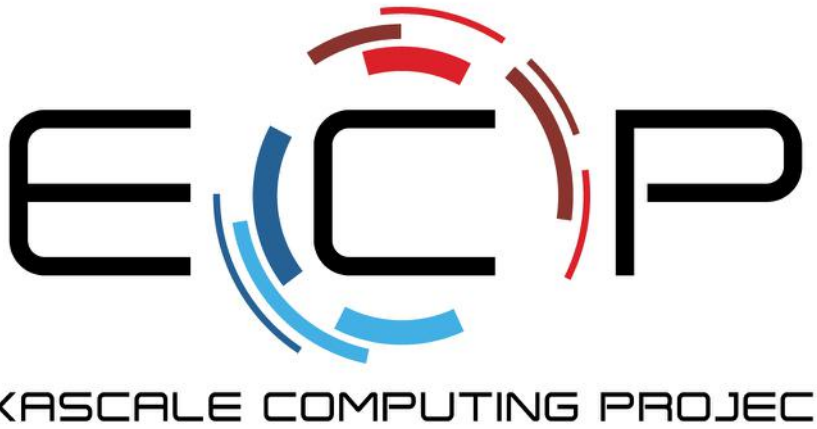


ECP Milestone Report

High-order algorithmic developments and optimizations for more robust exascale applications

WBS 2.2.6.06, Milestone CEED-MS38

Tzanio Kolev
Paul Fischer
Ahmad Abdelfattah
Natalie Beams
Jed Brown
Jean-Sylvain Camier
Robert Carson
Noel Chalmers
Veselin Dobrev
Yohann Dudouit
Leila Ghaffari
Aditya Y. Joshi
Stefan Kerkemeier
Yu-Hsiang Lan
Damon McDougall
David Medina
Misun Min
Abhishek Mishra
Will Pazner
Malachi Phillips
Thilina Ratnayaka
Mark S. Shephard
Morteza H. Siboni
Cameron W. Smith
Jeremy L. Thompson
Ananias Tomboulides
Stanimire Tomov
Vladimir Tomov
Tim Warburton

March 31, 2022

ECP Milestone Report
High-order algorithmic developments and optimizations for more robust exascale applications
WBS 2.2.6.06, Milestone CEED-MS38

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

March 31, 2022

ECP Milestone Report
High-order algorithmic developments and optimizations for more robust exascale applications
WBS 2.2.6.06, Milestone CEED-MS38

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	May 3, 2022	Original	

EXECUTIVE SUMMARY

The goal of this milestone was to improve the high-order software ecosystem for CEED-enabled ECP applications by making progress on efficient matrix-free kernels targeting forthcoming ECP architectures. Kernels include preconditioning, matrix-free monotonicity, gradients of high-order operators, matrix assembly for low-order methods and other operations that are critical for applications.

As part of this milestone, we also released the next version of the CEED software stack, CEED-5.0, released hipBone, a new performance-portable GPU-accelerated C++ version of the NekBone benchmark, improved meshing and application performance, developed methods for kernel fusion in MFEM, performed extensive studies on the Crusher MI250X system, and worked on number of additional software and algorithmic improvements.

The specific tasks addressed in this milestone were as follows.

- CEED-T17 (ADCD04-87): Matrix-free algorithms: monotonicity/positivity, matrix-free gradients (AD), solvers
- CEED-T18 (ADCD04-88): Support for matrix assembly and low-order methods optimizations
- CEED-T19 (ADCD04-89): GPU porting of additional CEED components based on profiling/application needs
- CEED-T20 (ADCD04-90): Public release of CEED-5.0

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The goal of this milestone was to improve the high-order software ecosystem for CEED-enabled ECP applications by making progress on efficient matrix-free kernels targeting forthcoming ECP architectures. Kernels include preconditioning, matrix-free monotonicity, gradients of high-order operators, matrix assembly for low-order methods and other operations that are critical for applications.

As part of this milestone, we also released the next version of the CEED software stack, CEED-5.0, released hipBone, a new performance-portable GPU-accelerated C++ version of the NekBone benchmark, improved meshing and application performance, developed methods for kernel fusion in MFEM, performed extensive studies on the Crusher MI250X system, and worked on number of additional software and algorithmic improvements.

2. NEW AND IMPROVED CEED SOFTWARE

2.1 The CEED 5.0 software distribution

The CEED distribution is a collection of software packages that can be integrated together to enable efficient discretizations in a variety of high-order applications on unstructured grids. CEED is using the Spack package manager for compatible building and installation of these software components, and also provides Docker images for containerized deployment and continuous integration.

Version CEED-5.0 released on March 31, 2022 contains 12 integrated packages ranging from low-level modular libraries to applications, provided together within the CEED meta-package. We list these packages below, listing some highlights for each of them since the last release. Note that many of these packages have had more than one feature release since CEED-4.0, some of which were reported in [38]; the list below reflects only the latest releases since CEED-4.0.

libCEED-0.10 $\mapsto$ Version 0.10.1 of libCEED was released on March 31, 2022. Some of the new additions in this release are: support for single precision; capability to assemble operators on GPUs; performance enhancements; various interface and error checking improvements; mini-app improvements. For more details, visit http://libceed.org.

MFEM-4.4 $\mapsto$ Version 4.4 of MFEM was released on March 21, 2022. Some of the new additions in this release are: support for AMG solvers on AMD GPUs; new hr-adaptivity and interface fitting of high-order meshes; high order Nedelec elements on tetrahedral meshes without reordering; GPU-enabled partial and element assembly for DG on AMR meshes; initial support for meshes with pyramidal elements; arbitrary order Nedelec and Raviart-Thomas elements on prisms; new and improved integrations with libCEED, CoDiPack, ParELAG; documentation for all releases at https://docs.mfem.org; 9 new examples and miniapps, including AD and Jupyter examples. For more details, visit http://mfem.org.

MAGMA-2.6.2 $\mapsto$ In this release from March 15, 2022 we added a number of batched routines and optimizations, most notably including variable-size batched routines for LU factorizations with partial pivoting and variable-size triangular matrix solvers. A number of bug fixes and performance tuning for AMD GPUs were added, as well as more MAGMA interfaces to hipBLAS routines. Interfaces to FP16 GEMM and mixed-precision FP32-FP16 GEMM and HGEMM benchmark were added. See the MAGMA 2.6.2 release notes for further details.

PETSc-3.17 $\mapsto$ Version 3.17 of PETSc was released on March 31, 2022. Some of the new additions in this release are: pure GPU parallel sparse matrix assembly using split-phase COO interface; can also assemble Hypre format via run-time options; CUDA and HIP algebraic multigrid setup and solves (some via Kokkos) and GPU support for Hypre; batch solvers using Kokkos; automatic distributed meshing for extruded triply periodic surfaces; many features related to unstructured mesh adaptivity, high order coordinate spaces, and performance.

PUMI-2.2.7 $\mapsto$ Version 2.2.7, released on March 14, 2022, adds support for Gmsh physical entity types for use with MFEM analysis requiring multiple material types (MFEM Issue #2726), a simplified MeshAdapt API that helps users avoid some common configuration mistakes, GitHub Actions CI testing, and support for XSDk v0.6.0. See the release notes for details.

Omega_h-10.1.0 ↦ Version 10.1.0 was released on March 15, 2022 and supports single-process meshes with mixed topological elements (tet, hex, prism, wedge, pyramid), retrieving a list of mesh entities classified on the closure of a given geometric model entity and storing fields on those mesh entities, and loading Simmetrix meshes.

Ratel ↦ Version 0.1.0 of Ratel was released on March 31, 2022. This is the initial release of the Ratel solid mechanics library, which is based upon libCEED and PETSc with support for efficient high-order elements and CUDA and ROCm GPUs. Ratel provides several examples for direct use as simulation drivers and as starters for more complete applications and workflows, to include static and quasistatic elasticity with material models for Neo-Hookean and Mooney-Rivlin hyperelastic materials at finite strain.

The CEED-5.0 distribution also contains stable packages that have not seen a new release since CEED-4.0, namely Laghos-3.0, Nek5000-19.0, Nekbone-17.0, NekCEM-c8db04b, and Remhos-1.0. The distribution is tested on several platforms including Linux (RHEL and Ubuntu), Mac OSX, and the representative heterogeneous HPC machines OLCF Summit, LLNL Lassen, and LLNL Corona. See `https://ceed.exascaleproject.org/ceed-5.0/` for details about native installation using Spack and containerized use via Docker, Singularity, and Shifter.

2.2 Omega_h developments to support MFEM

Supporting MFEM high order, conforming, adaptive analysis on unstructured meshes running on leadership class systems requires a GPU accelerated, distributed memory, mesh library.

PUMI, the Parallel Unstructured Mesh Infrastructure, provides high order conforming mesh adaptation on distributed memory systems [58, 34, 33] but is not a viable candidate for GPU acceleration given that the data structures and algorithms are setup for sequential, fine-grained modification operations. To meet the MFEM needs, Omega_h [35, 32] is being integrated with MFEM and extended to support high-order adaptation, mixed topological mesh entity types, and parametric geometric model queries on the GPU.

Omega_h distributed memory parallelism is through MPI and GPU acceleration uses either Kokkos and Thrust, CUDA and Thrust, or HIP and ROCThrust. All mesh data resides on the GPU and inter-process communications can optionally use GPU aware MPI to reduce CPU-GPU data movement. Mesh topology, and data associated with mesh entities, are stored in immutable GPU arrays. Mesh topology modification operations are evaluated in large batches identified by a maximal independent set operation after a layer of ghost elements is created to satisfy data dependencies at the mesh partition boundaries. Unlike PUMI's mutable mesh data arrays and incremental modification support, Omega_h performs the selected set of mesh modifications by rebuilding the entire mesh topology on a given process to effectively utilize the concurrency provided by GPUs.

Omega_h GPU accelerated extensions towards high order adaptive MFEM adaptive analysis include support for mesh instances combining simplex and non-simplex element topologies, integration with the EGADSlite CUDA port for improving geometric approximation, and a demonstration of the MFEM-Omega_h in-memory integration for an adaptive electrostatic volta analysis. Details of the design, implementation, and testing of these capabilities, are discussed in the remainder of this sub-section.

Mixed Mesh Representation. The original design and implementation of Omega_h [35, 31] was limited to simplex elements (triangles and tetrahedra). It is well know that the most effective application of high-order methods is with tensor product elements (quadrilaterals and hexahedron), In addition, there is an increasing capability of automatic mesh generation procedures to be able to create hexahedral dominate meshes (see Figure 1). To take advantage of these capabilities, efforts are underway to support mixed meshes for the standard set of finite element topologies (triangles, quadrilaterals, tetrahedra, wedges, pyramids and hexahedra) within Omega_h. The efforts to this point have focused on the extending the on-process, GPU performant, data structures. (The interprocess mesh links and mesh migration procedures used to support the distributed meshes are yet to be updated to account for the mixed mesh extensions.)

Omega_h employs a two-dimensional array that serves as the lookup table for adjacencies. To support mixed meshes the original 4x4 look-up table used to define simplex elements was extended to an 8x8 table that addresses the eight unique topological entities of vertex for the single 0-dimension entity, edge for the

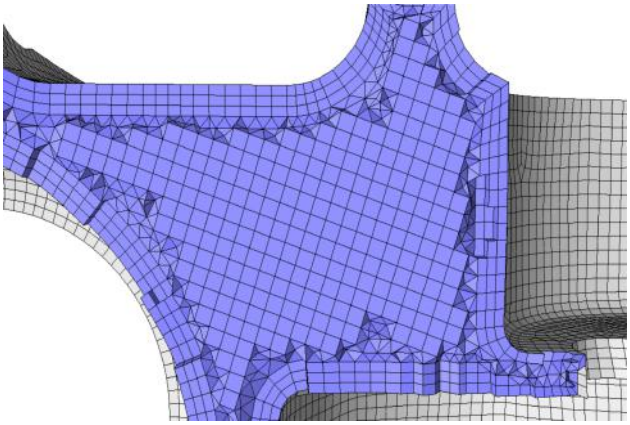


Figure 1: An automatically generated hexahedral dominated mixed mesh.

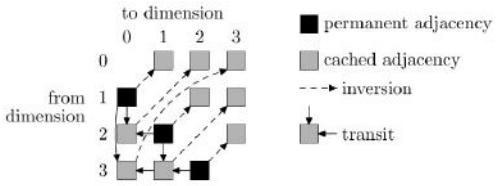


Figure 2: Mesh adjacency storage in Omega_h as per [35]

single 1-dimensional entity, triangle and quadrilateral for the two 2-D entities, and tetrahedron hexahedron, wedge and pyramid for the four 3-D entities. Each entry in the table contains an adjacency graph structure. Figure 2, shows the original 4x4 table used for simplex elements. It relies on the two arguments of 'from' dimension and 'to' dimension. The same concept holds for the mixed mesh 8x8 table, Figure 3, however, there is the added complexity of having two different face topologies and four different region topologies.

As is the case for simplex meshes, the edge-to-vertex downward adjacency for mixed meshes requires a single position of the lookup table. However, in the case of mixed meshes the face-to-edge downward adjacency graph is separated by topology type of the face, namely triangles-to-edge, and quadrilaterals-to-edge. In addition, distinct region-to-face downward adjacency graphs are stored for each region type. The wedge and pyramid require specific consideration since the face downward adjacency includes both triangles and quadrilaterals. Therefore, we store the region-to-face downward adjacency in two different graphs for each type of downward entity. Figure 3 shows a representation of a mixed mesh using one level adjacency graphs. In addition to the one level adjacency graphs, we can also derive adjacency information between any two of the eight mesh entities, and store it in the extended lookup table. Figure 4 shows the adjacency information

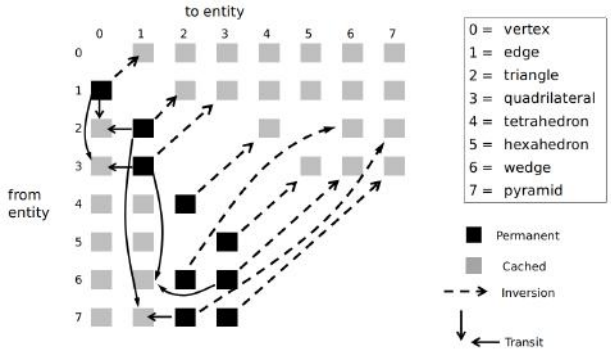


Figure 3: Overview of adjacency storage for mixed mesh

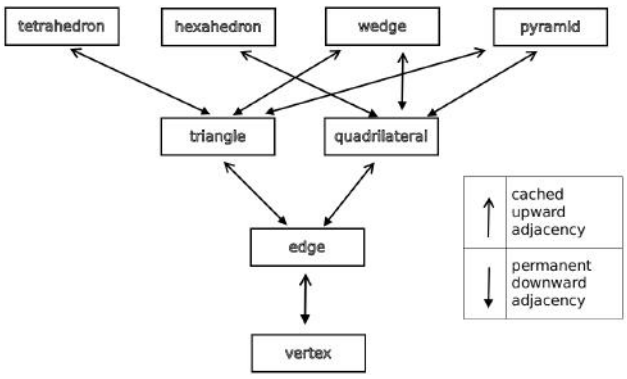


Figure 4: One level representation of a mixed mesh

in a one-level adjacency with both downward and upward adjacencies. It can be noted that each individual downward adjacency graph has a constant downward degree which is taken specific advantage of in the mesh storage.

The permanent downward adjacencies depicted in Figure 3 are populated in the adjacency lookup table during mesh creation in Omega_h. The downward adjacency graphs are used to construct the one level upward adjacencies via a traversal of the mesh. By having a full representation in which each mesh entity is represented we can associate fields, including element shape information with each of the mesh entities.

Improving Geometric Approximation. Improving the meshes approximation of the geometric model during mesh adaptation is critical to ensure that the domain of the problem being solved is the correct one. Towards this, methods to reliably reposition mesh vertices onto the geometric model boundary after mesh refinement via vertex motion and topological modifications have been developed on CPU based systems by relying on parametric queries to a CAD kernel [43]. The closed source Parasolid [64] and ACIS [13], and the open source OpenCASCADE [61], modeling kernels support these queries. EGADS, the Engineering Geometry Aircraft Design System, simplifies and hardens the OpenCASCADE interface and functionality [28]. Use of EGADS by engineering and simulation applications that require read-only access to the CAD, and want/need to avoid the dependency on OpenCASCADE, is provided by EGADSlite [27]. An experimental and undocumented port of EGADSlite to CUDA for execution on NVIDIA GPUs is provided with the EGADSlite release and can support the necessary parametric model queries needed by Omega_h on the GPU.

Leveraging EGADSlite-CUDA in Omega_h required building and testing EGADSlite with CUDA enabled, documenting that process, and then modifying Omega_h to call the CUDA APIs on the GPU. Building and testing EGADSlite-CUDA involved fixing compiler errors and a logic error in the non-recursive implementation of parametric coordinate evaluation, adding tests for GPU execution, and adding a CMake build system that exports configuration files for downstream libraries [6]. Using EGADSlite-CUDA in Omega_h required CMake build system changes to support separable compilation required by EGADSlite-CUDA and porting the routines that call existing EGADS APIs to query geometric model topology information and perform parametric evaluation to kernels that execute on the GPU.

The EGADSLite-CUDA integration in Omega_h was tested using the cone-cone test case from the Unstructured Grid Working Group [77]. The geometric model is a conical section created from the intersection of a box with two nested cones sharing an axis where the inner cone is subtracted from the outer. Figure 5 depicts the initial coarse mesh on the left, the adapted mesh without improvement to the geometric approximation in the middle, and the refined mesh created using queries to EGADSlite-CUDA on the GPU during Omega_h mesh adaptation on the right.

MFEM-Omega_h Adaptive Analysis. The electrostatic inclusion problem is used to demonstrate the in-memory adaptive simulation workflow in MFEM+Omega_h. In the inclusion problem a dielectric inclusion (or a set of inclusions) is embedded into a substrate of a different material. It is well known that for ellipsoidal shapes of the inclusion one can extract an representative volume element for which the Dirichlet boundary conditions are consistent with a uniform background field, the solution inside the inclusion is uniform and the solution outside the inclusion resembles the electric field of a dipole added to the uniform background

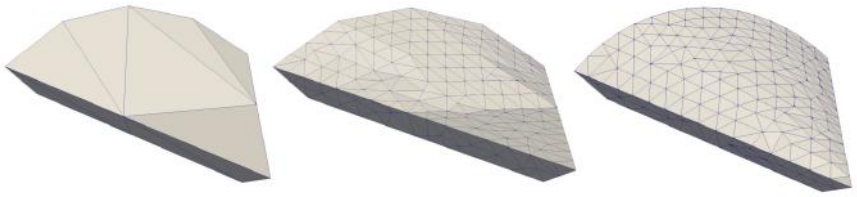


Figure 5: (left to right) The initial cone-cone mesh, the Omega_h adaptively refined mesh without improved geometric approximation, and the refined mesh created using queries to EGADSlite-CUDA on the GPU during Omega_h mesh adaptation

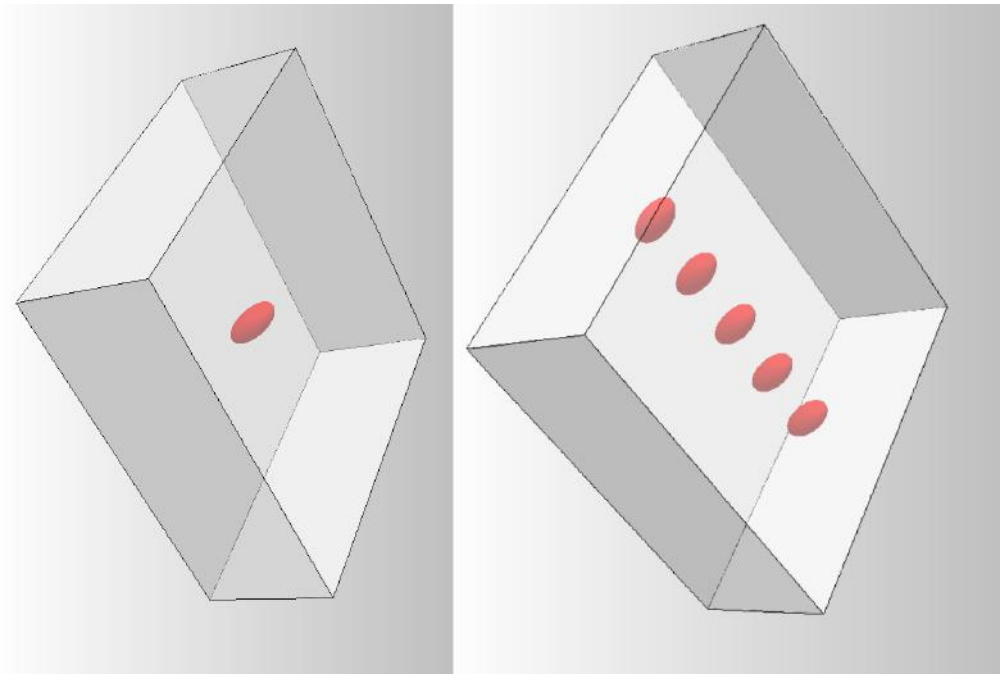


Figure 6: The simulation domain for the example test case; (left) the single inclusion problem and (right) the multi inclusion problem. For both cases the substrate (phase 1) occupies the space $\Omega = [-0.55, 0.55] \times [-0.55, -0.55] \times [0, 0.4]$ and the ellipsoidal inclusion(s) (phase 2) are located in the middle of the substrate. The substrate (phase 1) has dielectric permittivity ϵ_1 and inclusion(s) (phase 2) has dialectic permittivity ϵ_2.

field, Figure 6.

The governing equations and boundary conditions for the problem are as follows

$$\begin{aligned} \nabla \cdot (\epsilon(x) \nabla \phi) &= 0 \quad \text{for} \quad x \in \Omega \\ \phi &= -E_* \cdot x \quad \text{for} \quad x \in \partial\Omega \end{aligned} \tag{1}$$

where ϕ is a scalar field representing the potential and E_* is a prescribed (constant) vector denoting the background uniform electric field.

The inclusion problem is defined using the MFEM Volta mini-app as the starting point with modifications for the domain, governing equations, and boundary conditions described above. The rectangular domain with elliptical inclusions is subjected to Dirichlet boundary conditions consistent with a uniform electric field. After MFEM computes a solution on the nearly uniform initial mesh, element errors are computed with its Zienkiewicz-Zhu error estimator. The desired mesh size field is then computed from the errors which drives

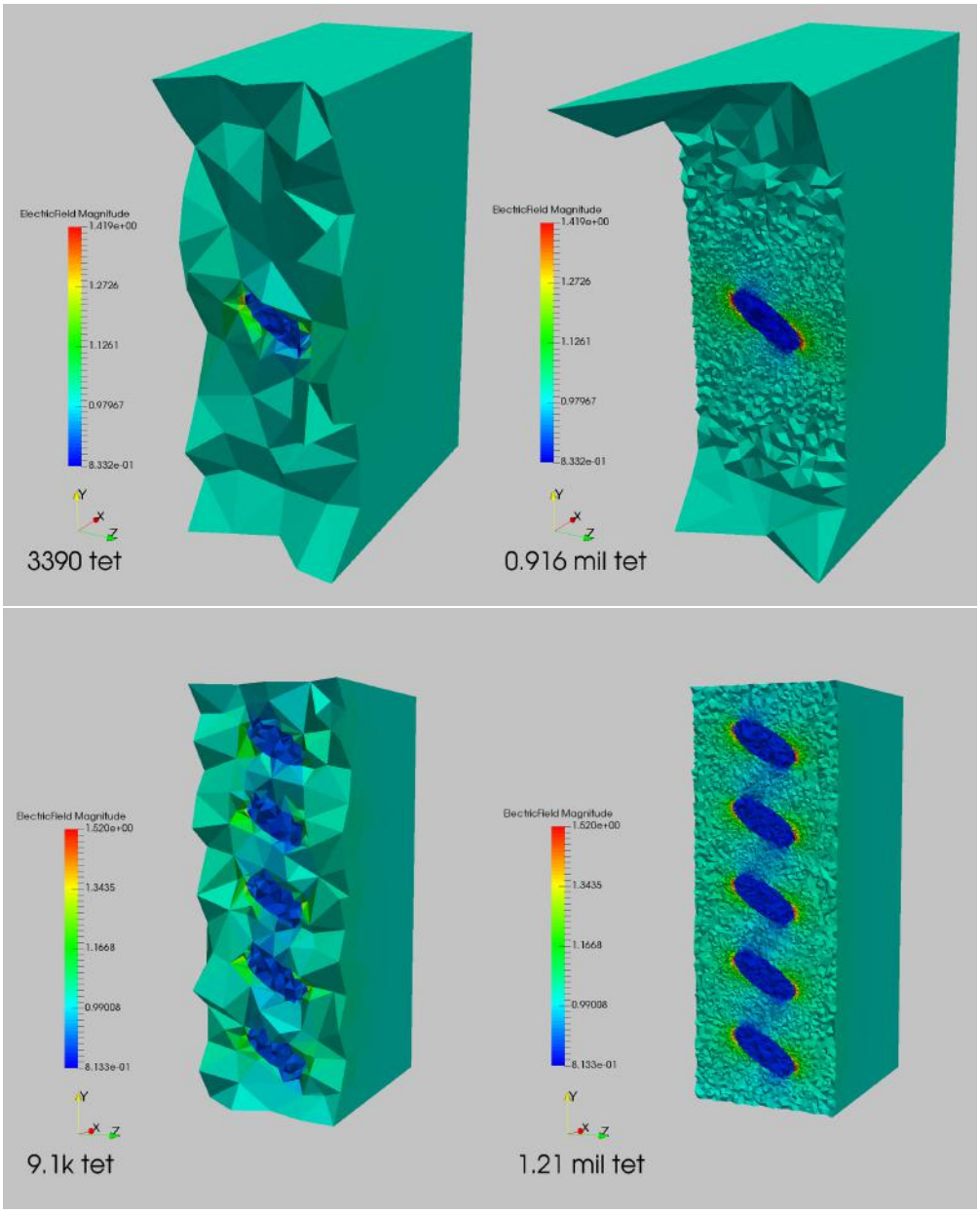


Figure 7: The electric field on the single inclusion (top) and multi-inclusion (bottom) initial and Omega_h adapted meshes.

GPU accelerated Omega_h mesh adaptation procedures employing cavity based refinement, coarsening and swapping operations. Figure 7 shows the initial and adapted meshes colored by the calculated electric field on a single (top) and multi-inclusion case (bottom).

2.3 hipBone: A performance-portable GPU-accelerated C++ version of the NekBone benchmark

The AMD Research team including Noel Chalmers, Abhishek Mishra, and Damon McDougall together with the CEED Virginia Tech team completed the initial development and testing of `hipBone v1.1.0` [9]. HipBone is an open source performance-portable proxy application for the Nek5000 (and NekRS) CFD applications [9]. It is a fully GPU-accelerated C++ implementation of the original NekBone CPU proxy application

with several novel algorithmic and implementation improvements which optimize its performance on modern fine-grain parallel GPU accelerators. Optimizations include a conversion to store the degrees of freedom of the problem in assembled form in order to reduce the amount of data moved during the main iteration and a portable implementation of the main Poisson operator kernel. We demonstrated near-roofline performance of the operator kernel on three different modern GPU accelerators from two different vendors. We present a novel algorithm for splitting the application of the Poisson operator on GPUs which aggressively hides MPI communication required for both halo exchange and assembly. Our implementation of nearest-neighbor MPI communication then leverages several different routing algorithms and GPU-Direct RDMA capabilities, when available, which improves scalability of the benchmark. We evaluated the performance of hipBone on three different clusters housed at Oak Ridge National Laboratory, namely the Summit supercomputer and the Frontier early-access clusters, Spock and Crusher. Tests demonstrated both portability across different clusters and very good scaling efficiency, especially on large problems.

HipBone details. HipBone, our re-implementation of the NekBone benchmark, is built upon a core set of libraries from the CEED open-source finite element library libParanumal (cf. [8]). These core libraries provide routines for simple hexahedral mesh generation, finite element operator construction, accelerated linear algebra and linear solvers, and nearest-neighbor communication collectives. As part of the libParanumal core, hipBone uses the Open Concurrent Compute Abstraction (OCCA) library (cf. [49]) to abstract between different parallel programming models such as OpenMP, OpenCL, CUDA, HIP, and SYCL. OCCA is a required dependency of libParanumal and provides a portable abstraction through a unified API which allows users to implement parallel kernel code in a (slightly decorated) C++ language, called OKL. At runtime, the user can specify which parallel programming model to target. OCCA then translates the OKL source code into the desired target language and Just-In-Time (JIT) compiles kernels for the user's target hardware architecture. Leveraging JIT compilation also allows different vendor compilers additional optimization, e.g. loop unrolling. The hipBone benchmark itself is fairly lightweight on top of the libParanumal libraries. In addition to simple vector operator kernels, the benchmark contains only two additional kernels. The first is used only once to populate a pseudo-random initial forcing vector, and the second is the performance-critical screened Poisson operator.

HipBone does not replicate the full functionality of NekBone at this point in time. Most notably, hipBone does no form of preconditioning in the CG iteration, while NekBone supports simple diagonal preconditioning as well as a much more sophisticated hybrid-Schwarz multigrid preconditioner detailed in [44]. We omit this functionality for now in order to first focus on optimized GPU performance, portability, and scalability of the Poisson operator and streaming operations in hipBone, leaving the topic of preconditioning on modern GPU accelerators for future study. There are, additionally, several key algorithmic/implementation differences between hipBone and NekBone. In the remainder of this section, we overview the most significant changes and their impact on performance.

DOF Storage. As described above, the NekBone benchmark stores the DOF vectors in scattered (E-vector) form, with N_L entries instead of N_G. This implementation choice has the benefit of allowing the MPI communication required in the Laplacian operator to be combined into a single gather-scatter operation, reducing MPI messaging latency costs. This is especially beneficial when strong-scaling a fixed problem size to many MPI ranks. The trade-off, however, is that vectors are now (potentially significantly) longer. Indeed, $\frac{N_L}{N_G} \approx \frac{(N+1)^3}{N^3}$, which is a still a 21.3% overhead at the relatively high order of $N = 15$. Furthermore, global reduction operations such as inner products require accessing an additional weight vector, leading to even more required data movement during each CG iteration.

Each of the vector operations in the CG iteration are of very low arithmetic intensity, i.e. they perform very few (often less than one) FLOPs per byte of data loaded from main memory. Even the element-local Laplacian operator has a fairly low arithmetic intensity, owing to the tensor-product form. Such low arithmetic intensity operations are often referred to as 'streaming' operations, and their runtimes t are well-modeled by an Amdhal-like model $t = \alpha + \beta B$, where α is some fixed latency cost, β is an effective asymptotic streaming bandwidth, and B is the amount of data moved in the operation. The reader is referred to [29] and [30] for a more complete discussion, and more recently [10] who studied streaming performance on several modern GPU-accelerators. With GPUs, the α term includes the latency cost of off-loading a kernel to the device and eventually synchronizing back with the host process. This is often referred to as a kind of 'kernel launch latency'. However, once a significant amount of data is moved in HBM (typically ∼10MB), performance

becomes dominated by the effective HBM bandwidth of the accelerator, β.

When targeting modern GPU-accelerators, the implementation choice of storing vectors in scattered form in NekBone has the effect of increasing data movement (therefore reducing effective performance) for large problem sizes. On the other hand, it also reduces MPI latency costs for small problem sizes, but in this regime the GPU's performance is primarily limited by kernel launch latency. With hipBone, we are interested in the case where the problem size is at least large enough to saturate the main memory bandwidth of the GPU. We therefore have made the implementation choice to store the DOF vectors in their assembled ordering. With this choice, vectors in the CG iteration become length N_G, rather than N_L, and some data motion, such as the accessing the weight vector in the global inner products, is avoided completely. This allows for significant effective performance improvements for large problems in each of the vector operations in the CG iteration. This design choice also aligns closely with the implementation of libParanumal's elliptic solvers, and the MFEM finite element library (cf. [2]).

Poisson Operator. The performance of the SEM Laplacian operator on hexahedral elements on modern GPU-accelerators has been studied previous by several authors. [60] presented an implementation of a high-order spectral element elliptic solver on GPU accelerators. In this work, the authors detail a Laplacian kernel that exploits fine-grain parallelism of the accelerator by mapping each of the elements to distinct threadblocks, and assigning each DOF in an element to an individual GPU thread. This approach creates a 3D thread structure of $(N+1)^3$ in each threadblock. This approach was limited to $N = 9$ due to the 1024 thread-per-block limit in CUDA, and $N = 5$ due to OpenCL's limit of 256 work-items per work-group on AMD GPUs. The approach of [60] was later extended by [69] as part of the Center for Efficient Exascale Discretizations (CEED) Exascale Computing Project co-design center[1]. In this work, the authors studied the GPU-implementation and optimization of several high-order finite element operators, including mass matrix multiplication, and diffusion matrix multiplication with/without higher-order quadrature rules. The case of the diffusion operator without higher-order quadrature is particularly useful for our hipBone implementation, as this aligns very closely with the screened Poisson operator in NekBone. The authors detail two optimized kernel implementations. The first follows the approach of [60] by using a 3D threadblock structure, but has additional optimizations such as processing several elements in a single threadblock for small polynomial orders, N. The second implementation uses a layer-by-layer processing of the hexahedral element, mapping individual GPU threads to $(N+1)^2$ nodes of a single face of the hexahedral (see for instance [66]). This implementation makes heavy use of register space, but allows for higher polynomial order N to be used. Both implementations heavily rely on shared memory as a scratchpad to store temporaries, and many of the FLOPs performed in the kernels have one or more of the inputs/outputs residing in shared memory. This can become a performance limiter as arithmetic intensity increases, as loading data to/from shared memory can be significantly slower than performing the FLOP.

In hipBone, we implement a single kernel to compute an intermediate vector $\mathbf{y}_L = (S_L + \lambda W) Z \mathbf{x}_G$, where W is a diagonal matrix of the inverse degree weights which can be defined as the diagonal matrix which satisfies $Z^T W = I$. We compute this intermediate vector $\mathbf{y}_L$ so that the final action of the Poisson operator A can be computed by gathering $\mathbf{y}_L$, i.e. $A\mathbf{x}_G = Z^T \mathbf{y}_L$, which requires MPI communication. To compute $\mathbf{y}_L$ we use a combined approach of the 3D and 2D threadblock structure kernels from [69], with some minor changes. First, for $N < 9$ we use the 3D threadblock kernel, while for larger N we use the 2D threadblock structure kernel. The decision to use the 2D threadblock kernel for $N = 9$ is made based on empirical performance measurements and current generation GPU accelerators, but is configurable. We also implement a blocking strategy in both kernel implementations, which allows for multiple elements to be processed by a single threadblock. This helps to minimize idle threads in warps/wavefronts. Finally, we slightly alter the storage of the geometric factors, G^e, so that all geometric factors at a given degree of freedom in an element are packed together. When SIMD lanes of a processor subsequently load the first geometric factor, the load will be strided since each lane loads factors for different degrees of freedom. However, this load populates the lowest-level cache, and the packed storage format makes the loads of the remaining geometric factors more efficient.

The fused action of $S_L + \lambda W$ and Z into a single kernel is a notable difference in our implementation compared to that presented by [69]. This fusing causes the reading of the element-local DOFs into the local memory on the processor to be an indirect read of $\mathbf{x}_G$. This requires additional indexing data to be stored in

[1] `http://ceed.exascaleproject.org`

memory and read-in by the kernel before accessing entries of $\mathbf{x}_G$, but the ordering of elements in the mesh and repetition of DOFs shared between elements can allow for some entries of $\mathbf{x}_G$ to be found in cache during this indirect load. With MPI parallelism, this indirect read also relies on a halo region to be populated with DOFs belonging to other processes. We detail this procedure, and the MPI parallelization of the gather operator Z^T, below. The total amount of data moved through main memory and the total FLOPs done by the Poisson operator can be estimated as follows. First, the load of $\mathbf{x}_G$ into element-local storage requires a 4 byte index, encoding a row of the scatter operator Z, to load each 8 byte entry of $\mathbf{x}_L$, totaling to $12N_L$ bytes loaded. With perfect caching of repeatedly used entries of $\mathbf{x}_G$, the amount of data moved from main memory can actually be as low as $8N_G + 4N_L$. The load of the six geometric factors in G^e and inverse degree count W for each of the N_L entries total to an additional $56N_L$ bytes, and the write of the final $8N_L$ bytes for the output of $\mathbf{y}_L$ brings the total data motion to approximately $8N_G + 68N_L \approx 8EN^3 + 68E(N+1)^3$ bytes, assuming perfect caching. The FLOP count, on the other hand, consists of $12E(N+1)^4 + 15E(N+1)^3$ FLOPS to apply S_L and $3E(N+1)^3$ additional FLOPS to add the λW contribution, for a total of $12E(N+1)^4 + 18E(N+1)^3$ FLOPS. With these estimates for data motion and FLOP count, we model the compute rate, R, of the screened Poisson operator kernel, in FLOPS, using a typical roofline model:

$$R = \min\left(C, \frac{12(N+1)^4 + 18(N+1)^3}{8N^3 + 68(N+1)^3} B\right), \tag{2}$$

where C is the peak compute rate of the processor in FLOPS, and B is the peak (bidirectional) bandwidth of the processor's main memory in bytes/s. As most current-generation GPU accelerators have peak FP64 compute rates typically measured in TFLOPS, and peak HBM memory bandwidths near 1 TB/s, we see that the low arithmetic intensity of the Poisson operator implies performance will be primarily limited by memory bandwidth, even for large degrees N. We demonstrate this on several accelerators in the computational tests below.

MPI Communication. The global mesh of hexahedral elements in hipBone is parallelized over multiple processes by partitioning the mesh evenly among the P processes. Message passing between process is then done via MPI, with a given processes requiring communication from processes that border it along a face, edge, or corner of an element. While the mesh used in NekBone and hipBone is a structured cube of uniform hexahedra, the message passing algorithms used assume no underlying mesh structure. This more accurately represents the larger Nek5000 and NekRS codes, which are generally unstructured.

In hipBone, we have re-written and extended the functionality of the `gslib` library, used in NekBone and Nek5000 for MPI communication, to a 'device-aware' gather-scatter library. This new library replicates the combined gather-scatter operations provided by `gslib`, and as well as its various communication algorithms, on both host memory or OCCA device buffers. The library also provides new gather, scatter, and halo exchange operations, which are needed for the screened Poisson operator in the assembled DOF ordering. Since each of these MPI communication operations in the new library can be cast as a sparse nearest-neighbor collective operation, then the underlying communication algorithms from the gather-scatter operation can be re-used for other operations. The library also supports leveraging GPU-Direct RDMA capabilities by directly passing pointers to device memory to MPI calls when the MPI library being used is "GPU-aware". This allows the MPI implementation to leverage high bandwidth GPU-to-GPU or GPU-network links in a system when they are present.

Our new device-aware gather-scatter library implements the following exchange routines for nearest neighbor collective communication:

- **All-to-all.** The simplest of the exchange routines, the All-to-all exchange performs all required data movement via a single `MPI_Alltoallv` function. This exchange heavily relies on existing optimized sparse all-to-all communication algorithms in the MPI implementation. Since this MPI function is one of the most general MPI collective patterns, the performance of this algorithm is typically not optimal.
- **Pairwise.** Another relatively simple exchange algorithm, the pairwise exchange implements the nearest neighbor collective operation by simply having every process send/receive all needed data to/from its neighbors, using several calls to `MPI_Isend` and `MPI_Irecv`. This exchange algorithm communicates using the maximum number of MPI messages, thereby incurring the maximum amount of messaging latency, but with direct routing moves the least amount of data possible. For very large problems where

the inter-process exchanges are sensitive to the bandwidth of the network and not messaging latency, this exchange likely performs well.

- **Crystal Router.** The crystal router algorithm performs the nearest neighbor collective via recursive hypercube folding. The algorithm is described in full by [40], and some performance results of this algorithm's use in Nek5000 were presented by [62]. For a P process grid, the algorithm can be summarized as follows: divide the grid in half and pair each process p_l in the lower half with a distinct process p_h in the upper half. Then, if process p_l has data needed by *any* process in the upper half, it sends that data to p_h, and vice versa for its partner p_h. The algorithm then proceeds recursively on the two halves of the grid. The crystal router performs the nearest neighbor collective in $\lceil \log_2(P) \rceil$ bidirectional messages, thereby minimizing the total number of messages sent/received. With a suitable distribution of the global mesh, such as that obtained by recursive mesh bisection, some messages may be avoided altogether. The amount of data sent and received in total, however, is larger than the pairwise exchange. This exchange routine, therefore, is primarily useful for smaller problems that are sensitive to network latency.

During the initial setup of the gather-scatter library, each of the exchange routines is timed, and the fastest exchange is selected for use in subsequent communication operations. When exchanging device-resident data, the gather-scatter library must either move the needed data to host memory, after which the host calls the MPI routines, or use GPU-aware MPI routines where available. In either case, a buffer of the data needed for communication is first extracted from the input vector using a device kernel. After this kernel completes, the user is free to queue additional work to the device to hide some or all of the time spent in MPI routines. Once the communication is complete on the host or device, the communication buffer is scattered back to its original order in the input vector. The exchanges, in addition to the MPI messaging latency costs, also incur device synchronization and kernel launch latency costs during the exchange as the host process must synchronize with the device to ensure the communication buffer is ready to be sent with MPI. When performing a crystal router exchange in device memory with GPU-aware MPI, we incur $\lceil \log_2(P) \rceil$ total device synchronizations and kernel launch latencies, as intermediate kernels are required to prepare the data to be sent in the next hypercube fold. As this exchange is primarily useful for small, latency-sensitive problems, it is unlikely for the device-resident crystal router to be the fastest exchange.

Overlapping halo and gather communication. At a high level, the Poisson operator application in hipBone is broken down into three stages: halo data communication, element-local Poisson operator application, and finally a gather operation. The first and third stages both require MPI communication, thus any efforts to hide this communication by local computation will greatly improve scaling efficiency. To describe how such communications are hidden, it is useful to first define a few terms relating to the degrees of freedom in elements owned by an arbitrary MPI process (Figure 8).

An arbitrary MPI process has 'ownership' of a number of spectral elements, meaning that in the scattered DOF storage each element, i.e. its constituent nodes, is stored entirely on a single distinct MPI rank. At the inter-process boundary between elements in the global mesh are nodes which are contained in two or more elements owned by different processes. We call these nodes, 'halo nodes', and all other nodes we call 'interior nodes'. 'Halo elements' are subsequently defined as elements which contain at least one halo node and 'interior elements' are all elements that are not halo elements. In Figure 8, we depict this for the case of two processes in two dimensions for simplicity. In the figure, each process owns nine quadrilateral elements with a degree $N = 3$ grid of interpolation nodes, and the inter-process boundary between them leads to several halo nodes in six halo elements. On each process, each of red-shaded elements is a halo element and the six remaining blue-shaded elements are interior elements. Within each halo element is a mixture of interior and halo nodes. In Figure 8, the halo nodes are colored red and the interior nodes are colored blue.

While there is a well-defined notion of 'ownership' of an element by a process, in the assembled DOF ordering it is less clear what process should 'own' the degree of freedom associated to a halo node. In hipBone, when defining the assembled DOF order, the owner of each halo node is chosen randomly, but fairly, from among the owners of the halo elements of which this node is a part. Consequently, when collecting the assembled DOFs into element-local form via the scatter operator Z, some DOFs must first be communicated via a halo exchange. The first step in the application of the screened Poisson operator in hipBone is to launch a kernel that extracts and packs buffers of halo node data to be communicated over MPI. Once complete, a

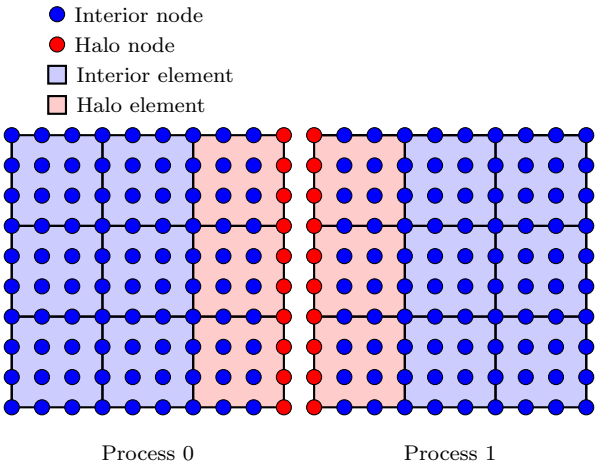


Figure 8: A two MPI process mesh arrangement of third-order 2D spectral elements.

kernel which computes the action of $(S_L + \lambda W)Z$ on *half of the interior elements* is launched. While this kernel is executing, halo data is communicated over MPI and packed into a halo region on the destination processes. Once complete, a kernel is launched to apply the action of $(S_L + \lambda W)Z$ to all the halo elements.

At this point, all that is left to do is to apply the operator to the other half of the interior elements, and then gather the intermediate vector $\mathbf{y}_L$. The interior nodes do not need to send any data to other process for the gather operation, and so the strategy for hiding MPI communication in the gather is similar to the strategy used for hiding the halo data communication. Following the kernel which computes the local operator application on the halo elements, we queue a kernel to gather values at the halo nodes into MPI buffers to be communicated as part of the global action of Z^T. Once the buffers are assembled, a kernel is launched to compute the action $(S_L + \lambda W)Z$ on the remaining half of the interior elements. Another kernel, which computes the entirely process-local action of the gather operator Z^T is also queued at this point. While both of these kernels are executing, the MPI communication required for the gather operator is performed, and the resulting gathered values at the halo nodes is written to the output vector. We show a graphical representation of this procedure in Figure 9. The figure illustrates a possible timeline of host and device activity during the application of the Poisson operator.

2.4 NekRS extensions for ExaWind collaborative study

In collaboration with ECP ExaWind team at NREL, we have continued examining the accuracy and computation performance of two open-source codes, Nek5000/RS and AMRWind, in comparison for simulating a GABLS benchmark problem representing the atmospheric boundary layer flows [51, 50].

In our previously developed large eddy simulation (LES) model for Nek5000/RS, the sub-grid-scale (SGS) dissipation was described through a high-pass filter (HPF) coupled with a relatively small, fixed, eddy viscosity and simulations were performed with wall boundary condition with the effective Reynolds, Re_{eff}=50,000 which was three orders of magnitude smaller than the actual $Re = 50M$ for the GABLS setup. From our extensive studies, we identified the necessity of a new SGS model and traction BCs, while we observed the mean velocity profiles change depending on the Re_{eff}. Our new implementation integrated into Nek5000/RS and validation are summarized in the following:

- Newly developed SGS modeling is added to Nek5000/RS and simulations are performed for Re=50M on Summit GPUs. The mean profiles of velocity and temperature are compared to those of AMRWind.
- Traction boundary condition is added to Nek5000/RS and performed for the GABLS benchmark problem on different resolutions of 0.39m, 0.78m, 1.56m, and 3.12m for convergence studies.

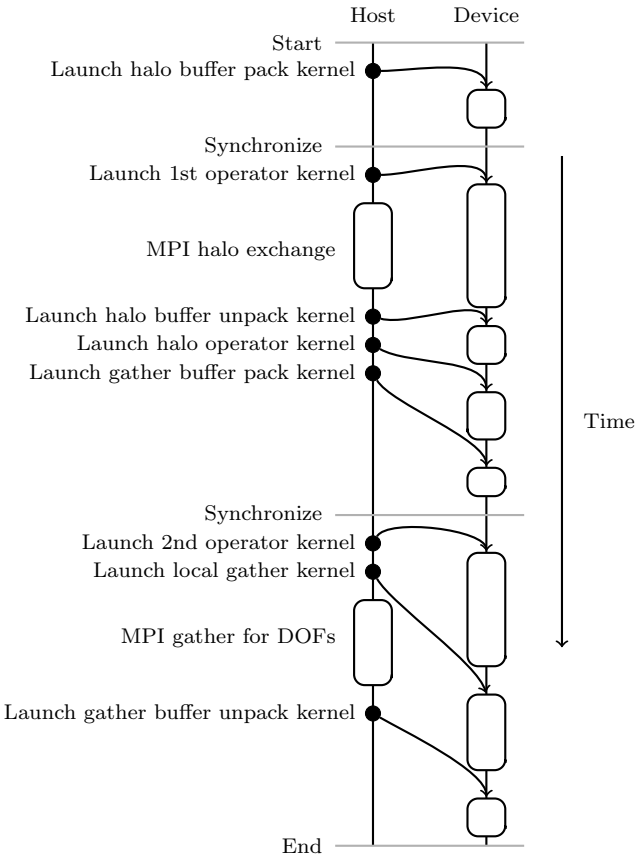


Figure 9: A timeline visualization of the operator application logic in hipBone. The illustration shows the splitting of the operator application into three kernels. Two kernels apply the operator to distinct halves of the interior elements, and the third kernel applies the operator to the halo element. This strategy allows hipBone to maximize the potential for hiding MPI communications for both the halo exchange and subsequent gather operation.

Our numerical results are based on LES, which requires enhanced dissipation to provide an energy drain at the grid scale. Consequently, we solve the incompressible Navier–Stokes (NS) and potential temperature equations in a *spatially filtered* resolved-scale formulation, expressed in nondimensional form as

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} - \frac{\partial \tau_{ij}}{\partial x_j} + f_i - \frac{\theta'}{\theta_0} g_i, \tag{3}$$

$$\frac{\partial \bar{u}_j}{\partial x_j} = 0, \tag{4}$$

$$\frac{\partial \bar{\theta}}{\partial t} + \bar{u}_j \frac{\partial \bar{\theta}}{\partial x_j} = -\frac{\partial \tau_{\theta j}}{\partial x_j}, \tag{5}$$

where an overbar denotes the LES filtering operation such that $\bar{u}_i$ is the ith component of the resolved-scale velocity vector, $\bar{p}$ is the pressure, and $\bar{\theta}$ is the potential temperature in the resolved scale. τ_{ij} and $\tau_{\theta j}$ are the stress tensors in the momentum and energy equations, which include (and are dominated by) subgrid-scale (SGS) modeling terms

$$\tau_{ij} = -\frac{2}{\mathrm{Re}} S_{ij} + \tau_{ij}^{sgs} = -\frac{1}{\mathrm{Re}} \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} \right) + \tau_{ij}^{sgs}, \tag{6}$$

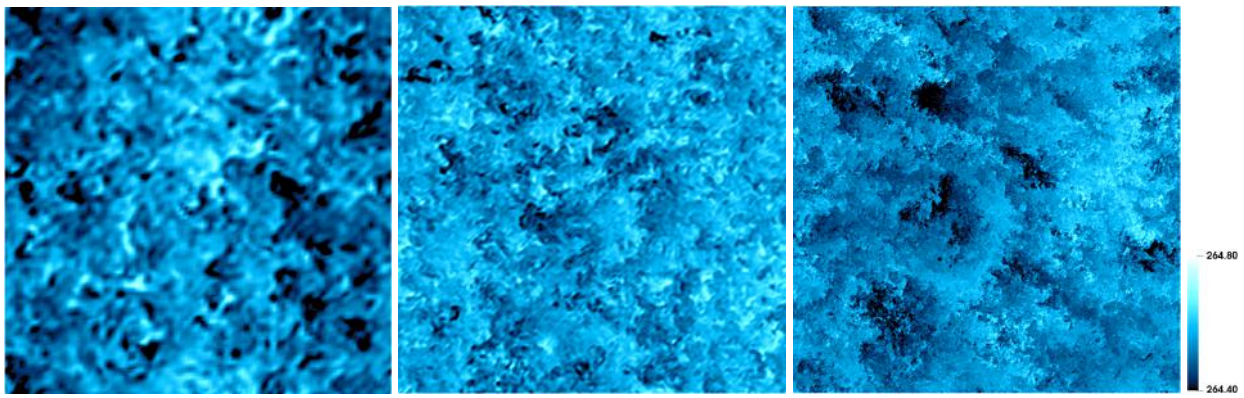


Figure 10: NekRS: potential temperature distributions in [K] at time 6h and z=100m on different resolutions of Δx= 3.12m (left), 1.56m (center), and 0.78m (right) corresponding to the number of grid points, n=128^3, 256^3, and 512^3, respectively. Δx represents the average grid-spacing for the spectral elements, $E = 16^3$, 32^3 and 64^3 and the polynomial order $N = 8$ on the domain 400m×400m×400m.

and

$$\tau_{\theta j} = -\frac{1}{\mathrm{Pe}}\frac{\partial \bar{\theta}}{\partial x_j} + \tau_{\theta j}^{sgs}, \tag{7}$$

where τ_{ij}^{sgs} and $\tau_{\theta j}^{sgs}$ are the subgrid scale stress tensors described in detail in the next sesction. The scalar θ'/θ_0 that dictates the sign and strength of the buoyancy force is obtained from

$$\frac{\theta'}{\theta_0} = \frac{\bar{\theta} - \theta_0}{\theta_0}, \tag{8}$$

where θ_0 is the reference potential temperature and f_i includes the Coriolis acceleration defined as

$$f_{c,i} = -2\epsilon_{i3k}\Omega\bar{u}_k, \tag{9}$$

where ϵ_{ijk} is the alternating unit tensor and Ω is the planetary rotation rate vector at the point of interest on the planet (which is dependent on latitude), and $j = 3$ corresponds to the vertical direction.

For the Nek5000/RS simulations the approach of [67] was used where the sub-grid-scale dissipation is split into two parts, an average eddy viscosity which is expressed in terms of mean flow quantities (anisotropic) and a fluctuating part (isotropic) which is effected through a high-pass filter (HPF). The SGS model of [67] is based on the following expression

$$\tau_{ij}^{sgs} = -2\nu_t\gamma S_{ij} - 2\nu_T \langle S_{ij}\rangle, \tag{10}$$

where the angle brackets $\langle\ \rangle$ denote averaging over the homogeneous directions and ν_T is an average eddy viscosity which needs to be expressed in terms of mean flow quantities. In (10), γ is an "isotropy factor", which accounts for variability in the theoretical SGS constants due to anisotropy of the mean flow, but also controls the transition between SGS and ensemble-average turbulence parameterizations. However, in [67] the fluctuating eddy viscosity, ν_t, is obtained using an eddy viscosity model based on the SGS turbulent kinetic energy equation, in which the shear production term is computed from the fluctuating velocities as suggested by [63]. Here, instead of an eddy viscosity SGS model for the fluctuating (isotropic) part, we use a HPF and therefore ν_t in equation (10) is by definition equal to zero. On the other hand, the expression for ν_T is derived so that the law-of-the-wall behavior can be recovered in the absence of any resolved turbulence. According to [67], a model consistent with this idea is as follows:

$$\nu_T = (C_K L_m)^2 \sqrt{2\langle S_{ij}\rangle\langle S_{ij}\rangle}, \tag{11}$$

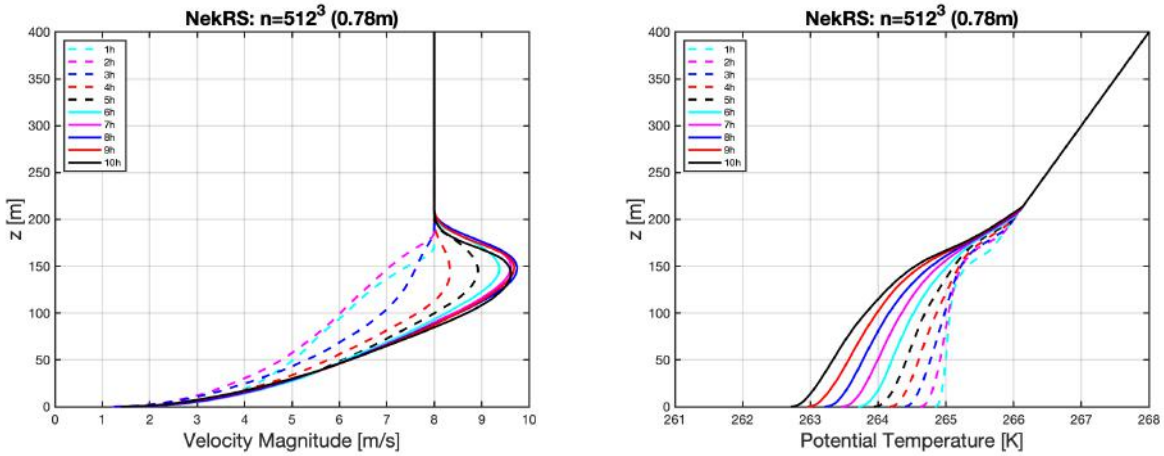


Figure 11: NekRS for ABL flows: mean profiles of the velocity magnitude and the potential temperature at each hour, 1h, 2h,...,10h. $n = 512^3$ (grid resolution of 0.78m). $n = EN^3$ with $E = 64^3$ and $N = 8$.

where S_{ij} is the strain rate tensor, C_K is a constant and $C_K L_m$ is a mixing-lqength scale. Equation 11 simplifies to

$$\nu_T = (C_K L_m)^2 \sqrt{\left(\frac{\partial \langle u \rangle}{\partial z}\right)^2 + \left(\frac{\partial \langle v \rangle}{\partial z}\right)^2}, \tag{12}$$

and by ignoring the mean wind turning with height at the first grid point implies that the wind-speed derivative is:

$$\sqrt{\left(\frac{\partial \langle u \rangle}{\partial z}\right)^2 + \left(\frac{\partial \langle v \rangle}{\partial z}\right)^2} \approx \frac{\partial U_s}{\partial z}, \tag{13}$$

where U_s is the average surface layer wind speed (i.e., $U_s = \langle \sqrt{u^2 + v^2} \rangle$). The choice of $C_K L_m$ is guided by the need to match Monin-Obukhov similarity theory in the wall region and the expression. In this theory, the wind-speed derivative becomes

$$\frac{\partial U_s}{\partial z} = \frac{u_\tau \phi_m}{\kappa z}, \tag{14}$$

where u_τ is the friction velocity, κ the von Karman constant, ϕ_m the Monin-Obukhov stability function for momentum. zsh:1: no such file or directory: /equa traction-type boundary condition at $z = z_1$, which states that the sum of the SGS and resolved momentum fluxes be equal to the surface stress, i.e.,

$$\left[\langle \tau_{uw}^{sgs} \rangle^2 + \langle \tau_{vw}^{sgs} \rangle^2\right]^{1/2} + \left[\langle uw \rangle^2 + \langle vw \rangle^2\right]^{1/2} = u_\tau^2. \tag{15}$$

This traction boundary condition in Nek5000/RS is imposed at the first grid point in the vertical direction, which is assumed to be a point inside the log-layer at a location $z = z_1$, where the boundary condition for the vertical velocity component is defined to be zero. For this reason the second term in (15) corresponding to the resolved momentum fluxes is identically equal to zero. In [67], a predictive relationship for the mean-field eddy-viscosity at the first grid point z_1, $\nu_T^\star = \nu_T(z_1)$, is obtained by invoking the approximation that the fluctuating components of strain are neglected compared to the mean strain so that only the horizontally-averaged SGS stress in (10) is retained. This leads to

$$\begin{aligned} \langle \tau_{uw}^{sgs} \rangle &= -\nu_T \frac{\partial \langle u \rangle}{\partial z}, \\ \langle \tau_{vw}^{sgs} \rangle &= -\nu_T \frac{\partial \langle v \rangle}{\partial z}. \end{aligned} \tag{16}$$

Upon substitution of (16) into (15) and making use of (13) and (14), the expression for the mean-field eddy-viscosity, $\nu_T^\star$, for Nek5000/RS becomes

$$\nu_T^\star = \frac{u_\tau \kappa z_1}{\phi_m(z_1)}. \tag{17}$$

Equation (17) provides an adaptive method for estimating the mean-field eddy viscosity needed to force the computed wind speed derivative to match with similarity theory at $z = z_1$. At any other height, similar to [67], we use

$$\nu_T = \nu_T^\star \frac{\kappa z_1}{u_\tau \phi_m(z_1)} \sqrt{2 \langle S_{ij} \rangle \langle S_{ij} \rangle}, \tag{18}$$

which follows directly from Equations (11), (13), and (14). In contrast to [67], a similar correction was also applied to the SGS potential temperature field, and $\tau_{\theta z}^{sgs}$ becomes:

$$\langle \tau_{\theta z}^{sgs} \rangle = -\nu_T \frac{\partial \langle \theta \rangle}{\partial z}. \tag{19}$$

Our Nek5000/RS results based on the new SGS model and traction boundary condition implementation are demonstrated in Fig 10 for the potential distribution and in Fig. 11 for the mean profiles at time 6 hours.

2.5 Kernel fusion in MFEM

The MFEM team continued performing research and development in generating optimized kernels and in reaching peak performance faster in high-order finite element simulations. Several CEED benchmarks have been released and are used as important benchmarks: there are the Mass and Stiffness matrices benchmarks: *BP1* (figure 12) and *BP3*, as well as the more recent preconditioned Poisson solvers: *BPS3* ([39]).

MFEM kernel fusion in BP1, BP3. We have presented in the CEED milestone report MS37 ([39]) the possibility of reaching peak performance faster on one benchmark, and one important remaining question is: can it be done in a sustainable way for other bigger problems. Fusing kernels is a real paradigm shift, where instead of reasoning like we have done so far with kernel launches, we are back to regular CPU-like code, but directly executing on the GPU. One general way to address this challenge is to gain in abstraction, using for example a more descriptive mathematical language. To achieve a sustainable way to generate these kernels, one possible way is to augment the compiler internal representation to handle this abstraction. For finite element methods the FEniCS project has formalized ten years ago the Unified Form Language: UFL. It is a domain specific language to declare finite element variational forms. The figure 12 shows the CEED BP1 benchmark written in UFL. It specifies the mesh, the finite element involved, the function space, the boundary conditions, the solution vector, the trial/test functions, as well as the linear and bilinear forms. Special function calls have been added to help with the implementation. Working at the compiler frontend allows to build a modular toolchain that transforms UFL to C++ and raw CUDA. It is a source-to-source transformation which uses the graph of all the kernels that are needed at runtime, the memory locations that are to be read / written or copied, allowing static analysis and optimizations.

If we try to fuse the algorithms presented for example in the figures 14 and 15 where one line typically represents a kernel launch, we are going to need at least a few building blocks:

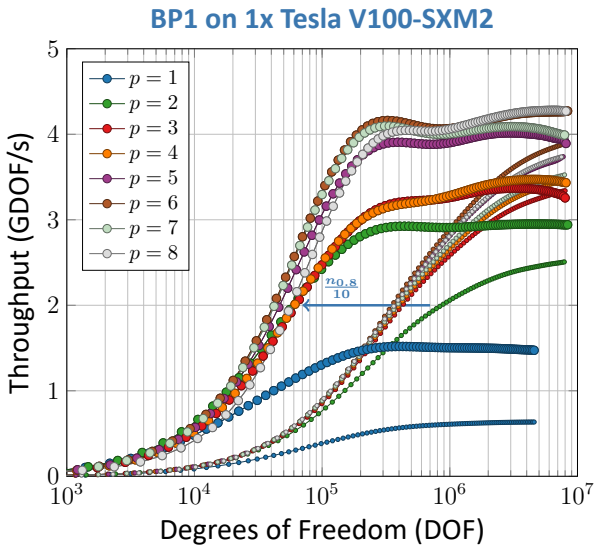
- we need a way to launch different kernels with different topologies: 1D kernels for vector operations, 2D/3D thread blocks kernels for the main partially assembled operators,
- we also need specific warp levels instructions for the dot products,
- as-well-as a way to synchronize all these different algorithmic parts.

One natural way on NVIDIA GPUs is to use the *Cooperative Groups* abstraction or programming model: we used the group partitioning for organization and used the group collectives for synchronizations.

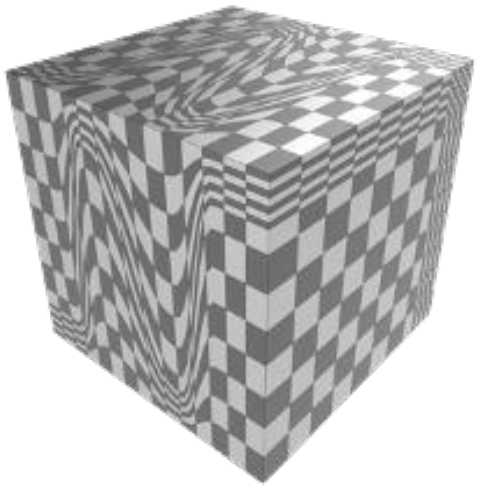
The first results presented in figure 13a so far were for the BP1 benchmark, a simple but important mass matrix solver. One main question is whether or not such gain in performance still holds for larger problems or solvers.

```
# CEED 3D Bake-off BP1, based on MFEM Example 1
order = 6

# Create the mesh
N = 16
mesh = UnitHexMesh(N,N,N)

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

# Define boundary condition.
bc = DirichletBC(fes)

# Define the solution vector x
x = Function(fes)
x = 0.0

# Define the trial and test functions
u = TrialFunction(fes)
v = TestFunction(fes)

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

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

# Solve the linear system a x = b
benchmark(a == b, x, bc)
```

Figure 12: CEED 3D Bake-off BP1 benchmark written in Unified Form Language used to generate fused MFEM kernels.



(a) BP1 kernel fusion results



(b) High order deformed Kershaw mesh

MFEM kernel fusion in BPS3. One recent CEED benchmark is the BPS3 bake-off problem, which is a full preconditioned Poisson solver. It is meant to test the scalability of high-order solvers on high-order meshes. The solver uses a higher order deformed Kershaw mesh, such as the one presented in figure 13b. Adding a preconditioner at the top-level conjugate gradient brings new elements:

- a first *outer* p-multigrid cycle with Chebyshev accelerated smoothing,
- a second *inner* low-order wavelet multigrid cycle with again Chebyshev smoothing,
- one *optional* inner-most conjugate gradient, depending on the options.

There are multiple different partially assembled diffusion operators: at each levels of the preconditioner conjugate gradient, for the outer p-multigrid cycle and its smoothing steps, for the inner wavelet multigrid cycle and smoothing steps, as well as for most inner conjugate gradient. There are also multiple dots products of different lengths required for each iteration. All of which represents a total of nearly a hundred of different kernels. Figure 14 presents all the differents kernels needed for the top root preconditioned Poisson conjugate gradient solver (*RootPCG*), the constrained diffusion operator (*RootDiffusionMult*), and the outer P multigrid cycle (*OuterPMultiGridCycle*) with the associated operators and smoothing steps.

All the different steps that led to full kernel fusion on this BPS3 solver can be illustrated through the NVIDIA NSIGHT Systems application for a typical run with tens-of-thousands of degrees of freedom, at order 6. There are six different steps that led to full kernel fusion. On the left of each figure 16, 17, 18, 19, 20 and 21, we can see the setup phase for the right-hand-side, the finite element spaces and the P and wavelet multigrid setups. The middle part if figure 16 shows one full benchmark of the pre-conditioned solver which takes without any fusion 95ms in this example. The magenta part is the outer p-multigrid where the different levels of each order-reduction can be visualised, using the transfer operators and the Chebyshev smoothing steps. The green part is for the inner wavelet multigrid, using nine levels before hitting the inner conjugate gradient. The blue part is for the inner most conjugate gradient: usually doing a handful of iterations, before going back all the way up through the wavelet interpolators and associated smoothing steps, the transposed transfer operators of the second p-multigrid V-cycles. Figure 17 shows the first fusion step, starting with the inner-most conjugate gradient bringing the benchmark time from 95 ms down to 88 ms. Figures 18, 19, 20 and 21 show all the impact of fusion and the associated timings are summarized in table 22.

MFEM MPI kernel fusion. Moving from a serial operator to the parallel version requires to transform it as a P^TAP operator. A typical boundary conditions constrained A operator, is prefixed and postfixed by communication operators. The prefix communication operator prepares the buffers, sends asynchronously the required data to the neighbors, works on some local vectors and finally unpacks received data to finalize the operation. The transposed operator is doing the same job in *opposite* order. The implementation has be done on NVIDIA's hardware with the *NVSHMEM* library which proposes a parallel API based on *OpenSHMEM*: providing a global address space for the multiple GPUs and allowing fine grain kernel-initiated communication operations. We are able to move from a standard GPU-to-CPU data transfers with the MPI interconnect model to a direct GPU-to-GPUs one. Figure 23a shows the last results obtained with a fully fused kernel with MPI communications for benchmark BP1. We are able to reduce the number of degrees of freedom to reach 80% of peak performance by a factor of five for the higher orders. It is the first time we see such improvement while adding the MPI communications overhead.

Figure 23b presents the three different implementations for the multi-GPUs dot-product: a *reference* one in blue based on the *cooperative groups* using the *atomic_system* call which is straight-forward but limited to a single node, a *standard* second one in orange presenting what MFEM typically uses (one first warp level reduction, followed by an MPI reduction on the CPU after some host transfer), and finally an *all-fused* one in green using the *NVSHMEM* call to compute the global reduction directly.

2.6 Recent GPU improvements in MARBL

Below we list several GPU-related developments that have recently improved the MARBL application. We stress that all of these activities have been carried out in close collaboration between the MARBL and MFEM teams. Most of resulting capabilities improved not only MARBL, but the MFEM project as well.

- Since the MFEM team added the HYPRE+CUDA infrastructure support (see MFEM PR `#1492`), the MARBL team has been exploring the use of HYPRE's GPU-based AMG preconditioners. Currently MARBL can solve linear systems on GPUs with HYPRE, and further optimization / porting work of related components is in progress. This capability is important for diffusion physics problems, e.g., in MARBL's radiation diffusion and MHD modules.

```
RootPCG(in b, out x):
    x = 0.0, CG.r = b                  # x = 0, r = b
    OuterPMultiGridCycle(CG.r, CG.z)   # prec: z = M r, d = z
    CG.d = CG.z, nom = CG.d * CG.r
    RootDiffusionMult(CG.d, CG.z)      # z = A d
    den = Dot(CG.z, CG.d)
    for (iter = 1; iter <= max_iter; ++iter)
        alpha = nom / den
        x += alpha * CG.d              #  x = x + alpha d
        CG.r -= alpha * CG.z           #  r = r - alpha A d
        OuterMGCycle(CG.r, CG.z)       # prec: z = M r
        betanom = CG.r * CG.z
        if (betanom <= r0)  break
        beta = betanom / nom
        CG.d[i] = beta*CG.d + CG.z     # prec: d = z + beta d
        RootDiffusionMult(CG.d, CG.z) #  z = A d
        den = CG.d * CG.z
        nom = betanom

RootDiffusionMult(x, y):
    y = 0.0, MG.W = x
    MG.W[D.ess_tdof_list] = 0.0
    RootDiffusionApply(MG.W, y)
    y[D.ess_tdof_list] = x[D.ess_tdof_list]

OuterPMultiGridCycle(x, y):
    MG.X[fine_level] = x
    MG.Y[fine_level] = 0.0
    for (level=fine_level; level>0; level--)
        for (k = 0; k < S.steps; k++) OuterSmoothingStep(level)  # Pre-SmoothingStep
        OuterDiffusionMult(level, MG.Y[level], MG.R[level])      # Compute residual
        MG.R[level] = MG.X[level] - MG.R[level]                  # R = X - R
        OuterTransferMultTranspose(level-1, MG.R[level], MG.X[level-1]) # Restriction
        MG.Y[level-1] = 0.0
    InnerWaveletMultiGridCycle(G.X[0], MG.Y[0])                  # COARSE solver
    for (level=1; level<=fine_level; level++)
        OuterTransferMult(level-1, MG.Y[level-1], MG.R[level])   # Prolongate
        MG.Y[level] += MG.R[level]                               # Update
        for (k = 0; k < S.steps; k++) OuterSmoothingStep(level)  # Post-Smoothing
    y = MG.Y[fine_level]

OuterDiffusionMult(level, x, y):
    y = 0.0, MG.W[level] = x
    MG.W[level][D.ess_tdof_list[level]] = 0.0
    DiffusionApply3D(MG.W[level], y)
    y[D.ess_tdof_list[level]] = x[D.ess_tdof_list[level]]

OuterSmoothingStep(level):
    MG.R[level] = A[level] * MG.Y[level])           # r = A x
    MG.R[level] = MG.X[level] - MG.R[level]         # r = b - A x
    OuterSmoother(level, MG.R[level], MG.Z[level])  # z = S r
    MG.Y[level] += MG.Z[level]                      # x = x + S (b - A x)

OuterSmoother(level, r, z):
    z = 0.0, S.r[level] = r
    for k in range(S.order):
        if (k > 0) S.r[level] = A[level] * S.r[level]
        S.r[level] *= S.dinv[level]
        z += S.coeffs[level][k] * S.r[level]
```

Figure 14: BPS3: Fully fused preconditioned conjugate gradient with all the differents kernels needed for the top root preconditioned Poisson conjugate gradient solver, the constrained diffusion operator, and the outer P multigrid cycle with its operators and smoother.

```
InnerWaveletMultiGridCycle(x, y):
   dX[fine_level] = x                                # X = x
   dY[fine_level] = 0.0                              # Y = 0.0
   for (level=fine_level; level>0; level-=1)
      for (k = 0; k < smoothing_steps; k++)
         InnerWaveletSmoothingStep(level, dX[level], dY[level])
      InnerWaveletDiffusionMult(dY[level], dR[level])
      dR[level] = dX[level] - dR[level]
      WaveletMult(dR[level], dX[level-1])            # Restriction
      dY[level-1][i] = 0.0                           # Y[level - 1] = 0.0
   InnerWaveletCG(dX[0], dY[0])                      # COARSE Wavelet solver
   for (level=1; level<=fine_level; level+=1)
      WaveletMultTranspose(dY[level-1], dR[level]) # Prolongate
      dY[level] += dR[level]                         # Update
      for (k = 0; k < smoothing_steps; k++)          # Post-Smoothing
         InnerWaveletSmoothingStep(level, dX[level], dY[level])
   y[i] = dY[fine_level]

InnerWaveletSmoother(level, X, Y):
    R = X
    Y = 0.0
    for (k = 0; k < smoother_order; ++k)
        if (k > 0) InnerWaveletDiffusionMult(R, H), R = H
        R *= Dinv  # Scale residual by inverse diagonal
        Y += C * R # Add weighted contribution to y

InnerWaveletSmoothingStep(level, x, y):
    InnerWaveletDiffusionMult(y, r)  # r = A y
    r = x - r                        # r = x - A y
    InnerWaveletSmoother(r, z)       # z = S r
    y[i] += z[i]

InnerWaveletDiffusionMult(level, d, z):
   WaveletMultTranspose(Ns[level], d, Wtd[level+1])
   for (l=level+1; l<fine_level; l+=1) WaveletMultTranspose(Wtd[l], Wtd[l+1])
   Wtw[fine_level] = Wtd[fine_level]
   Wtz[fine_level][i] = 0.0
   Wtw[fine_level][ess_tdof_list] = 0.0
   PADiffusionApply3D(Wtw[fine_level], Wtz[fine_level])
   Wtz[fine_level][ess_tdof_list] = Wtd[fine_level][ess_tdof_list]
   for (l=fine_level; l>(level+1); l-=1) WaveletMult(Ns[level], Wtz[level+1], z)

InnerWaveletCG(b, x):
   x = 0.0
   d = r = b
   nom = d*r
   InnerWaveletDiffusionMult(d, z)
   den = z*d
   for (iter = 1; iter <= max_iter; ++iter)
      alpha = nom / den
      x += alpha * d #  x = x + alpha d
      r -= alpha * z #  r = r - alpha A d
      betanom = r*r
      if (betanom <= r0) break
      beta = betanom / nom
      d = beta * d + r
      InnerWaveletDiffusionMult(d,z)
      den = d * z
      nom = betanom
```

Figure 15: BPS3: Fully fused preconditioned conjugate gradient with all the differents kernels needed for the inner wavelet multigrid cycle, the smoothing steps, and the last *optional* inner-most conjugate gradient, depending on the options.

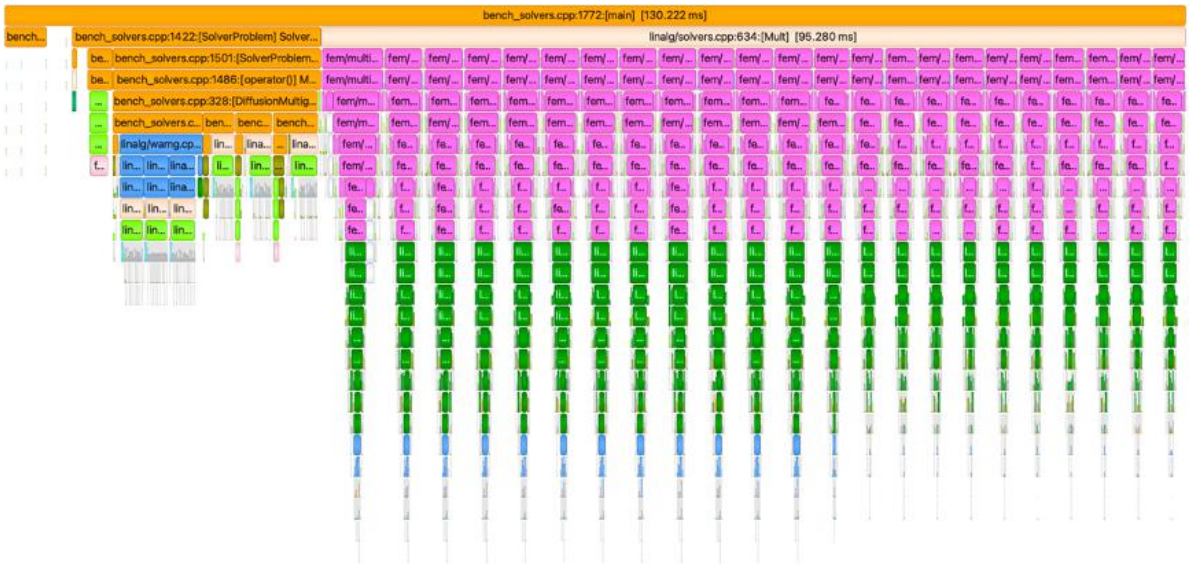


Figure 16: Without any fused step, the whole solver takes 95ms in average.

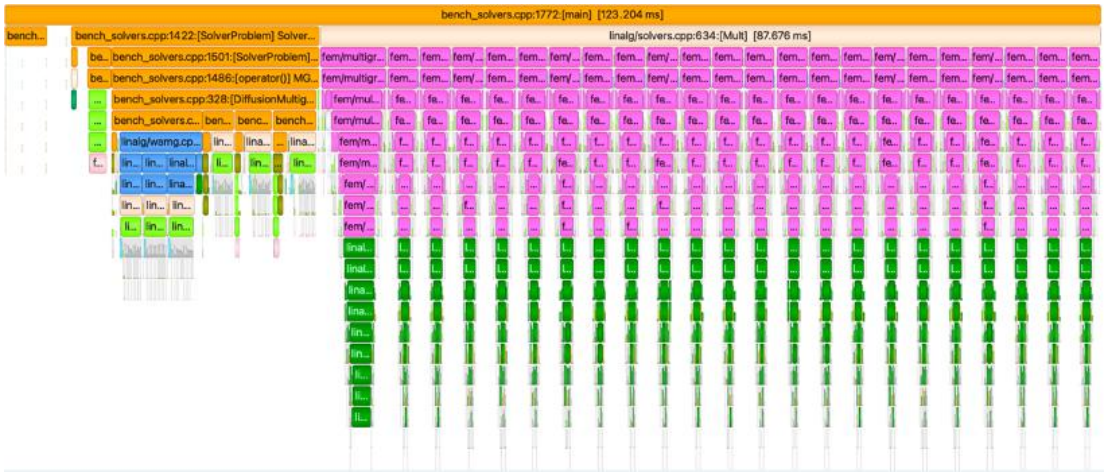


Figure 17: The first fusion step, starting with the inner-most conjugate gradient brings the benchmark time from 95 ms down to 88 ms.

- The MARBL team built on top of MFEM's GPU infrastructure and added support for using the HYPRE library built with HIP support, see MFEM PR `#2750`. Similar to the HYPRE + CUDA support added earlier, most of the MFEM examples and miniapps work transparently with HYPRE + HIP builds. This includes the `HypreBoomerAMG`, `HypreAMS`, and `HypreADS` solvers, which can now be used with AMD GPUs in both MFEM and MARBL.
- The MARBL team extended MFEM's GPU sparse linear algebra infrastructure by adding a hipSPARSE support in addition to the existing cuSPARSE, see MFEM PR `#2784`. The hipSPARSE option has led to noticeable performance improvements in MARBL.
- The MFEM team added matrix-free GPU support for several of the existing H(curl) integrators, see MFEM PR `#2856`. Utilizing these capabilities by the MHD module of MARBL is in progress.
- The MFEM team has added support for Discontinuous Galerking partial assembly on nonconforming grids, see MFEM PR `#2195`. This capability will be used by the remap phase of MARBL in the future.

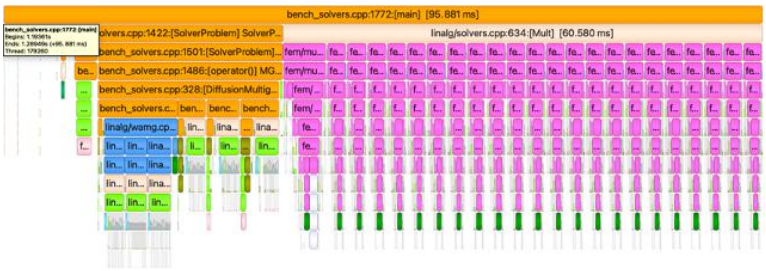


Figure 18: This step fuses the inner wavelet multigrid. Most of the software toolchain offline work is to prepare these transformations by populating the *contexts* which hold the needs of all the different kernels. This brings the benchmark time down to 61ms.

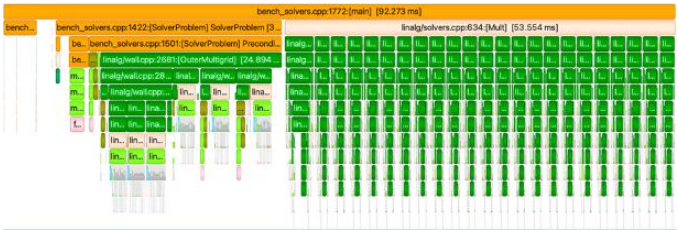


Figure 19: Intermediate steps can be introduced in order to allow graph optimizations of the kernel launches and the data movements. The best impact is when we are able to reduce the number of copies. This brings the benchmark time down to 54ms.

Figure 20: One important step is the fusion of the outer p-multigrid: it brings a factor of two here. The closest we are to having everything fused, the more gain we usually see. This brings the benchmark time down to 26ms.

Figure 21: The last part is the top preconditioned conjugate gradient. Everything is now generated, embedded and fused in a whole big kernel. For this benchmark and these options, we are going from 95ms down to 22ms, which represents a reduction by a factor of more than four.

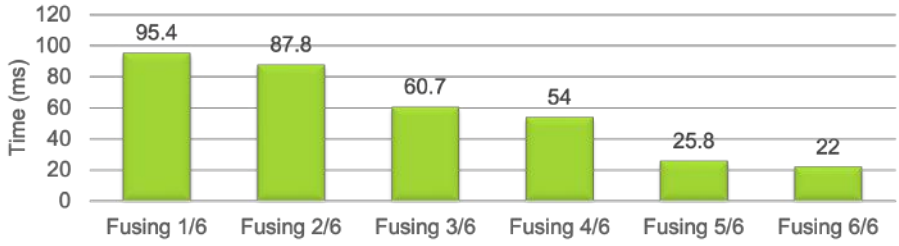
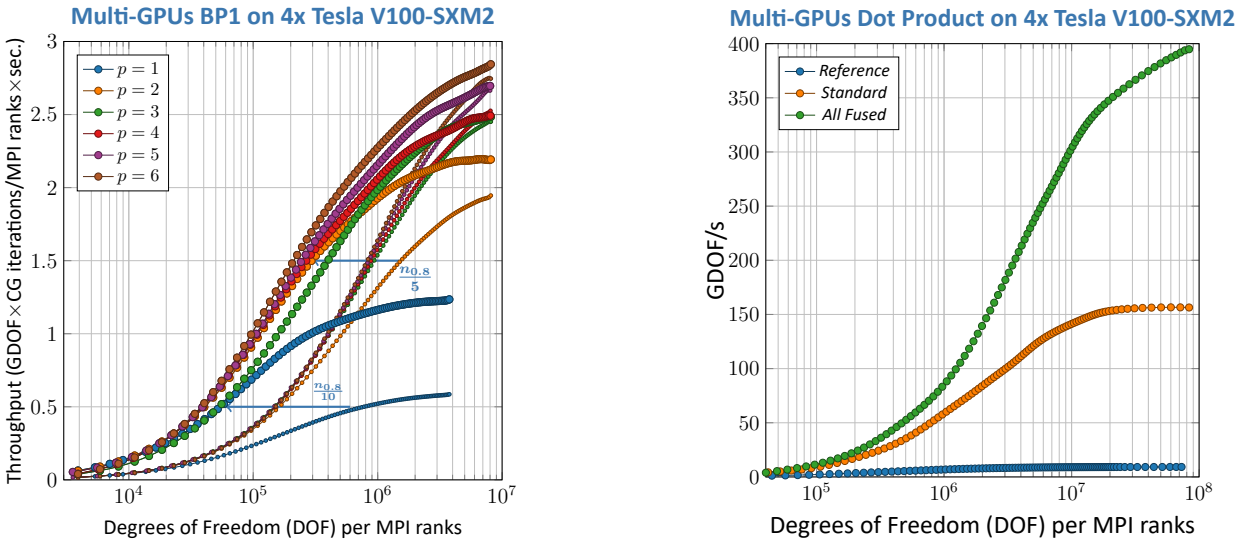


Figure 22: All the different steps that led to full kernel fusion on this BPS3 solver for a typical run with tens-of-thousands of degrees of freedom, at order 6. There are six different steps that leads to full kernel fusion.



(a) bench bp1p

(b) bench multi

Figure 23: MFEM parallel MPI BP1 kernel fusion

3. MATRIX-FREE PRECONDITIONING

3.1 Matrix-free preconditioning for Maxwell problems

While low-order-refined preconditioning has been a successful technique for solving Poisson-type problems, the direct application of this strategy to definite Maxwell problems in $H(\text{curl})$ has not been successful. One reason for this is that the curl operator has a nontrivial nullspace (the space of all irrotational vector fields), and this nullspace must be correctly handled by the preconditioner. In the case of the Poisson problem, the preconditioner must only treat the nullspace of the gradient operator, which consists of constant functions (for both the high-order and low-order-refined discrete versions).

Recent work [21, 54] has seen the development of a new basis for $H(\text{curl})$ and $H(\text{div})$ spaces that recovers the low-order-refined spectral equivalence, and hence also allows for the construction of efficient matrix-free preconditioners for definite Maxwell problems. By using this basis, one can assemble the corresponding low-order-refined problem (which is sparse, and the assembly can be performed with optimal memory requirements are computational complexity). Given the assembled low-order system, any effective preconditioner can be constructed and applied to the high-order problem. For example, *hypre*'s Auxiliary Maxwell Solver (AMS) has recently been ported to the GPU, and achieves good performance on these types of problems. A comparison of the effectiveness of this approach to the naive low-order-refined approach is shown in Figure 25 (left panel).

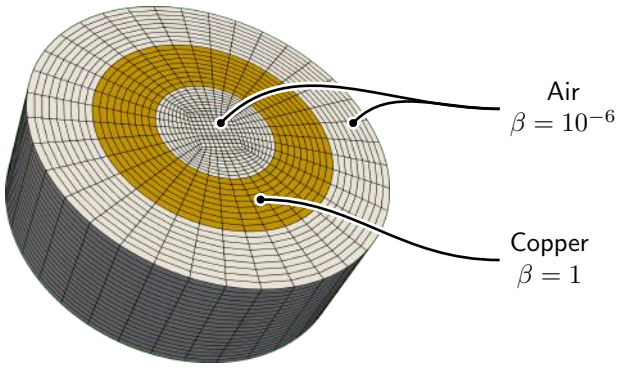


Figure 24: Schematic of "copper wire" definite Maxwell problem.

As an example, we consider the "copper wire" benchmark problem. This problem simulates electromagnetic diffusion of a copper wire in air. The governing equations are given by

$$\nabla \times \nabla \times \boldsymbol{u} + \beta \boldsymbol{u} = \boldsymbol{f},$$

where β represents the conductivity coefficient. This coefficient is given by a piecewise constant, with $\beta_{\text{air}} = 10^{-6}$ and $\beta_{\text{copper}} = 1$. This problem is solved on a mesh with 21,060 curved $\mathcal{Q}_3$ elements. A schematic of this problem and the computational mesh are shown in Figure 24. Comparing the new low-order-refined approach to a standard matrix-based AMS solver results in significant speedups and memory savings, in particular at higher orders, see Figure 25 (right panel).

3.2 Batched low-order-refined assembly in MFEM.

MFEM supports four different levels of assembly: `NONE`, `PARTIAL`, `ELEMENT`, and `FULL`. `NONE` corresponding to a "Matrix-free" form that computes all of its action on-the-fly without any substantial storage. `PARTIAL` corresponding to a partially-assembled form, which computes and stores data only at quadrature points. `ELEMENT` corresponding to a form assembled at element level, which computes and stores dense element matrices. `FULL` corresponding to a fully assembled form, i.e. a global sparse matrix in MFEM format.

All these different levels of assembly are compatible with device execution and build on top of each other. Starting from the CEED decomposition $A = P^T G^T B^T D B G P$, `PARTIAL` assembly stores D which contains the values at quadrature points. Reusing D, `ELEMENT` assembly computes the local dense matrices $A_e = B^T D B$. And finally, `FULL` assembly assembles a local sparse matrix by computing $A_L = G^T A_e G$.

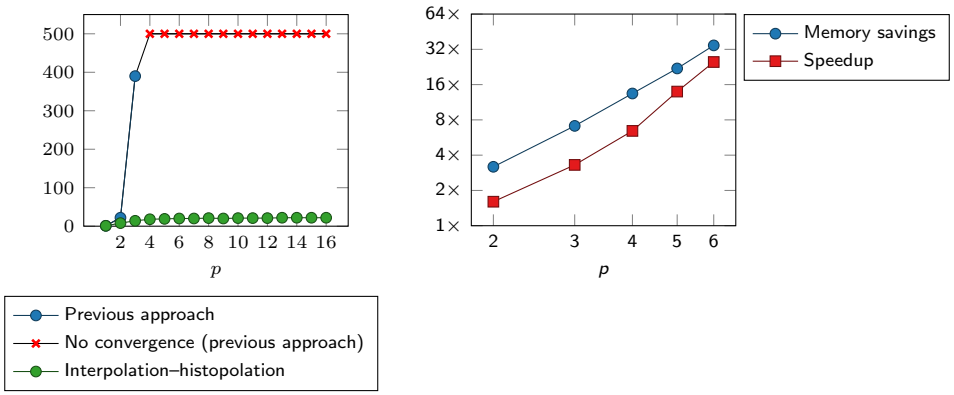


Figure 25: Left: comparison of convergence results using low-order-refined preconditioning in $H(\mathrm{curl})$. Right: speedup and memory savings on 3D "copper wire" Maxwell problem.

Table 1 summarize the different assembly cost for tensor-product elements. It is important to observe that `ELEMENT` and `FULL` assembly levels have significantly higher computational and storage cost relative to `PARTIAL` assembly. Figure 26 shows the difference in setup throughputs between `PARTIAL`, `ELEMENT`, and `FULL` assemblies on a V100 for a diffusion problem (BP3).

	Sparse Matrix	Element Matrix	Partial Assembly
FLOPs for assembly	$O(np^{3d})$	$O(np^{3d})$	$O(np^{d+1})$
Amount of storage	$O(np^{2d})$	$O(np^{2d})$	$O(np^{d})$

Table 1: Asymptotic computational and storage costs for the different levels of assembly in MFEM using tensor elements.

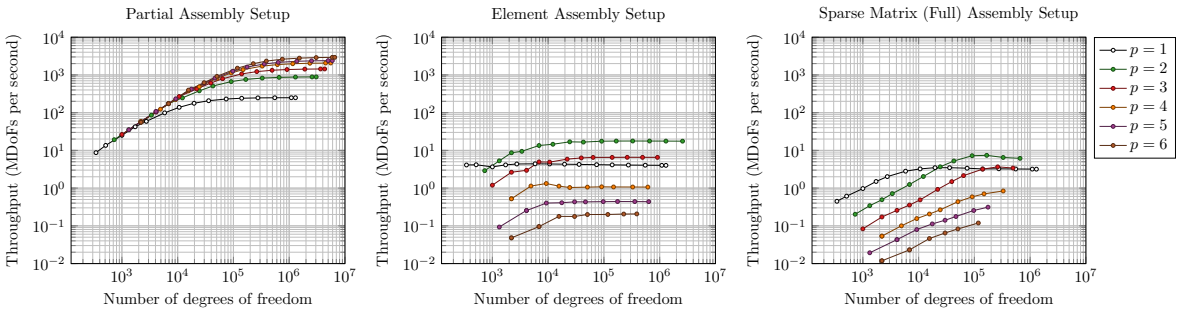


Figure 26: Setup throughput for the different levels of assembly for a diffusion operator (BP3) in MFEM using a NVIDIA V100.

New capabilities have been introduced in MFEM that allow for GPU-accelerated batched assembly of low-order-refined (LOR) discretizations, used for the matrix-free preconditioning of high-order operators. This capability complements the existing `FULL` assembly mode in MFEM, which builds on the `PARTIAL` and `ELEMENT` assembly modes to assemble the sparse matrix (CSR) representation of a finite element operator.

Low-order-refined operators possess additional structure that allow optimized assembly algorithms to achieve higher performance than purely unstructured algorithms. For example, each high-order element corresponds to a "macro-element" in the low-order refined mesh, which consists of a fixed number of subelements. These subelements are arranged in a topologically Cartesian structure, with fixed adjacency. Therefore, with each macro-element one can associate a sparse matrix with fixed sparsity pattern that can be

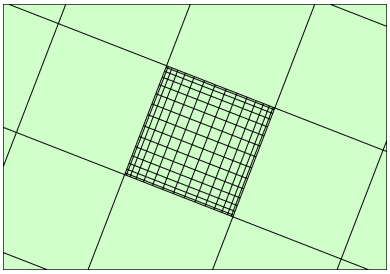


Figure 27: Illustration of a mesh of high-order elements, showing the low-order subelements of a single "macro element."

assembled into a global sparse matrix, reusing only the element restriction information from the high-order discretization.

A significant bottleneck in the assembly of the low-order-refined matrix is the construction of the refined mesh and the associated computation of the geometric factors. The construction of the mesh object is a preprocessing step typically performed on host. Because each macro-element possesses a topologically Cartesian structure, it is possible to perform the matrix assembly using only the topological information from the coarse (high-order) mesh with specialized kernels.

The assembly kernels in MFEM take as input the coordinates of the low-order mesh vertices. These coordinates can be thought of as Q-vectors (each vertex of the low-order mesh is analogous to a quadrature point in the high-order mesh), and are obtained by interpolating the high-order mesh coordinates at the low-order-refined points. Within the assembly kernel, each macro-element is assigned a block of threads, and one thread assembles each low-order subelement. The geometric factors (Jacobian matrices and determinants) are computed on the fly using the coordinate vectors, and then discarded. The subelement matrices are assembled into a macro-element sparse matrix. All macro-element sparse matrices share the same sparsity pattern, with a fixed upper bound on the number of nonzeros per row (for the diffusion operator, 9 nonzeros per row in 2D, and 27 nonzeros per row in 3D).

After obtaining the macro-element sparse matrices, a global CSR matrix is assembled, using the gather and scatter maps from the high-order element restriction operator. The global (parallel) CSR matrix can be obtained from the rank-local CSR matrices by computing the triple product $P^T AP$. This product is computed using *hypre*'s optimized matrix-matrix GPU kernels. Initial profiling results indicate that this triple product represents a significant portion of the assembly time, and so alternative parallel assembly strategies are being investigated.

The batched assembly strategy described here can result in significant performance gains, when compared with a fully unstructured assembly algorithm. The throughput of the batched algorithm, especially for larger problems, is several times higher than that of the unstructured algorithm. We further note that the unstructured ("`FULL`") throughput results omit the construction of the low-order-refined refine and its geometric factors, which represents a significant bottleneck. Including these preprocessing steps in the benchmark results in throughputs that are consistently less than 1 MDOF/s, compared with over 20 MDOF/s for the batched algorithm. A comparison of the assembly throughput results for the initial implementation (including costly preprocessing steps) and the current batched implementation are shown in Figure 28. Overall, a significant performance improvement of about two orders of magnitude is achieved for the LOR matrix assembly.

Furthermore, even without including the preprocessing steps (assuming, for example, that the refined mesh and its geometric factors have been precomputed), the new batched algorithms significantly outperform the unstructured algorithm. In Figure 29, we compare the throughput of the `FULL` assembly algorithm to the batched algorithm, where the `FULL` results omit both preprocessing steps and memory allocations. For the batched algorithm, we include two sets of results: one with full allocations, and one with preallocated memory (assuming, for example, that the sparse matrix has been preallocated, and only its entries are recomputed). The batched kernels reach a higher peak throughput and sustain this level for larger problems. Additionally, the throughput results using preallocated memory display favorable strong scaling results, reaching peak performance even for problems with a relatively small number of degrees of freedom per GPU.

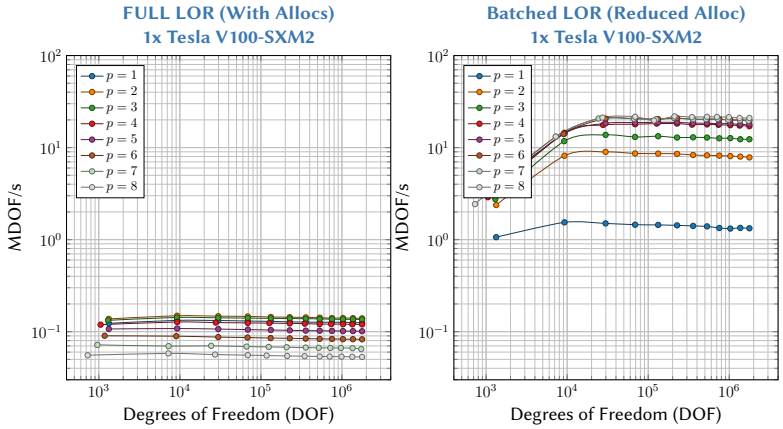


Figure 28: Low-order-refined matrix assembly throughput results on one Tesla V100 GPU (lassen). The left plot shows matrix-assembling using the `FULL` assembly level, including the preprocessing steps of creating the refined mesh and computing geometric factors. The right plot shows the throughput of the batched kernels, which do not require these expensive preprocessing steps, and additionally make use of a more performant specialized assembly algorithm.

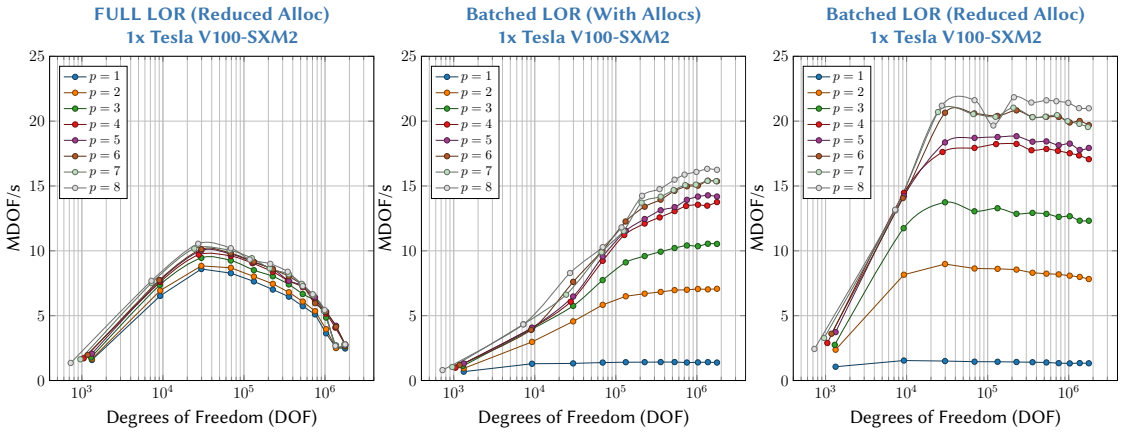


Figure 29: Low-order-refined matrix assembly throughput results on one Tesla V100 GPU (LLNL's *lassen*). The left plot shows matrix assembly using the `FULL` assembly level (omitting the expensive preprocessing step of creating the low-order refined mesh). The center plot shows the new batched algorithm, including *all* preprocessing steps and memory allocations. The right plot shows the throughput of the batched assembly alone, assuming the memory required for the sparse matrix has been pre-allocated.

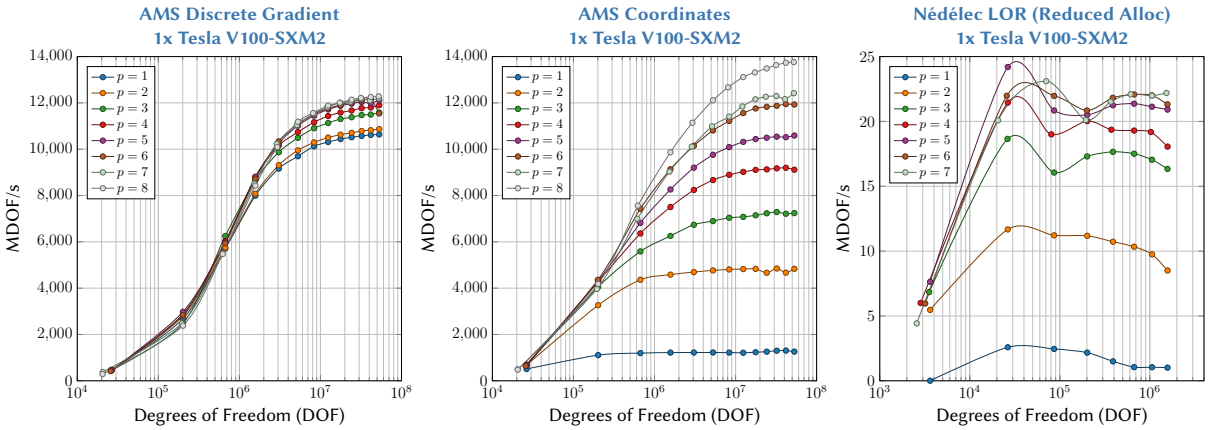


Figure 30: Low-order-refined matrix assembly for Maxwell problems with Nédélec elements. The left plot shows the throughput of the kernel that assembles the discrete gradient matrix needed for the AMS Maxwell solver. The right plot shows the throughput of the kernel that assembles the low-order refined definite Maxwell problem using Nédélec elements.

In addition to the assembly of the mass and diffusion systems (required by the CEED BPS bakeoff problem), MFEM also supports the batched LOR assembly of $H(\text{curl})$ problems using Nédélec elements. Making use of the low-order-refined solver techniques described in Section 3, definite Maxwell solvers such as AMS applied to the low-order-refined system can be used to precondition high-order Maxwell problems. Therefore, the efficient assembly of low-order-refined $H(\text{curl})$ systems is an important component of the preconditioner setup. The AMS solver provided by *hypre* requires as additional inputs a *discrete gradient* matrix, that maps a continuous function in the H^1 finite element space to its gradient, represented as a Nédélec grid function in $H(\text{curl})$, as well as coordinate vectors representing the vertices of the low-order-refined mesh. Therefore, MFEM provides kernels to assemble the low-order-refined discrete gradient matrix and coordinate vectors, which can then be provided to *hypre* to complete the solver setup. Throughput results for the assembly of Nédélec elements (and the associated discrete gradient and coordinate vectors) is shown in Figure 30. The discrete gradient and coordinate vectors can be computed very efficiently on GPU, resulting in peak throughput for large problems of roughly 12×10^9 DOFs/s and 14×10^9 DOFs/s, respectively. The major computational cost is associated with assembling the sparse matrix associated with the definite Maxwell discretization, which achieves a peak throughput of just under 2.5×10^7 DOFs/s.

3.3 Matrix-free preconditioning for pressure solve in Nek

A central component of many multiphysics problems is the need to solve Poisson problems that generally arise from fast-time-scale physics (e.g., acoustic or EM waves) being replaced by constraints (e.g., incompressibility or electroneutrality). These elliptic subproblems consequently constitute the stiffest and most time consuming substep in time-advancement of the overall system. As a result, there is a critical need for fast solvers. Matrix-free high-order methods that are based on local tensor-product polynomial expansions present their own challenges and opportunities for developing performant preconditioners. The challenges stem from the relative ill-conditioning of the stiffness matrix and, worse, the impact of effectively high-aspect ratio cells that heuristically result from the tensor-products of the Gauss-Lobatto-Legendre (GLL) nodal point spacing. The opportunities include the fact that these methods have a naturally nested approximation sequence that is realized by interpolating from lower polynomial orders to p' to $p'' > p'$, with $p \geq p''$ being the baseline approximation order. Additionally, the structured bases admit the use of fast tensor-product solvers so that local systems can be solved in a time that is roughly equivalent to that of forward operator evaluation. Finally, for nodal bases on hexahedral elements, there is spectral equivalence between the high-order discretization and a low-order ($p = 1$) FEM Laplace operator using the same nodal points [53, 7, 3, 54].

Here we summarize parallel-scaling results from [57] for spectral-element based Poisson problems using Chebyshev-accelerated Schwarz and Jacobi preconditioning schemes, with special focus on GPU architectures.

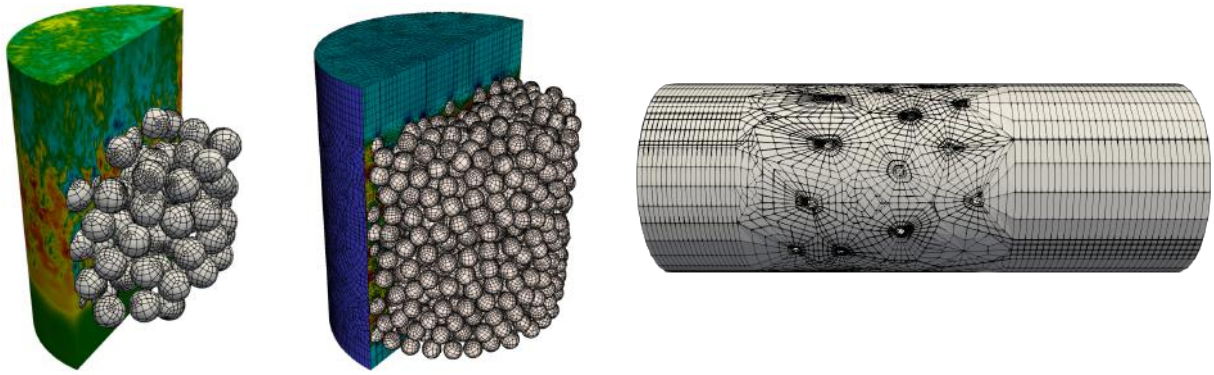


Figure 31: Navier-Stokes pebble-bed cases with (a) 146, (b) 1568, and (c) 67 spheres.

Iteration performance of the Chebyshev-accelerated schemes are compared with alternative methods, such as low-order preconditioners combined with algebraic multigrid.

Test Cases. Three challenging test cases are considered, corresponding to turbulent flow through a cylindrical packed-bed with S=146, 1568, and 67 spherical pebbles (resp., E=62K, 524K, and 122K). The 146 and 1568 pebble cases are from Lan and coworkers [41]. The first two bed flows are at Reynolds number $Re_D = 5000$, based on sphere diameter, D, while the 67 pebble case is at Reynolds number $Re_D = 1460$. The polynomial order is $p = 7$ for each case.

Time advancement is based on a two-stage 2nd-order characteristics timestepper with CFL=4 ($\Delta t = 2 \times 10^{-3}$ $\Delta t = 5 \times 10^{-4}$, and $\Delta t = 5 \times 10^{-5}$ for the 146, 1568, and 67 pebble cases). An absolute pressure solver tolerance of 10^{-4} is used. A restart at $t = 10$, $t = 20$, and $t = 10.6$ convective time units is used for the 146, 1568, and 67 pebble cases, respectively, to provide an initially turbulent flow. In all cases, solver results are collected over 2000 timesteps.

Navier-Stokes Results. We consider scalability of nekRS for the cases of Fig. 31. All simulations except one use GMRES(15) with an initial guess generated by A-conjugate projection onto 10 prior solutions [24]. Due to memory constraints, the 1568-pebble case with $P = 24$ uses GMRES(10) with only 5 solution-projection vectors. For each case, two pMG schedules are considered: $(7, 5, 3, 1)$ and $(7, 3, 1)$ for $p = 7$; and $(9, 7, 5, 1)$ and $(9, 5, 1)$ for $p = 9$. Other parameters, such as the Chebyshev order and the number of coarse grid BoomerAMG V-cycles are also varied. Results are shown in Fig. 32. The plots relate the effective work rate per node, measured as the gridpoints $n = Ep^3$ solved per second per node, to the time-to-solution. The y-axis notes the drop in the relative work rate, which corresponds to a lower parallel efficiency, as the strong scale limit is reached, while the x-axis denotes the time-to-solution. Each node consists of 6 GPUs, hence $P = 6 \times$ nodes.

In all the performance tests conducted, the pMG preconditioner with Chebyshev-Jacobi smoothing is outperformed by the other preconditioners, whether using one or two V-cycle iterations in the AMG coarse-grid solve. For each case, the fastest preconditioner scheme varies. In the 146 pebble case (Fig. 32a), using Cheby-RAS(2),(7,5,3,1) yields the smallest time per pressure solve. However, in the 1568 pebble case (Fig. 32b), SEMFEM is a moderate improvement over the second best preconditioner, Cheby-ASM(2),(7,5,3,1).

The Chebyshev-accelerated Schwarz schemes are not always the fastest, however. For the 67 pebble case (Fig. 32c), Cheby-Jac(2),(7,5,3,1) is comparable to Cheby-RAS(2),(7,5,3,1) and are the two fastest pMG based preconditioners. However, SEMFEM is significantly faster than the other preconditioners for this case.

A hybrid pMG/SEMFEM approach wherein SEMFEM is used as the coarse grid solver is considered. For $p = 7$, a $(7, 6)$ schedule with 2nd order Chebyshev-accelerated ASM smoothing on the $p = 7$ level and SEMFEM solver on the $p = 6$ level, denoted as Cheby-ASM(2),(7,6) + SEMFEM, is used. Similarly, Cheb-ASM(2),(7,5) + SEMFEM and Cheb-ASM(2),(7,3) + SEMFEM are considered. For $p = 9$, a $(9, 8)$,

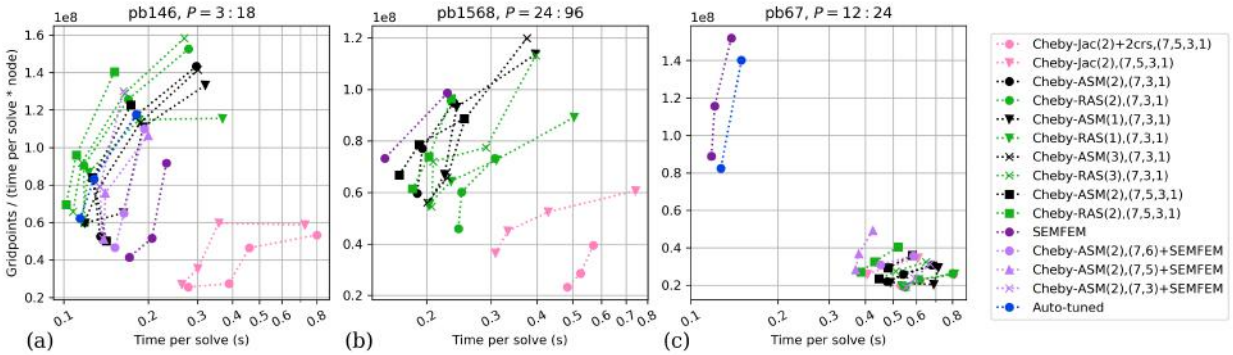


Figure 32: Strong scaling results on Summit for the Navier-Stokes cases of Fig. 31a,b,c.

$(9,7)$, $(9,5)$, and $(9,3)$ hybrid pMG/SEMFEM approach is considered, denoted as Cheb-ASM(2),(9,8) + SEMFEM, Cheb-ASM(2),(9,7) + SEMFEM, Cheb-ASM(2),(9,5) + SEMFEM, and Cheb-ASM(2),(9,3) + SEMFEM, respectively. In the pebble cases shown in Fig. 32a,c, this hybrid approach performs somewhere between the SEMFEM and Cheby-ASM(2),(7,3,1) preconditioners.

Across all cases, with exception to the 67 pebble case, pMG preconditioning with Chebyshev-accelerated ASM or RAS smoothing is the fastest solver or is comparable to SEMFEM. The choice of Chebyshev order and multigrid schedule, moreover, contributes only a modest $\approx$ 10-20% improvement to the overall time-to-solution in most cases, all else being equal. This makes the default Cheby-ASM(2),(7,3,1) or Cheby-ASM(2),(9,5,1) preconditioner reasonably performant. However, in order to avoid the situation encountered in the 67 pebble case, a simple auto-tuner sweep over the preconditioners considered can be used to identify a good preconditioner for the problem.

3.4 Local Fourier Analysis of multigrid and domain decomposition methods

Balancing Domain Decomposition by Constraints (BDDC) is a relatively new technique created by Dohrmann in 2003 [20]. BDDC is an improvement upon balancing domain decomposition (BDD) [45], which requires modifications to be effective for high-order problems, especially for three-dimensional problems. The dual primal version of finite element tearing and interconnecting (FETI-DP) [22] is closely related to BDDC and the two are effectively the same technique, under certain assumptions [46]. In BDDC, we use non-overlapping subdomains with an energy minimization problem to resolve jumps in the values of degrees of freedom over subdomain interfaces.

Local Fourier Analysis (LFA) [4, 79] is a powerful tool for predicting multigrid performance and tuning of multigrid components by examining the spectral radius and/or condition number of the *symbol* of the underlying operator, which enables sharp predictions for large-scale problems. LFA of BDDC was introduced by Brown, He, and MacLachlan [5].

`LFAToolkit.jl` [75], a Julia package for LFA of high-order finite element methods, can be used to analyze multigrid and domain decomposition methods for high-order finite elements. LFAToolkit provides the LFA-predicted convergence factor, given by the maximum spectral radius of the symbol of the operator, provides an estimate for the rate at which error is reduced by repeated application of the multigrid or domain decomposition method as an iterative solver.

The results in Table 2 provide the LFA convergence factor for two-grid high-order p-multigrid for a variety of coarsening rates and orders of Chebyshev smoother. The two-grid convergence factor degrades and as we coarsen more aggressively.

Jacobi and Chebyshev smoothers can provide rather poor LFA convergence factors for p-multigrid when aggressively coarsening by a factor of greater than 2, such as going from 4th order to linear elements in a single step. Using BDDC on single high-order element subdomains as a smoother for a p-multigrid v-cycle with aggressive coarsening can provide better LFA convergence factors.

In Table 3, we see the two-grid convergence factor for p-multigrid with the weighted Dirichlet BDDC smoother for the two-dimensional Laplacian. When compared to Table 2, we can see that the weighted

p_{fine} to p_{coarse}	$k=1$	$k=2$	$k=3$	$k=4$
$p=2$ to $p=1$	0.621	0.252	0.075	0.039
$p=4$ to $p=2$	0.607	0.281	0.085	0.047
$p=4$ to $p=1$	0.768	0.424	0.219	0.127
$p=8$ to $p=4$	0.669	0.278	0.110	0.055
$p=8$ to $p=2$	0.864	0.633	0.456	0.336
$p=8$ to $p=1$	0.956	0.873	0.795	0.730
$p=16$ to $p=8$	0.855	0.613	0.435	0.319
$p=16$ to $p=4$	0.938	0.822	0.719	0.634
$p=16$ to $p=2$	0.976	0.928	0.882	0.842
$p=16$ to $p=1$	0.992	0.975	0.959	0.944

Table 2: Two-grid convergence factor for p-multigrid with Chebyshev smoothing for the 2D Laplacian

p_{fine} to p_{coarse}	$\rho_{\min}$	ω_{opt}
$p=2$ to $p=1$	0.121	0.66
$p=4$ to $p=2$	0.272	0.48
$p=4$ to $p=1$	0.281	0.47
$p=8$ to $p=4$	0.409	0.38
$p=8$ to $p=2$	0.462	0.33
$p=8$ to $p=1$	0.462	0.32
$p=16$ to $p=8$	0.504	0.32
$p=16$ to $p=4$	0.579	0.25
$p=16$ to $p=2$	0.597	0.23
$p=16$ to $p=1$	0.597	0.23

Table 3: Two-grid convergence factor for p-multigrid with Dirichlet BDDC smoother for the 2D Laplacian

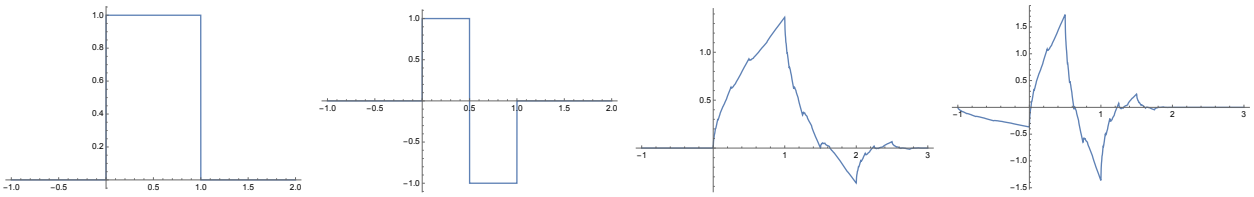
Dirichlet BDDC smoother provides a significantly improved two-grid convergence factor for aggressive coarsening with high-order elements when compared to fourth-order Chebyshev smoothing. Additionally, the weighted Dirichlet BDDC smoother provides analogous two-grid convergence factors when compared to more conservative coarsening strategies with second or third-order Chebyshev smoothing.

This analysis indicates that using Dirichlet BDDC as a smoother for p-multigrid may provide an attractive matrix-free preconditioning strategy.

3.5 Matrix-free wavelet preconditioning

Wavelet analysis was first developed in the early 1990s [15], the orthonormal smooth wavelet was then constructed [14], and the construction method was generalized and reformulated as multi-resolution analysis. Wavelets can be used as an analysis tool to describe mathematically the increment in information needed to go from a coarser approximation to a higher resolution approximation. Several kinds of wavelets have been proposed for various purposes, and the concept of wavelet has been extended to new types of basis functions. But its implementation [68] to an efficient computer code is not straightforward; research is still continuing for concrete problems [16].

Wavelets have provided a new method for decomposing a function or signal, just like a Fourier expansion. Given an orthogonal wavelet function in the continuous space, such as the Haar scaling and wavelet function



(a) Haar ϕ function **(b)** Haar ψ function **(c)** Daubechies ϕ function **(d)** Daubechies ψ function

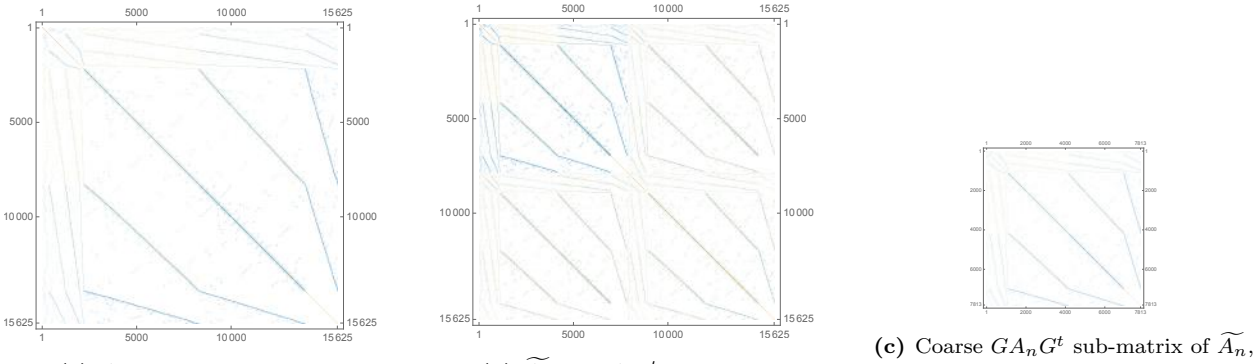
Figure 33: Haar and Daubechies (ϕ) scaling and wavelet functions (ψ).

presented on figures 33a and 33b, or the Daubechies wavelet presented on figures 33c and 33d, there corresponds an orthogonal matrix $W = \begin{pmatrix} G & H \end{pmatrix}^t$, G being the low-pass filter part of the wavelet operator, and H the high-pass filter part, that transforms vectors from the standard basis to the wavelet basis. If x is a vector, its wavelet representation is $\tilde{x} = Wx$, we can also represent two dimension transforms with such orthogonal matrix W: let A_n be a matrix at some *level* n in the standard basis, $\widetilde{A_n}$ is then the representation of A_n in the wavelet basis, given by:

$$\widetilde{A_n} = WA_nW^t = \begin{pmatrix} GA_nG^t & GA_nH^t \\ HA_nG^t & HA_nH^t \end{pmatrix}$$

The goal is to compute the wavelet transform of such operator A_n, the matrix of which is shown in figure 34a, resulting in a matrix shown in figure 34b and to approximate the operator at some next *level* $n+1$, $\widetilde{A_{n+1}}$ as shown in the figure 34c, by:

$$\widetilde{A_{n+1}} \simeq GA_nG^t$$



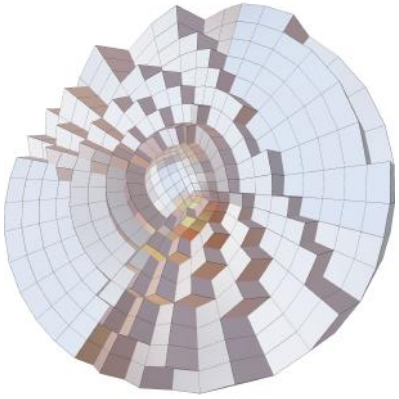
(a) A, size 15625, $\kappa = 470$ **(b)** $\widetilde{A_n} = WAW^t$, size 15625 **(c)** Coarse GA_nG^t sub-matrix of $\widetilde{A_n}$, of size 7813, $\kappa = 155$

Figure 34: 3D problem matrices and their associated conditioning number κ.

Assuming A^{-1} exists, the idea is to apply such wavelet transforms to *simplify* A^{-1} with $\widetilde{A^{-1}}$ whose orthogonal wavelet transformed matrix is given by $\widetilde{A^{-1}} = WA^{-1}W^t = (WAW^t)^{-1} = \tilde{A}^{-1}$ and use it as a preconditioner of a multigrid method.
The multigrid method is very useful in reducing the convergence time for solving systems of algebraic equations. A multigrid scheme consists of three key elements: operator representations at various scales, restriction and interpolation operators. The operator representations at the various scales enable the manipulation of iterative schemes on grids of different scales. The restriction and interpolation matrices are used to convert the residual error from one scale to its adjacent scales. Specifically, a restriction operator projects the residual error from a scale to its adjacent coarser scale, while an interpolation operator maps the residual error from a scale to its adjacent finer scale. The information transfer of the iterative solution among different scales is realized by the use of the restriction and interpolation operators. Wavelets have been widely used in various fields of research and the application of wavelet transform to numerical solver of differential equations has been studied and used [74, 11, 71, 70, 72]. Here we are continuing approaches introduced in [18, 73, 12, 65, 56, 19, 25, 59, 55]

to improve the efficiency of the multigrid method for the CEED BPS3 problem on GPU architectures. We are using wavelets to obtain new coarse grid, interpolation and restriction operators to build a fully matrix-free wavelet multigrid method.

Wavelet BP3 numerical validations. The effects and advantages of the wavelet preconditioner are confirmed with CEED's benchmarks BP3, with the use of different unstructured meshes, such as the ones presented in the figures 35a, 36a and 37a. The order is set to one, as the main objective here is to validate the implementation of the matrix-free wavelet preconditioner. Hypre version 2.24, AMGX version 2.2.0.132-opensource and CUDA 11.6 are used. The tables in figures 35, 36 and 37 present the results on these different input meshes, as well as the validation test problem 38 with variable coefficients. Different smoothing orders for the wavelet multigrid solver results have been reported to emphasis the impact of such parameter on the wavelet V-cycle. Although the number of iterations is not as scalable as in Hypre and AmgX, the values are comparable. This method does not require any matrix assemblies, which results in a short setup time but it does not benefit from the extensive background of AMG algorithms built in the Hypre and AmgX libraries.



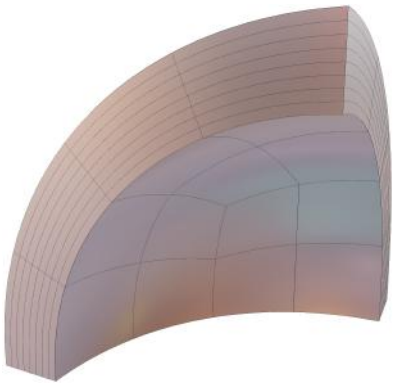
(a) Butterfly mesh

	Number of unknowns	Iterations	Levels
Hypre	11145	74	5
	83345	86	5
	644385	89	7
AmgX	11145	71	3
	83345	80	4
	644385	80	5
WAMG, smoothing order 2	11145	46	12
	83345	59	15
	644385	94	18
WAMG, smoothing order 5	11145	29	12
	83345	39	15
	644385	62	18

Figure 35: CEED BP3 iteration numbers on mesh 35a on one Lassen's NVIDIA Tesla V100

Wavelet BPS3 numerical validations. The results of the p-multigrid with wavelet coarse multigrid preconditioner are presented with CEED's BPS3 benchmarks in tables of figures 39, 40, 41 and 42. The right-hand-side of the problem is however set to a constant coefficient one for these runs. Each table has five different preconditioners:

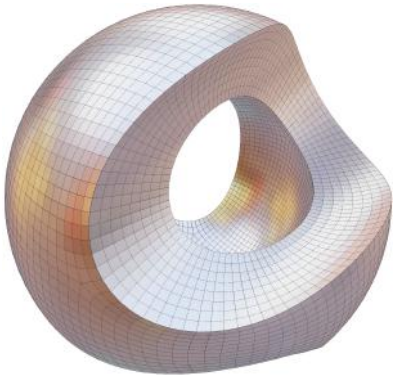
- *Jacobi* with a fast *setup* time, but which does not converge with the ϵ parameter different from one under 500 iterations or with increasingly larger meshes,
- *LORBatch* which uses the Hypre's HypreBoomerAMG preconditioner on the batched low-order-refined matrix introduced in 3.2,
- *MGFAHypre* is a p-multigrid solver with Chebyshev smoothing and Hypre's HypreBoomerAMG as coarse solver,
- *MGWavelets* is a p-multigrid solver with Chebyshev smoothing and partially assembled wavelet-multigrid V-cycle as coarse solver,
- *MGFAWavelets* is the same p-multigrid with Chebyshev smoothing and a fully assembled wavelet-multigrid V-cycle as coarse solver.



(a) Shell mesh

	Number of unknowns	Iterations	Levels
Hypre	1037	92	4
	7161	116	5
	53105	132	6
	408801	147	7
AmgX	1037	88	2
	7161	118	4
	53105	135	5
	408801	149	5
Wavelets-MG, smoothing order 2	1037	55	9
	7161	73	12
	53105	84	15
	408801	94	18
Wavelets-MG, smoothing order 5	1037	35	9
	7161	46	12
	53105	54	15
	408801	65	18

Figure 36: CEED BP3 iteration numbers on mesh 36a on one Lassen's NVIDIA Tesla V100

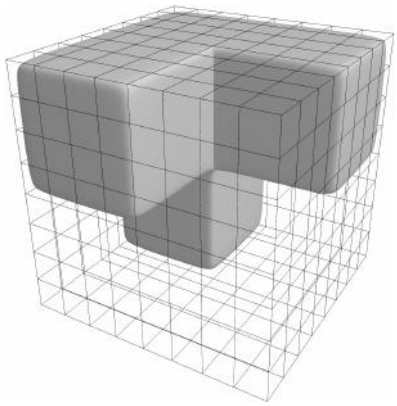


(a) Sculpture mesh

	Number of unknowns	Iterations	Levels
Hypre	28215	25	4
	208335	27	6
	1598751	30	6
AmgX	28215	21	3
	208335	22	4
	1598751	24	4
Wavelets-MG, smoothing order 2	28215	20	14
	208335	27	17
	1598751	43	20
Wavelets-MG, smoothing order 5	28215	15	14
	208335	20	17
	1598751	29	20

Figure 37: CEED BP3 iteration numbers on mesh 37a on one Lassen's NVIDIA Tesla V100

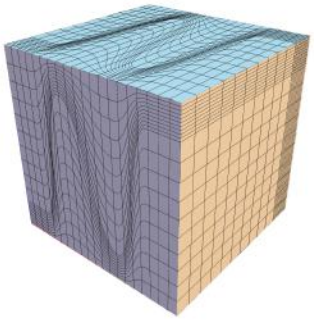
Each table is broken into two sections: one with $\epsilon = 1.0$ and one with $\epsilon = 0.3$. As behavioral differences between these two are discovered, both setup and solution times are provided. MGWavelets typically have the shortest setup time, mostly used to build the p-multigrid finite element space hierarchy. Even if the MGFAWavelets implementation uses completely assembled matrices, the setup time is almost unaffected. Having a entirely matrix-free *MGWavelets* implementation allows to fit 19 millions degrees-of-freedom problem on a single GPU with 16GB of memory. For $\epsilon = 1.0$, the p-multigrid does help with the number of iterations.



(a) Cube with variable mesh region coefficients: $\alpha = 1e+7$ in the presented region, $\beta = 1e-7$ otherwise.

	Number of unknowns	Iterations	Levels
Hypre	4913	16	4
	35937	19	5
	274625	22	6
	2146689	25	7
AmgX	4913	15	2
	35937	16	3
	274625	16	3
	2146689	16	4
Wavelets-MG, smoothing order 2	4913	12	11
	35937	20	14
	274625	33	17
	2146689	56	20
Wavelets-MG, smoothing order 5	4913	8	11
	35937	15	14
	274625	23	17
	2146689	39	20

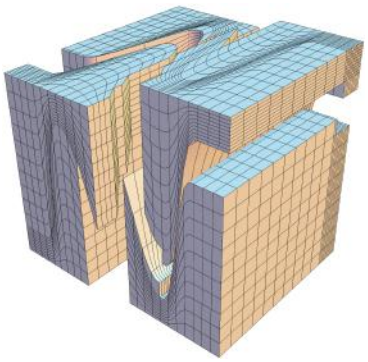
Figure 38: MFEM ex1p iteration numbers with mesh 38a and different coefficients in mesh regions on one Lassen's NVIDIA Tesla V100



(a) Kershaw mesh on 1 GPUs.

	ϵ	MPI	Ndofs	Niters	Tsetup	Tsolve
Jacobi	1.0	1	3M	366	0.03	0.57
LORBatch	1.0	1	3M	22	1.31	0.28
MGFAHypre	1.0	1	3M	9	0.93	0.13
MGWavelets	1.0	1	3M	12	0.59	0.17
MGFAWavelets	1.0	1	3M	11	0.63	0.16
Jacobi	0.3	1	3M	✗	0.03	✗
LORBatch	0.3	1	3M	61	1.32	0.92
MGFAHypre	0.3	1	3M	90	0.93	1.16
MGWavelets	0.3	1	3M	92	0.57	1.18
MGFAWavelets	0.3	1	3M	92	0.64	1.18

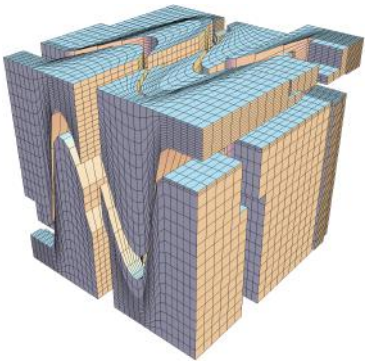
Figure 39: Iteration numbers for CEED serial order 6 BPS3 with mesh 39a, on one Lassen's NVIDIA Tesla V100. Timing are in seconds.



(a) Kershaw mesh on 4 GPUs.

	ϵ	MPI	Ndofs	Niters	Tsetup	Tsolve
Jacobi	0.3	4	3M	✗	0.05	✗
LORBatch	0.3	4	3M	60	2.18	0.75
MGFAHypre	0.3	4	3M	91	0.44	0.99
MGWavelets	0.3	4	3M	91	0.24	0.73
MGFAWavelets	0.3	4	3M	91	0.28	0.75
Jacobi	0.3	4	19M	✗	0.20	✗
LORBatch	0.3	4	19M	✗	✗	✗
MGFAHypre	0.3	4	19M	89	5.87	5.96
MGWavelets	0.3	4	19M	96	3.73	6.98
MGFAWavelets	0.3	4	19M	96	4.04	6.52

Figure 40: Iteration numbers for CEED parallel order 6 BPS3 with mesh 40a, on four Lassen's NVIDIA Tesla V100. Timing are in seconds.



(a) Kershaw mesh on 16 GPUs.

	ϵ	MPI	Ndofs	Niters	Thypre (included)	Tsetup	Tsolve
Jacobi	1.0	16	40M	✗		0.12	✗
LORBatch	1.0	16	40M	23	(4.01)	5.97	0.78
MGFAHypre	1.0	16	40M	11	(0.16)	1.20	0.27
MGWavelets	1.0	16	40M	26		0.72	0.52
MGFAWavelets	1.0	16	40M	25		0.84	0.48
Jacobi	0.3	16	40M	✗		0.12	✗
LORBatch	0.3	16	40M	65	(4.23)	6.27	2.33
MGFAHypre	0.3	16	40M	111	(0.15)	1.22	2.28
MGWavelets	0.3	16	40M	131		0.71	2.15
MGFAWavelets	0.3	16	40M	131		0.82	2.16

Figure 41: Iteration numbers for CEED parallel order 6 BPS3 with mesh 41a, on sixteen Lassen's NVIDIA Tesla V100. The overall *Tsetup* setup time includes Hypre's *Thypre* setup time. Timing are in seconds.

(a) Kershaw mesh on 128 GPUs.

	ϵ	MPI	Ndofs	Niters	Tsetup	Tsolve
Jacobi	1.0	128	158M	✗	0.06	✗
LORBatch	1.0	128	158M	23	3.96	0.69
MGFAHypre	1.0	128	158M	13	0.75	0.25
MGWavelets	1.0	128	158M	33	0.45	0.54
MGFAWavelets	1.0	128	158M	31	0.49	0.51
Jacobi	0.3	128	158M	✗	0.09	✗
LORBatch	0.3	128	158M	68	4.10	2.11
MGFAHypre	0.3	128	158M	126	0.89	2.54
MGWavelets	0.3	128	158M	137	0.44	2.30
MGFAWavelets	0.3	128	158M	137	0.53	2.26

Figure 42: Iteration numbers for CEED parallel order 6 BPS3 with mesh 42a, on 128 Lassen's NVIDIA Tesla V100. Timing are in seconds.

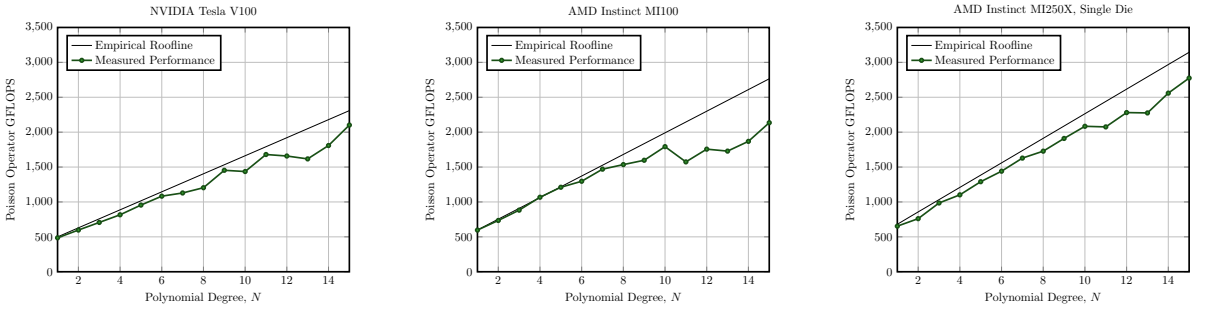
4. PERFORMANCE IMPROVEMENTS FOR FRONTIER ON MI250X GPUS

4.1 hipBone results on Summit, Spock, and Crusher

Computational Tests. This section describes the computational studies we performed on three different machines at the Oak Ridge Leadership Compute Facility (OLCF): Summit, Spock, and Crusher. Summit[2] is an IBM system, each node consisting of a dual socket configuration of two IBM POWER9 CPUs, six NVIDIA Tesla V100 GPUs, and one dual-rail 100Gbps Mellanox Connect X-5 EDR InifniBand network card. Spock[3] is an HPE system, each node consisting of a single socket AMD EPYC 7662 CPU, four AMD Instinct MI100 GPUs, and one HPE Slingshot 100Gbps network card. Finally, Crusher[4] is an HPE Cray EX supercomputer system serving as a Frontier early-access system, each node consisting of a single socket optimized 3rd Gen EPYC 64 core processor, four AMD Instinct MI250X accelerators, and four HPE Slingshot 200Gbps network interfaces.

On Summit, we use GCC v9.1.0 as our C++ compiler, CUDA v11.0.3 to JIT compile device kernels, and Spectrum-MPI v10.4.0.3 as the MPI implementation. On Spock, we use GCC v11.2.0 as our C++ compiler, ROCm v4.5.0 to JIT compile HIP kernels, and Cray-MPICH v8.1.12 as the MPI implementation. Finally, on Crusher we GCC v11.2.0 as our C++ compiler, ROCm v4.5.2 to JIT compile HIP kernels, and Cray-MPICH v8.1.12 as the MPI implementation. For each of these systems, we perform several performance and scaling studies and present their results below.

Poisson Operator Performance. Our first test measures the performance of the screened Poisson operator kernel on each of the three different GPU accelerators considered, as its performance is critical to the overall performance of the benchmark. To isolate the performance on each accelerator, we use a single NVIDIA Tesla V100, a single AMD MI100, and one of the two Graphics Compute Die (GCD) on the AMD Instinct MI250X Multi-Chiplet Module (MCM). It is important to note that while each GCD in the MI250X presents as a distinct GPU to the operating system, the two GCDs on the MI250X share the total power budget on the full MI250X MCM. Testing on a single GCD, therefore, allows for the GCD to potentially use more than half of the overall power budget of the MI250X, and operate at high clock frequencies than one would observe when running the same workload on both GCDs simultaneously. For the operator tests in this section, however, in which performance is predominately bound by HBM streaming rates, we have not observed any significant performance difference between kernels executed on a single GCD of the MI250X, or both GCDs in the MCM simultaneously.



(a) Performance of Poisson operator in hipBone on a single NVIDIA Tesla V100 shown with empirical roofline model. Roofline model uses an empirical peak streaming rate of $B = 820$ GB/s.

(b) Performance of Poisson operator in hipBone on a single AMD Instinct MI100 shown with empirical roofline model. Roofline model uses an empirical peak streaming rate of $B = 952$ GB/s.

(c) Performance of Poisson operator in hipBone on a single GCD of AMD Instinct MI250X shown with empirical roofline model. Roofline model uses an empirical peak streaming rate of $B = 1117$ GB/s.

Figure 43: Performance test of Poisson operator in hipBone for polynomial degrees $N = 1, \ldots, 15$. Results for each GPU accelerator are shown with empirically calibrated roofline model given in (2).

[2] https://docs.olcf.ornl.gov/systems/summit_user_guide.html
[3] https://docs.olcf.ornl.gov/systems/spock_quick_start_guide.html
[4] https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

We measure the performance of the Poisson operator kernel by time-averaging 50 back-to-back invocations of the kernel for each polynomial degree $N = 1, \ldots, 15$. For each degree, we use a fixed mesh of elements, chosen large enough so that the number of DOFs in the mesh is approximately $N_G \approx 40$ million. This helps to minimize effects such as kernel launch latency, threadblock scheduling latency and tail effects, and core clock variations due to temporary boosting. We also compute, for each device, an empirically calibrated roofline using the model in equation (2). We use as the peak bandwidth, B in the roofline model, not the quoted theoretical peak of each device's HBM memory, but rather an empirically measured peak streaming rate. To measure this rate, we use a simple streaming kernel in which each thread in the threadblocks only reads 8 FP64 values from main memory, and writes back one FP64 value. This 8:1 read-write ratio closely matches the data movement of the Poisson operator kernel, but is also idealized in that there is no unstructured load, no data re-use in cache, and the kernel uses a uniform threadblock size. We compute an asymptotic streaming rate for this kernel using the methodology described in [10], finding streaming rates of $B = 810$ GB/s for the NVIDIA Tesla V100, $B = 982$ GB/s for the AMD Instinct MI100, and $B = 1117$ GB/s for a single GCD of the AMD Instinct MI250X. We do not treat these streaming rates, nor the roofline model itself, as absolute bounds on the performance of the Poisson operator kernel. Rather, we use this roofline model as guidance for what one would expect the performance of the operator kernel to be, if it were to read and write data to main memory as efficiently as a pure streaming kernel.

We show the results of the Poisson operator performance test in Figure 43. We show the performance in GFLOPS of the operator kernel for each $N = 1, \ldots, 15$ on the NVIDIA Tesla V100, AMD Instinct MI100, and a single GCD of the AMD Instinct MI250X in Figures 43a, 43b, and 43c, respectively. For each device, we see that the measured operator performance closely matches the empirical roofline model for $N < 9$, indicating near-optimal performance of the 3D threadblock kernel described above. At $N = 9$ and beyond, wherein the kernel implementation switches to the 2D threadblock algorithm, performance becomes more variable, with some degrees falling significantly below the roofline. This is especially noticeable in the performance on the AMD Instinct MI100 which sees a performance dip at $N = 11$. While performance continues to climb after $N = 11$, the gap between the measured performance and the roofline is never fully recovered. Despite the departure from the roofline model on each GPU, all the devices achieve the highest GFLOPS at the highest polynomial degree $N = 15$, with the NVIDIA Tesla V100 achieving 2101.4 GFLOPS, the AMD Instinct MI100 achieving 2135.2 GFLOPS, and a single GCD of the AMD Instinct MI250X achieving 2774.9 GFLOPS.

The inefficiencies that appear in the Poisson operator kernel for large N may be due to the combination of high register pressure, masked threads, and large threadblocks. To gain some insight into these effects, we can consider some important metrics regarding the occupancy of the Poisson operator kernel. For each polynomial degree, N, the threadblock size of the 2D threadblock operator kernel is $(N + 1)^2$. This threadblock is then divided into several 32-lane-wide 'warps' on the NVIDIA Tesla V100, and 64-lane-wide 'wavefronts' on the AMD GPUs. Any additional threads in excess of the threadblock size are simply masked out when executing on the hardware. Each warp/wavefront then has access to a number of vector registers. On the NVIDIA Tesla V100, a warp is able to reserve a maximum of 255 128B vector registers (each register being 32 lanes of 4B words) within a Stream Multiprocessor (SM) on the GPU. Similarly, on the AMD GPUs each wavefront is able to reserve a maximum of 256 256B vector registers (each register being 64 lanes of 4B words) in each Compute Unit (CU). While terminology between the GPU vendors differ, they share the property that warps/wavefronts reserving large amounts of register space can substantially affect the number of warps/waves which can be simultaneously active on an SM/CU. Each GPU's SMs/CUs have vector register files that are divided evenly between four SIMD units, with both the NVIDIA Tesla V100 SMs and the AMD Instinct MI100 CUs having a total of 256KB of vector register space, and the AMD Instinct MI250X CUs having 512KB of vector register space. The 'occupancy' of a kernel then refers to the average number of warps/wavefronts which are active on each SM/CU during the kernel's execution. Other constraints, such as the requirement that all warps/wavefronts from a given threadblock must occupy the same SM/CU can also restrict the occupancy of a kernel.

Comparing the occupancy metrics of the operator kernel on the AMD Instinct MI100 in Table 4 to the achieved performance in Figure 43b, we see that the drop in performance observed at $N = 11$ corresponds to a drop in occupancy on each CU. The occupancy drop itself is caused by the threadblock size of 144 requiring one additional wavefront at the same register usage compared to $N = 10$. This third wavefront will also unfortunately have 48 of its lanes masked while executing. The subsequent $N = 12$ kernel on the AMD Instinct MI100 achieves the same occupancy as $N = 11$, and the performance sees a increase which

		N						
		9	10	11	12	13	14	15
	Block size, $(N+1)^2$	100	121	144	169	196	225	256
NVIDIA Tesla V100	Warps/Block	4	4	5	6	7	8	8
	Registers/Warp	80	80	104	90	82	102	100
	Occupancy (Warps/SM)	24	24	15	18	14	16	16
	Elements/SM	6	6	3	3	2	2	2
AMD Instinct MI100	Wavefronts/Block	2	2	3	3	4	4	4
	Registers/Wavefront	113	125	125	125	125	125	125
	Occupancy (Wavefronts/CU)	8	8	6	6	8	8	8
	Elements/CU	4	4	2	2	2	2	2
AMD Instinct MI250X	Wavefronts/Block	2	2	3	3	4	4	4
	Registers/Wavefront	126	124	124	124	124	124	124
	Occupancy (Wavefronts/CU)	16	16	12	12	16	16	16
	Elements/CU	8	8	4	4	4	4	4

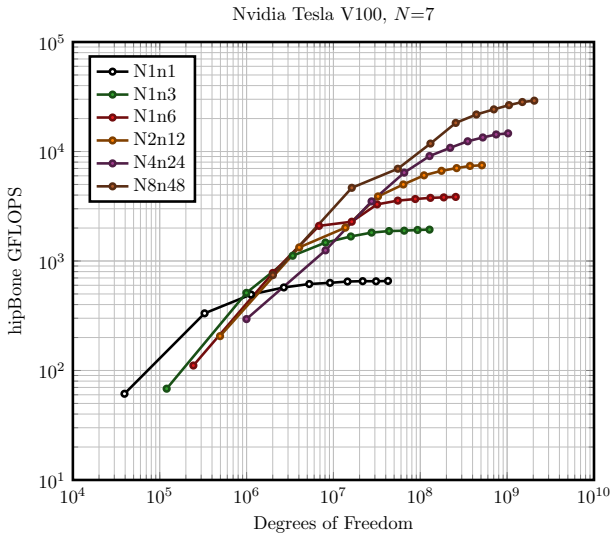
Table 4: Occupancy metrics for 2D threadblock algorithm for the Poisson operator kernel in hipBone. The number of warps/wavefronts must be sufficiently large to cover the threadblock size. Each warp/wavefront then uses some number of vector registers, which is the primary factor limiting SM/CU occupancy. The occupancy in terms of warps/SM or wavefronts/CU is listed for each kernel, along with the occupancy in terms of elements per SM/CU.

follows the trend of the roofline model, but shares the same inefficiencies as $N = 11$, albeit with less masked lanes. At $N = 13$ and onward, the kernel requires four wavefronts per threadblock and occupancy in terms of wavefronts/CU is recovered. The kernels, however, now do more computation with more wavefronts synchronizing their activity to coordinate accesses to shared memory. The performance continues to track upwards, but the inefficiency compared to the roofline model is never recovered.

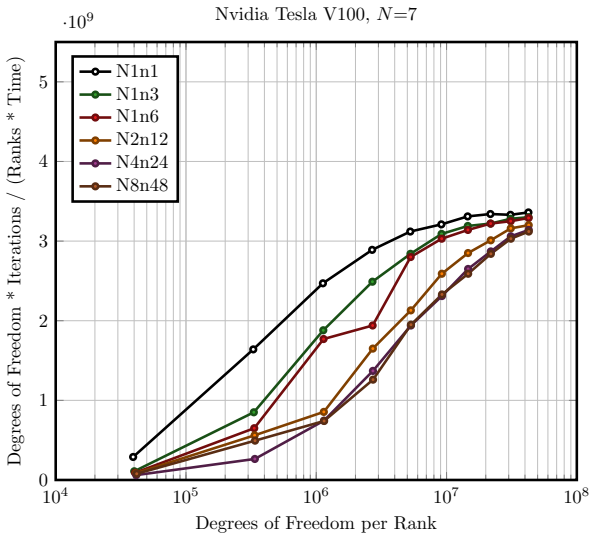
The occupancy metrics of the operator kernel on the AMD Instinct MI250X in Table 4 follows the same progression as the AMD Instinct MI100, but overall occupancy is immediately doubled due to the 2x larger register file size per CU. We see a similar qualitative trend in the achieved performance of the operator kernel on the AMD Instinct MI250X in Figure 43c as on the AMD Instinct MI100, but the dips in performance are less pronounced and performance stays closer to the empirical roofline model for high degrees. This efficiency improvement is likely due to the CU occupancy doubling.

Finally, comparing the performance trend of the NVIDIA Tesla V100 in Figure 43a to the occupancy metrics in Table 4, the effects of different occupancies and registers per warp between kernels are more difficult to reason about. Overall, the NVIDIA compiler uses significantly fewer registers per warp compared with the AMD compiler, allowing for a much higher occupancy per SM, especially at the moderately high degrees of $N = 9$ and 10. The occupancy decrease at $N = 11$, however, appears to correspond to an increase in relative efficiency, though the occupancy increase at $N = 12$ does not. The different number of registers per warp between these two kernels makes them difficult to compare, however, since the differences in the assembly generated by the compiler may have additional impacts.

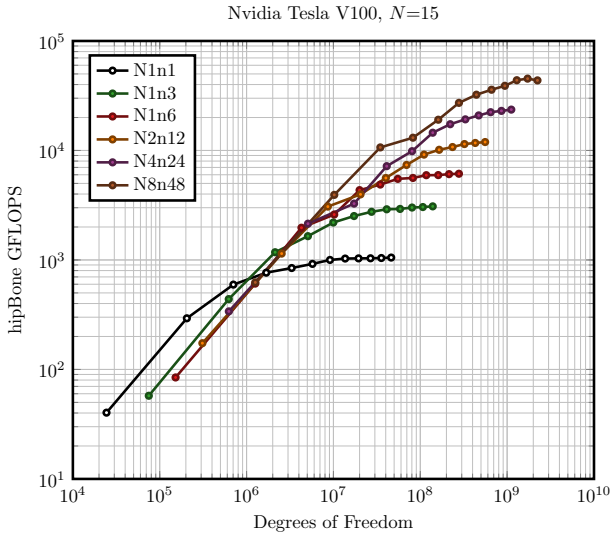
Scaling. Overall performance of hipBone on a single GPU device is almost completely determined by the streaming efficiency of the accelerator and the performance of the Poisson operator kernel, with latency effects such as kernel launch latency and host-to-device synchronization latency being important factors for small problem sizes. The design focus in hipBone, however, is to scale efficiently to many GPU devices by leveraging GPU-Direct RDMA technologies in GPU-Aware MPI libraries, and aggressively hiding communication times behind local computation. To demonstrate the efficacy of this design we present in this section a series of performance tests of the full hipBone benchmark, rather than simply the Poisson kernel in isolation as presented above. We scale hipBone to several MPI ranks, each using a GPU accelerator, on the ORNL Summit, Spock, and Crusher clusters, sweeping over a variety of problem sizes on several GPUs/nodes.



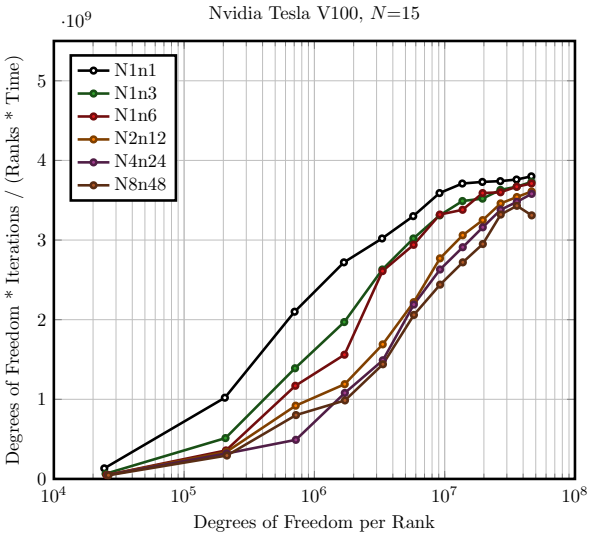
(a) HipBone FOM in GFLOPS on Summit for a variety of problem sizes using polynomial degree $N = 7$.



(b) Throughput, in terms of DOFS*Iterations/(Ranks*Time), of hipBone on Summit over a variety of problem sizes using polynomial degree $N = 7$.

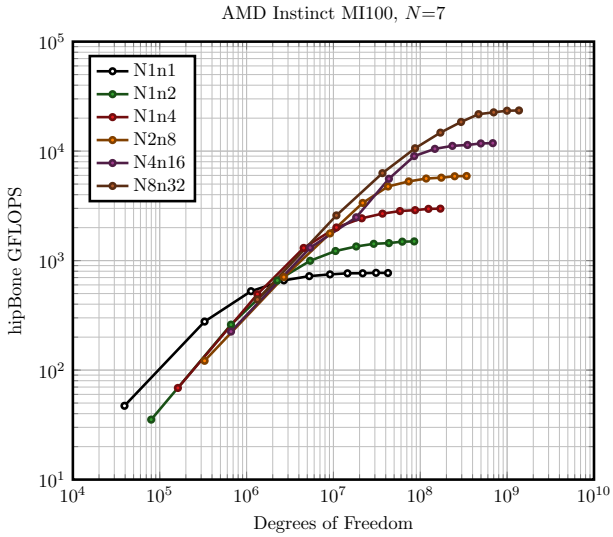


(c) HipBone FOM in GFLOPS on Summit for a variety of problem sizes using polynomial degree $N = 15$.

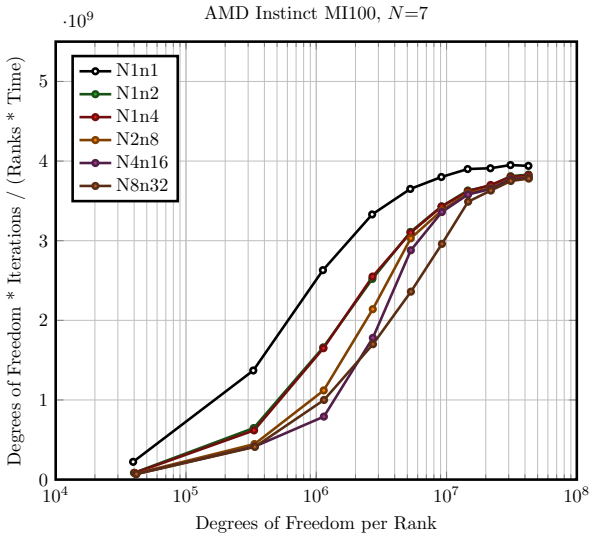


(d) Throughput, in terms of DOFS*Iterations/(Ranks*Time), of hipBone on Summit over a variety of problem sizes using polynomial degree $N = 15$.

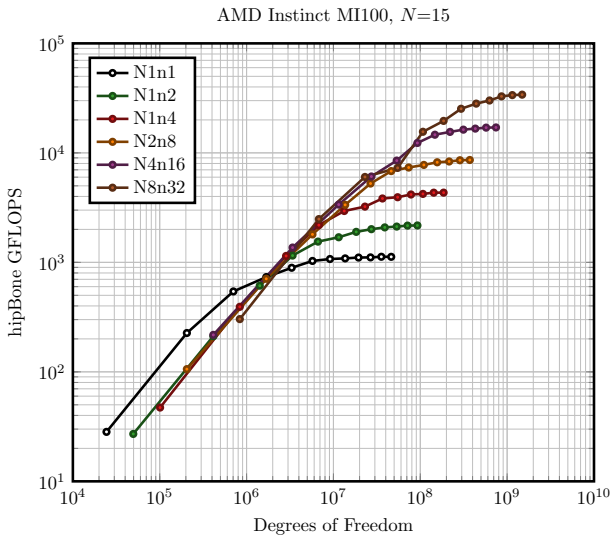
Figure 44: Performance of full hipBone benchmark on the ORNL Summit cluster using NVIDIA Tesla V100 GPUs. Performance is measured over a variety of problem sizes, on 1 to 48 MPI ranks, each utilizing a single NVIDIA Tesla V100.



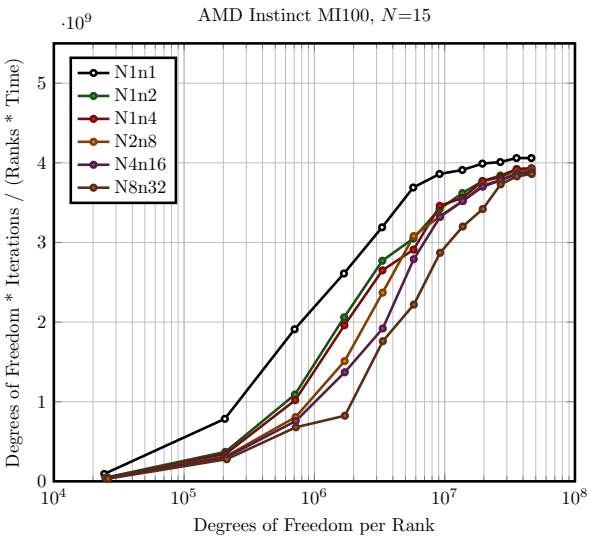
(a) HipBone FOM in GFLOPS on Spock for a variety of problem sizes using polynomial degree $N = 7$.



(b) Throughput, in terms of DOFS*Iterations/(Ranks*Time), of hipBone on Spock over a variety of problem sizes using polynomial degree $N = 7$.

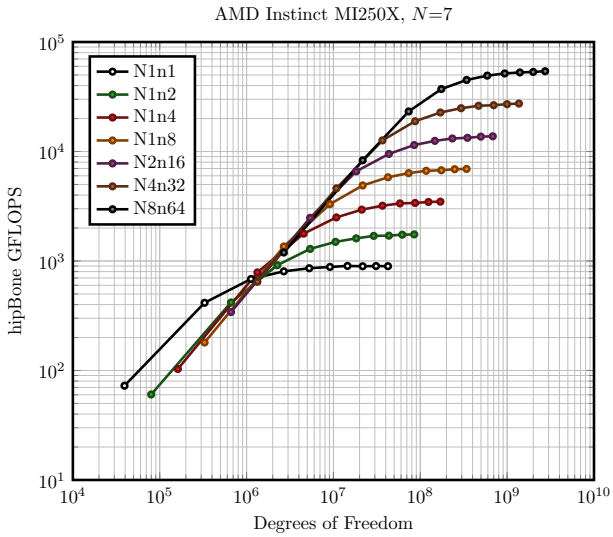


(c) HipBone FOM in GFLOPS on Spock for a variety of problem sizes using polynomial degree $N = 15$.

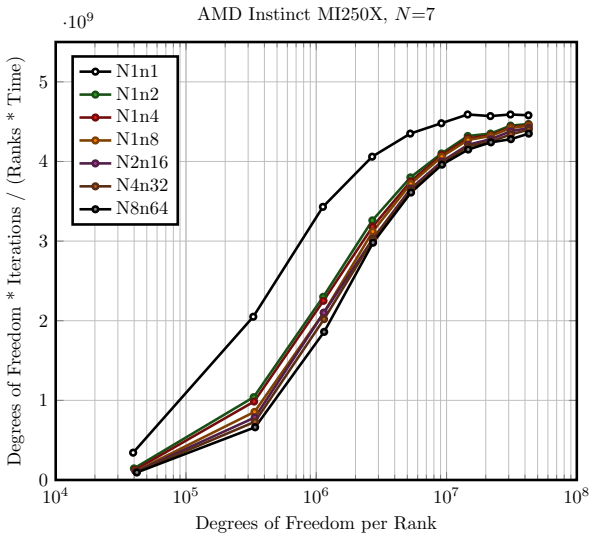


(d) Throughput, in terms of DOFS*Iterations/(Ranks*Time), of hipBone on Spock over a variety of problem sizes using polynomial degree $N = 15$.

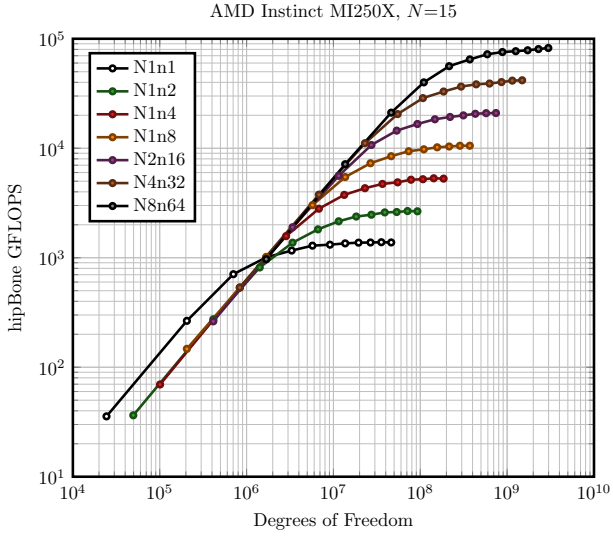
Figure 45: Performance of full hipBone benchmark on the ORNL Spock cluster using AMD Instinct MI100 GPUs. Performance is measured over a variety of problem sizes, on 1 to 32 MPI ranks, each utilizing a single AMD Instinct MI100.



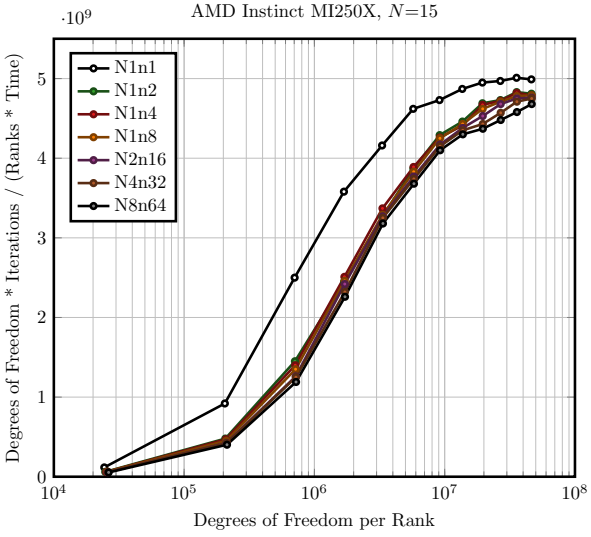
(a) HipBone FOM in GFLOPS on Crusher for a variety of problem sizes using polynomial degree $N = 7$.



(b) Throughput, in terms of DOFS*Iterations/(Ranks*Time), of hipBone on Crusher over a variety of problem sizes using polynomial degree $N = 7$.



(c) HipBone FOM in GFLOPS on Crusher for a variety of problem sizes using polynomial degree $N = 15$.



(d) Throughput, in terms of DOFS*Iterations/(Ranks*Time), of hipBone on Crusher over a variety of problem sizes using polynomial degree $N = 15$.

Figure 46: Performance of full hipBone benchmark on the ORNL Crusher cluster using AMD Instinct MI250X GPUs. Performance is measured over a variety of problem sizes, on 1 to 64 MPI ranks, each utilizing a single GCD of an AMD Instinct MI250x.

We restrict our attention to the two polynomial degrees, $N = 7$ and $N = 15$, for the scaling tests in this section. The choice of degree $N = 15$ is of course of interest for hipBone as a benchmark, as this degree maximizes the GFLOPS of the operator kernel, and thus maximizes the overall FOM. For other polynomial degrees N there will be little qualitative difference in the scaling behavior of hipBone for sufficiently large global meshes. Once there is enough local work in the Poisson operator kernels to completely hide the MPI communication time, performance (as in the single GPU case) is bound only by the asymptotic streaming rate of the GPU devices, which is not appreciably affected by N, and the streaming rate of the Poisson operator kernel which, as demonstrated in the tests above, is consistently near the roofline streaming rate for all N. For smaller problems, however, there can be differences between the scaling behaviors of different degrees due to the variety of different latency effects. We therefore include degree $N = 7$ in our scaling tests, which uses the 3D threadblock algorithm for the Poisson operator kernel, so that any qualitative difference in scaling behavior can be observed.

For each test, we sweep through several global mesh sizes, ranging from the very small ≈ 2500 degrees of freedom per MPI rank, to the very large $\approx 45M$ degrees of freedom per MPI rank. In this way, we collect performance data from both the extremely strong-scaled as well as weak-scaled regimes. On each cluster, we perform tests on an increasing number of GPUs, with small numbers of GPUs initially communicating entirely on-node using fast GPU-GPU data links, then over the network with GPU-Direct RDMA. Due to the different node architectures between the clusters, namely Summit nodes consisting of six GPU devices, Spock nodes consisting of four GPUs, and Crusher nodes consisting of four accelerators which appear to the OS as eight GPU devices, we scale the number of GPUs to eventually utilize a single full node, and continue scaling by using multiple full nodes. We label the configuration of each test as Nxny to indicate the execution was on x nodes of the cluster, using y total GPU devices.

We show the results of the scaling tests on the ORNL Summit, Spock, and Crusher clusters in Figures 44, 45, and 46, respectively. In each figure, we show the FOM of the hipBone application, given in GFLOPS, for $N = 7$ and 15 for each problem size.

It is difficult to immediately see how efficiently hipBone strong/weak scales from the plots of the FOM alone. One clear observation, however, is the significantly higher single rank performance of hipBone on small problem sizes. This can be attributed to some latency effects being completely avoided for single rank execution, as no MPI communication is required at all. Indeed, recalling the timeline of the Poisson operator application in Figure 9, we see that in addition to avoiding MPI latency itself, when no communication is required the two instances of host-device synchronization after packing data buffers for MPI can be skipped. Avoiding these significant sources of latency leads to better single rank performance for latency-sensitive problem sizes.

To obtain a more useful visualization of the scaling behavior of hipBone, we also show in each figure plots of the performance in terms of a 'throughput' metric, similar to that used in [23], defined as

$$\text{Throughput} = \frac{\text{DOFs} * \text{CG Iterations}}{\text{Number of Ranks} * \text{Time}}. \tag{20}$$

This throughput metric essentially normalizes the FOM measurements by computing a rate-of-work. As the benchmark is predominately streaming bandwidth limited, and the time taken to stream data scales linearly with amount of data moved, the work done by each CG iteration therefore scales linearly with the number of degrees of freedom, N_G. By multiplying the global problem size with the number of CG iterations done (which is a fixed 100 iterations), the numerator of (20) encodes a measure of the amount of work performed. The denominator of (20) then encodes an amount of computational resources by multiplying the number of process, each with a separate GPU, by the wall-clock time required to complete the CG iterations.

In each of the Figures 44, 45, and 46, we show the throughput metric of each scaling test plotted against a normalized DOFs-per-rank measure. Were all tests to weak scale perfectly in all regimes, the lines on these plots would collapse to a single curve, implying the same throughput per rank can be achieved even while increasing the number of compute resources (ranks), at every problem size. This is not what is typically seen in practice, as the smaller problem sizes typically under-saturate the compute resources and communication overheads lead to lower throughput as the problem is weak-scaled. At the very large problem sizes, however, we observe a clustering of throughputs near their peak values for all numbers of ranks. This regime demonstrates the ideal weak scaling regime in which the compute resources are saturated and MPI communication is effectively hidden.

	Ranks	Peak FOM (GFLOPS)	FOM per Rank (GFLOPS)
Summit, NVIDIA Tesla V100	1	1051.5	1051.5
	3	3086.1	1028.7
	6	6129.7	1021.6
	12	11935.5	994.6
	24	23611.4	983.8
	48	45294.6	943.6
Spock, AMD Instinct MI100	1	1124.8	1124.8
	2	2172.5	1086.3
	4	4333.0	1083.3
	8	8609.5	1076.2
	16	17065.0	1066.6
	32	34011.7	1062.9
Crusher, AMD Instinct MI250X	1	1386.1	1386.1
	2	2669.6	1334.8
	4	5316.9	1329.2
	8	10548.8	1318.6
	16	20968.0	1310.5
	32	41779.4	1305.6
	64	82375.7	1287.1

Table 5: Summary of peak FOM in GFLOPS recorded on several multi-GPU runs of hipBone on ORNL Summit, Spock, and Crusher clusters. The far right column shows the average FOM in GFLOPS per MPI rank for each run.

Examining the scaling results for Summit and Spock in Figures 44 and 45, respectively, we see a fair amount of variability in throughputs in the latency-sensitive strong-scaled regimes. This is somewhat expected, as we do no averaging of repeated runs on any of the clusters, and noise or jitter in communication latency, especially in multi-node tests, is normal. We see as well in the throughput plots a distinct separation of throughput curves for the multi-GPU runs which communicate entirely on-node with fast GPU-GPU links compared to the multi-node runs. Indeed, throughput on both clusters drops in the strong-scaled regime once communication travels over the network, with the throughput on Summit remaining fairly stable as the node count is increased to eight, and the throughput on the Spock cluster appearing to take additional impacts as node count increases. Comparatively, the throughput data from the Crusher cluster in Figure 46 indicates excellent scalability of hipBone to multiple MI250X GPUs. The tight clustering of all the multi-GPU throughput curves indicates very little additional latency and communication overheads as the number of ranks is increased, even to multi-node.

In the weak scaled regime of each test, we see excellent weak scaling efficiency even compared to the single rank FOM on each GPU. We list the peak FOM of the degree $N = 15$ tests in Table 5 wherein we see that when weak-scaled we observe 943.6 GFLOPS or higher on each NVIDIA Tesla V100, 1062.8 GFLOPS or higher on each AMD Instinct MI100, and 1287.1 GFLOPS or higher on each GCD of AMD Instinct MI250X. Comparing to other GPU performance values for NekBone in the literature, [37] used a version of NekBone with a native CUDA Poisson operator kernel to report ≈ 410 GFLOPS on a single NVIDIA Tesla V100 at degree $N = 9$. Figure 44a shows our hipBone benchmark exceeding this FLOP rate at the lower polynomial degree $N = 7$, achieving 657.6 GFLOPS a single NVIDIA Tesla V100 despite the lower arithmetic intensity. Performance on an NVIDIA Tesla K20X was also reported by [26] using an OpenACC version of NekBone with a CUDA Fortran Poisson operator kernel to achieve 78.2 GFLOPS at degree $N = 15$, while hipBone achieves a FLOP rate over 13 times higher the same degree on a single NVIDIA Tesla V100, despite an only 3.6x increase in peak memory bandwidth of the GPU. [26] also reported significant communication overhead which effected weak scaling. Using Table 5, we see that the NVIDIA Tesla V100 has weak scaling efficiency from 1 to 48 GPUs of 89.7%, while the AMD Instinct MI100 observes

a 94.5% weak scaling efficiency from 1 to 32 GPUs, and the AMD Instinct MI250X achieves 92.9% efficiency scaling from 1 to 64 GCDs. Considering just the scaling efficiency from two to eight nodes on each cluster, the NVIDIA Tesla V100 achieves a 94.9% scaling efficiency, while both AMD GPUs observe greater than 98% efficiency. This indicates that when sufficiently weak scaled, our optimizations in hipBone are very effective at hiding MPI communication overheads.

4.2 NekRS kernel performance: MI250X vs. A100

Figure 47 shows the performance of NekRS's critical kernels on Crusher/MI250X (1 GCD) vs ThetaGPU/A100. The kernels, `AxHelm`, `FDM`, and `advSub`, represent the computation of Helmholtz, fast diagonalization method, and advection operators and `gsLocal` represents the local gather-scatter operation. We observe performance changes depending on the polynomial order, N, and the precision, FP64/FP32. (FP32 is used only in some of the preconditioner subproblems.) We initially had poor performance for `AxHelm` FP32 but subsequently made significant improvements by removing padding in the `SMEM` array. Figure 47, top left and top right, demonstrates reasonable `AxHelm` performance in GFLOPS/s between MI250X and A100 after these issues were resolved. The speedup, A100/MI250X(1 GCD), remains consistent with 0.87–0.92 for `AxHelm` (FP64) and 0.90–0.94 for `AxHelm` (FP32), for varying $N = 5, 7, 9$.

However, Figure 47, center left and center right, shows relatively poor performance in `FDM` FP32 and `advSub`. MI250X(1 GCD) shows a significant slowdown with a speedup of only ∼0.6 for `FDM` FP32 with $N = 7$ and $N = 9$, compared to the A100, while having a better speedup of 0.81 with $N = 5$. We observe a similar poor performance in `advSub` with a speedup of ∼0.5 with $N = 5$ and $N = 9$, while a speedup of 0.81 is realized with $N = 7$.

In addition to `FDM` and `advSub`, we also identified relatively poor performance in the local gather-scatter kernel `gsLocal`. Figure 47, bottom, has a speedup of 0.48–0.59 with $N = 5, 7$ for `gsLocal` FP32, while a speedup of 0.79 is realized with $N = 9$. As our pressure-solve preconditioners are based on p-multigrid and mixed precision, it is critical to have consistent performance behavior across a range of N for both FP32 and FP64. We are looking into these kernels to identify the critical issues.

Challenges: We summarize here the outstanding issues on Crusher MI250X that are under discussion with the libParanumal and AMD teams.

- Poor `FDM` FP32 $N = 7, 9$ performance (0.6 × to A100)
- Poor `advSub` $N = 5, 9$ performance (0.5 × to A100)
- Poor `gsLocal` FP32 $N = 7, 9$ performance (0.5–0.6 × to A100)

TurbPipePeriodic, GPU=1, $E = 7840$, $N = 7$, $n/gpu = 2.6M$			
System	**Crusher (MI250X)** time(s)	**ThetaGPU (A100)** time (s)	**MI250X/A100** speedup
total solve	2.54e+1	1.80e+1	71%
makef	7.59e+0	5.57e+0	73%
velocitySolve	5.03e+0	4.01e+0	80%
pressureSolve	1.24e+1	7.96e+0	64%
preconditioner			
ASM smoother	7.48e+0	4.27e+0	57%
coarse grid	1.19e+0	1.05e+0	88%
projection	5.30e-1	3.85e-1	73%

Table 6: NekRS performance breakdown on Crusher vs ThetaGPU.

NekRS uses mixed precision in the preconditioner for the pressure-Poisson problem. In particular, the Schwarz-based solvers use fast-diagonalization to efficiently solve the local 3D Poisson subproblems on each subdomain (which comprises an extension of each tensor-product-based spectral element). The FDM kernel consists of a tensor-contraction in each direction, followed by pointwise inversion (in the wavespace of the discrete Poisson operator), followed by another set of three tensor contractions. Although superficially similar to the classic CEED BK1 kernel operation the FDM kernel uses a an element specific small matrix for each

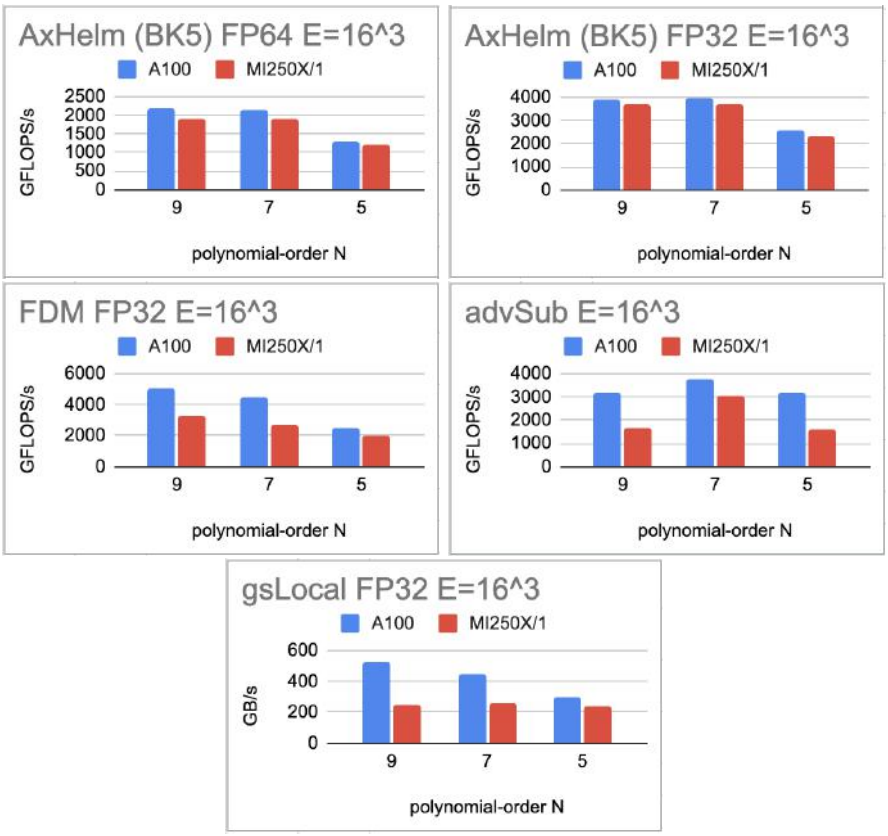


Figure 47: NekRS kernel performance on a single GPU of Crusher/MI250X vs ThetaGPU/A100. ($E = 4096$, $N = 5, 7, 9$).

tensor contraction in each direction. This renders previous highly optimized BK1 kernels less effective because of the reduced operator reuse. The initial implementation sustained up to 5 TFLOPS (FP32) on the NVIDIA A100, but less than 3 TFLOPS on the MI250X (and even worse for even-order expansions). As noted earlier, other FP64 kernels did not show such a discrepancy between the A100 and MI250X. Through extensive testing and tuning by Tim Warburton, the numbers for the FDM kernel were improved dramatically for both platforms, with the A100 reaching 7–8 TFLOPS (FP32) and a single GCD of the MI250X reaching around 5 TFLOPS (FP32) over the same range of p. Unfortunately, because of compiler and architectural differences, out of 14 new kernels developed, tuned, and tested for this operation the most performant kernel is not the same on the two architectures.

4.3 libCEED on Crusher

With the MI250x, we have the first AMD GPU with hardware support for double precision atomic addition. This is crucial for the `hip-gen` backend, which, like the `cuda-gen` backend from which it was ported, uses atomic adds during the transpose element restriction (gather) operation. In CEED-MS36 [38], we investigated the potential impact of not having hardware support for these atomic operations on an MI50 GPU. The left plot of Figure 48 shows the speedup in the BP3 (diffusion) MFEM benchmark when adding the `-munsafe-fp-atomics` flag to the `hiprtc` calls for compiling `hip-gen` kernels. This flag is required to emit the FP64 atomic add instructions in the resulting assembly code. The effects of this change on its own is modest; fourth order basis functions even experienced noticeable slowdown. However, we also consider the importance of adding launch bounds to the operator kernels, which is shown in the middle plot. When combining the launch bounds with the atomic adds (right plot), the overall effect is greater than either change individually. After making the improvements for MI250x performance shown in Figure 48, we obtain the performance seen

FDM FP32 Kernel Performance (GFLOPS)				
	A100	**MI250X**	**A100**	**MI250X**
p	pre-tune	pre-tune	post-tune	post-tune
3	1542	1032	2731	2774
4	2362	575	3735	3251
5	2835	2372	4352	4151
6	3130	653	5147	4775
7	2833	2849	5572	4346
8	4039	630	6866	5433
9	4979	2723	7044	5029
10	4745	621	8200	5334
11	5167	2375	8232	4742
12	4660	549	8294	5072

Table 7: Performance for the FP32 fast-diagonalization-method kernel, pre- and post-tuning, on a single NVIDIA A100 and a single GCD of an AMD MI250X. The number of degrees-of-freedom is $n = 2.5$M in each case.

in Figure 49 for the MFEM-libCEED diffusion benchmark on the Crusher early-access system.

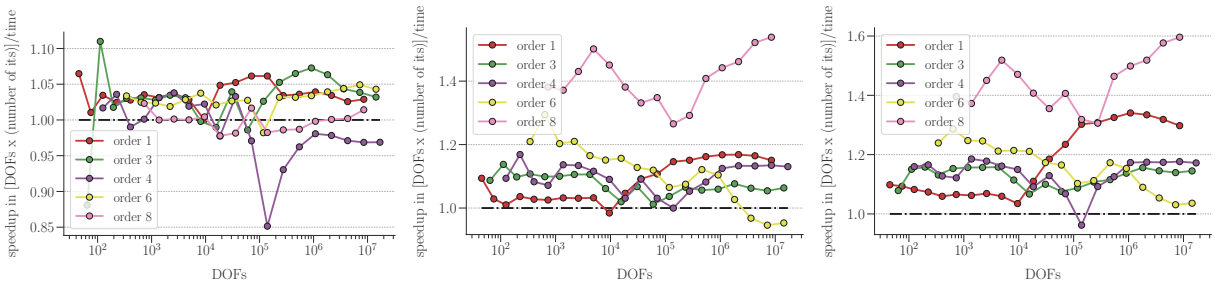


Figure 48: Speedup effects in the diffusion (BP3) MFEM benchmark performance for different incremental changes in the `hip-gen` libCEED backend on one GCD of an MI250x GPU, with ROCm 4.5. Left: giving the `-munsafe-fp-atomics` flag to `hiprtc`, to emit double precision atomic add instructions. Middle: adding launch bounds to the operator kernel. Right: atomics flag and launch bounds together.

The latest `hip-gen` kernels on MI250x achieve better performance than `cuda-gen` on a V100 GPU, as demonstrated in Figure 50. In these tests, we log how long it takes to apply the operator 2000 times–not counting a warm-up apply, which would include runtime compilation costs–followed by one device synchronization call, to ensure all kernels have completed. The average time to apply the operator is then computed. The left plot shows the usual CEED BP "DOFs processing metric," but with CG iterations replaced by "mults" – that is, operator multiplication actions. The achieved processing rate of the MI250x surpasses that of the V100 around 10^5 degrees of freedom, demonstrated in the right plot, which shows speedup for MI250x over V100; there is an initial lag for smaller problems, seen in the clustering to the right side of the CUDA data points along the x-axis on the left plot. This is been consistent for `hip-gen` on older AMD GPUs and for older ROCm versions. However, it is improved on MI250x compared to MI100, as demonstrated in Figure 51; note the minimum time per multiplication for MI250x is clearly shifted leftward for larger orders of basis functions. The peak performance is also improved compared to the MI100, shown in the right plot of Figure 51. This is, of course, expected, as the MI100 does not have double precision atomic addition support, and has fewer available registers and a lower theoretical peak double precision performance.

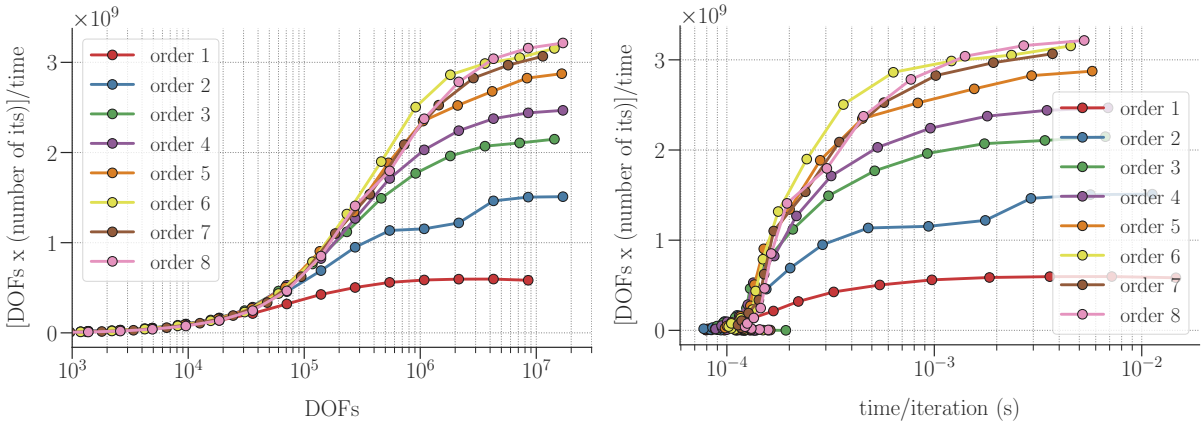


Figure 49: Diffusion (BP3) MFEM benchmark performance for the `hip-gen` libCEED backend on one GCD of an MI250x GPU, with ROCm 4.5. The figures show the DOFs processing metric versus: left, DOFs; right, average time per iteration in CG.

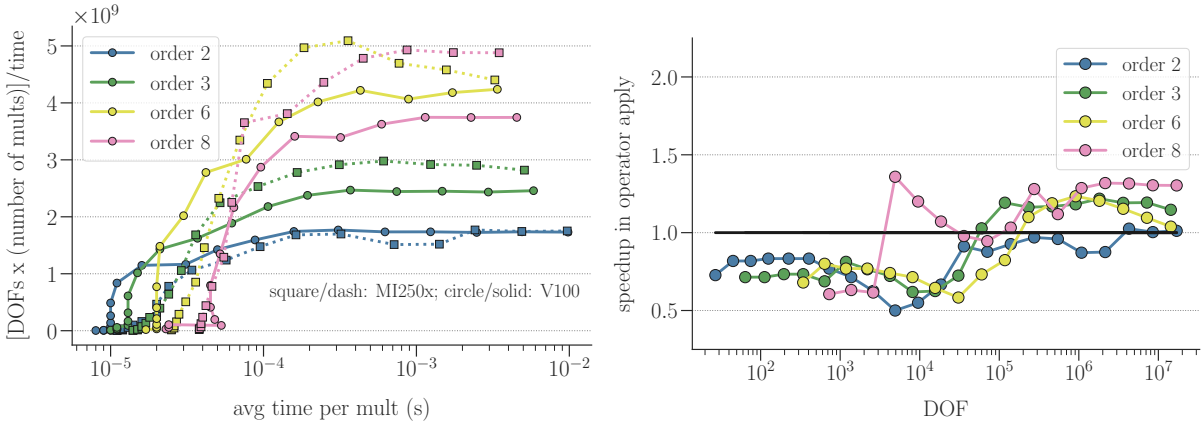


Figure 50: Comparison of average time to apply the libCEED operator for the `cuda-gen` backend (V100 GPU; CUDA 11) and the `hip-gen` backend (MI250x GPU; ROCm 4.5). The right plot shows speedup of MI250x over V100.

4.4 Tuning the MAGMA backend of libCEED on Crusher

The non-tensor basis action in the MAGMA backend is performed using standard general matrix multiplication (GEMM), either as $C = A \times B$ or as $C = A^T \times B$. Both matrices B and C are often short and very wide. It is, therefore, possible to split them into a batch of smaller matrices having the same number of rows, but fewer columns. This leads to the batch GEMM kernel being another candidate for performing the basis action. The decision of choosing between the GEMM and the batch GEMM kernel depends on many factors, such as

1. The input sizes of A, B, and C
2. The compute precision (e.g. single vs. double precision)
3. The transposition of A
4. The tuning of the vendor library and MAGMA's own GEMM kernels
5. The GPU architecture

The existing implementation of the basis action includes hard-coded decisions (using `if-else`) that are based on performance benchmark sweeps conducted on the NVIDIA V100 GPU only. Given the number of

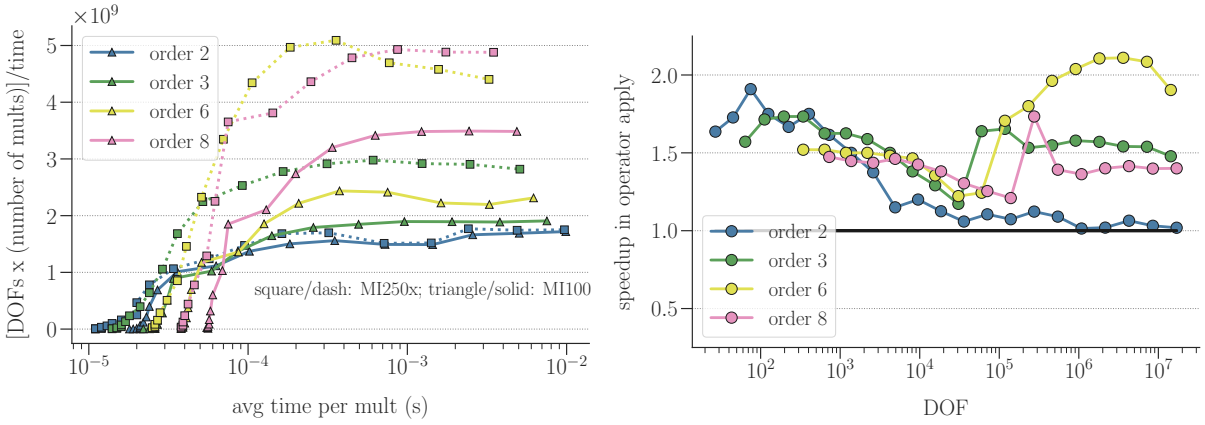


Figure 51: Comparison of average time to apply the libCEED operator for the `hip-gen` backend using one GCD of an MI250x GPU and one MI100 GPU, both with ROCm 4.5). The right plot shows speedup of MI250x over MI100.

factors impacting the decision, and the expansion of libCEED to support AMD GPUs, there is a need to implement a more robust decision-making layer that facilitates the support of different GPUs, while being easy to expand and maintain on the long term.

A current effort in the MAGMA backend is to implement a *GEMM selector* layer that decouples the decision making process from the GPU-specific data. The GEMM selector has two main components. The first one is a set of tuning data that are collected offline for every GPU model of interest. The data for each GPU is stored in lookup tables using C++ `std::vector` containers, as shown below.

```
// m, n, k, nb, use_magma?
std::vector< std::vector<int> > dgemm_nn_a100 =
{
    /* more entries */
    {84   , 8704 , 126  , 32   , 1    },
    {84   , 9216 , 126  , 64   , 0    },
    {84   , 9728 , 126  , 32   , 1    },
    {84   , 10240, 126  , 512  , 0    },
    /* more entries */
};
```

The use of `std::vector` containers makes it easy to expand or shrink the amount of offline data stored per GPU, without any changes to the way they are accessed. Each GPU has four lookup tables, two for single precision, and two for double precision. For each precision, one lookup tables covers the non-transpose basis action, while the other covers the transposed basis action. All data are collected and processed offline. A single record in the lookup table contains five entries. The first three represent the sizes (m, n, k) of the GEMM operation. The fourth entry represents the subdivision size nb of B and C. If a record has $n = nb$, that means that the regular GEMM routine achieves better performance than the batch GEMM routine for that record. The fifth entry is a flag that decides whether to use MAGMA's own kernels, or to call the routines provided by the vendor (e.g. cuBLAS or hipBLAS).

The second component of the GEMM selector is a search routine that extracts a decision given a lookup table. The new non-tensor basis action in MAGMA will get rid of the hard-coded decision, and instead invokes the search routine, which reads the settings of the operation (sizes, transposition, precision, and GPU architecture), and searches the corresponding lookup table. The sizes (m, n, k) are treated as a 3-component vector, and a closest match is searched for based on the minimum norm of the input vector with respect to the lookup table

Figure 52 shows preliminary results for the MFEM BP3 on Crusher (1 GCD of the MI250x GPU) using ROCM-5.0.0. In most cases, asymptotic speedups range between 20% to 30%. This is due to deviating

from the hard-coded decisions for the V100 GPU to a more robust implementation that loads GPU specific information to select the best performing kernel for a given size.

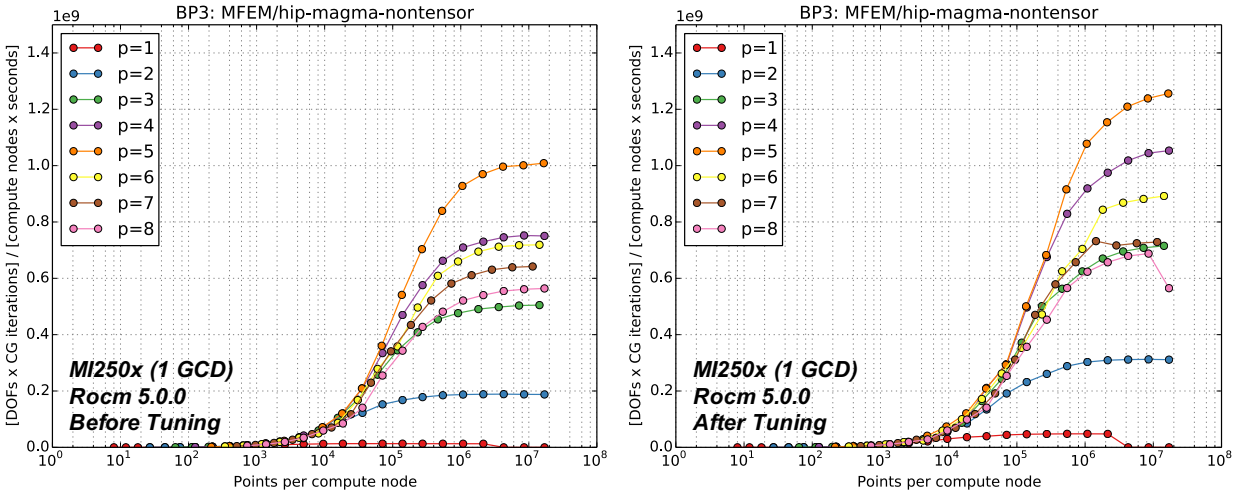


Figure 52: Speedup of the BP3 MFEM non-tensor benchmark using MAGMA without tuning (Left) and with tuning (Right) on Crusher MI250x GPUs using 1 GCD with ROCm 5.0.0

4.5 ExaSMR on Crusher MI250X vs. A100/V100/MI100

Table 8 demonstrates NekRS performance for ExaSMR's singlerod simulation on a single GPU. Simulations are performed for 500 steps and the average time-per-step, t_{step}, is measured in seconds for the last 400 steps. For a given system, the speedup is the ratio of t_{step} for ThetaGPU A100 to t_{step}. v_i and p_i represent the averaged iterations of the velocity components and pressure. Characteristic-based BDF2 with one substep is used for timestepping and the timestep size is Δt=1.2e-03 (CFL=1.82). Pressure preconditioning is based on p-multigrid with CHEBYSHEV+ASM smoothing and *hypre* AMG for coarse grid solve. Tolerances for pressure and velocity are 1e-4 and 1e-6, respectively. The results show that, for the current version of NekRS, a single GCD of the MI250X on Crusher realizes 79% of the performance of a single A100 on Theta. A breakdown of relative kernel performance is provided in the next section.

ExaSMR singlerod simulation, $E = 7168$, $N = 7$, $n = 2.4M$, $Re = 5000$.								
system	backend	v_i	p_i	t_{500}	t_{100}	t_{step}	t_{step} (A100)/t_{step}	ROCm
ThetaGPU/A100	CUDA	3.4	1.1	2.686e+01	5.805e+00	0.0526	1.00	-
Summit/V100	CUDA	3.4	1.1	3.592e+01	7.791e+00	0.0703	0.74	-
Spock/MI100	HIP	3.4	1.1	4.631e+01	9.905e+00	0.0910	0.57	v4.5
Crusher/MI250X	HIP	3.4	1.1	3.391e+01	7.364e+00	0.0663	0.79	v4.5.2

Table 8: NekRS V22.0.0(4647cc4e) performance on a single GPU.

4.6 ExaConstit improvements and initial Crusher results

As part of ExaAM's engagement with CEED, the ExaConstit team has worked with the CEED team over the past year to port ExaConstit to HIP. This process was overall a simple one, in large parts due to MFEM already having HIP support. ExaConstit was up and running on the OLCF's Spock and Crusher EA systems fairly quickly, allowing us to focus on performance differences between Summit and Crusher. We performed a node to node comparison using a portion of our test problem described here: `https://confluence.`

Path	Min time/rank (s)	Max time/rank (s)	Inc Avg time/rank (s)	Time % (total)
MPI Total	105.54556	1409.4917	394.423883	23.642297
ECMECH_TOTAL	109.968416	117.848298	113.154164	6.782845
Operator Total	884.078267	905.970167	895.295531	17.114556
Krylov Solver	1123.924269	1123.993362	1123.944614	47.015597
Mesh instantiation	73.964719	74.549154	74.293412	4.281634
Subtotal	2297.481231	3631.852681	2601.111604	98.836929

Table 9: Summit Caliper Annotations for most expensive scopes of ExaConstit

Path	Min time/rank (s)	Max time/rank (s)	Inc Avg time/rank (s)	Time % (total)
MPI Total	100.557671	2479.434488	382.853752	14.383828
ECMECH_TOTAL	79.064669	83.705589	81.538371	3.063439
Operator Total	516.98427	529.492162	523.952898	6.295452
Krylov Solver	2506.840296	2506.87883	2506.858972	74.623913
Mesh instantiation	38.582356	39.232815	38.979694	1.351549
Subtotal	3242.029262	5638.743884	3534.183687	99.718181

Table 10: Crusher Caliper Annotations for most expensive scopes of ExaConstit

`exascaleproject.org/display/ADSE10/FY22+Q2+ExaConstit+-+Stage+3` as this would reflect the actual ExaConstit use case within the ExaAM challenge problem.

While comparing Crusher and Summit runs, we saw a 1.6x slowdown with the Crusher system. In order to determine where potential areas of improvement could be made, we made use of the Caliper performance annotation tool (`https://github.com/LLNL/Caliper`). The results from these annotated runs are seen in Tables 9 and 10 with the top 5 most expensive portions of the code shown. The top 5 sections of code were examined as it can be seen from the subtotal row of each table that they account for 98-99% of the runtime. It can be seen from these tables that in almost every case Crusher outperforms Summit on a node per node comparison. However, this is not the case for the Krylov solver which suggestions something within could be improved for AMD hardware. After performing additional annotations within the Krylov solver code, the main culprit was found to be an operation within one of ExaConstit's matrix-free nonlinear integrators. It was found that the transpose prolongation operation within MFEM was to be 1.5x slower on Crusher than Summit. This operation is necessary for matrix-free methods as they take in E-Vector quantities and need to output T-Vector quantities for the matrix multiplication operations. Due to our current matrix-free formulations of our problem to run on the GPU, this transpose prolongation operation needs to be called every-time our Krylov solver has a matrix-vector operation using the linearized matrix.

In order to avoid this slowdown, the ExaCosntit and MFEM teams are working together to port over the full assembly integration code developed for the `BilinearFormIntegrator` class over to the `NonlinearFormIntegrator` class. This will avoid the transpose prolongation operation every Krylov iteration as the full assembly creates a sparse matrix that already operates on T-Vectors.

4.7 Preliminary Results for the MAGMA's Batch SVD on Crusher

Singular value decomposition (SVD) of many small matrices, i.e., batched SVDs, are needed in a number of GPU-accelerated simulations of interest for CEED. We developed and optimized them in MAGMA for both NVIDIA and AMD GPUs. Of particular interest are batched SVDs for 2x2 and 3x3 matrices, as they are applied to the Jacobians at each quadrature point as part of the `D` matrix evaluation. This is the case for example in the Laghos project, where there is need to accelerate and show impact through using these new batched SVDs.

The batched SVD developed is based on the one-sided Jacobi method. It applies a sequence of plane rotations to the right of the input matrix A until $A^T A$ implicitly convergences to a diagonal matrix. This algorithm is highly parallel and a very good fit for GPUs.

Figures 53 and 54 show the time-to-solution (in μs) to compute a batch of 1,000 SVDs in FP64 arithmetic, including the singular vectors' computation. Performance is given for the NVIDIA A100 GPU and the AMD MI100 GPUs, respectively. For the A100 GPU, the performance of MAGMA is compared against the batched SVD routine available in the cuSOLVER library, which is also based on the Jacobi SVD algorithm. For the MI100 GPU, the performance of MAGMA is compared against a batch SVD routine in the rocSOLVER

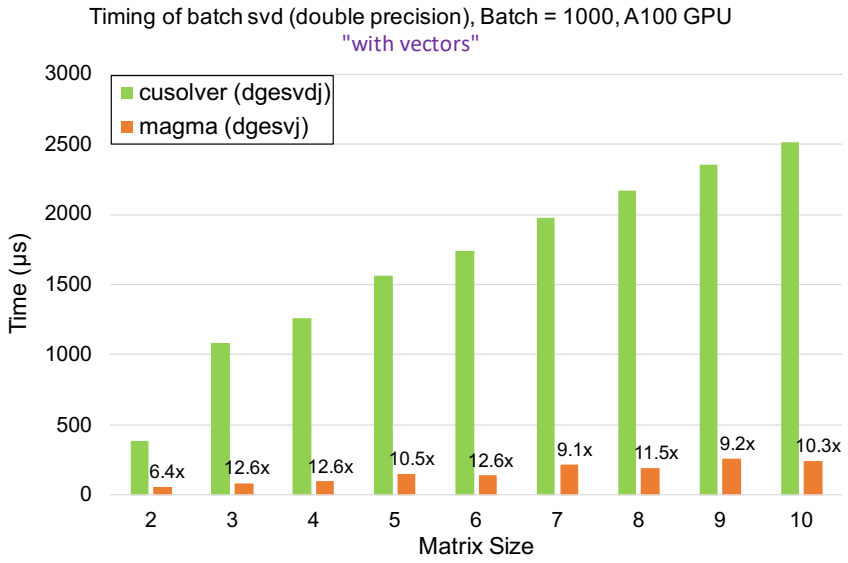


Figure 53: Speedup of MAGMA batched SVD vs. cuSOLVER batched SVD on the A100 GPU using CUDA-11.0. The SVDs computed are for a batch of 1,000 matrices in FP64 arithmetic and include the vectors computation.

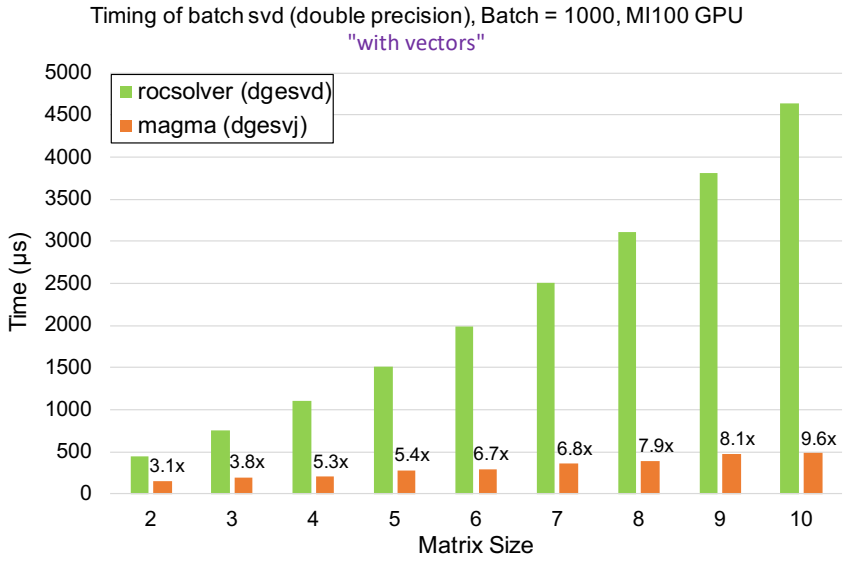


Figure 54: Speedup of MAGMA batched SVD vs. rocSOLVER batched SVD on the MI100 GPU using ROCM-5.0.0. The SVDs computed are for a batch of 1,000 matrices in FP64 arithmetic and include the vectors computation.

library, which is based on the QR iteration algorithm. MAGMA is up to 12.6× faster than cuSOLVER on the A100 GPU, and up to 9.6× faster than rocSOLVER on the MI100 GPU.

The MAGMA library has an ongoing effort to develop optimized batch SVD routines for both NVIDIA and AMD GPUs. The current batch SVD prototype uses the parallel one-sided Jacobi algorithm, and supports relatively small matrices such that all computations can be done in the shared memory of the GPU. For the MI250x GPU, square sizes up to 60×60 were successfully tested, where both the singular values and the singular vectors are computed. Figure 55 shows the MAGMA performance versus rocSOLVER for

sizes up to 10×10. Note that rocSOLVER batch SVD routine is based on the QR iteration algorithm, since no Jacobi-based routines currently exist. MAGMA is between 5.7× and 13.4× faster than rocSOLVER. Figure 56 also compares Spock and Crusher when running the MAGMA's batch SVD algorithm. With almost no code change, speedups between 1.5× and 2.5× are observed. There is potential room for even further improvement if specific tuning to the MI250x GPU is performed.

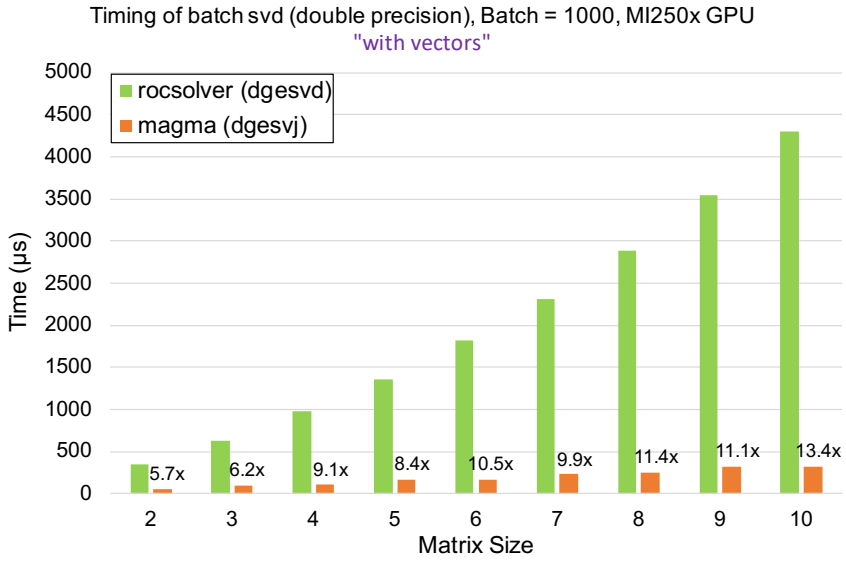


Figure 55: Speedup of MAGMA batched SVD vs. rocSOLVER batched SVD on the MI250x GPU (single GCD) using ROCM-5.0.0. The SVDs computed are for a batch of 1,000 matrices in FP64 arithmetic and include the vectors computation.

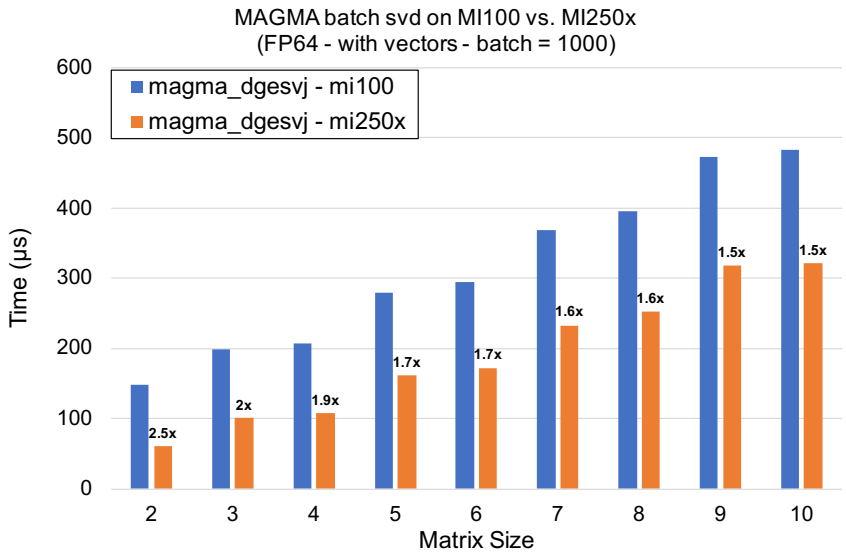


Figure 56: Performance portability of the MAGMA's batched SVD across Spock (MI100 GPU) and Crusher (MI250x GPU - single GCD) using ROCM-5.0.0. The SVDs computed are for a batch of 1,000 matrices in FP64 arithmetic and include the vectors computation.

4.8 Omega_h Crusher initial testing

An Omega_h benchmark case was ran on a single NVIDIA V100 on the OLCF Summit system and one GCD of an AMD MI250x on the OLCF Crusher system to collect baseline performance data on the MI250x after porting the CUDA backend to HIP. CUDA 10.1.243 and ROCm 5.0.0 to build the Omega_h CUDA and HIP backends, respectively, The benchmark case is a 3D Delta Wing from the Unstructured Grid Working Group with a mesh metric field derived from a laminar simulation [78, 42, 76]. The initial mesh has 500,000 tetrahedral elements and adapts (coarsening and refinement) to five million tetrahedra. On the V100 the mesh is adapted in 9.8 seconds and 10.5 seconds on one GCD of MI250x. In both runs the Omega_h memory pool is enabled to avoid expensive device memory allocation and deallocation API calls. Avoiding these calls on the V100 reduces the runtime by 80% and by 37% on the MI250x. Profiling indicates that two Omega_h memory movement functions `copies_to_linear_owners` and `single_host_to_device` are consuming an order of magnitude more time on the MI250x than the V100. Investigations into the cause of the slowdown is ongoing.

5. OTHER PROJECT ACTIVITIES

5.1 Nonconforming discontinuous Galerkin support in MFEM for AMR

A critical feature to application codes using MFEM is the ability to use nonconforming meshes for adaptive mesh refinement (AMR) strategies. For continuous Galerkin ("standard") finite element methods, nonconforming meshes can be handled either in the P or G operator. However, for discontinuous Galerkin finite element methods, a special treatment must be applied to nonconforming faces. MFEM v4.4 adds GPU enabled partial assembly and element assembly support for nonconforming DG FEM methods.

For discontinuous Galerkin finite element methods, the operator decomposition contains two parts, a volume contribution and a face contribution (see fig. 57):

$$A = A_{Vol} + A_{Faces} \tag{21}$$

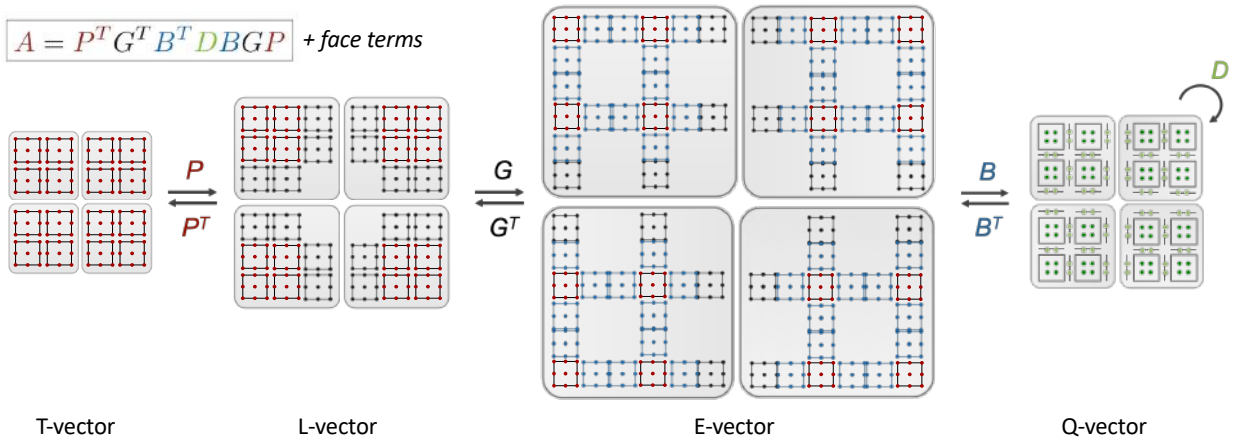


Figure 57: Operator decomposition in MFEM for discontinuous Galerkin finite element methods.

The volume contribution can be seen as a simpler version of the CEED operator decomposition:

$$A_{Vol} = B^T D B \tag{22}$$

The operator decomposition for the face contributions is represented by the formula (see fig. 58):

$$A_{Faces} = \sum_F B_F^T D_F B_F + B_F^T D_{F'} B_{F'} P_F \tag{23}$$

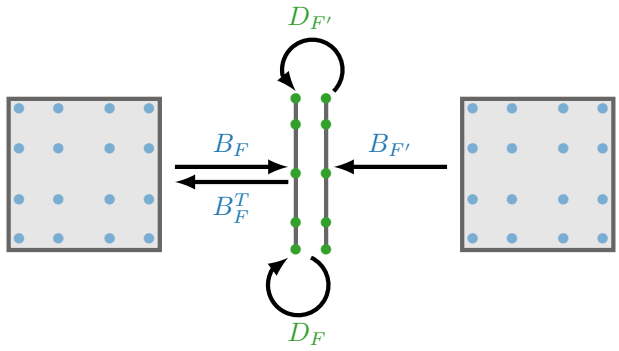


Figure 58: Operator decomposition in MFEM for discontinuous Galerkin finite element methods face contributions.

In the case of a nonconforming face $\tilde{F}$, the usual approach would be to use specialized interpolation operator $B_{\tilde{F}}$. This approach would be inefficient on GPU hardware due to the introduction of multiple $B_{\tilde{F}}$ operators. Instead, the approach in MFEM is to express $B_{\tilde{F}}$ as $B_F B_{\tilde{F}\to F}$. This approach allows to apply the $B_{\tilde{F}\to F}$ first only where needed on nonconforming faces, resulting in an homogeneous GPU computation. This approach also has the benefit that besides the introduction of the $B_{\tilde{F}\to F}$ operators, the rest of the DG operator decomposition remains identical to conforming meshes.

5.2 Enzyme integration with libCEED

Efficient use of matrix-free methods requires quadrature-point based linearization ("partial assembly") of forward (and possibly adjoint) operators. While many problems have great exploitable structure [**?**, 17] to reduce the memory footprint and operation count, it can be tedious to find these formulations and it slows development to have to develop the nonlinear residual and Jacobian action synchronously. Enzyme [52] is a new LLVM plugin with GPU support that provides split forward and reverse mode AD on LLVM intermediate representation (IR). We worked with William Moses to identify and fix bugs in the new forward-mode capability to provide derivatives of Neo-Hookean models with libCEED-managed storage of the "tape". On CPU, Enzyme identifies a straightforward and relatively low-memory representation (small tape) with more recomputation than is needed by a clever human, resulting in a modest increase in The optimizer applies the same vectorization as would be applied to hand-written code and the performance is on par. This allows rapid development of constitutive models since only the nonlinear forward model needs is written by a human. In our elasticity experiments, this amounts to writing the Piola-Kirchhoff tensor as a function of Green-Lagrange strain $\boldsymbol{S}(\boldsymbol{E})$ or writing the Kirchhoff stress as a function of Green-Euler strain $\boldsymbol{\tau}(\boldsymbol{e})$, with no need to represent the (4th order) linearized elasticity tensor or its action. Since Enzyme is language-agnostic by virtue of operating on LLVM IR, this opens the door to constitutive modeling in safer/more convenient languages, such as Rust and Julia, with no impact on execution performance or environment.

With Moses' help, we have run libCEED with hipRTC using Enzyme on ROCm-5.0 using our development machine (Radeon VII). This involved two small patches that Moses submitted upstream and we are in discussion with engineers at AMD on how to get these (or similar functionality) merged and eventually available on machines like Crusher.

5.3 Schwarz primitive extrusions

Volumetric extrusions of triply periodic minimal surfaces have garnered interest during the additive manufacturing revolution for a range of applications from tissue membranes [36] to metallurgy [1]. We consider the Schwarz Primitive surface, which exhibits interesting geometric and material nonlinearities. Prior finite element analysis of such models [48] using voxelized meshes [47] found that about 30k low order (Abaqus C3D8R) elements were needed to achieve an engineering tolerance of 1%. We consider conformal meshes that attain comparable accuracy with fewer degrees of freedom and much fewer elements. To generate such

meshes, we start with a 24-element 2D manifold mesh of a single unit cell embedded in 3D, replicated to the prescribed extent in each embedding dimension. This mesh is partitioned and distributed using ParMETIS, then refined with new nodes projected to the closest point on the implicit surface

$$\cos 2\pi x + \cos 2\pi y + \cos 2\pi z = 0.$$

The resulting manifold mesh is extruded normal to this surface to the prescribed thickness and number of layers. Figure 59 shows an example of such surfaces under loading approaching the plastic yield limit for typical photopolymer AM products, as computed by Ratel.

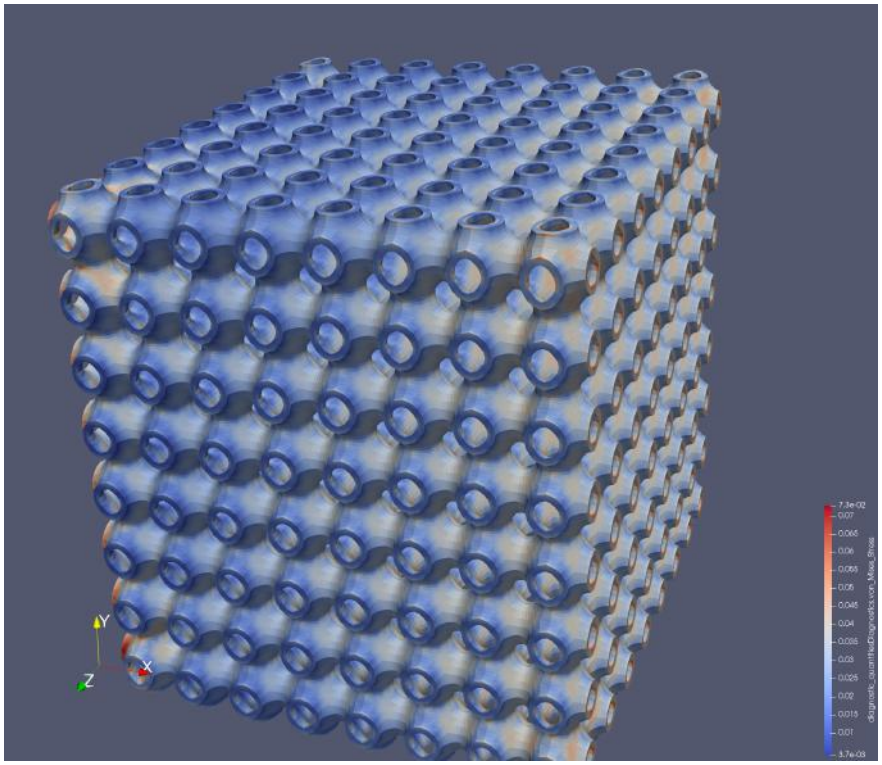


Figure 59: Extruded Schwarz Primitive surface under 12% compressive strain, colored by von Mises stress. The left wall is fixed and a force is applied to the right faces.

These domains provide excellent tests for solvers since they exhibit all compressive and bending modes, nonlinearities are activated at local and global scale, coarsening is inherently unstructured, and scaling is done by making the domain larger while achieving the same accuracy tolerances in contrast to the common practice of refining a simpler domain to achieve unrealistically tight accuracy tolerances. Our experience with matrix-free p-multigrid using PETSc's GAMG and hypre's BoomerAMG has shown that large efficiency improvements are available without at typical (loose) engineering accuracy tolerances. With reasonable element quality, preconditioner setup is still dominated by the AMG coarse problem (8x fewer dofs and 20x fewer nonzeros) when using Q_2 elements. The efficiency (dofs/cost) improves as the element order increases, but most industrial models have sufficient geometric complexity that Q_2 and Q_3 elements deliver sufficient accuracy on the coarsest geometry-resolving mesh that can be created. Figure 60 explores this effect for a large-deformation bending of a drilled out box using coarse meshes. This efficiency benefit of higher order bucks the prevailing industry recommendation of using low order elements for nonlinear elasticity, which has tacitly assumed assembled methods will always be used.

With new features in libCEED and PETSc, coarse matrix assembly is done entirely on GPUs, yielding an assembled matrix in the user's (run-time) choice of PETSc, hypre, cuSparse, rocSparse, and Kokkos formats. When combined with Enzyme (subsection 5.2), the user can focus entirely on expressing the nonlinear residual via libCEED qfunctions and have a matrix-free p-MG with AMG coarse solve running entirely on GPUs at a

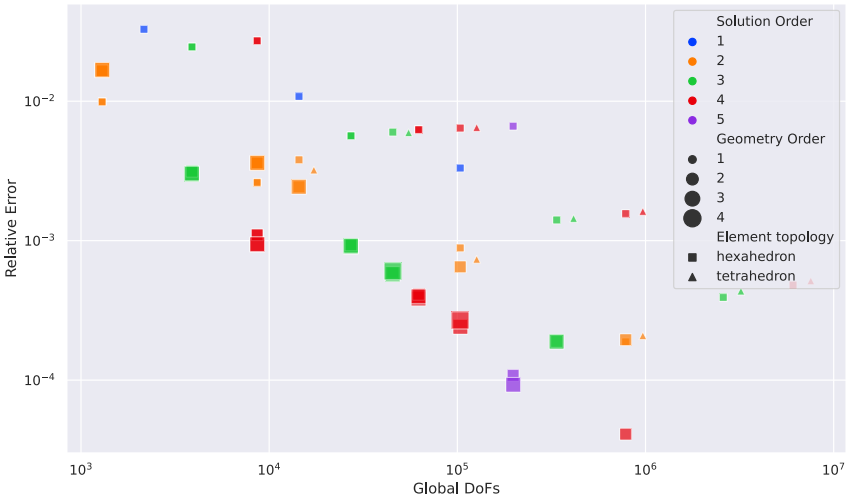


Figure 60: Accuracy study for the bending experiment, including both h and p refinement with low and high order geometry. The Pareto front is toward the lower left. We measure integrated strain energy as quantity of interest. This is a large deformation problem with concave surfaces and a fixed boundary condition on one surface, thus it has strain singularities. h-refinement with linear elements is consistently far from the Pareto front and h-refinement from any model on the front always moves one away from the optimal front. p-refinement with linear geometry elements usually helps from $p = 1$ to $p = 2$, then errors get larger as weak stress singularities on spurious reentrant corners (present in the linear mesh geometry but not in the smoothly concave geometric model) are resolved. Using second order geometry is usually enough to p-refine beyond engineering tolerances. Due to physical reentrant corners and Dirichlet-Neumann boundary transitions, there are always stress singularities so all methods converge at the same order under h-refinement. The constants are significantly better for high order methods, even in terms of number of degrees of freedom. When using p-multigrid, the Newton and linear solve costs decrease with increasing p (several times cheaper per dof than linear elements, due to more efficient quadrature and lower memory requirements). This part is only possible with matrix-free methods.

fraction of the cost per dof of traditional methods. These methods are being used in CU Boulder's PSAAP center, which is focused on bonded granular media.

5.4 Progress on mixed precision in libCEED

In [39], we detailed initial tests to determine the potential of mixed-precision computations within libCEED operator applications. Those tests focused on the "low-high" and "high-low" models of computation for the fused `cuda-gen` backend. Recall the libCEED operator decomposition, $A = \mathcal{E}^T B^T D B \mathcal{E}$. The "low-high" name indicates that the input and output vectors are in lower precision (float), while conversion to/from higher precision (double) happens in the element restriction, $\mathcal{E}$, the first and last portions of the computation. As the element restrictions do not involve much (or any) computation, "all" computation can be considered to take place in the higher precision. The "high-low" case, of course, is the opposite; the input and output vectors are in double precision, but conversion happens in the element restriction, and all other computation and data movement proceeds in single precision.

We have now conducted tests for the `hip-gen` backend on an AMD MI100 GPU as well, with noticeable differences in performance trends. In the following, we compare the speedup in the average time to apply the operator for "all-float," "low-high," and "high-low" operators compared to double precision. The results are shown over increasing problem sizes for three different orders of tensor-product basis functions. To obtain the average operator application times, each test timed 2000 successive operator applications followed by one

device synchronization call. (Note that the CUDA results are slightly different than the ones in [39], which used `nvprof` in a way that is not currently possible to replicate with `rocprof`; the CUDA tests were re-done to match the test case for HIP.)

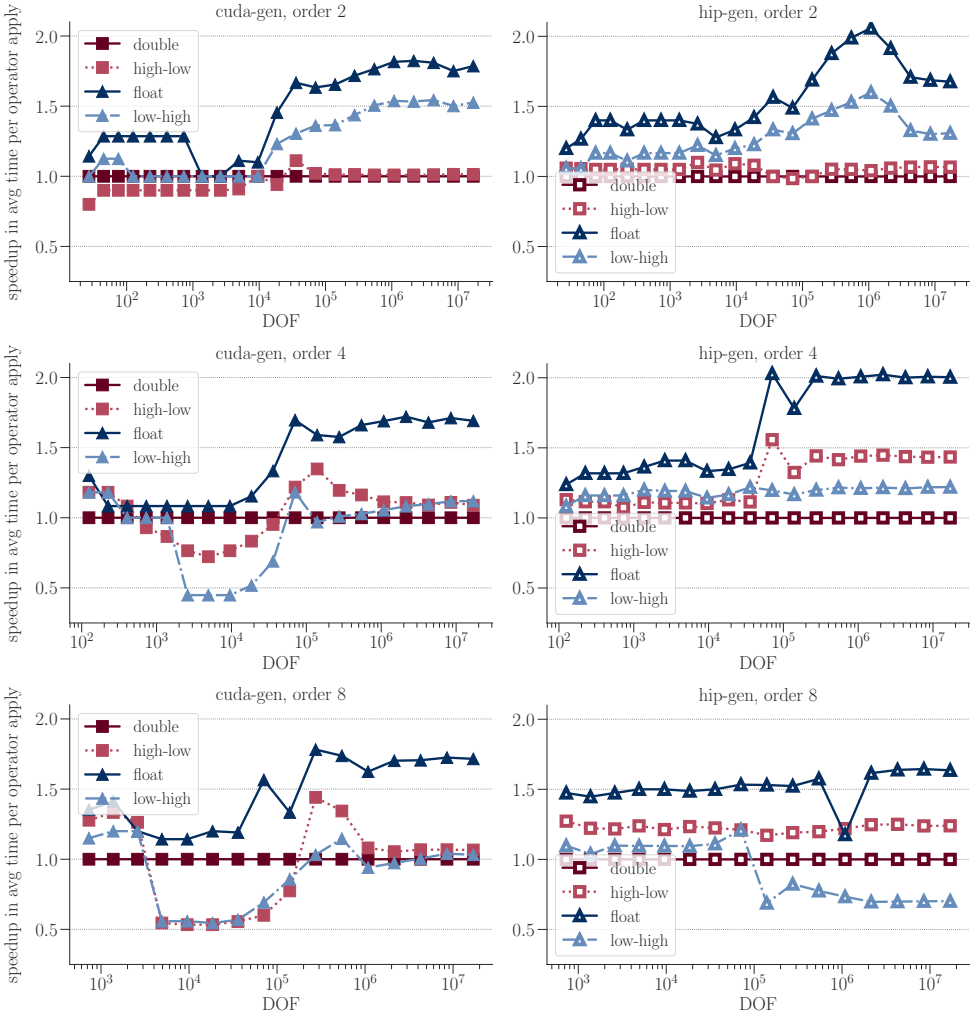


Figure 61: Comparison of the speedup obtained through mixed-precision operator application in the CUDA and HIP code generation backends, for tensor-product basis functions of order 2, 4, and 8. CUDA results obtained on one NVIDIA V100 GPU with CUDA 11; HIP results on an AMD MI100 GPU with ROCm 4.5.

We are most interested in the right side of each plot, where the degrees of freedom in the discretization exceed 10^5, knowing that this is the range required for libCEED's GPU backends to make efficient use of available resources. In this regime, we notice that overall, the MI100 results benefit much more than V100 from doing computation in lower precision (the pink squares/"high-low"), except when the order of basis functions is very low, as in the first row of plots, which is for second order functions. Both fourth and eighth order basis functions see moderate-to-good (20-50%) speedup from doing computation in lower precision on the MI100, but almost no benefit on the V100. CUDA and HIP both obtain the most speedup among "low-high" operators with second order basis functions; this is expected, as the operator itself is cheaper both in terms of memory movement and computation, so the major cost is reading and writing the input and output vectors from main memory, which is done in float for this case. We actually see sizeable slowdown from the "low-high" case on the MI100 when using eighth order basis functions (bottom right plot). Investigation into the compiler assembly output revealed that the additional register usage required by the "low-high" kernel in

this case (likely due to the extra conversions in the element restriction portion) reduces the occupancy of the kernel below that of the full double-precision operator.

As a final remark on comparing CUDA and HIP results for the code generation backends, we note that the `cuda-gen` backend chooses a block size at runtime based on `cuOccupancyMaxPotentialBlockSize`, while `hip-gen` makes its choice based solely on the number of basis function nodes and quadrature points per element. This is currently done for two reasons: first, because the HIP API lacks an equivalent to `cuOccupancyMaxPotentialBlockSize`, which allows for shared memory requirements that change with the thread block size, as required by the `*-gen` kernels, and second, because we also want to make use of the launch bounds, as discussed for MI250x in 4.3. The need to know accurate launch bounds at the time of kernel compilation complicates the use of a function like `cuOccpuancyMaxPotentialBlockSize`, because the kernel must be compiled before calling the function. The runtime block size selection of `cuda-gen` can (and does) sometimes select different block sizes for different precisions and mixed-precision versions of the operator kernel for the same problem. (We have considered the effects of adding launch bounds to `cuda-gen` kernels and using the same pre-set block sizes as `hip-gen`, and found very little effect of launch bounds on V100 performance.)

Next, we consider the effects of a particular mixed-precision use case for a non-fused backend, the deterministic MAGMA backend. Unlike the code generation backends, the other GPU backends launch separate kernels for each suboperator in the sequence $\mathcal{E}^T B^T D B \mathcal{E}$. This means that the "E-" and "Q-Vectors" which are intermediate data storage between the suboperators are fully formed, increasing data movement to and from main memory when compared to the fused `cuda-` or `hip-gen` backends. In this case, we may expect that being able to lower the precision of data movement will achieve a noticeable speedup, even if the basis (B) and QFunction (D) computations are done in higher precision. In the following, "low-high" now refers to a case where the basis and QFunction kernels perform conversion to and from double precision inside the kernels themselves, and do all computation in double precision; everything else, including the input and output vectors, is in single precision. We again compare CUDA/V100 and HIP/MI100 results for second, fourth, and eighth order basis functions.

For the V100, this mixed-precision mode does regain most of the all-single-precision speedup for second order basis functions, and around half as much for eighth order. But the MI100 results, as with the code generation backends, reflect that the balance between memory movement and computation is different than for the CUDA version, with almost no benefit to this "low-high" scheme for higher orders of basis functions; even second order basis functions achieve much less speedup than the all-float case, especially in comparison to the V100.

These results demonstrate the complexity involved in picking a mixed-precision scheme that will yield worthwhile speedup for a particular case; we expect further such complications when considering different GPUs from the same vendor, e.g. AMD MI100 versus MI250x or NVIDIA V100 versus A100.

Plans for Multi-precision libCEED Vectors and User API. Among libCEED developers, we have held discussions and planned for a new "multiprecision" framework for libCEED vectors. This framework will allow vectors in backends with mixed-precision capabilities to store multiple precisions of data at the same time. New vector read/write access functions will prompt data precision conversion, handled by the vector object, as necessary. Data validity will be maintained through differing behavior of read and write access; both read and write functions requesting a precision other than the current data's precision will result in data conversion, but after a read access, the vector will maintain both versions of the data, while a write-capable access will invalidate all but the new precision.

Further plans for the final user-facing mixed-precision API for application of mixed-precision libCEED operators is still under active development.

6. CONCLUSION

The goal of this milestone was to improve the high-order software ecosystem for CEED-enabled ECP applications by making progress on efficient matrix-free kernels targeting forthcoming ECP architectures. Kernels include preconditioning, matrix-free monotonicity, gradients of high-order operators, matrix assembly for low-order methods and other operations that are critical for applications.

As part of this milestone, we also released the next version of the CEED software stack, CEED-5.0,

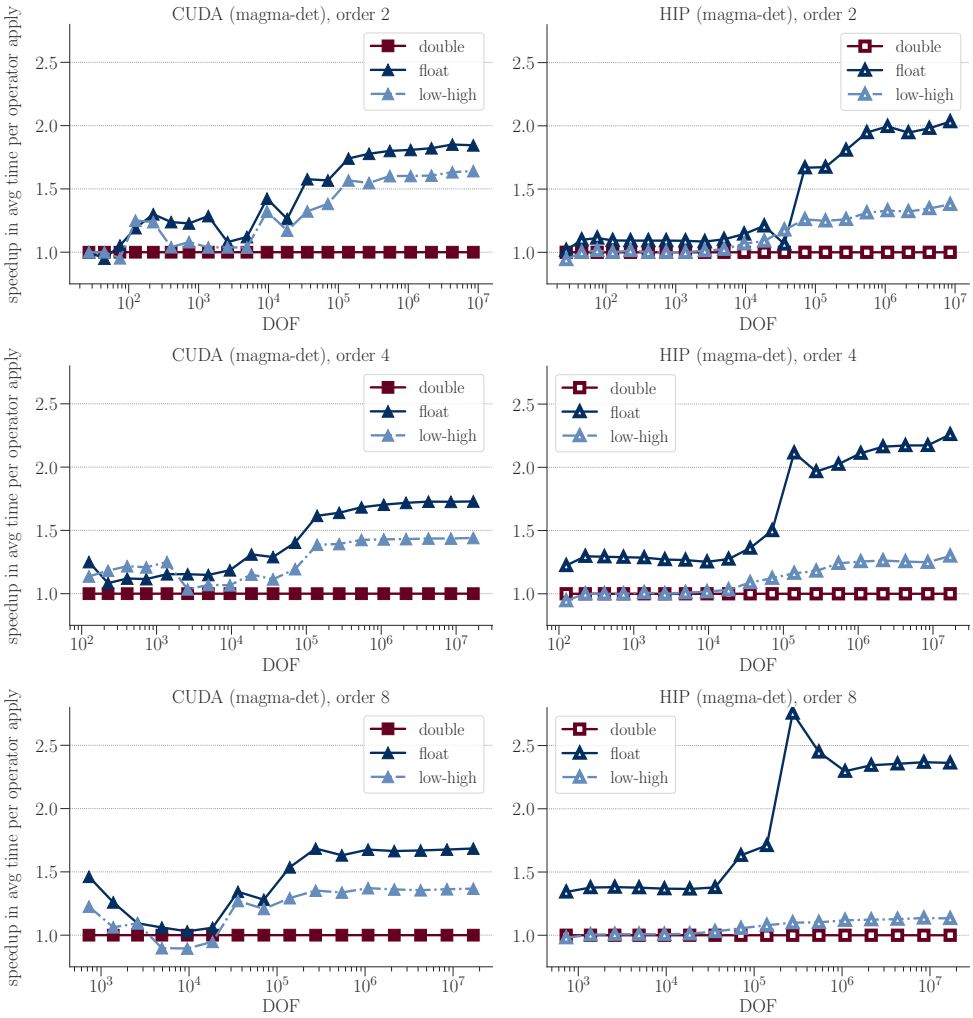


Figure 62: Comparison of the speedup obtained through mixed-precision operator application in the CUDA and HIP deterministic MAGMA backends, for tensor-product basis functions of order 2, 4, and 8. CUDA results obtained on one NVIDIA V100 GPU with CUDA 11; HIP results on an AMD MI100 GPU with ROCm 4.5.

described recent meshing and applications performance improvements and reported on the effort of porting to AMD GPUs for Frontier.

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-833372.

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] Nesma T Aboulkhair, Marco Simonelli, Luke Parry, Ian Ashcroft, Christopher Tuck, and Richard Hague. 3d printing of aluminium alloys: Additive manufacturing of aluminium alloys using selective laser melting. *Progress in materials science*, 106:100578, 2019.

[2] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, et al. MFEM: a modular finite element methods library. *Computers & Mathematics with Applications*, 81:42–74, 2021.

[3] P. Bello-Maldonado and P.F. Fischer. Scalable low-order finite element preconditioners for high-order spectral element Poisson solvers. 41:S2–S18, 2019.

[4] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.

[5] Jed Brown, Yunhui He, and Scott MacLachlan. Local Fourier analysis of balancing domain decomposition by constraints algorithms. *SIAM Journal on Scientific Computing*, 41(5):S346–S369, 2019.

[6] Cameron W. Smith. SCOREC EGADSliteCuda Github Repository. `https://github.com/SCOREC/egadslitecuda`, 2022.

[7] C. Canuto, P. Gervasio, and A. Quarteroni. Finite-element preconditioning of G-NI spectral methods. 31:4422–44251, 2010.

[8] N. Chalmers, A. Karakus, A. P. Austin, K. Swirydowicz, and T. Warburton. libParanumal: a performance portable high-order finite element library, 2020. Release 0.4.0.

[9] Noel Chalmers, Abhishek Mishra, Damon McDougall, and Tim Warburton. Hipbone: A performance-portable gpu-accelerated c++ version of the nekbone benchmark. *arXiv preprint arXiv:2202.12477*, 2022.

[10] Noel Chalmers and Tim Warburton. Portable high-order finite element kernels I: Streaming operations. *arXiv preprint arXiv:2009.10917*, 2020.

[11] Tony F. Chan, Wei-Pai Tang, and Wing Lok Wan. Wavelet sparse approximate inverse preconditioners. *BIT Numerical Mathematics*, 1997.

[12] European Fluids Engineering Division Conference, editor. *Wavelet-Preconditioned Conjugate Gradient Poisson Solver and Its Use in Parallel Processing: Application of Haar Wavelet*, volume Volume 1: Fora, Parts A and B of *Fluids Engineering Division Summer Meeting*, 7 2002.

[13] Jonathan Corney. *3D Modeling Using the Acis Kernel and Toolkit*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[14] Ingrid Daubechies. Orthonormal bases of compactly supported wavelets. *Communications on pure and applied mathematics*, 41(7):909–996, 1988.

[15] Ingrid Daubechies. *Ten lectures on wavelets*. SIAM, 1992.

[16] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *The Journal of Fourier Analysis and Applications*, 4:247–269, 1998.

[17] Denis Davydov, Jean-Paul Pelteret, Daniel Arndt, Martin Kronbichler, and Paul Steinmann. A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid. *International Journal for Numerical Methods in Engineering*, 121(13):2874–2895, 2020.

[18] Doreen De Leon. Wavelet operators applied to multigrid methods, 2000.

[19] Doreen De Leon. A new wavelet multigrid method. *Journal of computational and applied mathematics*, 220(1-2):674–685, 2008.

[20] Clark R Dohrmann. A preconditioner for substructuring based on constrained energy minimization. *SIAM Journal on Scientific Computing*, 25(1):246–258, 2003.

[21] Clark R. Dohrmann. Spectral equivalence of low-order discretizations for high-order $H(\mathrm{curl})$ and $H(\mathrm{div})$ spaces. *SIAM Journal on Scientific Computing*, 43(6):A3992–A4014, January 2021.

[22] Charbel Farhat, Michael Lesoinne, and Kendall Pierson. A scalable dual-primal domain decomposition method. *Numerical linear algebra with applications*, 7(7-8):687–714, 2000.

[23] Paul Fischer, Misun Min, Thilina Rathnayake, Som Dutta, Tzanio Kolev, Veselin Dobrev, Jean-Sylvain Camier, Martin Kronbichler, Tim Warburton, Kasia Świrydowicz, et al. Scalability of high-performance PDE solvers. *The International Journal of High Performance Computing Applications*, 34(5):562–586, 2020.

[24] Paul F Fischer. Projection techniques for iterative solution of Ax= b with successive right-hand sides. *Computer methods in applied mechanics and engineering*, 163(1-4):193–204, 1998. Publisher: Elsevier.

[25] V.M. Garcia, L. Acevedo, and A.M. Vidal. Variants of algebraic wavelet-based multigrid methods: Application to shifted linear systems. *Applied Mathematics and Computation*, 202(1):287–299, 2008.

[26] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Nekbone performance on GPUs with OpenACC and CUDA Fortran implementations. *The Journal of Supercomputing*, 72(11):4160–4180, 2016.

[27] Robert Haimes and John Dannenhoffer. *EGADSlite: A Lightweight Geometry Kernel for HPC*.

[28] Robert Haimes and Mark Drela. On the construction of aircraft conceptual geometry for high-fidelity analysis and design. In *50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Expo*, pages 1–21. AIAA, January 2012.

[29] Roger W Hockney. Characterization of parallel computers and algorithms. *Computer Physics Communications*, 26(3-4):285–291, 1982.

[30] Roger W Hockney. ($r\infty$, n12, s12) measurements on the 2-CPU CRAY X-MP. *Parallel Computing*, 2(1):1–14, 1985.

[31] Dan Ibanez. Omega_h GitHub repository, 2022. https://github.com/sandialabs/omega_h.

[32] Dan Ibanez and Mark Shephard. Mesh adaptation for moving objects on shared memory hardware. In *Proc. 25th Int. Meshing Roundtable*, pages 1–5, September 2016.

[33] Dan Ibanez and Mark S Shephard. Modifiable array data structures for mesh topology. *SIAM Journal on Scientific Computing*, 39(2):C144–C161, 2017.

[34] Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard. Pumi: Parallel unstructured mesh infrastructure. *ACM Transactions on Mathematical Software (TOMS)*, 42(3):17, 2016.

[35] Daniel Alejandro Ibanez. *Conformal mesh adaptation on heterogeneous supercomputers.* Rensselaer Polytechnic Institute, Troy, NY, 2016.

[36] Sebastian C Kapfer, Stephen T Hyde, Klaus Mecke, Christoph H Arns, and Gerd E Schröder-Turk. Minimal surface scaffold designs for tissue engineering. *Biomaterials*, 32(29):6875–6882, 2011.

[37] Martin Karp, Niclas Jansson, Artur Podobas, Philipp Schlatter, and Stefano Markidis. Optimization of tensor-product operations in NekBone on GPUs. *arXiv preprint arXiv:2005.13425*, 2020.

[38] 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.

[39] Tzanio Kolev, Paul Fischer, Natalie Beams, Jed Brown, Jean-Sylvain Camier, Noel Chalmers, Veselin Dobrev, 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, and Tim Warburton. ECP Milestone Report CEED-MS37: Port and optimize the CEED software stack to Aurora/Frontier EA, September 30, 2021.

[40] David Alex Lamb, Geoffrey C Fox, Margaret H Johnson, G Lyzenga, and S Otto. *Solving Problems on Concurrent Processors: General techniques and regular problems*, volume 1. Prentice Hall, 1988.

[41] Yu-Hsiang Lan, Paul Fischer, Elia Merzari, and Misun Min. All-Hex Meshing Strategies For Densely Packed Spheres. *Proceedings of the 29th International Meshing Roundtable*, pages 293–305, June 2021.

[42] Tobias Leicht and Ralf Hartmann. Error estimation and anisotropic mesh refinement for 3d laminar aerodynamic flow simulations. *Journal of Computational Physics*, 229(19):7344–7360, 2010.

[43] Xiangrong Li, Mark S Shephard, and Mark W Beall. Accounting for curved domains in mesh adaptation. *Int. J. Numerical Methods in Eng.*, 58(2):247–276, July 2003.

[44] James W Lottes and Paul F Fischer. Hybrid multigrid/Schwarz algorithms for the spectral element method. *Journal of Scientific Computing*, 24(1):45–78, 2005.

[45] Jan Mandel. Balancing domain decomposition. *Communications in numerical methods in engineering*, 9(3):233–241, 1993.

[46] Jan Mandel and Bedřich Sousedík. BDDC and FETI-DP under minimalist assumptions. *Computing*, 81(4):269–280, 2007.

[47] Ian Maskery, LA Parry, D Padrão, RJM Hague, and IA Ashcroft. Flatt pack: A research-focussed lattice design program. *Additive Manufacturing*, 49:102510, 2022.

[48] Ian Maskery, Logan Sturm, Adedeji O Aremu, Ajit Panesar, Christopher B Williams, Christopher J Tuck, Ricky D Wildman, Ian A Ashcroft, and Richard JM Hague. Insights into the mechanical properties of several triply periodic minimal surface lattice structures made by polymer additive manufacturing. *Polymer*, 152:62–71, 2018.

[49] David S Medina, Amik St-Cyr, and T. Warburton. OCCA: A unified approach to multi-threading languages, 2014.

[50] Misun Min, Michael Brazell, Ananias Tomboulides, Matthew Churchfield, Paul Fischer, and Michael Sprague. Large-eddy simulations of a benchmark turbulent atmospheric flow on graphical-processing units. to be submitted, 2022.

[51] Misun Min, Michael Brazell, Ananias Tomboulides, Matthew Churchfield, Paul Fischer, and Michael Sprague. Towards exascale for wind enery simulations. to be submitted, 2022.

[52] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[53] S.A. Orszag. Spectral methods for problems in complex geometry. *J. Comput. Phys.*, 37:70–92, 1980.

[54] Will Pazner, Tzanio Kolev, and Clark Dohrmann. Low-order preconditioning for the high-order de Rham complex. arXiv eprint 2203.02465. (Submitted for publication), March 2022.

[55] Fabio Henrique Pereira, Kleber Rogério Moreira Prado, and Silvio Ikuyo Nabeta. A new multilevel smoothing method for wavelet-based algebraic multigrid poisson problem solver. *Journal of Microwaves, Optoelectronics and Electromagnetic Applications*, 10(2):379–388, 2011.

[56] Fábio Henrique Pereira, Sérgio Luís Lopes Verardi, and Sılvio Ikuyo Nabeta. A wavelet-based algebraic multigrid preconditioner for sparse linear systems. *Applied Mathematics and Computation*, 182(2):1098–1107, 2006.

[57] M. Phillips, S. Kerkemeier, and P. Fischer. *Tuning Spectral Element Preconditioners for Parallel Scalability on GPUs*, pages 37–48. 2022.

[58] PUMI: Parallel unstructured mesh infrastructure, 2016. http://www.scorec.rpi.edu/pumi.

[59] Padmanabha Reddy and Nagendrappa Bujurke. An improved wavelet based preconditioner for sparse linear problems. *Applied Mathematics*, 1(5):370–376, 2010.

[60] J-F Remacle, Rajesh Gandham, and Tim Warburton. GPU accelerated spectral finite elements on all-hex meshes. *Journal of Computational Physics*, 324:246–257, 2016.

[61] Open CASCADE SAS. Open CASCADE modeling kernel. `https://www.opencascade.com/content/latest-release`, 1999.

[62] Michael Schliephake and Erwin Laure. Performance analysis of irregular collective communication with the crystal router algorithm. In *International Conference on Exascale Applications and Software*, pages 130–140. Springer, 2014.

[63] U. Schumann. Subgrid scale model for finite difference simulations of turbulent flows in plane channels and annuli. *Journal of Computational Physics*, (18):376–404, 1975.

[64] Siemens. Parasolid 3D geometric modeling engine. `http://www.plm.automation.siemens.com/en_us/products/open/parasolid/`, 1986.

[65] Joel Solé Rojals. *Optimization and generalization of lifting Schemes: application to lossless image compression*. Universitat Politècnica de Catalunya, 2006.

[66] Nichole Stilwell. *gNek: A GPU accelerated incompressible Navier Stokes solver*. Rice University, 2013.

[67] P. Sullivan, J. McWilliams, and C.H. Moeng. A subgrid-scale model for large-eddy simulation of planetary boundary-layer flows. *Bound.-Layer Meteor.*, 71:247–276, 1994.

[68] Wim Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM Journal on Mathematical Analysis*, 29(2):511–546, 1998.

[69] Kasia Świrydowicz, Noel Chalmers, Ali Karakus, and Tim Warburton. Acceleration of tensor-product operations for high-order finite element methods. *The International Journal of High Performance Computing Applications*, 33(4):735–757, 2019.

[70] Nobuatsu Tanaka. A review of wavelet-based conjugate gradient method for solving poisson equations. *Science bulletin of Josai University, Special Issue*, 5:71–79, 1998.

[71] Nobuatsu Tanaka. *A Wavelet-Based Conjugate Gradient Method for Solving Poisson Equations*, chapter 3, pages 45–73. SIAM, 1998.

[72] Nobuatsu Tanaka. Wavelet-preconditioned conjugate gradient poisson solver and its parallel processing. *Comput. Mech.*, 23(2):555–561, 1999.

[73] Nobuatsu Tanaka. Parallel processing of haar-wavelet-preconditioned conjugate gradient methods. *Transactions of the Japan Society for Computational Engineering and Science*, 2001:20010003–20010003, 2001.

[74] Nobuatsu Tanaka, Haruo Terasaka, Takeshi Shimizu, and Yukio Takigawa. Incomplete discrete wavelet transform and its application to a poisson equation solver. *Journal of Nuclear Science and Technology*, 33(7):555–561, 1996.

[75] Jeremy L. Thompson. LFAToolkit. `https://github.com/jeremylt/LFAToolkit.jl`, 2021.

[76] Unstructured grid adaptation working group, 2022. https://ugawg.github.io/.

[77] Unstructured Grid Adaptation Working Group. Adapt Benchmarks: Cone-Cone. `https://github.com/UGAWG/adapt-benchmarks/tree/master/cone-cone`, 2022.

[78] Z.J. Wang, Krzysztof Fidkowski, Rémi Abgrall, Francesco Bassi, Doru Caraeni, Andrew Cary, Herman Deconinck, Ralf Hartmann, Koen Hillewaert, H.T. Huynh, Norbert Kroll, Georg May, Per-Olof Persson, Bram van Leer, and Miguel Visbal. High-order cfd methods: current status and perspective. *International Journal for Numerical Methods in Fluids*, 72(8):811–845, 2013.

[79] Roman Wienands and Wolfgang Joppich. *Practical Fourier analysis for multigrid methods*. CRC press, 2004.