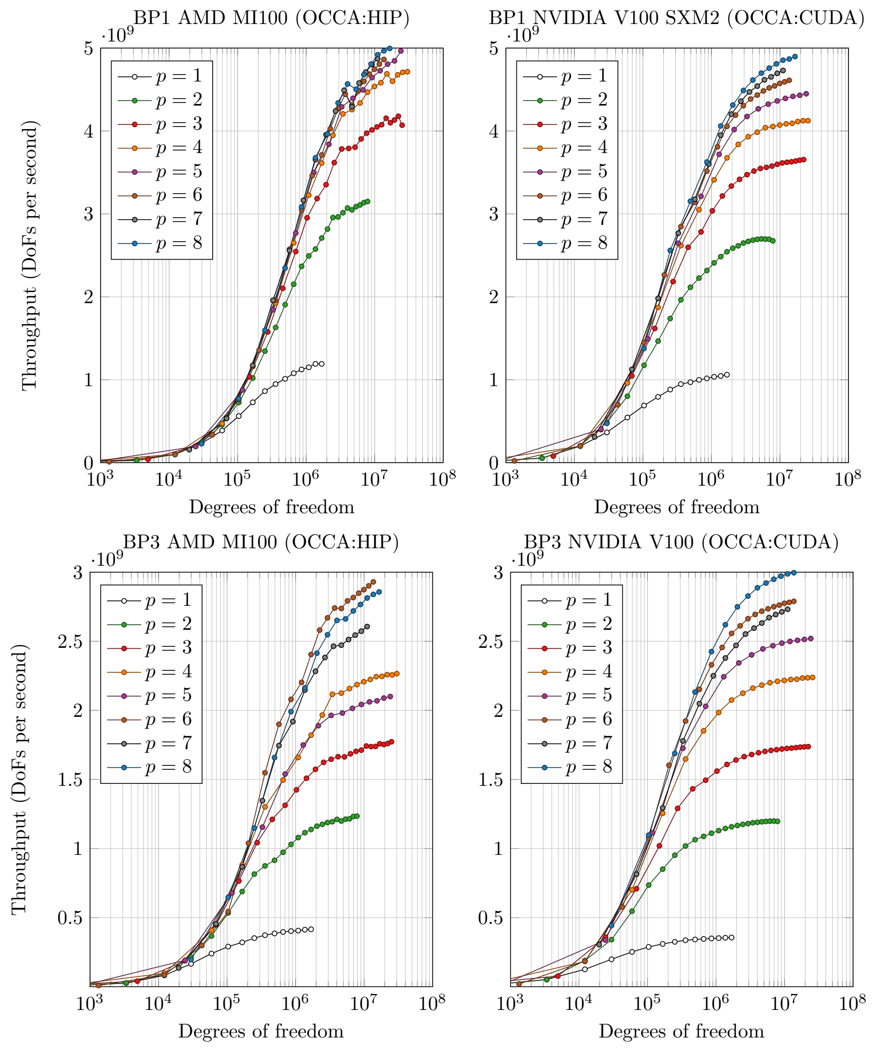


Figure 14: Performance of the benchParanumal version of the CEED benchmark problem BP1 (top) and BP3 (bottom) on a single GPU of HPE/Tulip AMD Instinct$^{\text{TM}}$ MI100 (left) and a single NVIDIA V100 PCI-E (right).

5.2 for performance projections) and an error indicator based stopping criterion to avoid over solving the linear system

In Figure 16 we show experimentally measured throughputs for a typically configured CEED BPS5 running a sequence of problems of different sizes on both the MI100 and V100 using the same kernels expressed in the OCCA kernel language (OKL). Notably the MI100 and V100 have very similar asymptotic throughput at larger problem sizes with the former overtaking the latter for sufficiently large problems. The performance curves show some differences for smaller problem size indicating a slightly higher launch overhead for the AMD GPU despite the Spock kernels being launched with the experimental `AMD_DIRECT_DISPATCH` option enabled. Also in Figure 16 we show the per kernel throughput for a fixed size mesh with 24^3 elements of degree 7. We see for this problem with approximately $4.7M$ global degrees of freedom that the streaming kernels running on the MI100 are consistently faster than the V100. The picture is a little more mixed for the matrix-vector kernels there is some variants in which V100 outperforms the MI100. Overall these results suggest that for sufficiently large problems the MI100 and V100 are relatively well matched as predicted by their similar aggregate on-core shared memory capacity which dictates their overall occupancy for the most time consuming matrix-vector kernels.

Finally, we note that the results in this section are subject to change as the kernel implementations and

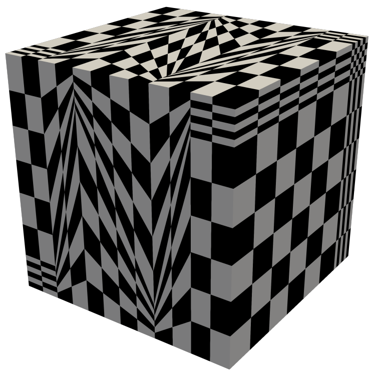


Figure 15: Illustration of a 12^3 element hexahedral mesh subjected to the piecewise linear Kershaw coordinate transform used in the CEED BPS5 benchmark problem.

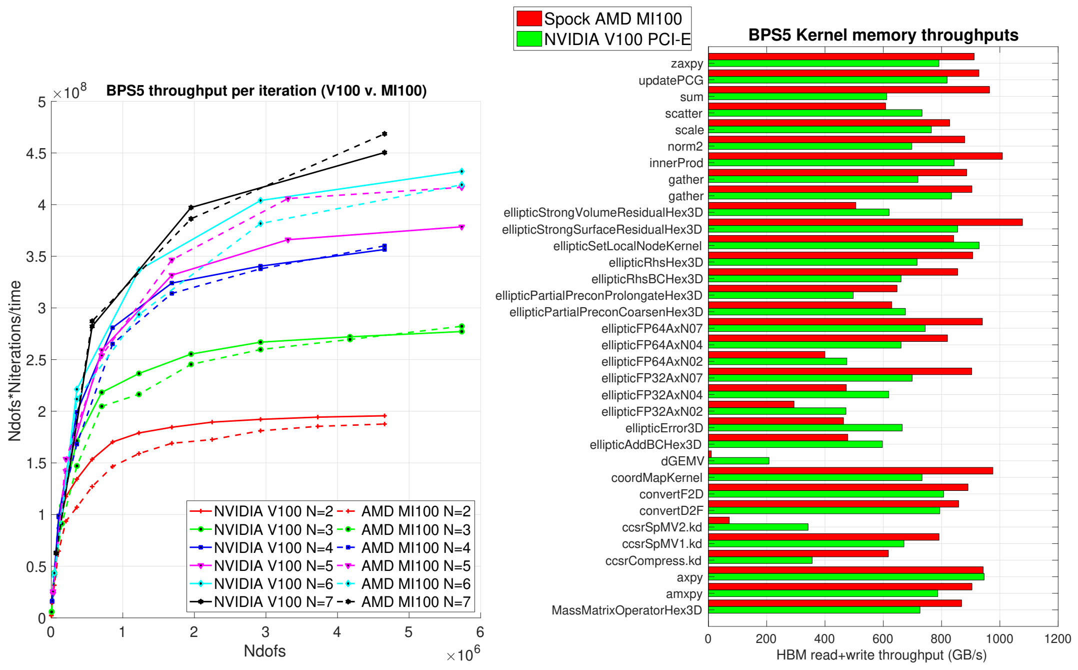


Figure 16: Left: comparison of throughputs for BPS5 on AMD MI100 (spock) and NVIDIA V100 PCI-E GPU. Right: Comparison of kernel HBM throughputs for libParanumal BPS5 kernels on AMD MI100 (Spock) and an NVIDIA V100 PCI-E GPU with discretization: E=24^3 degree 7 hexahedral elements which amounts to 4.6M global degrees of freedom.

structure of the multigrid hierarchy are further optimized for AMD GPUs.

3.4 OCCA DPC++ Backend

OCCA v1.2.0 (see Section 6.6) includes a new DPC++ backend made possible by a joint project between ALCF, Intel, and Shell. The main highlights of the DPC++ backend are:

- Using freely available DPC++ implementations, this backend can be run on CPUs, GPUs, and FPGAs.

- Memory allocation/deallocation is handled using the DPC++ USM functions, which are now part of the SYCL 2020 standard.
- OKL kernels are translated to lambda functions for transparency and performance purposes.
- The OKL `@outer` and `@inner` loop indices are mapped to work-group IDs and work-items, respectively in a similar fashion to OpenCL.

During development the Intel oneAPI DPC++ compiler was used. Intel GPU hardware was used for testing, including the JLSE Aurora testbeds at ALCF. In the near future testing will be expanded to included other DPC++ implementations and other vendor hardware. The main contributors to the development of the DPC++ backend were Anoop Madhusoodhanan Prabha (Intel), Cedric Andreolli (Intel), Kris Rowe (ALCF), Phillipe Thierry (Intel), Saumil Patel (ALCF). Future work includes profiling and optimizing build + launch of the inlined lambdas.

4. CEED PERFORMANCE ON FUGAKU AND A100

In this section we report CEED performance results on non-ECP hardware, including the Fugaku's A64FX chip and the NVIDIA A100 GPU architecture.

4.1 Nek5000/NekRS Advances on Fugaku

The principal activities for Nek5000/RS on Fugaku have been the development of a parallel reader that has no MPI-I/O dependencies and initial port and tuning of NekRS to the A64FX architecture.

Half-scale system access to Fugaku was awarded to the Nek team in late March, 2021. We were able to submit jobs for Nek5000 using 1/2 Fugaku but experienced problems in reading the meshes for larger jobs because Fugaku did not have MPI-I/O support. Jobs on more than 2000 nodes would stall at the mesh reading stage. For this reason, we implemented a new stand-alone parallel reader which will be utilized in both Nek5000 and NekRS. An Argonne Summer Intern, Yimin Lin (Rice University) implemented the new distributed reader including single-interface support for different files (rea, par, re2, map, ma2, co2) used in Nek5000/NekRS. Reading is done by a user-defined number of ranks and the requisite parallel communication is performed in $\log P$ time using communication utilities in `gslib`. The new code has been tested on Summit at small scale and will be tested up to the full-scale system on Summit before testing on Fugaku.

More recently, NekRS has been ported to Fugaku and, with significant help from the RIKEN team (Miwako Tsuji and Mitsuhisa Sato), the principal BK5 kernel has been highly optimized. We will continue this collaboration and anticipate additional speedup in the future. The team intends to submit an SC2022 paper on this topic.

4.2 MFEM's A64FX Results

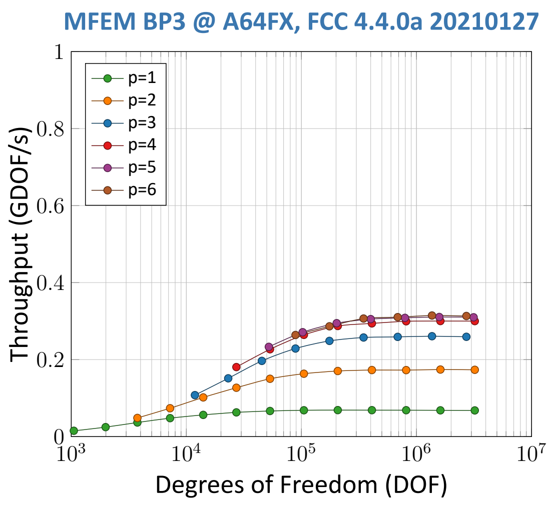
Following the initial work in 2021, the CEED team at LLNL continued optimizing the MFEM library to the Fugaku architecture and performed an incremental performance evaluation. The BP3 benchmark used is described in the CEED milestone report CEED-MS6 [7]. It is a scalar PCG with stiffness matrix: we solve $A\underline{u} = \underline{f}$, where A is the Poisson operator.

Different compiler configurations were tested initially: Fujitsu's C++ compiler (FCC 4.4.0a 20210127), as well as the GNU GCC 10.2.0. Performance updates with Fujitsu's C++ compiler (FCC 4.6.1 20210812) and the GNU GCC 11.2 are presented in the Figures 17 and 18.

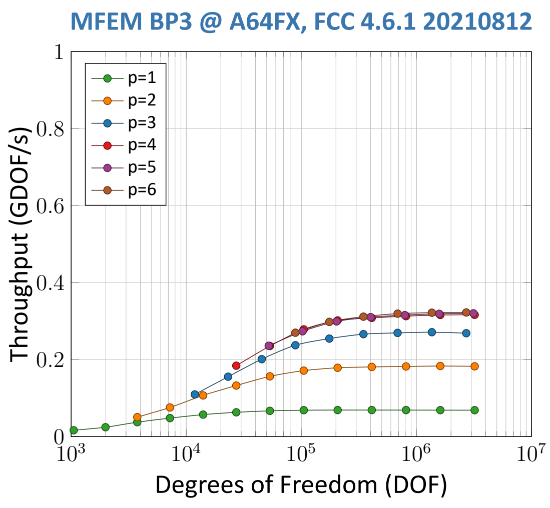
Since the Fujitsu's C++ compiler performance results were obtained with the `Kfast` option, in Figure 19(a) we present also the performance gain when enabling this option in the GCC compiler.

4.3 ExaSMR Single Rod Performance on A100 GPUs

We demonstrate a single GPU performance on ANL's ThetaGPU system (which has NVIDIA A100 GPUs) for NekRS full solver simulations for ExaSMR's singlerod problem with the problem size of 2.4M grid points per GPU. Our tests are extended to Aurora-development system for this problem. Table 4 shows the performance in comparison to other GPU platforms. The simulations are for 500 steps and the averaged timing per step,

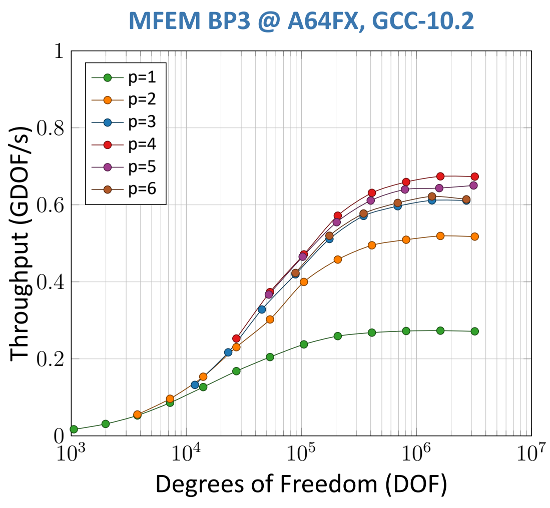


(a) BP3 with Fujitsu fcc-4.4.0 compiler

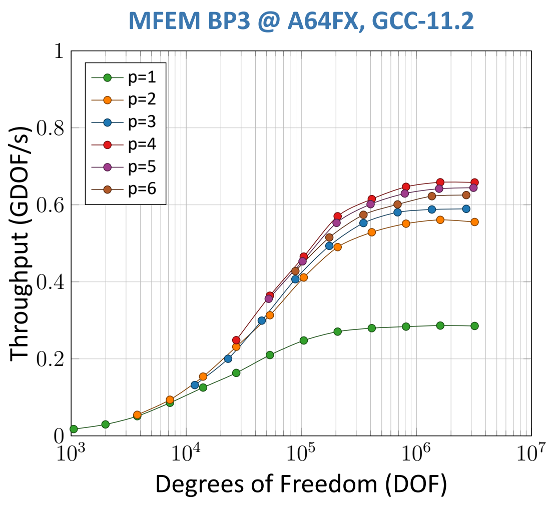


(b) BP3 with Fujitsu fcc-4.6.1 compiler

Figure 17: MFEM BP3 benchmark on one A64FX chip with Fujitsu fcc-4.4.0 (*a*) and fcc-4.6.1 (*b*) compilers.



(a) BP3 with GCC 10.2



(b) BP3 with GCC 11.2

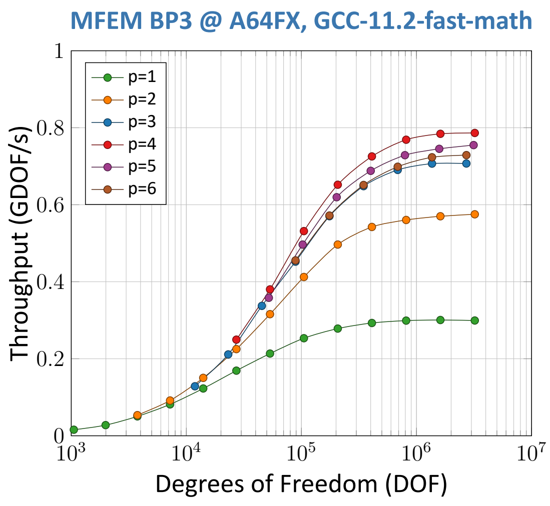
Figure 18: MFEM BP3 benchmark on one A64FX chip with GCC 10.2 (*a*) and 11.2 (*b*) compilers.

t_{step}, is measured in seconds for 101-500 steps. R is the ratio of t_{step} to Summit V100. The timestep size is Δt=1.2e-03 (CFL=7.3). Characteristic-based BDF2 with 1 substeps is used for timestepping. Tolerances for pressure and velocity are 1e-4 and 1e-6, respectively.

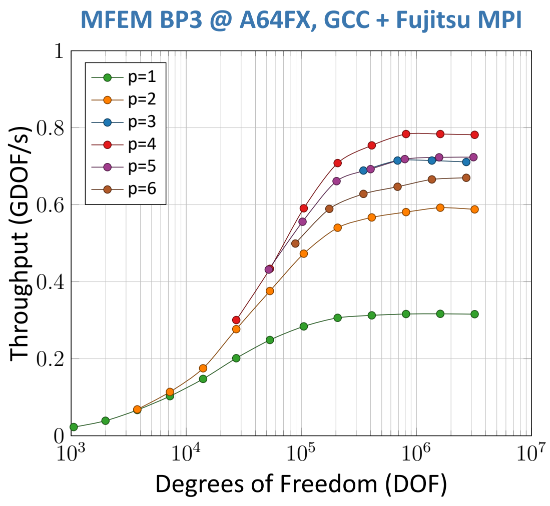
4.4 MFEM BP1 Results and Tensor-Core MMA Kernels for A100 GPUs

The new MFEM *fast* and *XFL* kernels from Section 2 were ported on the NVIDIA A100 GPU achitecture and we report the obtained speedups in Figures 20 and 21.

Additionally for the A100 chip, we collaborated with Peng Wang and Jiqun Tu from NVIDIA to re-



(a) BP3 with GCC 11.2 `ffast-math` and `OpenMPI 4.1.1`.



(b) BP3 with GCC 11.2 `ffast-math` and Fujitsu's `MPI`

Figure 19: MFEM BP3 benchmark on one A64FX chip with GCC 11.2 `ffast-math` and `OpenMPI 4.1.1` (a) or Fujitsu's `MPI` (b).

NekRS for singlerod simulation, $E = 7168$, $N = 7$, $n = 2.4M$, $Re = 5000$					
system	backend	t_{500}	t_{100}	t_{step}	R
Summit/V100	CUDA	4.089e+01	9.018e+00	0.0795	1.00
ThetaGPU/A100	CUDA	2.503e+01	5.587e+00	0.0485	0.61
Spock/MI100	HIP	5.007e+01	1.103e+01	0.0975	1.22
Tulip/MI100	HIP	4.870e+01	1.060e+01	0.0953	1.19

Table 4: NekRS performance on a single GPU.

interprete the partially assembled tensor contraction kernels as a new type of general matrix multiply + tensor transpose operations kernels using the *tensor core* matrix multiply-add (MMA) parallel thread execution (PTX) instructions. This new type of kernels uses for the first time all the hardware available on the NVIDIA GPU chip by the use of the FP64 tensor cores.

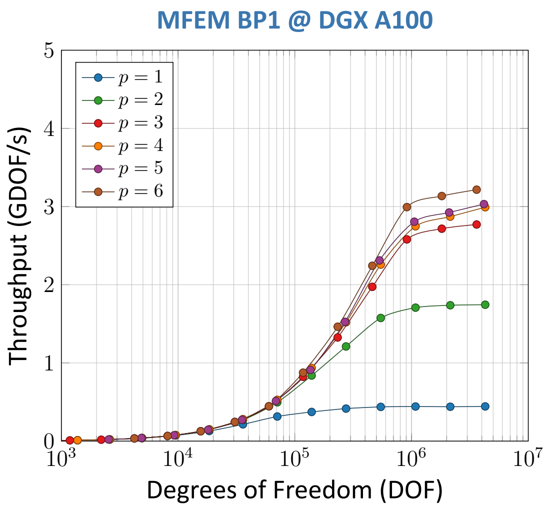
The instruction can be inserted into CUDA C++ code via the inline PTX syntax. It is a warp-synchronous, collective M-by-N-by-K matrix operation: the matrix data elements are automatically distributed between the hardware threads. Only the `mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64` is available on the A100 hardware, but useful enough to recast the tensor products into a series of matrix multiply-add operations. Figure 22 presents the MFEM *MMA* kernels for the BP1 benchmark on a single NVIDIA A100 GPU.

5. ADDITIONAL TOPICS

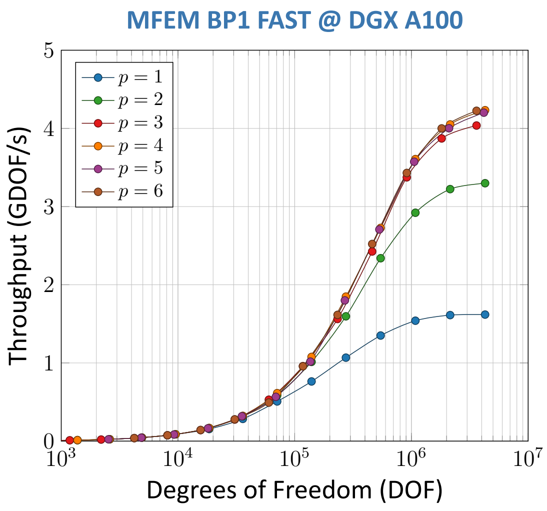
This section covers additional research performed by the CEED team on topics such as just-in-time (JIT) compilation at scale and the use of mixed precision algorithms in high-order methods.

5.1 JIT in Applications at Scale

The easiest way of compiling code at run-time - also called *just-in-time* (JIT) - requires to execute `system()` calls in order to launch compilation commands MFEM embeds the code to be compiled at run-time as standard strings, as well as all the directory paths and ahead-of-time (AOT) compilation commands used for the library itself. Recent version of MPI implementations usually claim that: `In general, if your application calls system() or popen(), it will likely be safe.` However, there are still some implementations



(a) Deterministic kernels



(b) *Fast* non-deterministic kernels

Figure 20: MFEM BP1 benchmark on a single NVIDIA A100 GPU for the deterministic (*a*) and *fast* non-deterministic (*b*) kernels.

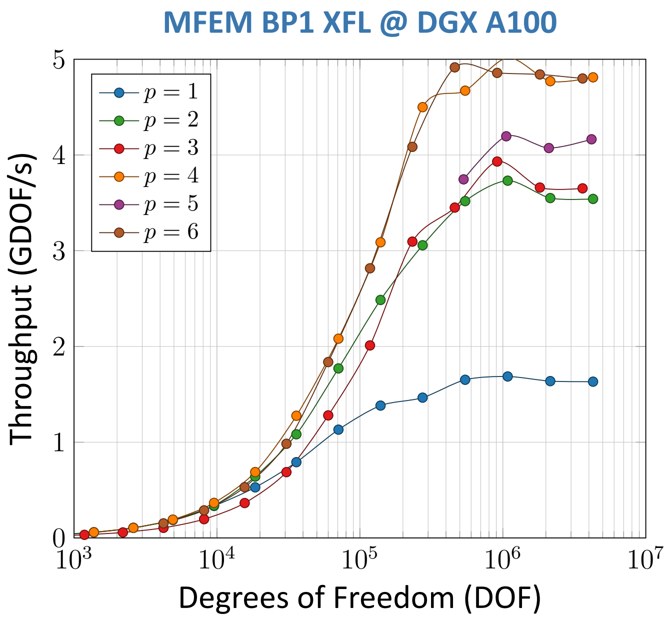


Figure 21: Reduced $N_{0.8}$ obtained with the *XFL* kernels on CEED BP1 benchmark on a single NVIDIA A100 GPU.

that have limitations, depending if they rely on specific software or network stacks.

There are three ways to be able to launch these commands:

- direct `fork()` and `system()` calls,
- through the `MPI_Comm_Spawn()` interface,

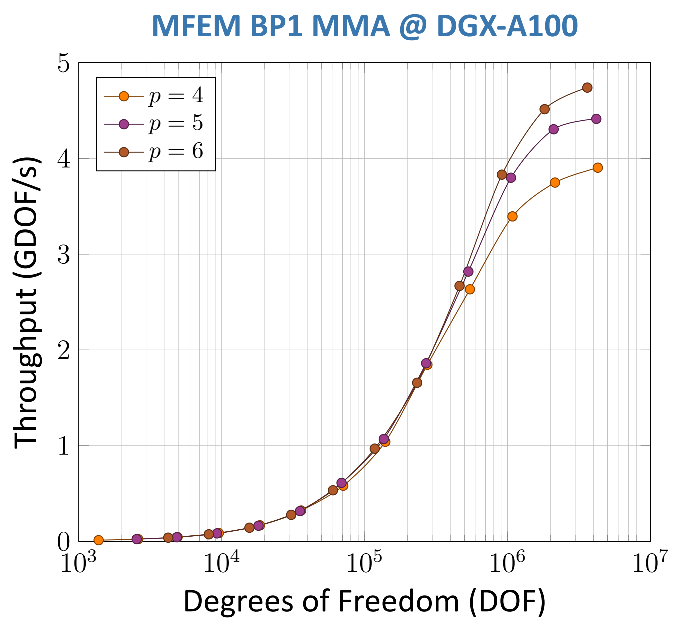
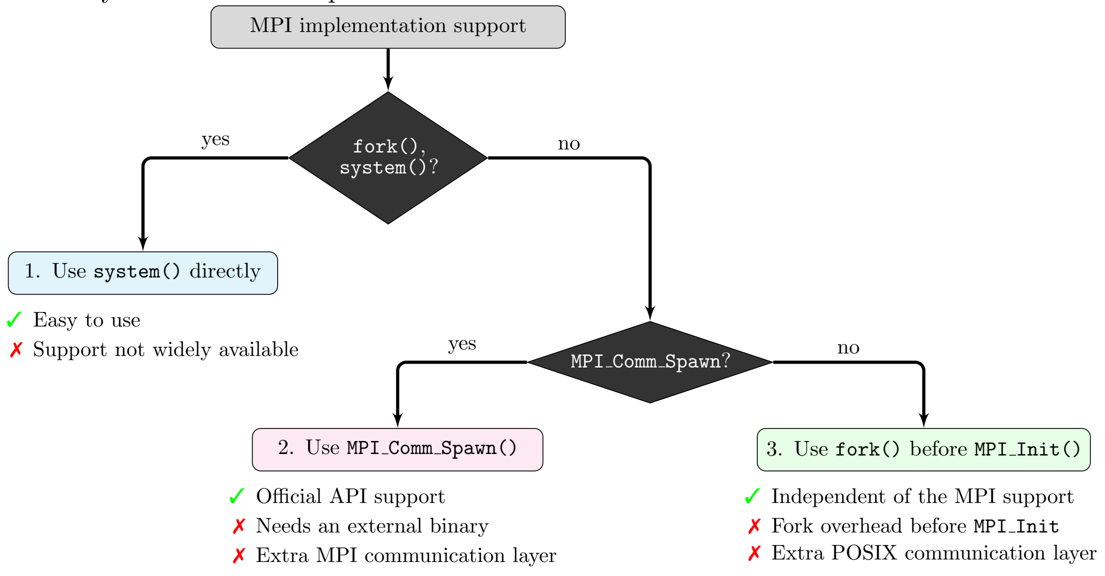


Figure 22: *MMA* kernels BP1 benchmark on a single NVIDIA A100 GPU.

- by *forking* before `MPI_Init` and keeping one thread for the direct `fork()` and `system()` calls.

Each way has been implemented and explored in a research branch of MFEM, to be able to deal with all different MPI limitations that have been encountered on the different HPC systems tested on. Sequential and parallel support have been tested on `RedHat` Linux, `MacOS` and HPC environments such as Sierra at LLNL. Parallel runs have been tested and validated with up to 32768 `CPU` MPI ranks and 1024 `GPU` MPI ranks.

Here is the decision tree followed by the MFEM implementation to allow executing compilation commands on the variety of HPC or desktop environments:


MPI implementation support
fork(), system()?
yes
no
1. Use system() directly
✓ Easy to use
✗ Support not widely available
MPI_Comm_Spawn?
yes
no
2. Use MPI_Comm_Spawn()
✓ Official API support
✗ Needs an external binary
✗ Extra MPI communication layer
3. Use fork() before MPI_Init()
✓ Independent of the MPI support
✗ Fork overhead before MPI_Init
✗ Extra POSIX communication layer

Direct `fork()` and `system()` calls support. Using directly the `fork()` and `system()` calls from MPI ranks is straightforward and has been the first implementation in order to have the JIT working, both for sequential and parallel runs. Most Linux and MacOS platforms offer these support, but also Sierra at LLNL.

However, MPI processes support for fork(), system(), or popen() system calls are not guaranteed. As stated in the OpenMPI FAQ (https://www.open-mpi.org/faq/?category=tuning#fork-warning), it depends on a lot of different factors, that includes: the operating system, the underlying compute hardware or the network stack, interactions with other middle-ware in the MPI process. We have encountered issues with some MPI implementations. One example: fork() is not available on Blue Gene, consequently system() is not supported too on such systems. In some cases, the MPI will determine that it is not safe to fork(). The MPI implementation can register a pthread_atfork() callback to print a warning when the process forks. If we called fork() or system() from the MPI program we got a warning, such as the following one:

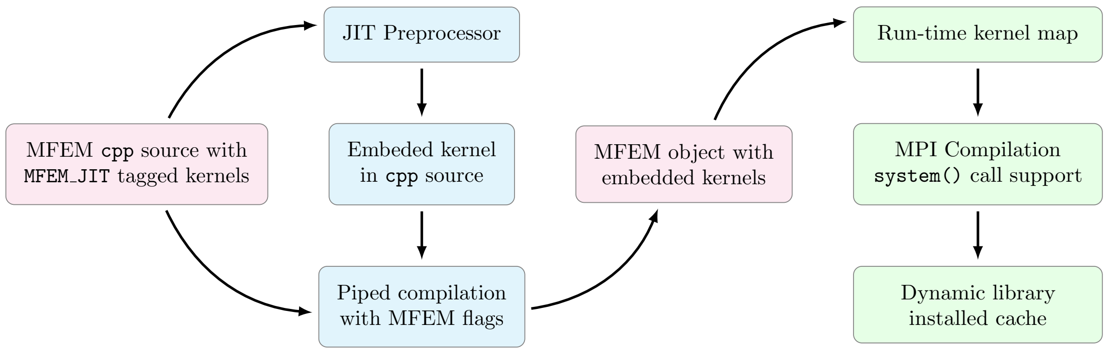
```
An MPI process has executed an operation involving a call to the
"fork()" system call to create a child process. Open MPI is currently
operating in a condition that could result in memory corruption or
other system errors; your MPI job may hang, crash, or produce silent
data corruption. The use of fork() (or system() or other calls that
create child processes) is strongly discouraged.

If you are absolutely sure that your application will successfully and
correctly survive a call to fork(), you may disable this warning by
setting the mpi_warn_on_fork MCA parameter to 0.
```

This is a serious warning which is due to the partial/lack of support for the fork() system call within the software stack used to deal with the underlying network layers. That's why the standard purposely refrains from incorporating OS-specific things like extended process control. The only officially supported way to execute other programs from within an MPI application is the use of MPI_Comm_spawn.

MPI_Comm_Spawn() MPI application interface. Another official way to launch other binaries is by the use of the MPI_Comm_Spawn() MPI application program interface. MPI_Comm_Spawn() spawns a number of identical binaries and in the MFEM JIT case, one single executable, capable of launching the different compilation commands. MPI_Comm_Spawn() starts identical copies of the MPI program specified by one command, establishing communication with them and returning another communicator. The spawned processes are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents. The communicator returned by MPI_Comm_Spawn() contains the parent processes in the local group and the child processes in the remote group: it allows to communicate between the two groups. However, the support for MPI_Comm_Spawn() is not fully supported on the different MPI implementations, which motivated the need for another support, independent of the underlying MPI implementation, to allow the direct fork() and system() calls.

Forking before MPI_Init, one thread for the fork() and system(). *Forking* before the MPI_Init call offers the advantage of being independent of the underlying MPI implementation. One disadvantage it brings is the duplication of each process launched through the mpirun command. All of the forked processes except one are returning, giving back the resources as only one process is kept alive to handle the compilation commands. One other advantage this method offers compared to the MPI_Comm_Spawn() in an HPC environment is the fact that no additional ranks are needed when requesting the number of nodes to the batch job scheduler, whereas the MPI_Comm_Spawn() approach do need extra allocation requirements or some MPI run options (such as oversubscribe).

MFEM JIT compilation process.


MFEM cpp source with MFEM_JIT tagged kernels
JIT Preprocessor
Embeded kernel in cpp source
Piped compilation with MFEM flags
MFEM object with embedded kernels
Run-time kernel map
MPI Compilation system() call support
Dynamic library installed cache

5.2 Potential of Mixed Precision in CEED Algorithms

The use of 64-bit floating point operations (FP64) has been a well established de facto standard in scientific computing. However, the performance of 32-bit (FP32) operations is often at least twice faster than FP64 operations on modern architectures. The reasons for this is that the FP32 arithmetic on modern processors is usually twice as fast as the FP64 arithmetic, and the number of bytes moved through the memory system is halved. The revolution in machine learning and artificial intelligence (AI) has influenced hardware design and brought interest in fast 16-bit floating-point arithmetic (FP16), since many AI applications do not require the FP32 or FP64 accuracy. Parameters for the IEEE FP16, FP32, and FP64 arithmetic precisions, and their respective peak performances on NVIDIA V100 and AMD MI100 GPUs are given in Table 5. The large accelerations that can be achieved by lowering the precision in parts of the computation has motivated us to explore the potential of using combinations of these precisions, referred to as mixed precision, to accelerate CEED Algorithms while still retaining desired accuracy.

Table 5: Parameters for the IEEE FP16, FP32, and FP64 arithmetic precisions, and their respective peak performances on NVIDIA V100 and AMD MI100 GPUs. "Range" denotes the order of magnitude of the smallest subnormal ($x_{\min,s}$), and largest and smallest positive normalized floating-point numbers. The peak 16-bit performances reported* by vendors vary depending on the instructions and special hardware acceleration used, e.g., the peak performance of 125 Tflop/s for Nvidia V100 uses FP16-TC (tensor cores) with inputs FP16, while the outputs and the computations are performed in full (FP32) precision.

Arithmetic	Size	Range			Unit round-off	Peak Tflop/s	
	(bits)	$x_{\min,s}$	$x_{\min}$	$x_{\max}$		V100	MI100
BFloat16	16	9.2×10^{-41}	1.2×10^{-38}	3.4×10^{38}	3.9×10^{-3}	N/A	92*
FP16	16	6.0×10^{-8}	6.1×10^{-5}	6.6×10^{4}	4.9×10^{-4}	125*	184*
FP32	32	1.4×10^{-45}	1.2×10^{-38}	3.4×10^{38}	6.0×10^{-8}	15.7	23
FP64	64	4.9×10^{-324}	2.2×10^{-308}	1.8×10^{308}	1.1×10^{-16}	7.8	11.5

Mixed Precision Preconditioning in libParanumal

The CEED GPU implementations of high-order hexahedral finite element operations by and large operate in the memory bound limit due to the low arithmetic intensity of the streaming operations like gather/scatter, conjugate gradient updates, reductions, as well as element local matrix-free matrix-vector products expressed in terms of tensor contractions. The CEED team has repeatedly verified that the computational time taken to complete one of these operations on the GPU is well modeled by

$$T^{op} = T_0 + \frac{B^{op} \times N_{\text{dof}}}{W^{op}_{\max}}, \tag{6}$$

where T_0 is a (nearly) universal overhead associated with launching the kernel on the GPU, N_{dof} is the number of degrees of freedom processed in the operation, B^{op} is the number of bytes of data transacted with global DEVICE memory, and $W^{op}_{\max}$ is the asymptotic global DEVICE memory throughput rate achieved for large N_{dof}. This model reflects the observation that a GPU typically streams data efficiently and kernel computation time scales linearly with problem size at a fixed rate once accessing data out of cache.

There are only a limited set of options for optimizing a memory bound solver: reduce the number of outer iterations and/or reduce the amount of data moved by the preconditioner. There are other pressing issues related optimizing message passing phases for multi-GPU computations however these are not addressed in this section. One obvious way to reduce the amount of on-GPU data movement when applying the preconditioner is to use reduced precision, i.e. reduce the number of bytes required per degree of freedom B^{op}. In the earliest days of GPGPU computing Strzodka and Göddeke experimented with using single precision preconditioners to both reduce the amount of data movement and to reduce the overall number of double precision operations as the contemporary GPUs did not support native double precision in hardware [9]. Furthermore they observed at the time that multigrid solvers [19] and also multigrid preconditioners could be robust to using selective use of reduced precision. For modern server grade GPUs such as the NVIDIA V100 or A100 and the AMD MI100 there is substantial support for double precision floating operations so the precision used for compute is less important than the precision at which data is stored.

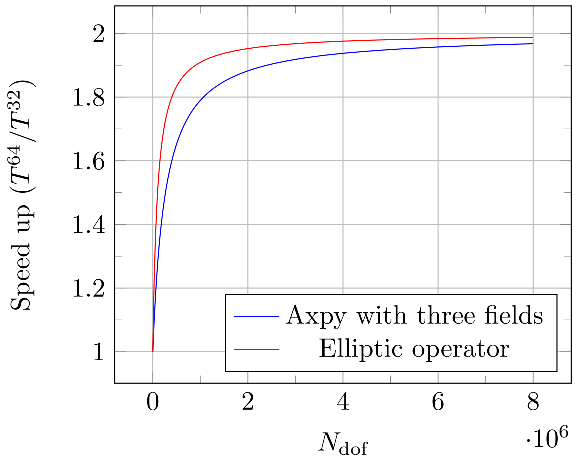


Figure 23: Comparison of theoretical speed ups for vector streaming operations and elliptic matrix-free matrix-vector multiplication for a range of N_{dof} on a theoretical GPU proxy for the NVIDIA V100. The model assumes a launch time $T_0 = 4\mu s$ and asymptotic throughput of $W_{\max} \approx 800GB/s$ for all kernels.

We can use the compute time model from Equation 6 to theoretically model the speed up one might expect when switching an operation from using say double precision to single precision by the following. For instance when evaluating of the spectral element stiffness matrix-vector product (as defined in the CEED BK5 benchmark kernel) the speed can be modeled approximately with

$$S^{Ax} = \frac{T^{Ax-64}}{T^{Ax-32}} = \frac{T_0 + \frac{8\times 8\times N_{\mathrm{dof}}}{W_{\max}^{Ax-64}}}{T_0 + \frac{8\times 4\times N_{\mathrm{dof}}}{W_{\max}^{Ax-32}}} \tag{7}$$

reflecting that this operation loads an input value and six geometric factors per local degree of freedom, and outputs one result value per local node. In most cases we have been able to optimize the matrix-free vector kernels to within 80-90% of peak throughput for a given GPU depending on the manufacturer and model. We benchmarked the kernels for instance on the NVIDIA V100 and measured asymptotic throughput up to $W_{\max}^{Ax-32} \approx W_{\max}^{Ax-64} \approx 800GB/s$ with launch times of approximately $4\mu s$ depending on the specifics of how deep the on DEVICE queue is stacked. In Figure 23 we visualize the theoretical speed up as a function of N_{dof} for the spectral matrix-vector product when reducing from double to single precision. For small N_{dof} there is no speed up as launch cost dominates the computation time as one might expect. However asymptotically the speed up does indeed approach a factor of 2 which we expect when halving the amount of data moved. In fact we can estimate that to get to within 80% of doubling the throughput by switching from double to single precision we need $N_{\mathrm{dof}} \geq 150,000$.

The speed up for simple streaming operations is a little less rosy. For instance performing an `axpy` operation requires $N_{\mathrm{dof}} \geq 400,000$ to achieve 80% of the maximum throughput doubling when switching from double to single precision. Performing a reduction operation requires $N_{\mathrm{dof}} \geq 1.2M$ to achieve 80% of the maximum throughput improvement.

We consider a concrete problem to put the above estimates into perspective. Consider a mesh of 6000 degree 7 hexahedra. The finest grid at the top level of a heterogeneous multigrid cycle has $N_{\mathrm{dof}} \approx 2M$. However at the degree 4 level is only has $N_{\mathrm{dof}} \approx 384,000$ and at the coarsest degree one p-multigrid level this mesh has a mere $N_{\mathrm{dof}} \approx 6000$. Thus only the highest levels will show major benefits from reducing the precision although it certainly will not harm throughput performance and will reduce storage requirements at the lower p-multigrid levels.

The *libParanumal* library developed at Virginia Tech has supported global precision change from inception in late 2016. In 2018 the library was updated to support limited multi-precision capabilities within the preconditioner but this was subsequently refactored out. In 2020 more extensive support for multi-precision was restored within a development fork of *libParanumal*. This update allows the library to use double precision (FP64) for the majority of set up on the HOST and in the outer Krylov iterations on DEVICE while using single precision arithmetic for the majority of the multigrid cycle (FP32). This is accompanied with other multi-precision capabilities in sparse matrix representation on DEVICE.

Toward Mixed Precision in Gingko + MFEM

Recent improvements to the integration of the Ginkgo linear algebra library [2, 1] in MFEM increase the options for using Ginkgo's mixed-precision capabilities in MFEM. Specifically, MFEM and Ginkgo solvers and preconditioners can now be used interchangeably, and Ginkgo can use MFEM's matrix-free operators in its solvers. The Ginkgo project has a mixed-precision algebraic multigrid preconditioner in active development that offers mixed-precision multigrid to MFEM when a sparse matrix is available for preconditioning, as is the case when using the low-order refined (LOR) operator to build a sparse matrix for preconditioning purposes, demonstrated previously in CEED-MS36 [13]. The Ginkgo mixed-precision AMG allows the user to have double precision on the finest level – interacting with MFEM double-precision vectors and the LOR matrix – while computing and storing data in single precision on other levels in the multigrid hierarchy. Similar to the p-multigrid example given above, there is a limit to the benefit of executing the coarser levels in lower precision, as the finest levels dominate the time, yet there is no compelling reason to convert back to double precision at the bottom of the V-cycle, and we do benefit from reduced storage requirements.

In Table 6 we consider a diffusion problem on two simple meshes provided by MFEM: "star" (a two-dimensional mesh) and "beam-hex" (a three-dimensional mesh). The LOR matrix is used to create a basic Ginkgo AMG preconditioner. The preconditioner has aggregation-based coarsening with one pre- and post-sweep of Jacobi smoothing on each level, and does four iterations of Jacobi smoothing on the coarse mesh in lieu of a coarse grid solve. The iteration counts for the full-double-precision case are compared to those of the mixed-precision preconditioner across a range of high-order discretization basis function orders; we also consider multiple values of mesh refinement, increasing the total degrees of freedom in the problem. We confirm that there is very little change in the effectiveness of the preconditioner when using the mixed-precision version compared to double precision, with iteration counts nearly identical, and reiterate that no changes were required to MFEM's operators or solvers in order to use the mixed-precision preconditioner. Performance improvements and additional options for Ginkgo's AMG preconditioning are planned.

Plans for Mixed Precision Support in libCEED

As an initial step toward mixed-precision performance enhancements of libCEED, we have added the ability to compile the libCEED library to use single precision everywhere. Furthermore, we have conducted several tests of CUDA performance in single precision. This offers a ceiling on performance for mixed single and double usage and validates the previously-discussed model illustrated in Figure 23, as we will demonstrate in the following.

In order to be able to run benchmarks for both single and double precision, a new implementation of the CEED diffusion benchmark (BP3) was created. This implementation does not rely on MFEM, as we have generally used to report performance for the GPU backends previously, due to the lack of support for single precision in MFEM. Instead, the entire problem setup is done with libCEED and low-level C code modified from libCEED tests. The linear form integration producing the right-hand side vector for the linear system is done through the creation and application of a libCEED mass operator. To achieve as much consistency as possible with the MFEM benchmark results, the handling of essential boundary conditions with the matrix-free operator is taken from MFEM's implementation, with a simple kernel for setting constrained degrees of freedom; default grid and thread block settings match those of MFEM. The conjugate gradient (CG) solver used is from the Ginkgo linear algebra library [2, 1], in which all solvers take the floating point precision as a template parameter, making it simple to switch between `float` and `double`.

mesh	order	DOF	double	mixed
star	2	5281	32	32
		20801	49	45
		82561	70	69
	3	11761	43	41
		46561	63	60
		185281	97	96
	4	20801	51	51
		82561	78	74
		328961	129	128
	5	32401	60	61
		128801	91	94
		513601	147	147
	6	46561	72	71
		185281	107	107
		739201	196	199

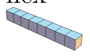
mesh	order	DOF	double	mixed
beam-hex 	2	37281	47	48
		279873	79	82
	3	120625	74	78
		924385	139	138
	4	279873	115	112
		2167425	212	215
	5	539601	155	157
		4205601	310	309

Table 6: Comparison of number of iterations for full double precision AMG preconditioner versus mixed precision with the Ginkgo library. The preconditioner is generated for the low order refined matrix corresponding to the high-order matrix-free operator.

First, to illustrate the relative cost of the libCEED operator application compared to other portions of the time spent in the iterative solver, we split the GPU activities into four categories: operator apply, performed by libCEED; setting the essential boundary conditions; performing vector updates and reduction operations required by CG at each iteration; and all other activities. For two sizes of meshes, the breakdown of these four activities is shown for the `cuda-gen` backend in Figure 24. The dominant costs are clearly the libCEED operator application and the vector operations, with the operator accounting for the majority of the time as the mesh size is increased. (The `cuda-gen` backend uses a fused operator kernel; in the case of a non-fused backend, such as `magma-det`, the domination of the operator application is even greater, as seen in Figure 25.)

Figure 26 details the speedup obtained from single precision compared to double for the `cuda-gen` backend, the best-performing CUDA backend for tensor-product elements. In the right plot, we show the speedup of the average time per iteration of the CG solver. The left plot comes from data extracted from the `nvprof` profiler tool and shows the speedup of the libCEED operator application kernel itself. The same information is provided for the `magma-det` backend in Figure 27.

In terms of the time per iteration metric, very little overall speedup is achieved for smaller problems, while the overall speedup trend is clearly a blend of the behavior of the vector operations and operator applications, with the dampening of the operator speedup especially apparent for the `cuda-gen` backend. There is a clear trend that we must have more than 100,000 degrees of freedom per GPU to achieve noticeable speedup, and more than a million DOF per GPU to reach the maximum speedup. These results line up nicely with the model given in Eq. 7 and the related discussion based on the V100 GPU. In particular, we confirm that the operator application achieves its peak speedup more quickly than the entire CG iteration. The results also demonstrate the utility of the model for predicting the range of required problem sizes to reach 80% of maximum speedup, given that each CG iteration involves operator application, vector updates, and vector reduction.

Mixed-precision libCEED operators. The addition of mixed-precision computations within libCEED itself is an area of active development. There are many possible combinations of data and computation precision within a libCEED operator. Consider the schematic in Figure 28, which represents the different kernels/functions inside a libCEED operator (element restriction, $\mathcal{E}$, basis action, B, and QFunction, D) as well as the data flow of input and output vectors. The dashed line on the left indicates that the L-Vectors are the inputs and outputs that interact with user code, while the E- and Q-Vectors are internal to libCEED.

Since we are generally memory bound with tensor-product elements, we benefit from being able to reduce movement to and from main memory as much as possible. In terms specific to the libCEED workflow of

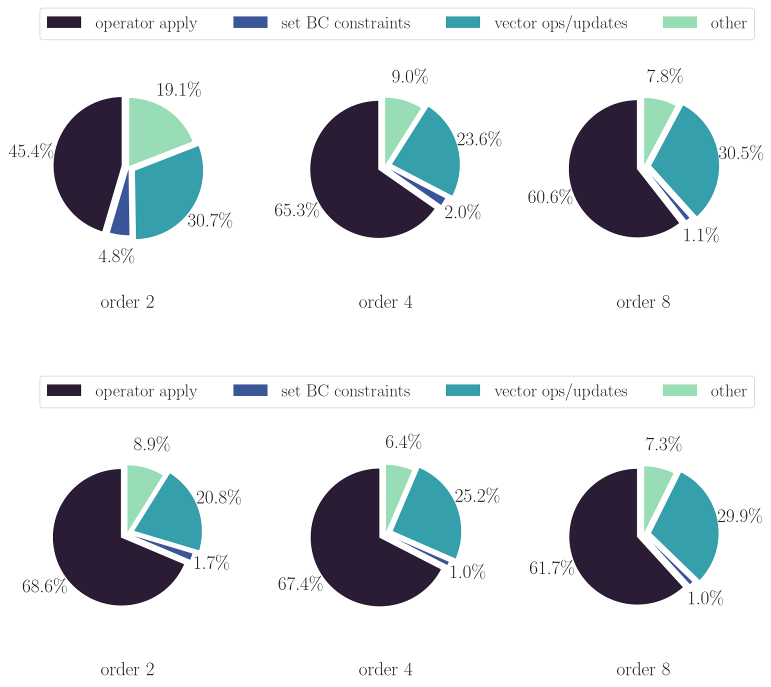


Figure 24: Breakdown of average GPU time per iteration of CG for the diffusion problem with the `cuda-gen` backend, on a mesh of: (top) $16 \times 16 \times 16$ and (bottom) $32 \times 32 \times 32$ hexahedral elements. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

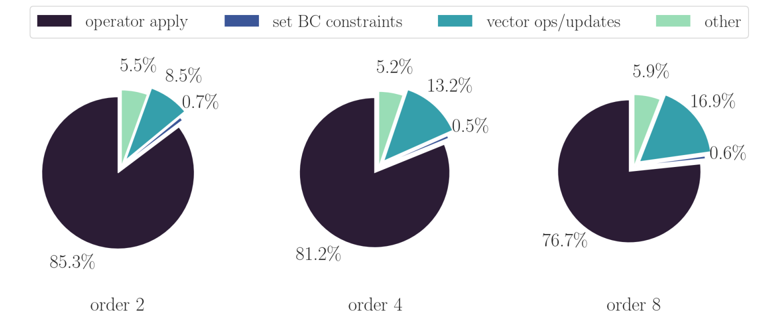


Figure 25: Breakdown of average GPU time per iteration of CG for the diffusion problem with the `magma-det` backend, on a mesh of $32 \times 32 \times 32$ hexahedral elements. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

Figure 28, this could take the form of using lower precision to represent the internal E- and Q-Vectors, while converting locally to higher precision inside the basis and QFunction kernels and performing computation in the higher precision. The benefit of this approach will depend on the backend in question. For example, in the non-fused GPU backends, the entire E- and Q-Vectors get written to memory at the end of their respective kernels, while in the fused backends (`cuda-gen` and `hip-gen`), everything is designed to be kept in registers or shared memory. This indicates that the non-fused backends would see greater relative speedup from this approach. The CPU backends, on the other hand, aim to keep the E- and Q-Vector data in cache,

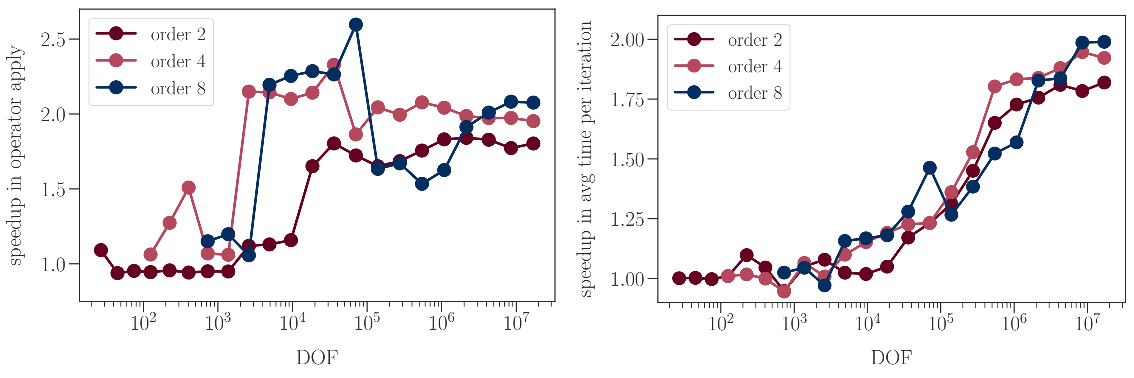


Figure 26: Speedup of the `cuda-gen` backend for the diffusion benchmark problem (BP3) when using single precision instead of double, for three orders of basis functions. The speedups are for: left, the time spent in applying the matrix-free operator with libCEED; right, the average time per iteration of CG. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

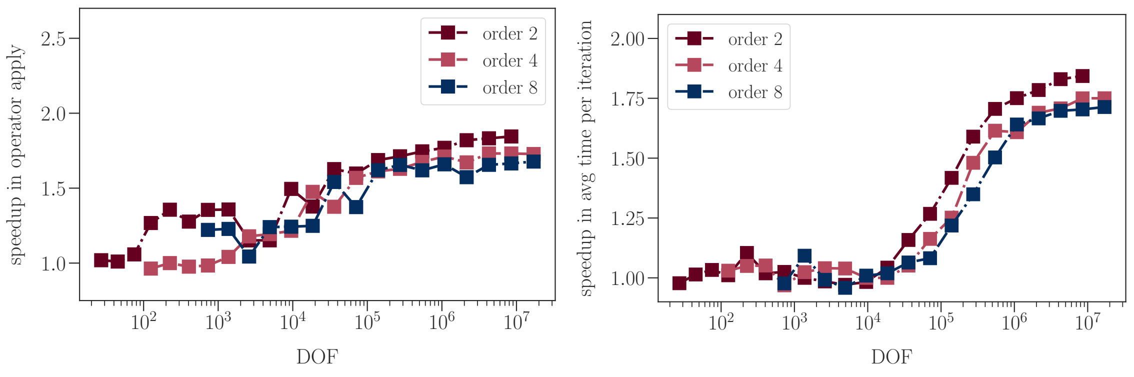


Figure 27: Speedup of the `magma-det` backend for the diffusion benchmark problem (BP3) when using single precision instead of double, for three orders of basis functions. The speedups are for: left, the time spent in applying the matrix-free operator with libCEED; right, the average time per iteration of CG. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

making them more analogous to the fused GPU backends in this regard, despite having separate kernels for each operation like the non-fused GPU backends.

Another possibility is that we would want to perform some of the computational kernels in a lower precision, especially in the case of non-tensor basis functions or a particularly expensive QFunction. Figures 29 and 30 demonstrate the percentage of GPU time spent in each component of the diffusion operator application for the non-fused `magma-det` backend and the speedup achieved by performing these components in single precision, respectively. The QFunction becomes the most expensive portion of the operator application for the large problem sizes of most interest to the GPU backends, though the basis computations increase in relative cost for higher orders of basis functions. The relative cost of each portion of the operator will depend, of course, on the operator itself, especially as libCEED allows users to define their own QFunctions. The relative cost will also depend on the hardware in question, such that CPU and GPU backends are not expected to benefit uniformly from performing a particular component of the operator in lower precision. We also emphasize that these results are for a tensor-product basis, and the relationship of basis action to QFunction in terms of cost is also dependent on the type of basis. All of this indicates that libCEED's interface for allowing mixed-precision computation needs to be flexible and adaptable – in the spirit of libCEED itself.

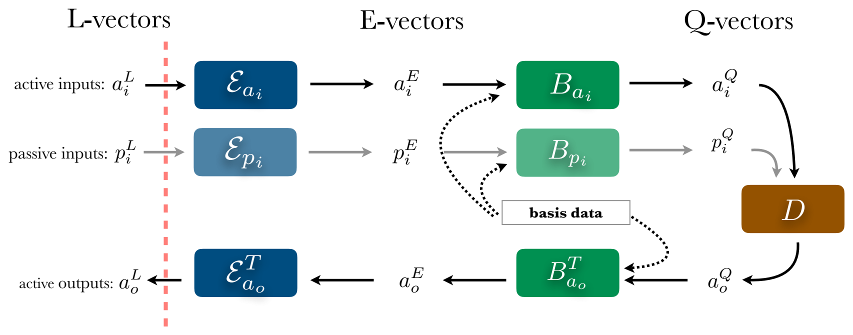


Figure 28: Representation of libCEED operator as data flow between the kernels/functions

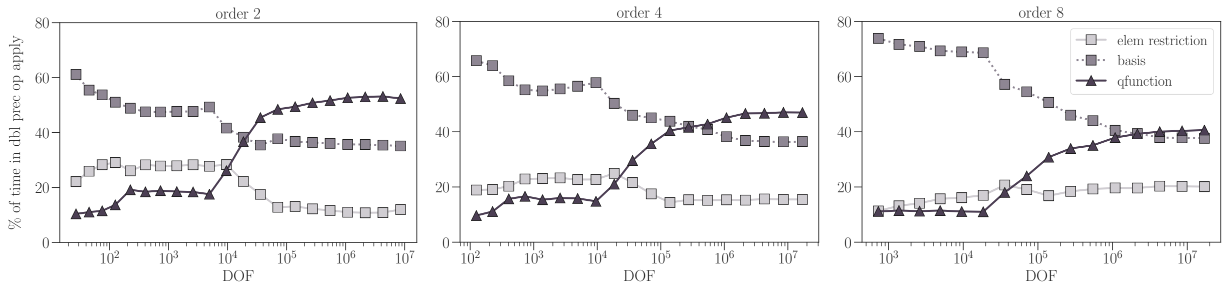


Figure 29: Percentage of time spent in each of the individual components of the libCEED operator for a full-double-precision application of the diffusion operator in BP3 with the `magma-det` backend. Three orders of basis functions are shown. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

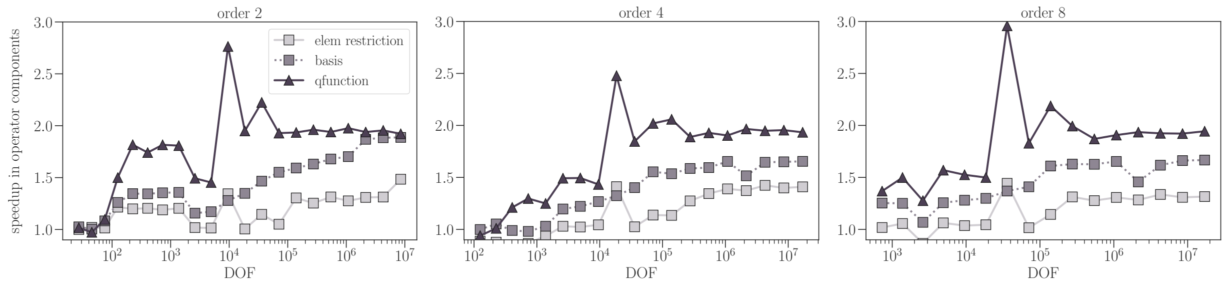


Figure 30: Speedup for each individual component of the `magma-det` backend inside the diffusion benchmark (BP3), for three orders of basis functions. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

To demonstrate initial mixed-precision performance, we will consider two cases that could be considered to reside at different ends of the mixed-precision spectrum: "high-low" and "low-high". In the case of "high-low," the input and output L-Vectors will be in double precision, but they are converted to single precision upon output from the element restriction ($\mathcal{E}$), and all data movement and computation thereafter proceeds in single precision, before finally converting back to double precision in the transpose element restriction ($\mathcal{E}^T$). The "low-high" case is the opposite, where the input and output vectors are in single precision, but all computation and intermediate values are in double precision inside the operator. This represents the case where we only require lower precision in the code immediately outside libCEED, but may want to perform the computation in higher precision to accumulate less round-off error. Note that the basis data shown as "internal" (to libCEED) input to the basis actions in Figure 28 is stored in the "second" precision, meaning the "high-low" operator stores this data in single precision and reads it into shared memory at the beginning of the fused kernel in single precision, in contrast to the "low-high" case.

In Figures 31 and 32 we see the speedup in the libCEED operator application and average time per iteration, respectively, for the fused `cuda-gen` backend in the Ginkgo-libCEED BP3 implementation. The all-float case is also shown for comparison. We note that the precision of the CG solver will match that of the input/output L-Vectors, which means the "low-high" case benefits from performing the CG vector operations – the second-costliest part of the solve, as shown in Figs. 24 and 25 – in single precision.

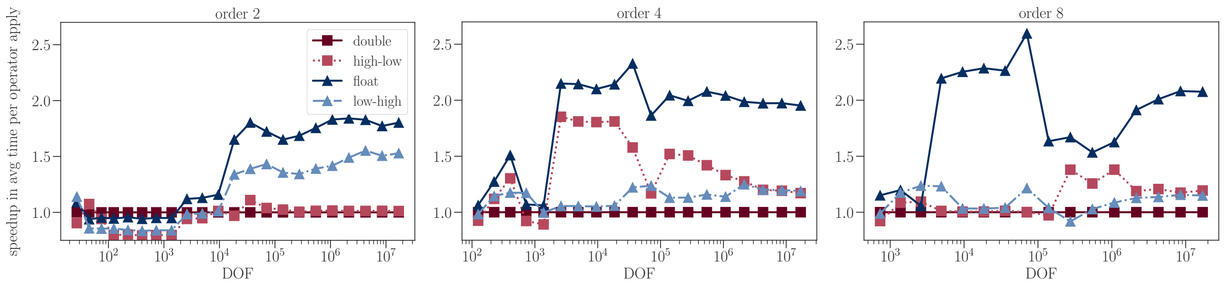


Figure 31: Speedup compared to full double precision for two mixed-precision modes, "low-high" and "high-low," for three orders of basis functions. All-single precision is also shown for comparision, with the double precision line = 1 as reference. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

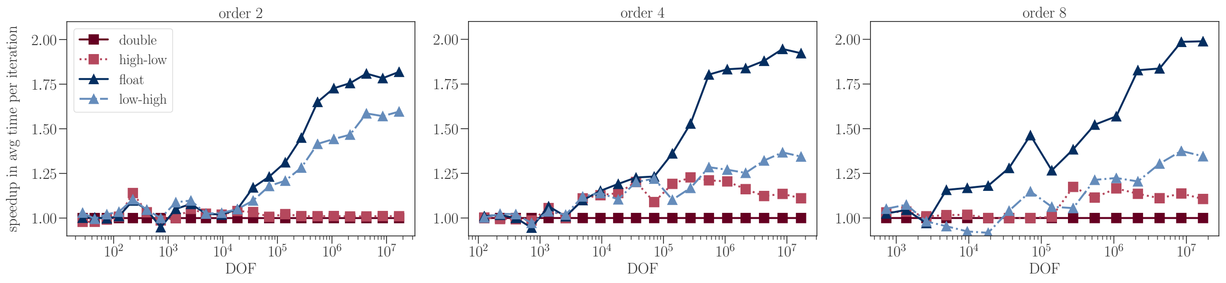


Figure 32: Speedup in average time per iteration in CG, compared to full double precision for two mixed-precision modes, "low-high" and "high-low," for three orders of basis functions. All-single precision is also shown for comparison, with the double precision line = 1 as reference. Results obtained on an NVIDIA V100 GPU with CUDA 11.0.

When the basis function order is low, as in the left plots for second order basis functions, there is no benefit to "high-low." This is to be expected, since the reading and writing of the L-Vectors from and to main memory is by far the limiting factor, and "high-low" is still using double precision for L-Vectors. The "low-high" case, on the other hand, sees speedup, albeit below that of the all-float case. However, as we increase the order of the basis functions, we do see some benefit from applying the operator in lower precision, with around 20% speedup in operator application for the largest problem sizes. This speedup is enough to retain modest speedup in average time per iteration for the "high-low" case as well, despite the CG computations still taking place in double precision.

Mixed Precision in MAGMA

The MAGMA library supports high-performance linear algebra routines in multiple precisions (FP64, FP32, and some FP16) that can be used as building blocks of mixed-precision solvers in libCEED. This includes various matrix factorizations, BLAS and Batched BLAS, various routines for casting vectors from one precision to another, as well as iterative Krylov solvers and preconditioners.

MAGMA also has a number of mixed-precision solvers that illustrate how the aforementioned building blocks are combined in complete solvers. Algorithm 1 outlines the main idea – a matrix factorization is computed in FP32 precision and used as preconditioner in an iterative refinement process where the residual and the approximate solution updates are computed in FP64 precision. This iteration process, known as mixed precision iterative refinement, has been well studied in the past and is known to converge to FP64 accuracy under minor requirements for the matrix (e.g., not to be too ill conditioned), and the solver/preconditioner does not have to be based on Gaussian Elimination. MAGMA has supported three mixed-precision iterative refinement solvers since 2010 – one for general matrices where the low-precision solver uses LU factorization with partial row pivoting, another for symmetric and positive definite matrices that uses a Cholesky factorization, and a third one for least squares problems using QR factorizations [21]. We have also investigated and shown how the technique works for sparse matrices where the preconditioners use sparse direct (multifrontal or supernodal) factorizations, or where the preconditioner is replaced by another iterative solver (leading to solvers known in the literature as *inner-outer iteration*) [5].

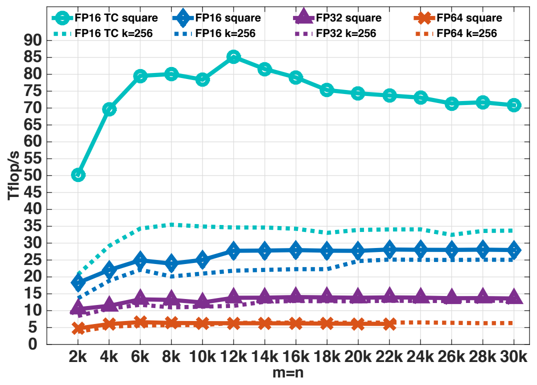
Algorithm 1 Mixed precision, Iterative Refinement $Ax = b$ Solver

```
1:  Compute the LU factorization of A, e.g., $LU = PA$          ▷ (FP32)
2:  Solve $Ly = Pb$                                              ▷ (FP32)
3:  Solve $Ux_0 = y$                                             ▷ (FP32)
4:  for $k = 1, 2, \ldots$ do
5:      $r_k \leftarrow b - Ax_{k-1}$                            ▷ (FP64)
6:      Solve $Ly = Pr_k$                                        ▷ (FP32)
7:      Solve $Uz_k = y$                                         ▷ (FP32)
8:      $x_k \leftarrow x_{k-1} + z_k$                           ▷ (FP64)
9:      Check convergence
10: end for
```

The mixed-precision solvers in MAGMA were recently enhanced to support half-precision Tensor Core (FP16-TC) and we demonstrated that these methods can provide up to 4× speedup and FP64 accuracy compared to the FP64 solvers [11, 10]. These results are illustrated on Figure 33. Figure 33 Left shows the performance of the rank-k GEMMs typically used in the LU factorizations on Nvidia V100 GPUs (providing upper bound for the performance of the factorization), and Figure 33 Right shows the performance acceleration achieved using the new mixed-precision solvers. The condition numbers of the matrices used and the number of iterations to reach FP64 accuracy are also given for the different solvers. The outer iteration in these solvers use GMRES.

The mixed-precision iterative refinement solvers were also ported to HIP to support AMD GPUs. Figure 34 Left shows the performance from the MAGMA xGEMM benchmark in the FP64, FP32, and FP16 arithmetic, respectively, on the MI100 GPUs in Spock. The currently achievable performance on these rank-k GEMMs (using ROCm 4.2) is lower than the theoretical peaks. Also, there are a lot of variations in the performance, e.g., in the range of 10 to 30 TFLOPs for both FP32 and FP16. We observe that the rank-k FP16 GEMMs are about the same performance as the FP32 ones. Because of this, we developed the mixed-precision iterative refinement using just FP32-FP64 arithmetic, shown on Figure 34 Right. The speedup is about two times, as expected. The mixed-precision solver converged to FP64 accuracy from 3 iterations on the smallest problem to 16 iterations for the largest problem, using random matrices.

The mixed-precision solvers and their building blocks for either CUDA or HIP are available through the recent MAGMA 2.6 release.



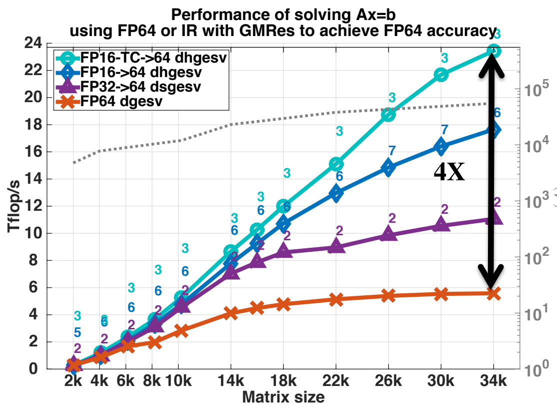
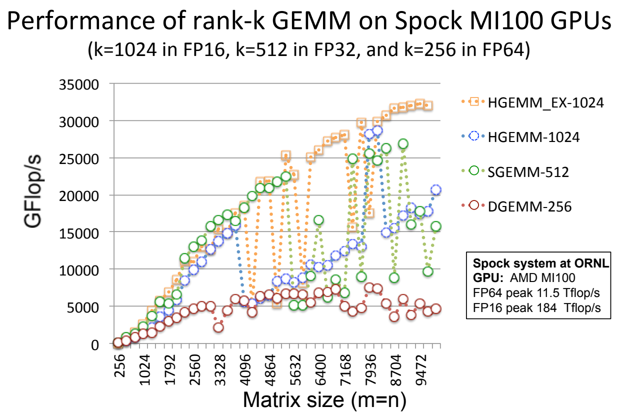


Figure 33: Left: Performance of the rank-k GEMMs typically used in the LU factorizations on Nvidia V100 GPUs. **Right:** Performance comparison (computed as $(\frac{2}{3}n^3 + \frac{3n^2}{2} - \frac{n}{6})$ / time in GFlop/s) of the FP64 solver and the mixed-precision solvers in MAGMA, achieving FP64 accuracy.



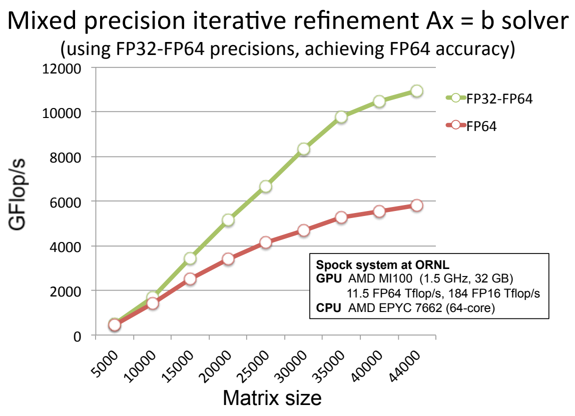


Figure 34: Left: Performance of the rank-k GEMMs typically used in the LU factorizations on GPUs. **Right:** Performance comparison (computed as $(\frac{2}{3}n^3 + \frac{3n^2}{2} - \frac{n}{6})$ / time in GFlop/s) of the FP64 solver and the mixed-precision FP32-FP64 solver in MAGMA, achieving FP64 accuracy. All experiments use ROCm 4.2.

6. SOFTWARE RELEASES

During the milestone period, the CEED team released new versions of 6 of its packages, including major releases of MFEM, NekRS and OCCA.

6.1 MFEM-4.3

Version 4.3 of MFEM was released on July 29, 2021. Some of the new additions in this release are:

- Variable order spaces, serial p- and hp-refinement.
- Low order refined discretizations, solvers and transfer.
- Preconditioners for advection-dominated systems.
- Support for GPU solvers from hypre and PETSc.
- GPU-accelerated mesh optimization algorithms.

- Explicit vectorization for Fujitsu's A64FX ARM microprocessor.
- Support for high-order Lagrange meshes in VTK format.
- New and improved integrations with FMS, Caliper, libCEED, Ginkgo.
- 11 new examples and miniapps.

For more details, see the interactive documentation at `https://mfem.org` and the full CHANGELOG at `https://github.com/mfem/mfem/blob/v4.3/CHANGELOG`.

6.2 NekRS-21.1

Version 21.1 of NekRS was released with new support for flexible GMRES, constant flow rate, time step controller for target CFL, improved runtime statistics, support for ROCm version (>v4.0), nonlinear artificial viscosity methods for scalar transport, FEMSEM preconditioner, low-storage A-orthogonal projection for initial guesses, updated user-defined file (nekrs.upd) for runtime modifications, validated key/value input in parfile, and various bug fixes. These improvements add to earlier NekRS features that include: k-τ RANS models, low-Mach formulation, run-time averaging for turbulence statistics, ALE with elliptic solvers for mesh motion, conjugate heat transfer, full stress formulation for variable-property support, and recycling boundary conditions.

6.3 FMS-0.2

Version 0.2 of FMS, CEED's high-order field and mesh specification was released on September 10th, 2021. Some of the new additions in this release are:

- Lightweight API to represent general finite element meshes + fields
- I/O in ASCII and Conduit binary format
- FMS visualization support in VisIt v3.2
- Common, easy to use framework

For more details, see the CHANGELOG at `https://github.com/CEED/FMS/blob/v0.2/CHANGELOG`.

6.4 MAGMA-2.6.1

MAGMA 2.6.1 was released on July 13, 2021. This is a major release that added HIP support for AMD GPUs (former hipMAGMA) as part of MAGMA. Support was added for MAGMA Sparse and mixed-precision solvers on AMD GPUs, as well as numerous performance improvements for AMD GPUs. See `https://icl.utk.edu/magma/software/` for download and further details.

6.5 PUMI-2.2.6

PUMI version 2.2.6 was released and includes improved high order mesh adaptation and field support, support for Simmetrix SimModSuite 15.0-210501, support for GCC 11, and a bug fix for GCC 10. Additional details are available at `https://github.com/SCOREC/core/issues/341`.

6.6 OCCA-1.2.0

This section provides detailed review of the new features, capabilities, bug fixes, and other changes that were part of the OCCA v1.2.0 release. In addtion to the features described below, see also Section 3.4 for the new DPC++ Backend in OCCA which was part of this release.

The release includes contributions from the main architect of OCCA David Medina as part of the CEED project and also by members of the OCCA and CEED development community including: Noel Chalmers (AMD Research), Damon McDougall (AMD Research), Anoop Madhusoodhanan Prabha (Intel), Cedric Andreolli (Intel), Kris Rowe (ALCF), Phillipe Thierry (Intel), Saumil Patel (ALCF), Jed Brown (UColorado at Boulder), Stefan Frijters (Shell), Malachi Timothy Phillips (UIUC). Text from the constituent pull requests making up the OCCA v1.2.0 release have been adapted for inclusion in this report format.

occa::forLoop and inlined kernels. The OCCA v1.2.0 release includes several new experimental features to the OCCA library. They are still in the experimental stage, mainly due to performance reasons. We found an initial approach to enabling inlined lambdas and are experimenting with them to explore the trade offs between capability and performance. The OCCA v1.2.0 release includes increased support for inlining OCCA kernels in a single-source fashion whereas in previous non experimental versions this was very much an experimental feature. As a basic example of inlining an OCCA kernel in single-source code we generate here a for-loop that loops through $[0, N)$ tiled by `tileSize` with the following syntax:

```
occa::forLoop()
  .tile({N, tileSize})
  .run(scope, OCCA_FUNCTION([=](const int index) -> void {
    // ...
  }));
```

The tiling can alternatively be done manually through the inclusion of `.inner` and `.outer` member function calls that can be for instance translated into for CUDA threads and thread-blocks respectively by the CUDA backend.

```
occa::forLoop()
  .outer(occa::range(0, N, tileSize))
  .inner(tileSize)
  .run(scope, OCCA_FUNCTION([=](const int outerIndex, const int innerIndex) -> void {
    const int index = innerIndex + (tileSize * innerIndex);
    // ...
  }));
```

Inlining idexing & multi-dimension kernels. The OCCA v1.2.0 release enables inlined single source kernels to use multilevel parallelism with inner and outer loops. Here we give an example where an index array is passed rather than a simple `occa::range`. Additionally, this `@inner` loop has 2 dimensions so the expected `OCCA_FUNCTION` should be supplied with an `int2` variable for the inner indices

```
occa::array<int> indices;
// ...
occa::forLoop()
  .outer(indices)
  .inner(X, Y)
  .run(scope, OCCA_FUNCTION([=](const int outerIndex, const int2 innerIndex) -> void {
    // ...
  }));
```

`occa::array` and Functional Programming. The OCCA v1.2.0 release provides a wrapper on `occa::memory` objects which includes type information and contains some of the core map and reduce functional methods.

```
const double dynamicValue = 10;
const double compileTimeValue = 100;

occa::scope scope({
  // Passed as arguments
    {"dynamicValue", dynamicValue}
  }, {
  // Passed as compile-time #defines
    {"defines/compileTimeValue", compileTimeValue}
});

occa::array<double> doubleValues = (
  values.map(OCCA_FUNCTION(scope, [](int value) -> double {
    return compileTimeValue + (dynamicValue * value);
  }));
);
```

The release includes a helper method `occa::range` which implements most of the `occa::array` methods but can be used without allocating data before iteration. It is useful if there is no specific input/output but still need to call a generic map or reduce function.

```
// Iterates through [0, 1, 2]
occa::range(3).map(...);

// Iterates through [3, 4, 5]
occa::range(3, 6).map(...);

// Iterates through [6, 5, 4]
occa::range(6, 3).map(...);

// Iterates through [0, 2, 4]
occa::range(0, 6, 2).map(...);

// Iterates through [6, 4, 2]
occa::range(6, 0, -2).map(...);

// No-op since there isn't anything to iterate through
occa::range(6, 0, 1).map(...);
```

The `occa::array` class now supports the following functionality:

- Core methods: `forEach`, `mapTo`, `map`, `reduce`
- Reduction types: `every`, `max`, `min`, `some`
- Re-indexing: `reverse`, `shiftLeft`, `shiftRight`
- Utilities: `cast`, `clamp`, `clampMax`, `clampMin`, `concat`, `dot`, `fill`, `slice`
- Search: `findIndex`, `find`, `includes`, `indexOf`, `lastIndexOf`

Atomics. The OCCA v1.2.0 release includes new experimental support for the natural inclusion of atomic operations with the OCCA kernel language (OKL) [2] The initial support for the OCCA GPU backend modes (HIP, CUDA, OpenCL) do not yet have general atomics implemented but do have the following basic updates:

```
  @atomic value += update;
  @atomic value -= update;
  @atomic value &= update;
  @atomic value |= update;
  @atomic value ^= update;
```

An example of an inlined atomic expression:

```
@atomic *ptr += value;
```

Alternatively code blocks can be made atomic (where possible) as

```
@atomic {
  *ptr += value;
}
```

and even generic @atomic blocks are also possible

```
@atomic {
  *ptr  += value;
  *ptr2 += value2;
}
```

Refactor of the OCCA code transformation infrastructure. In OCCA v1.2.0 the way statement and expression code transformations are performed have been fully rewritten. A functional `occa::lang::array` class was introduced to help with statement and expression iteration and transformation. Additionally the `occa::lang::expr` class helps create expressions easily without having to worry about pointers or underlying node objects.

[2] Although `@atomic` is fully available for Serial and OpenMP modes there is probably still room for improvement in the OpenMP implementation.

Improved handling of OCCA memory slicing. OCCA v1.2.0 includes changes to the internal handling of how references to backend memory allocations are counted so they can be more reliably destroyed, and prevent memory access violations via invalidated pointers. These changes are entirely internal to OCCA and do not change the public API, or alter current behavior of OCCA applications. Summary of issue: Prior to v1.2.0 `modeMemory_t` objects were not track whether if they were created by slicing another `modeMemory_t` object. Consequently, only the original `modeMemory_t` object can free the underlying device memory allocation, and would do so even if there still existed `modeMemory_t` objects referencing the device memory. This resulted in dangling references. As an example

```
{
  occa::memory o_a;
  {
    occa::memory o_A = device.malloc<float>(2*entries); //Create a new buffer
    o_a = o_A + entries; //  o_a is a slice into o_A
  } //o_A is destroyed here because it's reference counting is separate from o_a.
    // The underlying device memory is also destroyed

  // o_a is now invalid, but doesn't know the original device memory is free'd

  // Using o_a will result in an access violation
  myKernel(entries, o_a); //Fault

} //o_a destroyed here, but since it's not the origin, there's no double free
```

Noel Chalmers contributed a pull request to address this behavior. The new buffer objects wrap device memory allocations and track their `modeMemory_t` references, created via slicing. The buffer and wrapped device allocation are destroyed once their reference count reaches zero.

7. OTHER PROJECT ACTIVITIES

7.1 CEED Annual Meeting

The CEED project held its fifth annual meeting August 3-4, 2021 with participation from 97 researchers from 8 national labs, 20 universities and 8 companies.

The goal of the meeting was to report on the progress in the center, deepen existing and establish new connections with ECP hardware vendors, ECP software technologies projects and other collaborators, plan project activities and brainstorm/work as a group to make technical progress. In addition to gathering together many of the CEED researchers, the meeting included representatives of the ECP management, hardware vendors, software technology and other interested projects.

7.2 MFEM Community Workshop

The MFEM team at LLNL is hosting the inaugural MFEM Community Workshop, a one-day virtual workshop designed to bring together users of the MFEM library, application scientists, researchers, students, and others who take an interest in finite elements. The goals of the workshop are to foster collaboration among users and developers of the MFEM, share the latest MFEM features with the broader community, deepen application engagements, and solicit feedback to guide future development directions for the project.

This workshop will be held on October 20, 2021, and has already seen over 150 attendees registered, from 25 countries and 82 institutions. The workshop will include talks both from members of the CEED project, and from outside contributors who have used and extended MFEM in their application domains. This workshop is intended to be an annual occurrence, with both virtual and in-person components in the future.

7.3 Efficient Sub-Domain Fields and MFEM Integration in Omega_h

APIs were added to Omega_h to support reverse classification queries (e.g., what mesh faces are classified on a given set of geometric model faces) and associate user data with the mesh entities returned by those queries. These APIs address the ability to define Omega_h field structures on specific groups of mesh entities of a selected dimension without the need to store a field on all the mesh entities of that dimension. A common case is a varying natural boundary condition applied to a model face. Now that field information can be associated with the mesh entities classified on that model face without the need to define a field overt all mesh faces in the entire finite element mesh. Having such field information in an Omega_h structure is needed to support proper field transfer during conforming mesh adaptation.

In addition, initial in-memory support for conforming mesh adaptive loops using MFEM and Omega_h on simplex meshes with low-order fields was implemented. This includes conversion from the MFEM partitioned mesh, and fields defined on it, to the Omega_h representation before mesh adaptation and conversion back to MFEM after adaptation.

7.4 ECP GitLab CI at OLCF

Following instructions given at the ECP annual meeting, and with support from ORNL, GitLab CI running on the OLCF Ascent system (a small version Summit) was setup for Omega_h.

8. CONCLUSION

In this milestone we ported the CEED software stack, including Nek, MFEM and libCEED to the Frontier and Aurora early access hardware, and worked on optimizing the performance on AMD and Intel GPUs.

As part of this milestone, we also released new versions of six of the CEED software packages, including major releases of MFEM, NekRS and OCCA, organized the fifth CEED Annual meeting and continued to improve the performance of high-order methods on a variety of hardware including V100, MI100, A100 and A64FX systems.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the 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—to support the nation's exascale computing imperative.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL-TR-827178.

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. Support was also given by the Frontier Center of Excellence.

REFERENCES

[1] Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software*, 5(52):2260, 2020.

[2] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. Ginkgo: A modern linear operator algebra framework for high performance computing, 2020.

[3] R. Balakrishnan, R. Kotamarthi, and P. Fischer. Large eddy simulation of isothermal and non-isothermal turbulent flows in ventilated classrooms. In *ECOFTAC 13th Conference on Engineering, Turbulence, Modelling and Measurements*, Rhodes, Greece, 2021.

[4] Jed Brown, Veselin Dobrev, Som Dutta, Paul Fischer, Kazem Kamran, Tzanio Kolev, David Medina, Misun Min, Thilina Ratnayaka, Mark Shephard, Cameron Smith, and Jeremy Thompson. CEED ECP Milestone Report: Propose high-order mesh/data format, June, 2018.

[5] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4), July 2008.

[6] N. Chalmers and T. Warburton. streamParanumal: Portable CEED benchmark streaming tests, 2020. Release 1.0.0.

[7] Veselin Dobrev, Jack Dongarra, Jed Brown, Paul Fischer, Azzam Haidar, Ian Karlin, Tzanio Kolev, Misun Min, Tim Moon, Thilina Ratnayaka, Stanimire Tomov, and Vladimir Tomov. ECP Milestone Report CEED-MS6: Identify initial kernels, bake-off problems (benchmarks) and miniapps, July, 2017.

[8] A. Fabregat, F. Gisbert, A. Vernet, S. Dutta, K. Mittal, and J. Pallaré. Direct numerical simulation of the turbulent flow generated during a violent expiratory event. *physflu*, 33:035122, 2021.

[9] Dominik Göddeke, Robert Strzodka, and Stefan Turek. *Accelerating double precision FEM simulations with GPUs.* Univ., 2005.

[10] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas Higham. Mixed-precision iterative refinement using tensor cores on gpus to accelerate solution of linear systems. *Royal Society of London. Proceedings A. Mathematical, Physical and Engineering Sciences*, September 2020.

[11] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[12] David S Kershaw. Differencing of the diffusion equation in lagrangian hydrodynamic codes. *Journal of Computational Physics*, 39(2):375–395, 1981.

[13] Tzanio Kolev, Paul Fischer, Anthony P. Austin, Andrew T. Barker, Natalie Beams, Jed Brown, Jean-Sylvain Camier, Noel Chalmers, Veselin Dobrev, Yohann Dudouit, Leila Ghaffari, Stefan Kerkemeier, Yu-Hsiang Lan, Elia Merzari, Misun Min, Will Pazner, Thilina Rathnayake, Mark S. Shephard, Morteza H. Siboni, Cameron W. Smith, Jeremy L. Thompson, Stanimire Tomov, and Tim Warburton. ECP Milestone Report CEED-MS36: High-order algorithmic developments and optimizations for large-scale GPU-accelerated simulations, March 31, 2021.

[14] Kai Liu, Majid Allahyari, Jorge S Salinas, Nadim Zgheib, and S Balachandar. Peering inside a cough or sneeze to explain enhanced airborne transmission under dry weather. *Scientific Reports*, 11, 1:1–9, 2021.

[15] J. Malm, P. Schlatter, P.F. Fischer, and D.S. Henningson. Stabilization of the spectral-element method in convection dominated flows by recovery of skew symmetry. *J. Sci. Comp.*, 57:254–277, 2013.

[16] B.E. Merrill, Y.T. Peet, P.F. Fischer, and J.W. Lottes. A spectrally accurate method for overlapping grid solution of incompressible Navier-Stokes equations. *J. Comput. Phys.*, 307:60–93, 2016.

[17] Ketan Mittal, Som Dutta, and Paul Fischer. Nonconforming Schwarz-spectral element methods for incompressible flow. *Computers and Fluids*, 191, 2019.

[18] Ketan Mittal, Som Dutta, and Paul Fischer. Multirate time-stepping for the incompressible Navier-Stokes equations in overlapping grids. *jcp*, 437:110335, 2020.

[19] Robert Strzodka and Dominik Göddeke. Mixed precision methods for convergent iterative schemes. *EDGE*, 6:23–24, 2006.

[20] Kasia Świrydowicz, Noel Chalmers, Ali Karakus, and Timothy Warburton. Acceleration of tensor-product operations for high-order finite element methods. *arXiv preprint arXiv:1711.00903*, 2017.

[21] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[22] David Zwick and S. Balachandar. A scalable Euler–Lagrange approach for multiphase flow simulation on spectral elements. *IJHPCA*, 34, 3:316–339, 2020.