ECP Milestone Report
Performance tuning of CEED software and first wave apps
WBS 2.2.6.06, Milestone CEED-MS20

Stanimire Tomov
Pedro Bello-Maldonado
Jed Brown
Jean-Sylvain Camier
Veselin Dobrev
Jack Dongarra
Paul Fischer
Azzam Haidar
Tzanio Kolev
Elia Merzari
Misun Min
Aleks Obabko
Scott Parker
Thilina Ratnayaka
Jeremy Thompson
Ahmad Abdelfattah
Vladimir Tomov
Tim Warburton

September 28, 2018

ECP Milestone Report
Performance tuning of CEED software and first wave apps
WBS 2.2.6.06, Milestone CEED-MS20

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

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

September 28, 2018

ECP Milestone Report
Performance tuning of CEED software and first wave apps
WBS 2.2.6.06, Milestone CEED-MS20

Approvals

Submitted by:

Tzanio Kolev, LLNL
CEED PI

Date

Approval:

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

Date

Revision Log

Version	Creation Date	Description	Approval Date
1.0	September 28, 2018	Original	

EXECUTIVE SUMMARY

The goal of this milestone was the performance tuning of the CEED software and first wave apps.

In this milestone, the CEED team developed optimization techniques and tuned for performance the CEED software. Specifically, the focus was on the following:

- Fast finite element operator storage and evaluation using partial assembly/matrix-free algorithms that take advantage of tensor-product element structure;
- Architecture optimizations, including the development of performant discretization libraries targeting heterogeneous systems with multi-core CPUs, GPUs, and/or many-core CPUs in single node, cluster, or large-scale ECP supercomputer configurations;
- Global kernels for finite element operators, including the development of algorithms and libraries for fast gather-scatter exchanges for action assembly on unstructured graphs and for components needed for global linear solvers.

A main part of this milestone was the performance tuning of the CEED first-wave ECP applications. This included the ExaSMR application – Coupled Monte Carlo Neutronics and Fluid Flow Simulation of Small Modular Reactors (ORNL), and the MARBL application – Next-Gen Multi-physics Simulation Code (LLNL).

The artifacts delivered include performance improvements in CEED's 1st wave of applications, and tuned CEED software for various architectures through a number of backends, freely available in the CEED's repository on GitHub. See the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed` for more details.

In addition to details and results from the above R&D efforts, in this document we are also reporting on other project-wide activities performed in Q4 of FY18 including: CEED's second annual meeting, a successful minisymposium at the premier international conference on high-order methods (ICOSAHOM), making the CEED milestone reports publicly available, new Laghos release, and other outreach efforts.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The goal of this milestone was the performance tuning of the CEED software and first wave apps.

In this milestone, the CEED team developed optimization techniques and tuned for performance the CEED software. Specifically, the focus was on the following:

- Fast finite element operator storage and evaluation using partial assembly/matrix-free algorithms that take advantage of tensor-product element structure;
- Architecture optimizations, including the development of performant discretization libraries targeting heterogeneous systems with multi-core CPUs, GPUs, and/or many-core CPUs in single node, cluster, or large-scale ECP supercomputer configurations;
- Global kernels for finite element operators, including the development of algorithms and libraries for fast gather-scatter exchanges for action assembly on unstructured graphs and for components needed for global linear solvers.

A main part of this milestone was the performance tuning of the CEED first-wave ECP applications. This included the ExaSMR application – Coupled Monte Carlo Neutronics and Fluid Flow Simulation of Small Modular Reactors (ORNL), and the MARBL application – Next-Gen Multi-physics Simulation Code (LLNL).

The artifacts delivered include performance improvements in CEED's 1st wave of applications, and tuned CEED software for various architectures through a number of backends, freely available in the CEED's repository on GitHub. See the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed` for more details.

2. ARCHITECTURE OPTIMIZATIONS

2.1 Fast Algorithms

2.1.1 Pre-assembled vs. Partial Assembly

While a global (parallel) sparse matrix is a good representation of a PDE operator discretized with low-order elements, a global parallel matrix is a poor choice when discretizing with high-order elements, due to the large cost of both the memory transfer and floating point operations.

CEED is developing an alternative operator format, based on the CEED low-level API, that allows efficient operator evaluation that is optimal in memory and nearly-optimal in FLOPs cost. The CEED low-level API, libCEED operates with the foundational components of finite element operators, described by the decomposition given in Figure 1.

To achieve high-performance, it is critical to take advantage of the tensor-product structure of both the finite element basis and the quadrature rule to efficiently apply the action of B without necessarily computing its entries. This is generally know as sum factorization. In the case where we precompute and store the D matrix, we call the algorithm partial assembly.

The low-level API allows us to provide multiple high-level APIs that are necessary to enable application to take advantage of CEED-developed high-order technologies at the level they are comfortable with, while taking advantage of the tensor-product structure of the finite element basis and the quadrature rule to enable efficient implementations on modern hardware.

Illustration of the benefits of matrix-free vs. assembled matrices for each element can be demonstrated with the following results on NVIDIA V100 GPUs. First, we derive a simple but effective performance model – the kernels of interest are strictly memory bound since each element requires a unique matrix and performs a matrix-vector product that cannot gain performance from reusing cached data. An estimate on the throughput in nodes per second is bound by:

$$\text{GNODES/s} = (\text{device bandwidth in GB})/(\ \text{sizeof(dfloat)} * (\text{Np} + 2)\ ),$$

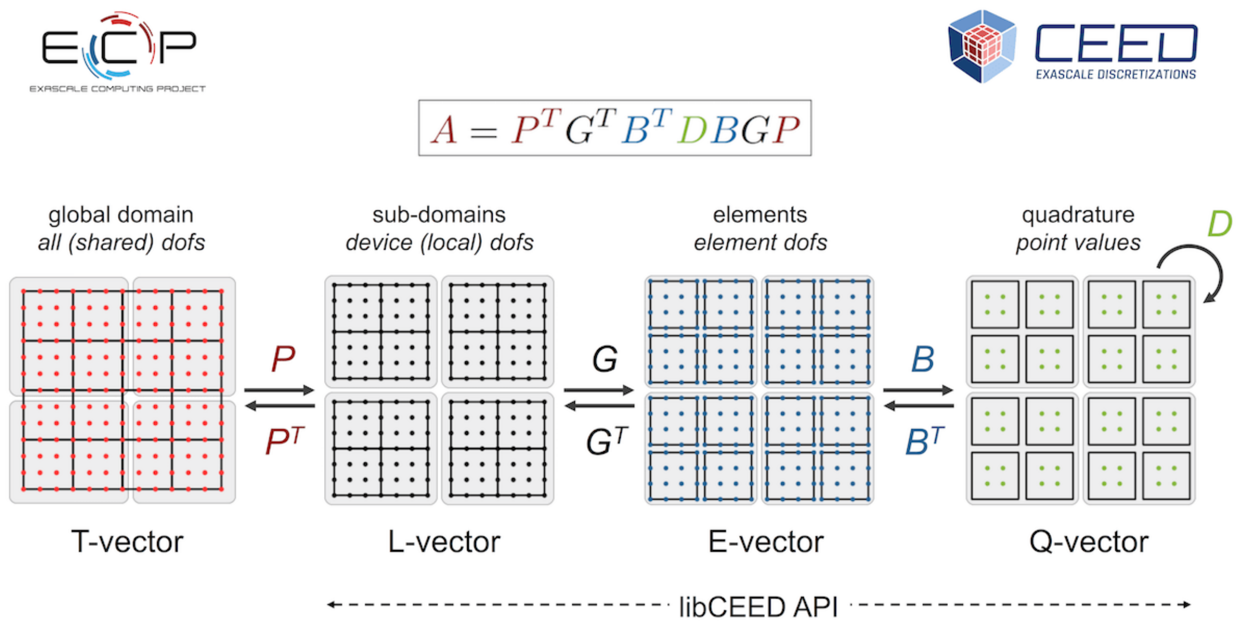


Figure 1: High-order finite element operators representation in the CEED low-level API, libCEED.

where Np is the number of basis functions, e.g., for a standard tetrahedral element of degree N this is given by

$$\text{Np} = (\text{N}+1) * (\text{N}+2) * (\text{N}+3)/6.$$

There are relatively few optimization options for implementing and tuning kernels that use pre-assembled matrices since the operation reduces to streaming the matrices from device memory and caching the vector in shared memory. Starting from our early work first reported in milestone CEED-MS13 [2], we have studied and illustrated on particular kernels the main optimization which is important at low order of SIMD cramming so that we use close to a multiple of 32 threads per thread-block [21]. The results of the comparison are summarized in Figure 2. Shown is the node throughput rates measured on an NVIDIA V100 for assembled element matrices (Left; demoted *partially assembled matrix version*) and for the matrix-free version (Right).

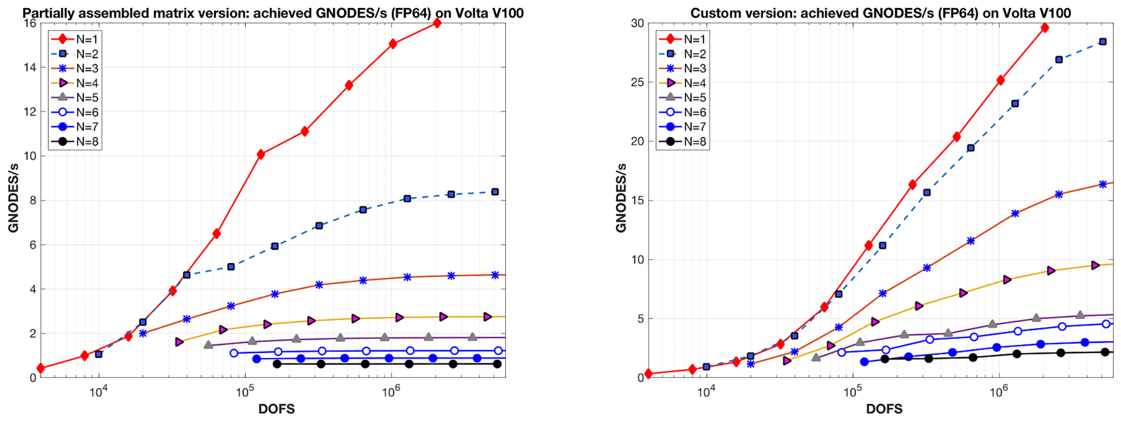


Figure 2: Node throughput rates measured on an NVIDIA V100 for assembled element matrices (Left) vs. matrix-free (Right).

We note that the achieved throughput of the matrix-free version is significantly faster. Further detail on our analysis and conclusions are available in [21].

2.1.2 BLAS vs. Custom Kernels

The BLAS (Basic Linear Algebra Subprograms) standard is an API for performing common, basic linear algebra routines. The appeal of using the BLAS API is that it is an accepted standard and implementations have been highly optimized for various architectures (e.g., by vendors in their corresponding math libraries like cuBLAS, MKL, ESSL, etc.). Thus, algorithms written in terms of BLAS benefit from both functional and performance portability across architectures.

A significant part of the work performed by the CEED finite element codes consists of operations that can be expressed as BLAS routines (i.e., dot products, vector updates, matrix-vector, and matrix-matrix multiplications). We usually develop custom kernels that combine several BLAS-type operations to reduce memory traffic or memory usage. For instance, we often implement *matrix-free* operations where we apply the action of a matrix on a vector without storing the matrix explicitly (as described in Section 2.1.1).

Thus, it is of high interest to compare the two approaches, quantify the differences in performance, and derive a strategy for developing fast high-order algorithms. We did extensive optimizations and comparison studies in order to gain insight into the strong and weak scalability characteristics of each approach.

Figure 3 compares the performance in GFlop/s of the two approaches on NVIDIA V100 GPU. DOFS in the figures refers to DOFS=Np*E, where Np is the number of basis functions as given in Section 2.1.1, and E is element count chosen from meshes with [1,000 2,000 4,000 8,000 16,000 32,000 64,000 128,000 256,000 512,000] elements.

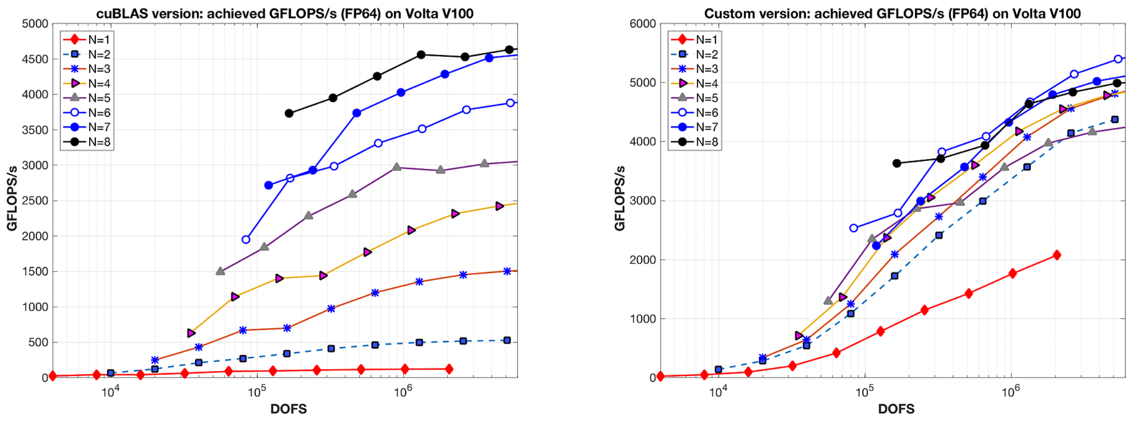


Figure 3: Performance in GFlop/s on NVIDIA V100 GPU on stiffness matrix action on tetrahedral elements using cuBLAS (Left) vs. custom build kernels (Right).

The custom kernel uses our best performing OCCA kernel [20]. The cuBLAS approach uses a matrix-matrix product (CUBLAS DGEMM) plus a streaming kernel to apply the chain rule that relates derivatives computed with respect to local elemental coordinates to derivatives in physical coordinates.

Our conclusion is that the custom kernel clearly “wins”, especially for lower degrees. At the lowest orders the computation is memory bound and using cuBLAS in the above manner requires us to write out 7 intermediate values per FEM node and then read them in again in the chain rule kernel. The custom version of the kernel compute the matrix product and chain rule without saving intermediate results, while achieving roofline performance through efficient blocking (see [20] for further details).

In summary, the custom approach scales better and cuBLAS-based version starts to be competitive at N=7. Although the cuBLAS implementation exercise is simpler than designing, implementing, testing, modeling, and tuning bespoke kernels it also induced significantly more data movement and required excessive temporary data storage. Finally, the tuned OCCA kernels deliver strong performance at all orders in contrast to cuBLAS which underpierformed until N=7.

In addition, we emphasize that in our case, we needed a two-step approach: cuBLAS function followed by chain rule kernel, which means we launch two kernels. In addition, we were forced to allocate more array space to hold the intermediate results. The conclusions of this study might had been very different if we were able to allocate the same amount of array space for both approaches and replace the custom bake-off approach with just one cuBLAS function. This is an approach that we pursue in one of the libCEED backends – the MAGMA backend – that we discuss in Section 2.3.4.

Further detail on our analysis and conclusions are available in [20].

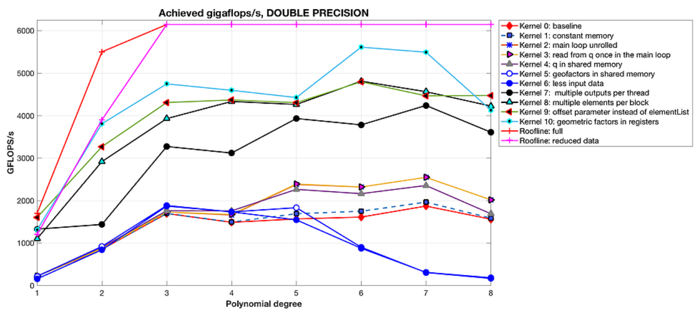
2.1.3 Kernel Optimizations

We did a number of kernel performance optimizations for GPUs. Our results in this section show that there is a substantial difference in performance between basic GPU code and highly optimized GPU code. To illustrate the effect of performance optimizations and tuning, we use a simple, yet demanding (high arithmetic intensity) matrix-vector kernel coming from FEM. The piece of code in question is executed multiple times per time step for time dependent flow calculations that might require hundreds of thousands of time steps, hence we would like to make it as fast as possible. We used OCCA in the implementations but the same techniques can be equally well applied directly in CUDA and OpenCL.

The code is executed for polynomial degrees N=1,2, ..., 8, using a mesh with $E = 320,000$ affine tetrahedral elements. The number of nodes per element is given by the formula: Np = (N+1)(N+2)(N+3)/6, as in the previous subsections.

First, we describe how many flops (per element) do we perform. In our particular case this is: $20 * p_{Np}^2 + 20 * p_{Np}$, where p_{Np} is the number of nodes per element. Note that p_{Np} increases cubically with the polynomial degree and number of flops is proportional to the sixth power of the polynomial degree - exhibiting the dreaded curse of dimensionality. Thus, the number of flops for $N = 1..10$ is respectively 400, 2200, 8400, 25200, 63840, 142800, 290400, and 547800.

We developed a sequence of ten optimizations that we describe in detail in [19]. The performance results are summarized in Figure 4 for an NVIDIA Titan V GPU in double precision arithmetic (Left).



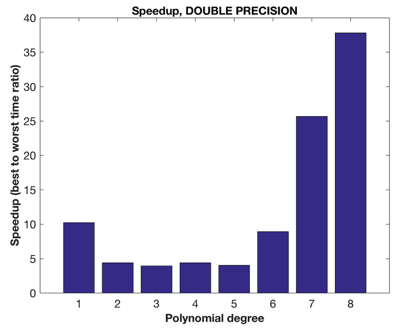


Figure 4: Performance results for ten optimizations [19] on an NVIDIA Titan V GPU in double precision arithmetic (Left), and corresponding speedup effects (Right).

In conclusion, it is clear that there is still a gap between the roofline and the achieved performance. The gap is more pronounced for mid-range degrees. We hypothesize that this gap results from L1 and L2 cache misses. For example, in double precision and N=8, the matrices alone take more than 4MB, which overflows even L2 cache. Also, note that we have used manufacturer peak floating point performance for the roofline plateaus. For an empirical estimate of the actual achievable peak performance on the Titan V see the earlier blog entry discussing the results from occaBench. Using the empirical peak suggest that the above results are close to best achievable performance for this kernel.

2.2 libParanumal

The libParanumal library is an experimental test-bed for exploring plugin GPU capable components that can be integrated into existing high-order finite element codes as optional accelerator modules. This library is being actively developed at Virginia Tech as part of the CEED project. Figure 5 shows the base library structure.

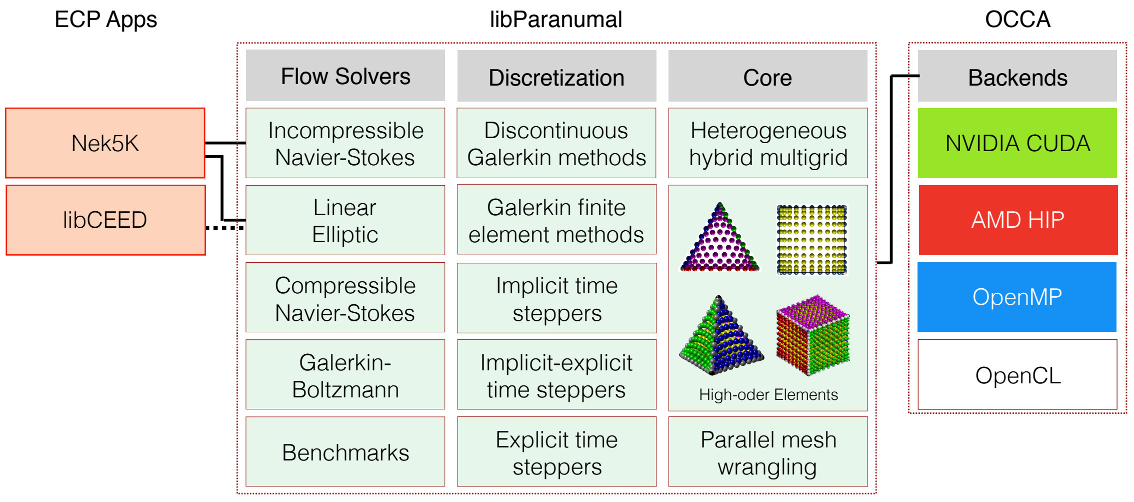


Figure 5: libParanumal: Library Structure and Flow Solvers with Plugins for Existing DOE ECP Packages.

The library uses the newly released OCCA 1.0 to deliver native CUDA/OpenCL/OpenMP performance for GPU and/or CPU calculations. The library includes carefully optimized kernel implementations of the most computationally intensive calculations for high-order finite element operations (see for example [16]).

The core sub-libraries of libParanumal include linear solvers and multigrid preconditioners that are fully OCCA accelerated and are customized specifically for linear systems stemming from high-order finite element discretization. The library further includes reference high-order finite element based solvers for incompressible Navier-Stokes; compressible Navier-Stokes; Galerkin-Boltzmann based gas dynamics; linear acoustics; and elliptic Poisson potential problems. A partial list of libParanumal capabilities follows:

A. Supported elements:
   - Meshes consisting of triangles, quadrilaterals, tetrahedra, or hexahedra.
   - Lagrange basis functions up to degree 15.
   - Partial support for Bezier-Bernstein basis functions.

B. Elliptic solver:
   - Linear Poisson and screened Poisson potential solvers.
   - GPU optimized matrix-vector products.
   - Hybrid p-type multigrid and algebraic multigrid preconditioned conjugate gradient solver.
   - Sparse matrix or nearly matrix-free algebraic multigrid for coarse levels of multigrid hierarchy.

D. Heterogeneous accelerated flow solvers:
   - Linearized Euler equations.
   - Isothermal compressible Navier-Stokes solver with:
     * Upwind discontinuous Galerkin discretization in space.
     * Dormand-Prince adaptive Runge-Kutta integration in time.

  - Isothermal Galerkin-Boltzmann gas dynamics solver with:
    - Penalty flux DG discretization in space.
    - Adaptive semi-analytic (pointwise exponential) integration in time.
    - Multi-axial quasi-perfectly matched absorbing layer far field boundary condition.
  - Incompressible Navier-Stokes solver with:
    - Choice of continuous FEM or interior penalty DG in space.
    - Extrapolation-BDF integration in time.
    - Sub-cycling (Operator Integration Factor Splitting) for advection.

E. Dependencies: MPI, gslib, OCCA.

The initial open source release of libParanumal library [22] was committed on 8/1/18. Publications related to libParanumal that have been submitted include [9, 16, 8].

It is now possible to use libParanumal as an optional flow simulation engine callable from within Nek5000, using its internal curvilinear hexahedral mesh representation. For more details, see Section 3.1.

2.3 libCEED-0.3

The next release of the CEED API library, libCEED v0.3, was released with several new features, significant performance improvements, improved continuous integration, and many new tests with code coverage reports (currently about 90%).

2.3.1 *Active and passive fields*

In this significant change to the public interface, `CeedQFunction`s now takes any number of named input and output arguments while `CeedOperator` connects them to the actual data, which may be supplied explicitly to `CeedOperatorApply()` (active) or separately via `CeedOperatorSetField()` (passive). This interface change enables reusable libraries of `CeedQFunction`s and composition of block solvers constructed using `CeedOperator`.

2.3.2 *Optimized CPU backend*

A concept of blocked restriction has been added to libCEED and used in an optimized CPU backend. Although this is typically not visible to the user, it enables effective use of arbitrary-length SIMD while maintaining cache locality. This approach is essential for performance at relatively low order and competitive any time the number of elements per core is significant. A different optimization strategy (based on internal vectorization) will be needed for optimal SIMD performance with high order elements in the strong scaling limit where the number of elements per core is smaller than the SIMD length.

This CPU backend also implements an algebraic factorization of tensor product gradients to perform fewer operations than standard application of interpolation and differentiation from nodes to quadrature points. This algebraic formulation automatically supports non-polynomial and non-interpolatory bases, thus is more general than the more common derivation in terms of Lagrange polynomials on the quadrature points.

2.3.3 *Initial non-tensor bases capability*

libCEED has an initial capability to handle elements with non-tensor bases. Due to a lack of a standardized convention for quadrature on triangles, there is currently not a constructor for Lagrange elements of an arbitrary order for non-tensor bases. The user must provide the quadrature points and weights, in addition to the full interpolation and gradient matrices.

This capability is currently implemented in the CPU family of backends: the reference backend, the optimized CPU backed, and the template backend. These new bases use the same `CeedBasisApply` and `CeedBasisCreate` functions, but with a new constructor, shown below.

```
CeedBasisCreateH1(ceed, CEED_TRIANGLE, 1, P, Q, interp, grad, qref, qweight, &b);
```

2.3.4 MAGMA backend

A main component of the performance tuning efforts in the CEED software has been the extension of the MAGMA backend. The MAGMA backend relies on the MAGMA library to provide the libCEED functionalities following the libCEED API specifications. The main differentiation with the other backends is that the MAGMA backend approach is based on the use of BLAS, as well as codesign efforts to extend the BLAS standard and provide highly tuned implementations that will overcome the performance limitations currently associated with the BLAS approach, as explained in Section 2.1.2.

Batched BLAS While the use of BLAS has proven to be very effective in developing portable and efficient software, the existing BLAS is not adequate for batched computations involving thousands of (or more) small problems. This is the case in high-order method, where the operators are matrix-free, and the operator evaluations and applications require batched operations over the finite elements, as described in Section 2.1.1. To address this drawback, we have been leading efforts to codesign, discuss, and formalize details related to a batched BLAS API standard. This is an ongoing effort to reach out to various ECP application and software developers, and to a broad community of scientific computing users, as well as library and application developers. Main ECP hardware vendors already provide some Batched BLAS routines in their math libraries. Recent activities on these co-design efforts included a BoF at SC17, two mini-symposium sessions at SIAM PP18 and a session at GTC18.

We have developed, tuned, and released through MAGMA the most used Batched BLAS. The MAGMA backend in CEED needs Batched matrix-matrix double-precision multiplication kernels (DGEMMs) that we developed and tuned for the latest architectures [12]. For example, on a V100 GPU for square matrices of size 32, we achieve an execution rate of about $1,600$ gigaFLOP/s in double-precision arithmetic, which is 95% of the theoretically derived peak for this computation on a V100 GPU. These results outperform currently available state-of-the-art implementations such as vendor-tuned math libraries, including Intel MKL and NVIDIA CUBLAS, as well as open-source libraries like OpenBLAS and Eigen. Figure 6 illustrates the performance obtained on batched DGEMMs for two 10-core IBM Power8 CPUs (Left), NVIDIA V100 GPU (Center), and a 10-core Intel Xeon E5-2650 v3 (Haswell) CPU (Right).

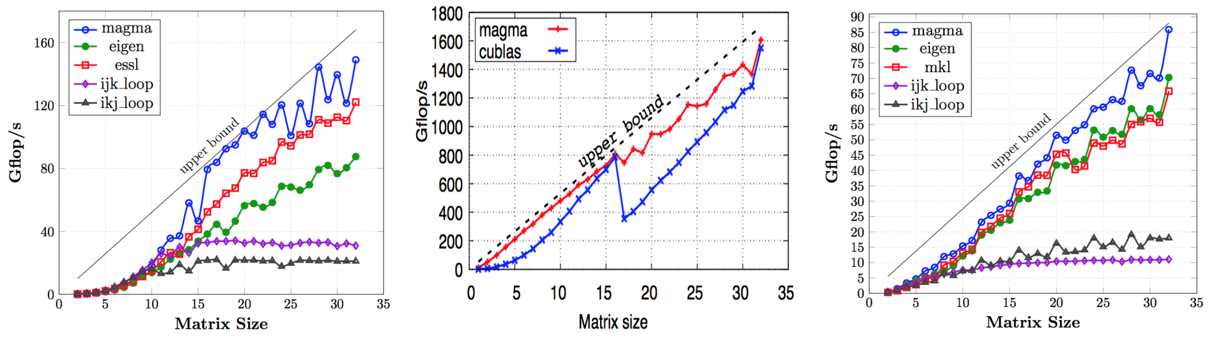


Figure 6: Batched DGEMM performance on two IBM Power8 CPUs (Left), NVIDIA V100 GPU (Center), and an Intel Haswell CPU (Right).

Fast Tensor Contractions through Fusing Batched BLAS Kernels The CEED operators have tensor-product element structure. We take advantage of that by representing the tensor contractions as sequence of matrix-matrix multiplications. For high-order methods, where the data for single DGEMM does not fit in the fast memory (e.g., L1 cache and registers for GPUs), we apply blocking techniques and use implementations that are in the current Batched BLAS libraries. However, when the matrices fit in fast memory, it is beneficial to fuse the sequence of operations for the tensor contractions in a single kernel. This removes the intermediate write to global memory of results from one Batched BLAS call, and possibly read from global memory for the next Batched BLAS call. This was explained in Section 2.1.1. Fusing kernels is critical for performance for these smaller sizes contractions as they minimize communications, and these are the best performing kernels that we have implemented for various cases so far.

Custom kernels though do not follow the BLAS API, may be more difficult to read (and hance understand and modify when needed), and we have to maintain them, tune across architectures, and develop custom kernels for many applications. Therefore, we have also been targeting developments based on Batched BLAS but with the possibility to fuse batched kernels into a single batched kernel. For example, the high-order operators can be expressed as a batch over the finite elements e, e.g.,

$$\texttt{batch<e=0..nelems>}\{\ B_e^T D_e.*(B_e A_e {B_e}^T) B_e\ \},$$

where the matrices involved are small and dense (except D_e that is diagonal), and depend on the particular element e. The effect of this optimization is shown in Figure 7 for NVIDIA V100 GPUs. Note that even for these small sizes, performance is still memory bound but gets close to the compute peak of the machine (which is around $7,000$ GFlop/s).

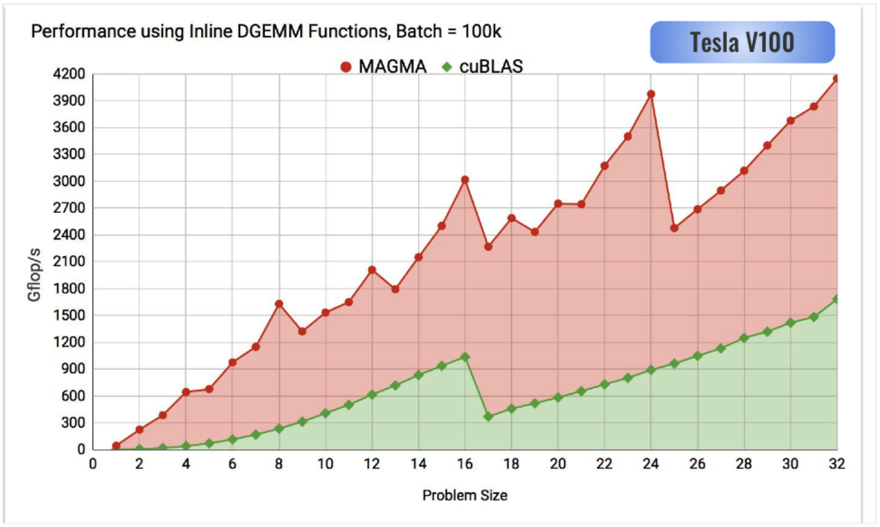
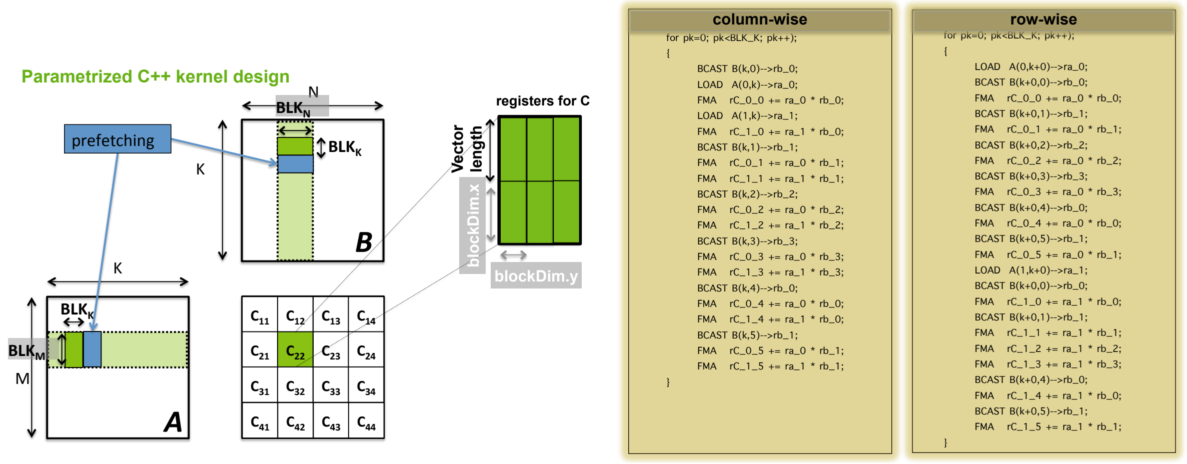


Figure 7: Performance of fused vs. un-fused Batched DGEMMs of small sizes on NVIDIA V100 GPU.

We enabled this optimization by providing device interfaces to BLAS that can be used to derive custom algorithm that fuse Batched BLAS calls. The data used by the new fused batched kernel is loaded at the beginning into shared memory and registers and subsequently used by the device BLAS calls through the shared memory or register shuffling (when needed). Finally, the result is written back to the main memory only once. Our efforts that initially introduced this approach were first reported in the CEED-MS13 report [2] and were further developed, extended, and tuned for more kernels in this milestone.

Kernel Auto-Generation and Autotuning Process for Performance Portable Versions The kernels developed for the MAGMA backend in CEED were parameterized and implemented using C++ features, including templates and overloaded functions. The gemm kernel design in particular for small matrix sizes is illustrated in Figure 8. The matrix C is split into blocks C_{ij} of size $BLK_M \times BLK_N$ that can be computed in parallel. The idea is that since C is where the computations are accumulated and the final result written, it is better to keep as large a part of C as possible in registers during the accumulation of the multiplication. Note that this one-level design of blocking is especially designed for small matrices; for larger matrices, a design with multiple levels of blocking may be better in order to account for blocking on the possibly multiple levels of the architecture's memory hierarchy layers. Any particular block C_{ij} of C will be held in registers for either the CPU or GPU case. The number of rows in C_{ij} is better to be multiple of the vector length for CPUs, or multiple of the number of threads in the "x" dimension for GPUs. Also, the number of columns will be dependent on the available registers (CPUs or GPUs) and on the number of threads in the "y" dimension for the GPU case. There is a sliding window of size $BLK_M \times BLK_K$ that reads data of the matrix A and, similarly, a sliding window of size $BLK_K \times BLK_N$ that reads data from the matrix B. This data can be read into register or into cache (shared memory or register in case of the GPU kernel). The innermost loop will multiply the green portion of A and B and will accumulate the result into the green portion of C. Note that the blue portion of A and B corresponds to the prefetching when it is enabled by the kernel generator (the kernel generator will generate two kernels w/o prefetching). The windows of A and B slide horizontally and vertically, respectively, and once finished, the block of C contains the final results of $A \times B$. This result is multiplied by α (when α is not equal to one) and added to the corresponding block of the matrix C (loaded from the main memory and multiplied by β—when β is not equal to one—before the addition, and the result is stored back into the main memory). If β is zero, the results of the multiplication are directly stored into the main memory.



column-wise

```
for pk=0; pk<BLK_K; pk++);
{
        BCAST B(k,0)-->rb_0;
        LOAD  A(0,k)-->ra_0;
        FMA   rC_0_0 += ra_0 * rb_0;
        LOAD  A(1,k)-->ra_1;
        FMA   rC_1_0 += ra_1 * rb_0;
        BCAST B(k,1)-->rb_1;
        FMA   rC_0_1 += ra_0 * rb_1;
        FMA   rC_1_1 += ra_1 * rb_1;
        BCAST B(k,2)-->rb_2;
        FMA   rC_0_2 += ra_0 * rb_2;
        FMA   rC_1_2 += ra_1 * rb_2;
        BCAST B(k,3)-->rb_3;
        FMA   rC_0_3 += ra_0 * rb_3;
        FMA   rC_1_3 += ra_1 * rb_3;
        BCAST B(k,4)-->rb_0;
        FMA   rC_0_4 += ra_0 * rb_0;
        FMA   rC_1_4 += ra_1 * rb_0;
        BCAST B(k,5)-->rb_1;
        FMA   rC_0_5 += ra_0 * rb_1;
        FMA   rC_1_5 += ra_1 * rb_1;
}
```

row-wise

```
for pk=0; pk<BLK_K; pk++);
{
        LOAD  A(0,k+0)-->ra_0;
        BCAST B(k+0,0)-->rb_0;
        FMA   rC_0_0 += ra_0 * rb_0;
        BCAST B(k+0,1)-->rb_1;
        FMA   rC_0_1 += ra_0 * rb_1;
        BCAST B(k+0,2)-->rb_2;
        FMA   rC_0_2 += ra_0 * rb_2;
        BCAST B(k+0,3)-->rb_3;
        FMA   rC_0_3 += ra_0 * rb_3;
        BCAST B(k+0,4)-->rb_0;
        FMA   rC_0_4 += ra_0 * rb_0;
        BCAST B(k+0,5)-->rb_1;
        FMA   rC_0_5 += ra_0 * rb_1;
        LOAD  A(1,k+0)-->ra_1;
        BCAST B(k+0,0)-->rb_0;
        FMA   rC_1_0 += ra_1 * rb_0;
        BCAST B(k+0,1)-->rb_1;
        FMA   rC_1_1 += ra_1 * rb_1;
        FMA   rC_1_2 += ra_1 * rb_2;
        FMA   rC_1_3 += ra_1 * rb_3;
        BCAST B(k+0,4)-->rb_0;
        FMA   rC_1_4 += ra_1 * rb_0;
        BCAST B(k+0,5)-->rb_1;
        FMA   rC_1_5 += ra_1 * rb_1;
}
```

Figure 8: Design and auto-generation of the dgemm kernel: parameterized C++ dgemm kernel design (Left) and auto-generated code for the dgemm kernel for the same configuration but with two different inner loop order (Right).

The same methodology applies when any of the matrices is transposed, and the code generation is always handled automatically. C_{ij} is always of size $BLK_M \times BLK_N$ and the reading of A and B always happens following the block design (e.g., contiguous block of the size $BLK_M \times BLK_K$ and $BLK_K \times BLK_N$, resp., for the Non-Transpose). As a result, the transpose is implicitly coded through the innermost loop when the data is already in cache. Moreover, the description here was provided for square matrices, but the same applies for rectangular matrices as well. The matrix C is always split over blocks, and therefore the case of rectangular matrices can be generalized to follow the same methodology. This is also valid for the GPU implementation. We also note that, since the read/store happens by block, a matrix stored in row-major format can also be handled by the same techniques. In this case, the window slides vertically on A and horizontally on B. It can also be handled by flipping the operations from non-transpose to transpose. For example, if the matrix A is the only matrix stored in row-major and the operation is $C = A \times B$, then this can be computed by the $C = A^T \times B$ kernel where A is considered stored in column-major format.

The ultimate goal is to explore all possible kernel configurations, called "*the autotuning search space*," and provide a clear description of the kernel generation and the autotuning process to be performed in order to get the best performance. As described above, for every architecture, there might be a very large number of possibilities for designing the matrix-matrix multiplication kernel. If we take for example an 8×8 matrix, on a hardware that has 16 256-bit AVX-2 registers, we can decide to hold all of B in registers and keep loading/reloading A and C, or we can decide to use only 8 registers to hold a portion of B and minimize the number of loads/reloads on A and C, and so on. The same scenario will be applicable to C and to A. Thus, the decision of how many registers we must dedicate to each array (e.g., A, B, and C) can generate many configurations (about a thousand). Furthermore, one configuration might be good for one matrix size but bad for other matrix sizes. In addition to that, there is the loop order: should the innermost loop go "*row-wise*" or "*column-wise*," should we implement the *ijk*, *ikj*, *kij*, or other loop orders? Thus, for every loop order configuration, since we have about one thousand configurations for the registers, one might end up with about ten thousand configurations. This is what makes up the search space. Then, in order to exploit such a large search space of possibility in the shortest time, we apply an aggressive pruning technique to reduce it. A condition of the pruning is that only the kernel configurations that have absolutely no chance of achieving good performance be eliminated.

Because our design is parameterized, once all the possible and acceptable configurations are created, the kernel generator creates one or many kernels for every configuration. For every configuration, the difference between the kernels can be the fashion of the innermost loop, e.g., "*row-wise*" or "*column-wise*," the whole nested loop order (e.g., *ijk*, *ikj*, *kij*, etc.), the instruction order, etc. For example, a configuration specifies the blocking sizes (BLK_M, BLK_N, and BLK_K) and the number of registers allocated for each variable A, B, and C. Then, the generator creates many possible kernels for this configuration. An example of two CPU generated kernels for the same configuration (2 registers for A, 4 registers for B, and 6 registers for C) is depicted in Figure 8, Right.

This new flexible and automated design for code and configuration generation enables us to easily design kernels for any architecture and to tune them and find the best kernel for each. This automated design did not exist in our previous work where we had to have different code snippets for every architecture and then tune it. Furthermore, we were able to extract from this tuning process the best configuration for these small sizes and write a parameterized C++ code for prefetch and loop unrolling on CPUs [12].

3. APPLICATIONS PERFORMANCE IMPROVEMENTS

3.1 Nek5000/libParanumal Performance Benchmarks on V100

As introduced in Section 2.2, libParanumal is a software developed by Tim Warburton's group at Virginia Tech, focusing on designing algorithms for finite element analysis that fully exploit the parallelism and data movement capabilities of GPUs. liParanumal supports GPU with OCCA provided with a unified APT for interacting with backend device APIs (OpenMPI, CUDA, OpenCL). Currently, its primary target is to run finite element solvers on CUDA GPUs. libParanumal can read Triangle, Quad, Hex and Tet meshes and solve incompressible Navier-Stokes, compressible Navier-Stokes, elliptic and few other equations on these meshes. Also, continuous and discontinuous Galerkin discretizations are also available. Various solver parameters can be tweaked by using a configuration file that is read before the numerical simulations are started. The skeleton of libParanumal is written in C++ and the main computational kernels are written in OCCA which can be compiled for different accelerator devices using CUDA, OpenCL, etc.

Nek5000 can successfully interface with libParanumal and can run complicated simulations using libParanumal as the main solver engine. Nek5000 can use libParanumal in parallel using MPI as well. We can read in a Nek5000 mesh, partition the mesh on Nek5000 side and then set the required data structures on libParanumal side to drive the required solvers from Nek5000. The process of setting up Nek5000 to use libParanumal can be described in five main steps:

1. Initialize the libParanumal library using the MPI communicator from Nek5000.
2. Initialize the Nek5000 mesh on libParanumal side provided with the required boundary conditions and get a handle for the mesh. Also, setup the data structures to be used for the solvers.
3. Write a header file for the initial condition and boundary conditions.
4. Setup the solvers with required parameters and solve the problem.
5. Cleanup the created handles.

Figure 9 demonstrates approximately 4× speedup on a single GPU, compared to previous Nek5000 + OpenACC version for simulating the singlerod mesh ($E = 2560, N = 7$) in Figure 10 (left) by Nek5000 + libParanumal which has the highly optimized tuned kernels in the OCCA backend. In particular, the simulations using 1 GPU ($n/p = 1,310,720$ grid points per GPU) and 8 GPUs ($n/p = 163,840$ grid points per GPU) performs faster than the one using 64 CPU cores ($n/p = 20,480$ grid points per CPU). Table 1 shows the performance profiling on ANL/JLSE DGX-1 Volta server using 8 GPUs, demonstrating 22% for computing the elliptic operator and 8% for the gather-scatter operations.

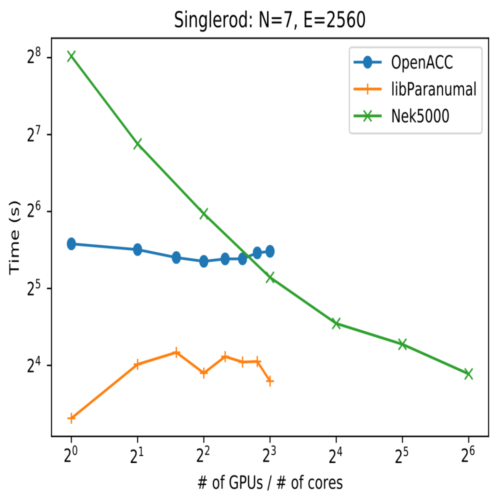


Figure 9: Strong-scaling on V100 for ExaSMR singlerod simulations by libParanumal (+ Nek5000), OpenACC (+ Nek5000), and Nek5000 (CPU).

We note the poor scaling performance on multi-GPU for this case, which was always meant as a single GPU performance test. This is expected as the numbers of degrees of freedom is likely not enough to even saturate a single GPU. For the pure OpenACC version saturation occurs at over 3,000,000 degrees of freedom per GPU. Future studies will characterize the saturation limit for this version and a more extensive strong-scaling and weak-scaling studies with the larger problem with 17x17 rods ($E = 221,600$) in Figure 10 (right) on OLCF Summit once it becomes available for us to access.

3.2 ExaSMR: Algorithmic Performance Improvements

The CEED team is engaged with the ECP ExaSMR team for performance improvements through advanced algorithmic developments as well as performance tuning. For the thermal-hydraulics analysis, hundreds of thousands of flow channels comprise turbulent flow with very fine solution scales. The channels are typically hundreds of hydraulic diameters in length.

For full reactor-core simulations, the ExaSMR strategy is to use Reynolds-Averaged Navier Stokes (RANS) in the majority of the core with more detailed large eddy simulations (LES) in critical regions. In addition, while the turbulence is challenging to resolve, it tends to reach a statistically fully-developed state within just a few channel diameters, whereas thermal variations take place over the full core size. This poses a challenge for coupled calculations. It is unpractical and too expensive to consider performing a full LES calculations. Some acceleration to couple the solution through the use of steady-state solvers is likely necessary.

Table 1: Performance profile of libParanumal on ANL/JLSE DGX-1 Volta server using 8 GPUs for singlerod mesh with ($E = 2560, N = 7, n = 1,310,720$).

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   22.84%  1.42900s      3341  427.72us  420.99us  435.55us  _occa_ellipticPartialAxIpdgHex3D_0
                   17.51%  1.09593s     88566  12.374us  1.7270us  48.511us  _occa_scaledAdd_0
                   16.98%  1.06224s     32005  33.189us  9.6320us  134.05us  _occa_ellipticPartialAxHex3D_0
                   11.38%  712.03ms      3341  213.12us  205.18us  220.48us  _occa_ellipticPartialGradientHex3D_0
                    8.18%  511.86ms     36620  13.977us  5.1840us  57.568us  _occa_gatherScatter_0
                    6.48%  405.65ms     37212  10.901us  1.5040us  40.544us  _occa_dotMultiply_0
                    4.25%  265.88ms      9873  26.929us  17.855us  32.832us  _occa_innerProduct_0
                    1.39%  87.292ms     34970  2.4960us  1.0880us  39.999us  _occa_vectorAddKernel_0
                    1.19%  74.446ms      1898  39.223us  30.944us  45.535us  _occa_weightedInnerProduct2_0
                    1.16%  72.393ms     20439  3.5410us  1.6310us  12.928us  [CUDA memcpy DtoH]
                    1.11%  69.162ms      4221  16.385us  2.3990us  82.879us  [CUDA memcpy DtoD]
      API calls:   67.96%  5.63226s    236848  23.780us  1.3250us  1.0378ms  cuStreamSynchronize
                   24.65%  2.04328s    338961  6.0280us  4.2300us  21.429ms  cuLaunchKernel
                    6.40%  530.61ms     20439  25.960us  16.255us  130.81us  cuMemcpyDtoH
                    0.70%  58.150ms      4221  13.776us  6.5470us  3.8550ms  cuMemcpyDtoD
                    0.70%  58.150ms      4221  13.776us  6.5470us  3.8550ms  cuMemcpyDtoD
```

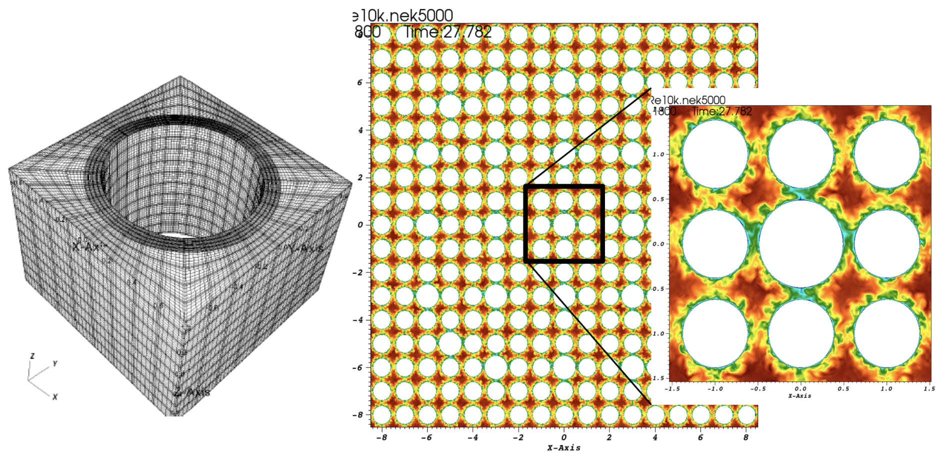


Figure 10: Spectral-element meshes for single rod (left) and 17×17 rods (right).

To accelerate the time-to-solution, CEED team is collaborating with ExaSMR team to develop fully implicit and steady state solvers for thermal transport and RANS. For the nonlinear Navier-Stokes (and RANS) transport, the Jacobian-free Newton Krylov (JFNK) routines from NekCEM's drift-diffusion solver [18] have been imported to Nek5000 and tested on several benchmark problems. This new steady-state solver includes an inexact (Jacobi-free) formulation based on a first-order Taylor series expansion [10].

3.2.1 Jacobian-free Newton Krylov Method Implementation into Nek5000

In JFNK approach, we consider the semi-discrete form based on spectral element method for the unsteady Navier-Stokes problems with a steady-state solution as

$$\frac{\partial u}{\partial t} = f(u) \to 0, \text{ as } t \ \to \infty, \tag{1}$$

and we define a pseudo-timestep using BDF1 with a time step size τ^n and introduce g_n at each time step which will approximate the root of $f(u)$ as $t \ \to \infty$:

$$g_n(u) := \frac{u - u^{n-1}}{\tau^n} - f(u) \to 0, \text{ as } t \ \to \infty. \tag{2}$$

At each pseudo-timestep, we solve $g_n(u) = 0$ by Newton iterations:

$$u_k^{n-1} = u_{k-1}^{n-1} + s_{k-1}^{n-1}, \tag{3}$$

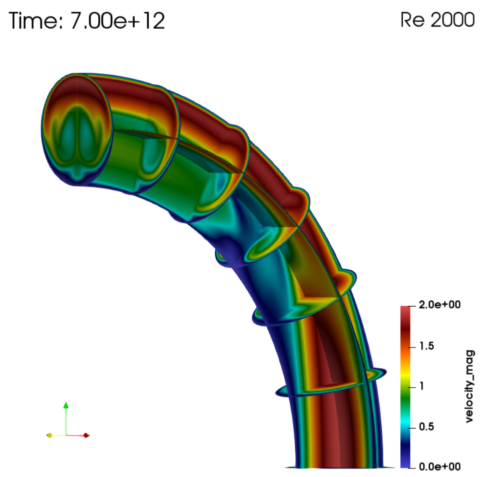
where s_k^n is obtained by GMRES solving a linear system

$$\mathbf{J}_{k-1}^{n-1} s_{k-1}^{n-1} = -g_n(u_{k-1}^{n-1}). \tag{4}$$

Within GMRES the action of the Jacobian matrix-vector product is approximated by a first order Taylor expansion for an arbitrary vector s, where f is approximated by BDF1 solver with Δt small enough:

$$\begin{aligned} \mathbf{J}_{k-1}^{n-1} s &= [g_n(u_{k-1}^{n-1} + \epsilon s) - g_n(u_{k-1}^{n-1})]/\epsilon & (5)\\ &= s/\tau^n - [f_n(u_{k-1}^{n-1} + \epsilon s) - f_n(u_{k-1}^{n-1})]/\epsilon. & (6) \end{aligned}$$

Figure 11 shows the case of Dean's flow (left) with rapid convergence to steady state solution with ~ 18 pseudo-time steps (right).



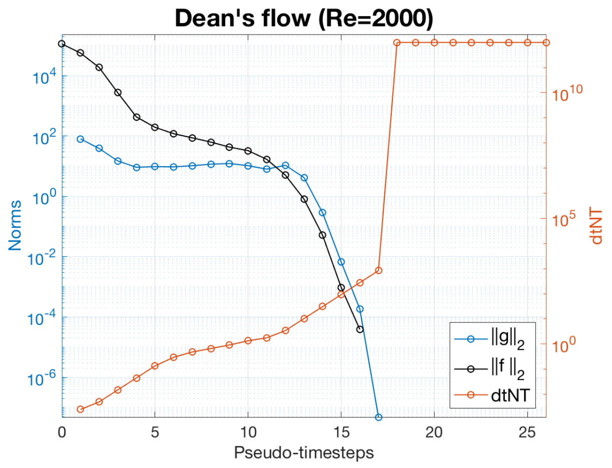


Figure 11: JFNK, converging to steady-state solutions for Dean's flow with $Re = 2000$ (left) and the convergence of the norms $\|f\|_2$ and $\|g\|_2$ with increasing pseudo-timestep sizes (dtNT) at each pseudo-timestep (right).

3.2.2 RANS Model in Nek5000 with Jacobian-free Newton Krylov Method

We extended the JFNK method for a RANS model [17] that is a regularized k-ω model, including the turbulent kinetic k and the specific dissipation rate ω in addition to the velocity field $\mathbf{v}$. The model describes the turbulent properties of the incompressible flows with

$$k = \frac{\langle u'^2 \rangle + \langle v'^2 \rangle + \langle w'^2 \rangle}{2}, \tag{7}$$

where u', v' and w' are fluctuation component of velocity vector around the ensemble-averaged mean velocity vector $\mathbf{v} = (u, v, w)$ governed by

$$\frac{\partial(\rho \mathbf{v})}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) = -\nabla p + \nabla \cdot \left[(\mu + \mu_t) \left(\nabla \mathbf{v} + \nabla \mathbf{v}^T - \frac{2}{3} \nabla \cdot \mathbf{v} \right) \right], \tag{8}$$

$$\frac{\partial(\rho k)}{\partial t} + \nabla \cdot (\rho k \mathbf{v}) = \nabla \cdot (\Gamma_k \nabla k) + G_k - Y_k + S_k, \tag{9}$$

$$\frac{\partial(\rho \omega)}{\partial t} + \nabla \cdot (\rho \omega \mathbf{v}) = \nabla \cdot (\Gamma_\omega \nabla \omega) + G_\omega - Y_\omega + S_\omega, \tag{10}$$

where μ is the molecular viscosity and μ_t is the turbulent viscosity with the continuity equation for incompressible flow

$$\nabla \cdot \mathbf{v} = 0. \tag{11}$$

Figure 12 demonstrates the number of pseudo-time steps to converge to the steady state solutions for a 2D turbulent channel problem with 57 pseudo-time steps for $Re = 10,935$ and 140 pseudo-time steps for $Re = 100,000$, whereas the second-order backward difference formula (BDF) with extrapolation (EXT) takes more than 500,000 timesteps. Experiments are currently ongoing with an extension to 3D RANS problem through CEED/ExaSMR collaboration.

3.2.3 Preconditioning Strategies for Steady Advection-Diffusion and Navier-Stokes

These recent developments are the first step in nonsymmetric system solvers for Nek5000. The next steps include larger problem sets, including $Pe > 10,000$ with the 17×17 rod bundle ($E > 200,000$ elements) to generate temperature profiles at low cost. The idea is to temporarily freeze the velocity field. The velocity

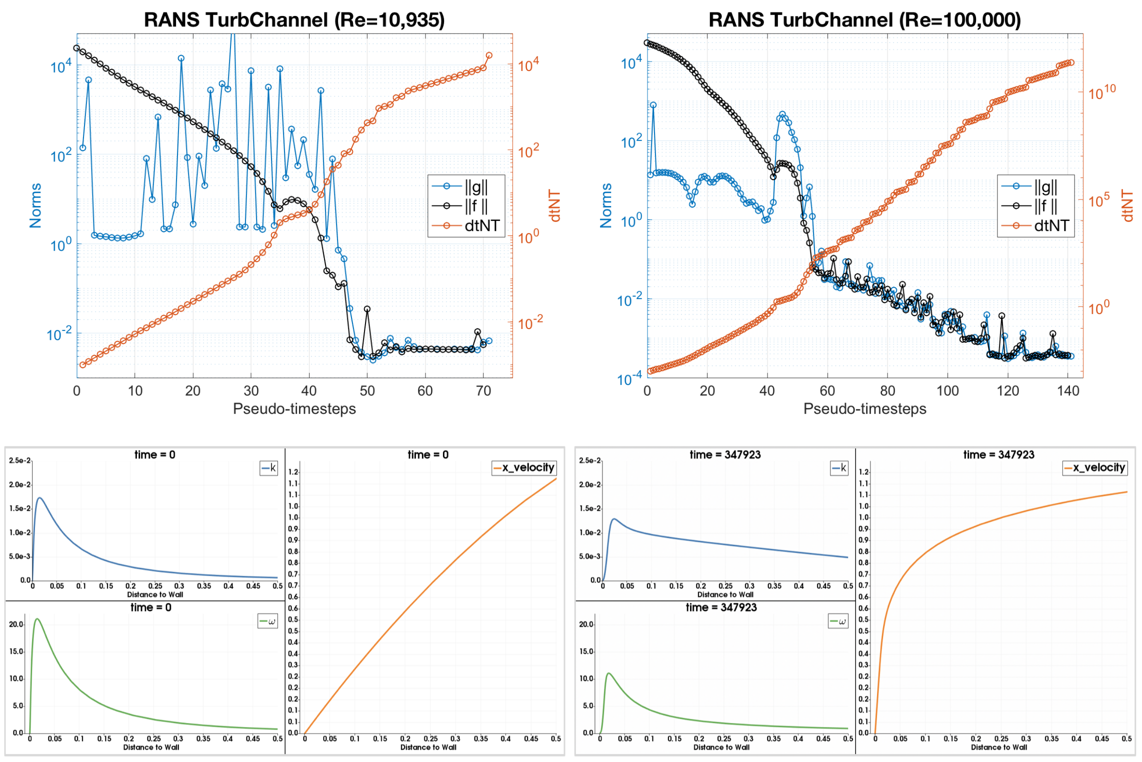


Figure 12: Jacobian-free Newton Krylov pseudo-time stepping, converging to steady-state solutions for a RANS turbchannel model with $Re = 10935$ (top left) and $Re = 100,000$ (top right). Profiles of the initial condition (bottom left) and steady state solution (bottom right) for k, ω and the x-component of the velocity.

rapidly reaches a statistically steady state while the hydraulic diameter D_h, is much shorter than the channel length L_t, which governs the thermal development time. By freezing the expensive-to-generate velocity field, we hope to accelerate equilibration of the thermal field without having to laboriously compute tens of thousands of transient turbulent eddies. Once the initial thermal transient has passed, LES of the thermal and hydrodynamics will be evolved in concert.

The steady-state advection-diffusion and steady-state Navier-Stokes (NS) equations share a nonsymmetric, non-positive-definite character arising from advective transport. Preconditioning these systems for tensor-product-based spectral element methods presents unique challenges and opportunities. Following earlier work for the Poisson and Stokes problems [4, 7, 6, 11], we develop a p-multigrid (PMG) strategy that uses overlapping Schwarz solves for a smoother at each level. For the steady advection-diffusion problem, PMG is used directly as a preconditioner within a Krylov subspace projection (KSP) method such as GMRES. For the Navier-Stokes, PMG is part of a larger preconditioner that includes restriction of velocity search directions to the space of divergence-free fields through a projection technique that we describe later. We begin with a short description of the advection-diffusion preconditioner and subsequently explain how PMG is a key component in the Newton-based nonlinear NS solver in next section.

3.2.4 p-Multigrid for Steady Advection-Diffusion

The smoother for our current p-multigrid preconditioner is based on a simple Richardson iteration involving a preconditioner, M. Coupling this with a coarse-grid correction leads to the two-level preconditioner given by the following pseudo-code.

$$\textit{Approximate the solution to } L\underline{x} = \underline{b};\ \underline{x}_0 = 0: \tag{12}$$

$$\begin{aligned} \underline{x}_1 &= \underline{x}_0 + M(\underline{b} - L\underline{x}_0) \\ \underline{r}_1 &= \underline{b} - L\underline{x}_1 \\ \underline{r}_c &= J_c^T \underline{r}_1 \\ \underline{x}_c &= L_c^{-1} \underline{r}_c \qquad (13) \\ \underline{x} &= J_c \underline{x}_c \quad (\text{return } \underline{x}) \end{aligned}$$

Here, L is assumed to be the spectral element advection-diffusion operator, $M \approx L^{-1}$ is the Schwarz smoother, and $L_c = J_c^T L J_c$ is the coarse grid system with J_c an interpolator from polynomial degree $p' < p$ to polynomial degree p within each spectral element. The coarse grid problem (13) can be solved by an invocation of the same two-level strategy, starting on a coarser grid, which leads to classic p-multigrid, or by an iterative KSP scheme. We formally define the overlapping Schwarz smoother with the Galerkin form:

$$M \;:=\; \sum_{e=1}^{E} R_e^T L_e^{-1} R_e, \tag{14}$$

where R_e is a rectangular Boolean matrix that restricts a global set of basis coefficients, $\underline{u}$ to a subset associated with Ω^e and its corresponding region of overlap, $\underline{u}_e = R_e \underline{u}$. The local systems are $L_e := R_e L R_e^T$.

The challenge with this method is to solve $L_e \underline{u}_e = \underline{r}_e$. L_e is dense and, if formed explicitly, has $O(p^6)$ nonzeros, which is prohibitive for $p > 3$. Fortunately, L_e can be applied in only $O(p^4)$ operations with only $O(p^3)$ storage, which is optimal. For elliptic problems, L_e is spectrally equivalent to its low-order finite-element counterpart on the same Gauss-Lobatto-Legendre nodes, which allows one to approximate L_e^{-1} by solving a sparse system [4, 13]. The equivalence is lost, however, for the advection-dominated case; the high-wavenumber eigenvalues of the low-order operator tend to zero whereas the high-order ones do not. Moreover, despite the tensor-product form of the underlying basis coefficients, L^e is generally *not separable*, particularly when advection is added.

3.2.5 Approximate Separable Operators

In the absence of advection, L^e is separable for certain geometries, which means that it can be expressed in the tensor-product form

$$L_e \;=\; B_3 \otimes B_2 \otimes A_1 + B_3 \otimes A_2 \otimes B_1 + A_3 \otimes B_2 \otimes B_1, \tag{15}$$

where A_j is the one-dimensional stiffness matrix associated with the jth direction (i.e., r, s, or t in the reference element $\hat{\Omega} := [-1, 1]^3$), and B_j is the associated mass matrix. Since B_j is symmetric positive definite (SPD), it is possible to find a matrix of eigenvectors, S_j, and a diagonal matrix of eigenvalues, Λ_j, satisfying $A_j S_j = B_j S_j \Lambda_j$, which leads to the diagonalization of L_e,

$$L_e \;=\; S \Lambda S^{-1}, \tag{16}$$

and its inverse,

$$L_e^{-1} \;=\; S \Lambda^{-1} S^{-1}. \tag{17}$$

Here, the full block-eigenvectors and eigenvalues are

$$S \;:=\; S_3 \otimes S_2 \otimes S_1, \tag{18}$$

$$S^{-1} \;:=\; S_3^{-1} \otimes S_2^{-1} \otimes S_1^{-1}, \tag{19}$$

$$\Lambda \;:=\; I \otimes I \otimes \Lambda_1 + I \otimes \Lambda_2 \otimes I + \Lambda_3 \otimes I \otimes I. \tag{20}$$

We note that, in 3D, (17) provides *one of the fastest possible solution strategies* for solving the Schwarz substeps in (14) for the p in the range of 2 to 20. S and S^{-1} are inexpensive to apply as tensor contractions that are expressible as fast matrix-matrix products. Λ is trivially inverted because it is diagonal.

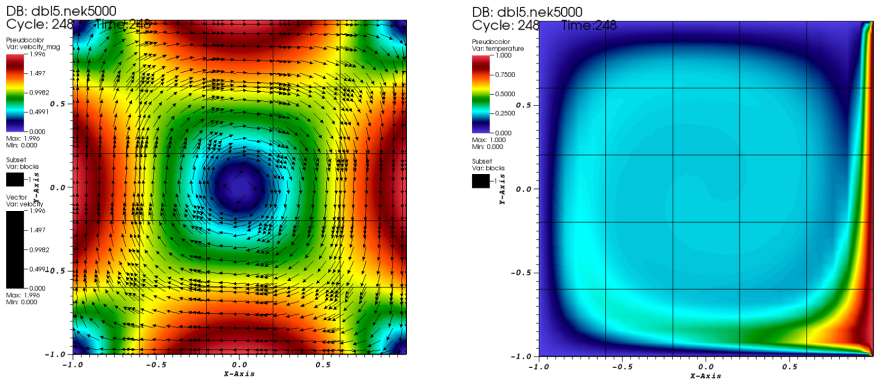


Figure 13: Double-glazing model: pre-defined velocity (left) and steady-state solution for temperature (right) with $Pe = 100$.

E	No Precon iter#	LU iter#	FDM iter#
1×1	37	16	-
2×1	79	22	23
2×2	170	22	19
3×3	319	19	18
4×4	482	18	16
5×5	632	17	15

Table 2: Double-glazing model with varying E for $N = 7$: GMRES iteration # without and with preconditioning using the LU and the FDM (FDM) for steady-state temperature solution with a pre-defined velocity.

The challenge for advection-diffusion is that L_e is generally not separable, meaning it does not have the form (15). In two dimensions, for the case of rectilinear elements with a velocity field of the form $\mathbf{c} = (c_x(\mathbf{x})\, c_y(\mathbf{x}))$, L_e takes the form

$$L_e = B_y \otimes A_x + A_y \otimes B_x + B_y \otimes \text{diag}(c_x) B_x D_x + \text{diag}(c_y) B_y D_y \otimes B_x. \tag{21}$$

Even in the simplest case where, say, $c_y = 0$ and $c_x = b(y)a(x)$, the fast diagonalization form (17) would require simultaneous diagonalization of B_y, A_y, and $b(y)B_y$, which is not possible unless two of the three matrices commute.

Fortunately, there have been recent developments in generating *separable approximations* to any given matrix L_e that require only the evaluation of matrix-vector products of the form $\underline{w} = L_e \underline{u}$ (which is fast) and not explicit formation of L_e (which is prohibitively expensive) [15]. We introduce the basic concepts for the two dimensional case, where we would seek an approximation of the form

$$L_e \quad \approx \quad A \otimes B \, + \, C \otimes D, \tag{22}$$

which could be solved by the fast diagonalization method. To simplify the exposition, we'll assume that all matrices on the right of (22) are $p \times p$ and that L_e is therefore $p^2 \times p^2$. We note that both A and B have p^2 entries such that $A \otimes B$ really comprises only $2p^2$ pieces of information, despite being of full rank (assuming that A and B are invertible). In particular, we note that

$$A \otimes B \quad := \quad \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{1p}b_{1p} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{1p}b_{2p} \\ \vdots & & \ddots & \vdots \\ a_{p1}b_{p1} & a_{p1}b_{p2} & \cdots & a_{pp}b_{pp} \end{pmatrix} \tag{23}$$

$$= \quad \mathit{shuffle}(\,\underline{a}\,\underline{b}^T), \tag{24}$$

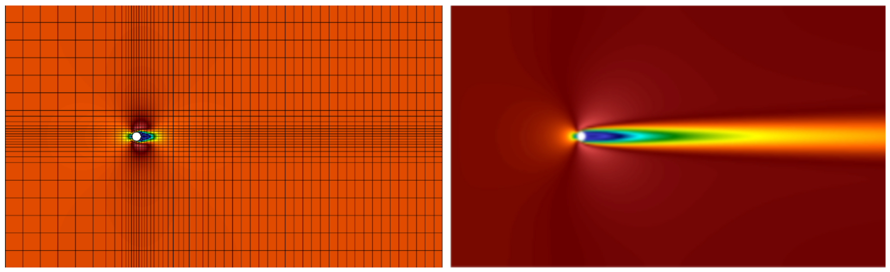


Figure 14: Flows past a cylinder ($E = 1472$): pre-defined velocity (left) and steady-state solution for temperature (right).

	$Pe = 100$				$Pe = 500$				$Pe = 1000$			
$N+1$	LU	FDM	LU (sec)	FDM (sec)	LU	FDM	LU (sec)	FDM (sec)	LU	FDM	LU (sec)	FDM (sec)
6	15	15	0.52	1.10	59	171	7.39	29.99	181	300	21.03	93.45
8	11	16	0.96	1.65	26	33	4.82	7.99	49	280	15.68	137.5
12	10	13	6.05	6.94	12	18	6.54	7.92	15	20	10.21.	15.71
16	10	14	28.73	22.66	11	14	26.71	17.38	12	14	28.42	18.85
20	10	19	95.58	62.03	11	14	83.58	41.61	11	14	84.00	40.97

Table 3: Flows past a cylinder with varying degrees N: GMRES iteration # and timings (sec) on a single CPU using tensor-product based fast diagonalization method (FDM) for steady-state temperature solution with frozen velocity.

where $\underline{a} = [a_{11}\, a_{21}\, \dots\, a_{pp}]^T$, $\underline{b} = [b_{11}\, b_{21}\, \dots\, b_{pp}]^T$, and *shuffle* is a permutation operator that rearranges the entries of $\underline{a}\,\underline{b}^T$ to the form (23).

For obvious reasons, we refer to $A \otimes B$ as a rank-1 tensor and $A \otimes B + C \otimes D$ as a rank-2 tensor. We are thus seeking a rank-2 tensor that approximates L_e, which is to say that we are seeking the rank-2 matrix

$$L_2 \;:=\; \underline{a}\,\underline{b}^T \;+\; \underline{c}\,\underline{d}^T \quad\approx\quad \mathit{shuffle}^{-1}(L_e) \;=:\; S_e \tag{25}$$

It is well-known that the closest rank-2 approximation to S_e, in the Frobenius norm, is given by entries in the singular value decomposition (SVD), $S_e = U\Sigma V^T$. Specifically,

$$L_2 \quad=\quad \underline{u}_1\sigma_1\underline{v}_1^T \;+\; \underline{u}_2\sigma_2\underline{v}_2^T. \tag{26}$$

The approximation, therefore, simply requires identification of the first two terms in the SVD of S_e, which can be found by iterative methods [15] that require fast matrix-vector products with L_e and forward and inverse *shuffle* operations applied to vectors.

Working with summer student Pablo Brubeck from UIUC, we have implemented approximate separable solvers for the fast diagonalization method (FDM) in Nek5000 for the advection-diffusion problem in both 2D and 3D. In the multilevel Schwarz context, it is used as a smoother at several levels. For 2D, we have compared the method to exact LU factorizations of L_e for the particular case of thermal transport past a heated cylinder at several Peclet numbers. (LU was deemed too expensive to test directly in 3D.)

Our preliminary results are summarized here, followed by more details: Table 2 presents comparisons of iteration counts with and without preconditioning with varying mesh resolution for a fixed polynomial order. Table 3 shows that, for sufficiently high polynomial orders ($N = p$), the FDM is faster than LU for each Peclet number, $Pe = 100$, 500, and 1000. Given the relative complexity estimates of p^{2d} for LU and p^{d+1} and p^d work and storage estimates for the FDM, it is clear that the FDM would be the best option in 3D. Tables 4 and 5 illustrate the effectiveness of this preconditioning strategy for the ExaSMR examples with single-rod and long single-rod cases in 3D as a function of Peclet number and domain size. These results are relatively new and thus still under investigation but, overall, the approach is promising for these challenging nonsymmetric problems.

Preconditioner Pe	$Pe = 100$ FDM	$Pe = 1000$ FDM
10	81	-
20	50	468
50	26	234
100	14	124
200	-	67
500	-	30
1000	-	16

Table 4: Singlerod ($E = 2560$ and $N = 7$): GMRES iteration # for steady-state temperature solution for $Pe = 100$ and $Pe = 1000$ with mismatched Pe in the FDM-based preconditioning.

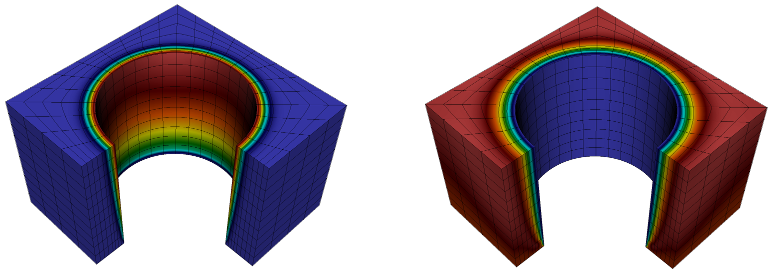


Figure 15: A singlerod ($E = 2560$, $N = 7$) and its steady-state solution for temperature with $Pe = 1000$ (left) with pre-defined frozen velocity (right).

Table 2 shows the case of the 2D double-glazing problem for the temperature solution with $Pe = 100$ and the pre-defined velocity shown in Figure 13. The GMRES iteration counts without preconditioning increase dramatically as the mesh resolution increases with larger E for a fixed $N = 7$, while the iteration counts drop down significantly with the cases of LU and FDM preconditioning.

Table 3 shows the case for 2D flows past a cylinder ($E = 292$) with varying polynomial degrees N. It demonstrates the GMRES iteration counts for steady-state temperature solution with frozen velocity by the steady-state advection-diffusion solver using the FDM. The results show the validation of the FDM implementation in comparison to the LU method, demonstrating the iteration numbers reduce as N increases and stays relatively constant as the Peclet number also increases.

Table 4 shows the case for solving the temperature with a frozen velocity for $Pe = 100$ and $Pe = 1000$ on a singlerod mesh ($E = 2560, N = 7$) shown in Figure 10 (left) with the field profiles in Figure 15. In particular, we studied mismatched Pe in the FDM-based preconditioning. The experiments show that the GMRES iterations actually reduce as the Pe in preconditioning chosen close to the problem Pe, with 14 iterations for $Pe = 100$ and 16 iterations for $Pe = 1000$. We also observe more speedup with smaller iteration counts in 3D when using relatively smaller N than the ones in 2D case of Table 3.

Table 5 demonstrates the cases for solving the temperature with higher Pe, including a larger singlerod mesh ($E = 12000, N = 7$) which is longer in z direction. The GMRES counts for $Pe = 1000$ stays around $16 \sim 18$ with matching $Pe = 1000$ in preconditioning for both experiments. For larger $Pe = 10000$, we currently obtained the data only up to $Pe = 1000$ in the preconditioning, with iteration counts around $122 \sim 135$ while we expect it would significantly reduce with the matching $Pe = 10000$ in preconditioning which have to be updated.

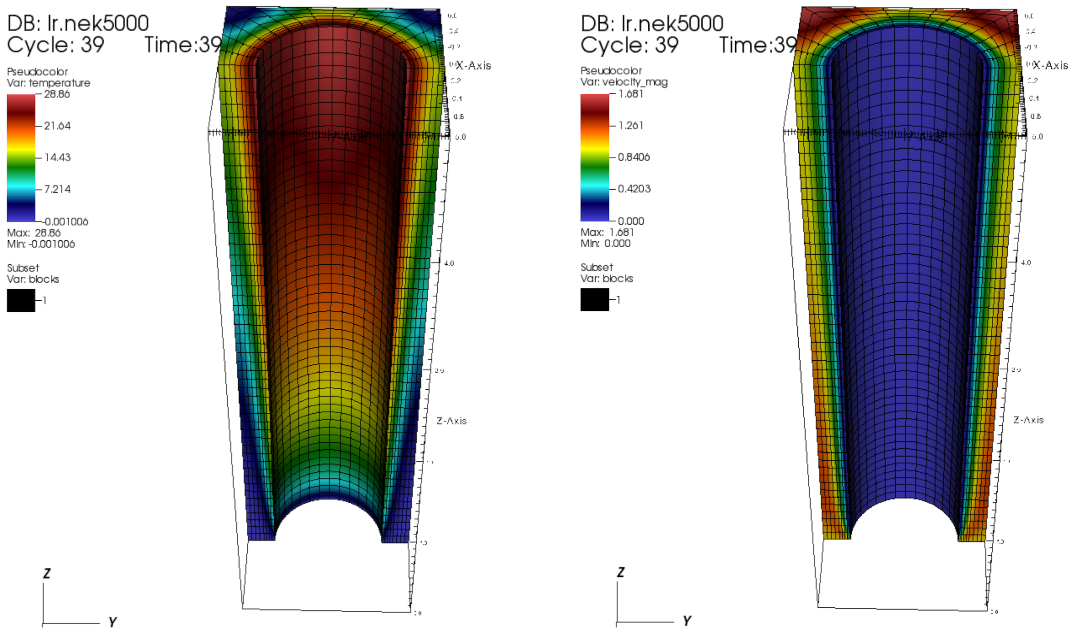


Figure 16: A long singlerod ($E = 2560 \times 5$, $N = 7$) and its steady-state solution for temperature with $Pe = 1000$ (left) with pre-defined frozen velocity (right).

singlerod			long singlerod		
Preconditioner Pe	$Pe = 1000$ FDM	$Pe = 10000$ FDM	Preconditioner Pe	$Pe = 1000$ FDM	$Pe = 10000$ FDM
500	30	-	500	34	-
1000	16	122	1000	18	135

Table 5: Singlerod ($E = 2560$ and $N = 7$) and long singlerod ($E = 12000$ and $N = 7$): GMRES iteration # for steady-state temperature solution for $Pe = 1,000$ and $Pe = 10,000$ with mismatched Pe in the FDM-based preconditioning.

3.2.6 FDM Preconditioning Extension to Steady Navier-Stokes

Here, we briefly outline extensions of the advection-diffusion preconditioner to the steady NS equations. Following [4], we write the algebraic form for the steady NS equations as

$$\begin{bmatrix} \mathbf{A} & -\mathbf{D}^T \\ -\mathbf{D} & 0 \end{bmatrix} \begin{pmatrix} \underline{\mathbf{u}} \\ \underline{p} \end{pmatrix} = \begin{pmatrix} \mathbf{B}\underline{\mathbf{f}} - \mathbf{C}(\underline{\mathbf{u}})\underline{\mathbf{u}} \\ 0 \end{pmatrix}, \tag{27}$$

where $\mathbf{A}$ constitutes the diffusion operator, $C(\underline{\mathbf{u}})\underline{\mathbf{u}}$ is the nonlinear convection term, and $\mathbf{D}$ is the discrete divergence operator. Let $\mathbf{W}$ be a diagonal SPD matrix and rewrite (27) in the algebraically equivalent form,

$$\begin{bmatrix} \mathbf{W} & -\mathbf{D}^T \\ -\mathbf{D} & 0 \end{bmatrix} \begin{pmatrix} \underline{\mathbf{u}} \\ \underline{p} \end{pmatrix} = \begin{pmatrix} \mathbf{B}\underline{\mathbf{f}} - \mathbf{C}(\underline{\mathbf{u}})\underline{\mathbf{u}} + (\mathbf{W}-\mathbf{A})\underline{\mathbf{u}} \\ 0 \end{pmatrix} = \begin{pmatrix} \underline{\mathbf{g}} \\ 0 \end{pmatrix} \tag{28}$$

With the SPD Schur complement, $E := \mathbf{D}\mathbf{W}^{-1}\mathbf{D}^T$, we can solve directly for the divergence-free velocity,

$$\underline{\mathbf{u}} = \left[\mathbf{I} - \mathbf{W}^{-1}\mathbf{D}^T E^{-1}\mathbf{D}\right]\mathbf{W}^{-1}\underline{\mathbf{g}} \tag{29}$$

$$= \mathbf{P}_W \mathbf{W}^{-1}\underline{\mathbf{g}}, \tag{30}$$

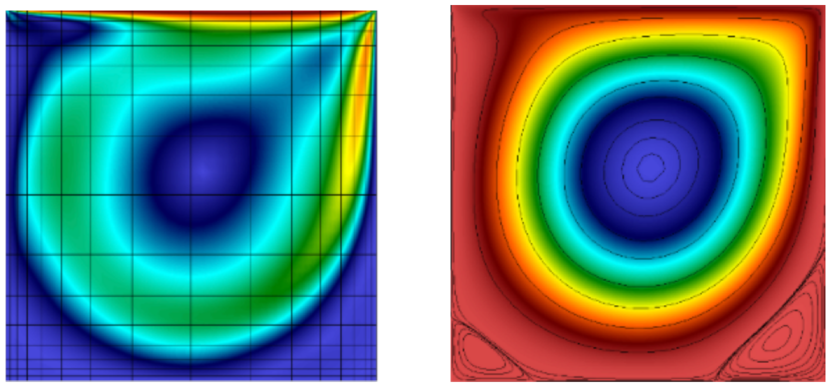


Figure 17: Lid-driven velocity (left) and stream lines (right) with $Re = 500$.

Preconditioner	$Re = 100$	$Re = 500$
Re	FDM	FDM
10	255	921
20	250	868
50	246	744
100	252	608
200	-	499
500	-	522

Table 6: Navier-Stokes Example ($E = 144$ and $N = 7$): Total number of GMRES iterations for the steady-state flow solution of a lid-driven cavity at $Re = 100$ and $Re = 500$ with mismatched Re in the FDM-based preconditioning.

where $\mathbf{P}_W := \left[\mathbf{I} - \mathbf{W}^{-1}\mathbf{D}^T E^{-1}\mathbf{D}\right]$ is a projector onto the space of divergence-free functions. The solution $\underline{\mathbf{u}}$ of (27) is also a solution of the fixed-point iterator (29) and thus the root of

$$\begin{aligned}
\mathbf{F}(\underline{\mathbf{u}}) &= \underline{\mathbf{u}} - \mathbf{P}_W\mathbf{W}^{-1}\underline{\mathbf{g}} \\
&= \underline{\mathbf{u}} - \mathbf{P}_W\mathbf{W}^{-1}\left[\mathbf{B}\underline{\mathbf{f}} - \mathbf{C}(\underline{\mathbf{u}})\underline{\mathbf{u}} + (\mathbf{W} - \mathbf{A})\underline{\mathbf{u}}\right] \\
&= \underline{\mathbf{u}} - \mathbf{P}_W\mathbf{W}^{-1}\left[\mathbf{B}\underline{\mathbf{f}} - \mathbf{C}(\underline{\mathbf{u}})\underline{\mathbf{u}} - \mathbf{A}\underline{\mathbf{u}}\right] - \mathbf{P}_W\underline{\mathbf{u}} \\
&= \mathbf{P}_W\mathbf{W}^{-1}\left[\mathbf{C}(\underline{\mathbf{u}})\underline{\mathbf{u}} + \mathbf{A}\underline{\mathbf{u}} - \mathbf{B}\underline{\mathbf{f}}\right]. \qquad (31)
\end{aligned}$$

Equation (31) requires $\mathbf{P}_W\underline{\mathbf{u}} = \underline{0}$, which will be true after a single iteration of (29).

For Newton's method, the Jacobian for the nonlinear problem (31) is

$$\mathbf{J}(\underline{\mathbf{u}}) = \mathbf{P}_W\mathbf{W}^{-1}\left[\mathbf{C}'(\underline{\mathbf{u}}) + \mathbf{A}\right],$$

which we recognize as (almost) an advection-diffusion system subject to projection onto a divergence-free space. We take as our preconditioner for $\mathbf{J}$ the matrix

$$\mathbf{M}(\underline{\mathbf{u}}) = \mathbf{P}_W\mathbf{W}^{-1}\left[\mathbf{C}'(\underline{\mathbf{u}}) + \mathbf{A}\right]^{-1},$$

which entails solving an advection-diffusion system ($\left[\mathbf{C}'(\underline{\mathbf{u}}) + \mathbf{A}\right]^{-1}$), followed by projection onto a divergence-free space (which requires a multigrid solve for the pressure). We have verified that, despite being rank deficient as a result of the divergence-free constraint, this preconditioner manages to cluster the resultant eigenvalues close to 1 and results in an effective iteration scheme. Of course, it is in reality too expensive to solve $\left[\mathbf{C}'(\underline{\mathbf{u}}) + \mathbf{A}\right]$ exactly, so we apply a single sweep of the Schwarz-PMG scheme outlined in the preceding section to each component of the velocity.

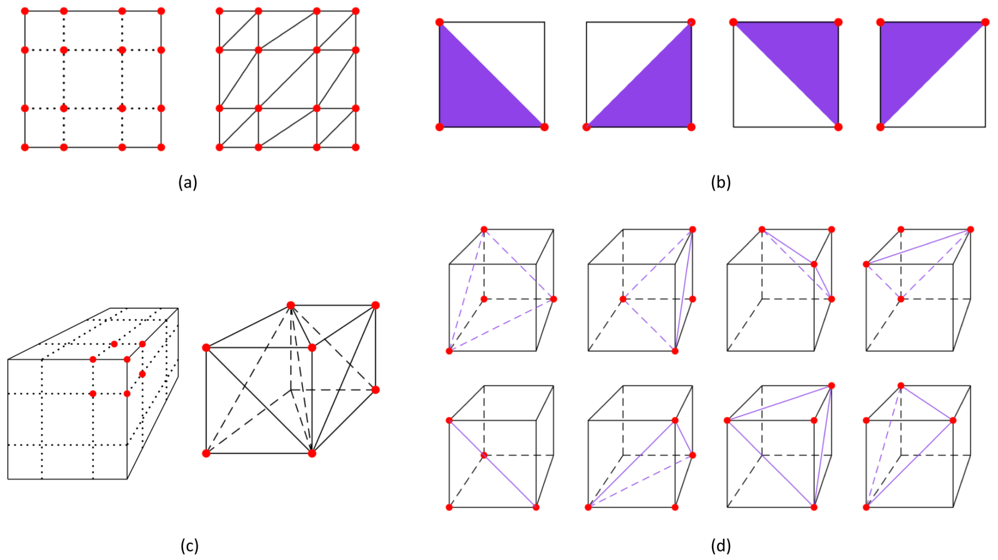


Figure 18: FEM preconditioning discretizations in 2D and 3D: (a) GLL points for $N = 3$ in a 2D quadrilateral spectral element and corresponding triangular discretization for the P_1 basis functions, (b) FEM meshing of a rectangular element with one triangle per vertex for a total of 4 low-order FEM elements, (c) GLL points for $N = 3$ in a 3D hexahedral spectral element and corresponding 6 tetrahedral discretization for the P_1 basis functions, (d) FEM meshing of a hexahedral element with one tetrahedron per vertex for a total of 8 low-order FEM elements.

As an initial test, we have applied the projected-FDM preconditioning strategy for our exact Jacobian Newton Krylov (EJNK) solver for the NS equations. Table 6 demonstrates the case of the lid-driven cavity problem ($E = 144, N = 7$) for $Re = 100$ and $Re = 500$ shown in Figure 17. The FDM iterations are the *total* GMRES iterations required for a sequence of Newton iterations. For $Re = 100$ and $Re = 500$, the total Newton iterations are 6 and 9, respectively. For the fluid problems, the FDM gives also a better speedup as the Re in the preconditioning gets close to the actual Re.

3.2.7 Low-Order FEM Preconditioning for Pressure

We accelerate the pressure solver using effective sparse preconditioner based on the first-order finite-element method (FEM), extended from [3]. The FE discretization is applied on the same Gauss-Lobatto-Legendre (GLL) points as the SE discretization. Figure 18 illustrates P_1 discretizations for a rectangular and hexahedral element with one triangle per vertex and one tetrahedron per vertex, respectively.

For the pressure solver, we consider the Poisson equation in $\mathbb{R}^d$, $d = 2$ or 3, defined by

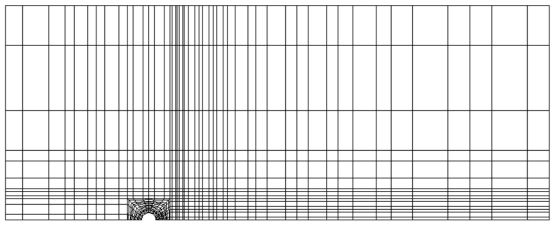
$$-\nabla^2 u = f \quad \text{for } u, f \in \Omega \subset \mathbb{R}^d. \tag{32}$$

The SE discretizations is based on the weak form, *Find* $u \in X_0^N$ *such that*

$$(\nabla v, \nabla u)_N = (v, f)_N \quad \forall v \in X_0^N = \{\phi_j(\mathbf{x}) : \phi_j \text{ vanishes on } \partial\Omega_D\}, \tag{33}$$

where the basis function $\phi_l = l_{\hat{i}}(\xi)l_{\hat{j}}(\eta)l_{\hat{k}}(\gamma)$ $(l = \hat{i} + (N+1)\hat{j} + (N+1)^2\hat{k})$ is defined by a tensor-product form of the one-dimensional Legendre-Lagrange interpolation polynomials $l_{\hat{i}}(\xi)$, $l_{\hat{j}}(\eta)$, $l_{\hat{k}}(\gamma)$ for $(\xi, \eta, \gamma) \in [-1, 1]^3$, approximating

$$\mathbf{x}^e(\xi, \eta, \gamma) = \sum_{\hat{i}\hat{j}\hat{k}} \mathbf{x}^e_{\hat{i}\hat{j}\hat{k}} l_{\hat{i}}(\xi)l_{\hat{j}}(\eta)l_{\hat{k}}(\gamma) \quad \text{and} \quad u^e(\mathbf{x}) = \sum_{\hat{i}\hat{j}\hat{k}} u^e_{\hat{i}\hat{j}\hat{k}} l_{\hat{i}}(\xi)l_{\hat{j}}(\eta)l_{\hat{k}}(\gamma) \tag{34}$$



	$N = 3$		$N = 7$		$N = 11$	
E	HS	FEM	HS	FEM	HS	FEM
93	25	17	43	18	78	18
372	40	17	60	17	102	20
1488	57	15	85	18	136	19
1744	34	16	44	19	54	20

Figure 19: A mesh for a flow past cylinder $E = 93$ (top). GMRES iteration counts (bottom) with varying N for low-order FEM using one-per-vertex elements.

on the non-overlapping elements Ω^e such that $\Omega = \cup\Omega^e$. Discretizing (32) by (34) we obtain the matrix form

$$A\underline{u} = B\underline{f}, \tag{35}$$

where the entries of A and B are defined as

$$A_{ij} = (\nabla\phi_i, \nabla\phi_j)_N \quad \text{and} \quad B_{ij} = (\phi_i, \phi_j)_N. \tag{36}$$

Here we introduce finite-element-based matrices A_F and B_F having entries

$$A_{F,ij} := (\nabla\psi_i, \nabla\psi_j), \quad B_{F,ij} := (\psi_i, \psi_j), \tag{37}$$

where $\psi_j(\mathbf{x})$ is a linear finite element basis function based on a (tetrahedral) triangulation of the mapped GLL points, $\mathbf{x}^e_{\hat{i}\hat{j}}$, from the SE discretization and $(\cdot,\cdot)$ is the standard L^2 inner product for the linear basis functions. We also consider the diagonal (lumped) mass matrix $B_d := diag(\sum_j B_{F,ij})$.

Three different preconditioners are considered:

$$M^{-1} = A_F^{-1} \qquad \text{Weak preconditioner, } P^w \tag{38}$$

$$M^{-1} = A_F^{-1} B_F B^{-1} \qquad \text{Strong preconditioner, } P^s \tag{39}$$

$$M^{-1} = A_F^{-1} B_d B^{-1} \qquad \text{Strong diagonal preconditioner, } P^{sd} \tag{40}$$

The preconditioning behaviors of these preconditioners are discussed in details for 2D and 3D geometries [1]. These preconditioners are robust with high aspect ratio, show bounded iteration counts with h- and p-refinement, and have lower iteration counts than scalable hybrid-Schwarz (HS) multigrid preconditioner used in Nek5000 [11].

Figure 19 demonstrates the GMRES iteration counts (bottom) for solving (35) with random right-hand side on the cylinder mesh (top). While the HS preconditioner is highly sensitive to high-aspect ratio elements, FEM preconditioner sustains at a constant level.

Figure 20 shows the run history for a full Navier-Stokes plus heat transfer simulation for 500 timesteps. We note that the iteration counts drop dramatically after the start of the computation because we generate highly accurate initial guesses to the pressure by projecting the solution onto the space of a handful (typically, 8–20) prior pressure solutions [5]. We can see that the one-per-vertex scheme requires about one-third the number of pressure iterations as HS. The total execution time for the full simulation was measured to be 2946.71 (sec) for HS and 2615.87 (sec) for one-per-vertex, thus achieving a 12 % reduction in overall computation time.

Figure 21 shows the iteration history for 250 time steps for the TCC-III engine geometry, deforming with time using the arbitrary Lagrangian-Eulerian (ALE) formulation described in [14]. A particular challenge

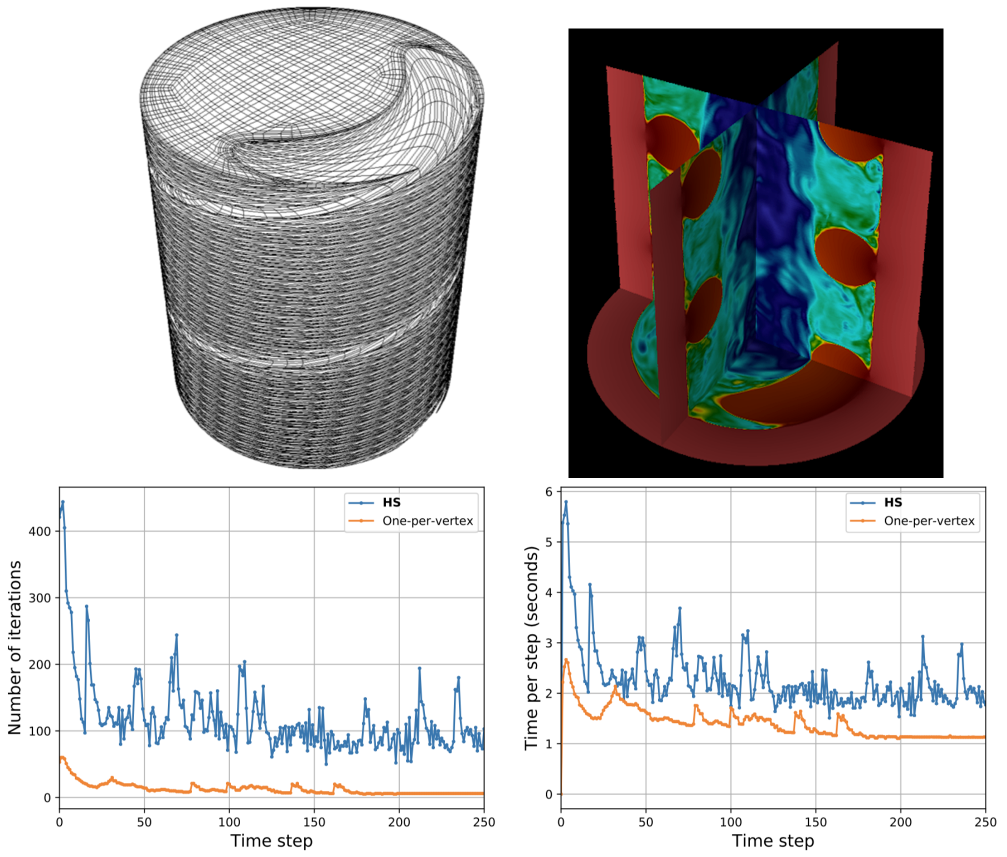


Figure 20: A mesh for wire-coil insert (top left) and temperature distribution in a wire-coil geometry (top right). time per time step (bottom right) with $(E, N) = (5720, 3)$.

for this problem is that the timestep is quite large, corresponding to a Courant number of $C \approx 4$, which is enabled through the ALE+characteristics-based timestepper developed in [14]. Higher Courant numbers typically lead to higher pressure iteration counts because the initial residual scales with timestep size, even in the context of time-projection [5]. Thus the baseline iteration count for this example is high, which provides more opportunity for improved preconditioning strategies. The preconditioner is constant throughout each restarted simulation. While A and B are functions of time because of the mesh motion, A_F is taken to be fixed at its initial instantiation at the start of each restarted computation. Here, we have the eight-fold reduction in iteration counts for the one-per-vertex algorithm with a significant reduction in time-to-solution compared to the HS method showing 2.82(sec) vs. 1.54 (sec) for the HS and one-per-vertex case, respectively, thus achieving an overall improvement of 46 % in the time spent at each Navier-Stokes step.

3.3 MARBL: Next-Gen Multi-Physics Simulation Code

3.3.1 Initial BP1 and BP3 Results on LLNL's Sierra Machine

One of the key components of explicit time-stepping algorithms with general high-order finite elements is the inversion of a global mass matrix, as exemplified by the momentum equation solve in the Lagrangian phase of the MARBL application. In the scalar case, this mass matrix inversion corresponds to CEED's benchmark BP1. Below we report some initial results for this benchmark on LLNL's Sierra machine, that are based on MFEM's new GPU *engine* interface (see below). We are also reporting results for BP3, which is relevant to diffusion problems, similar to the radiation-diffusion or MHD physics packages in MARBL. For more details on the specifications of BP1 and BP3 see `http://ceed.exascaleproject.org/bps`.

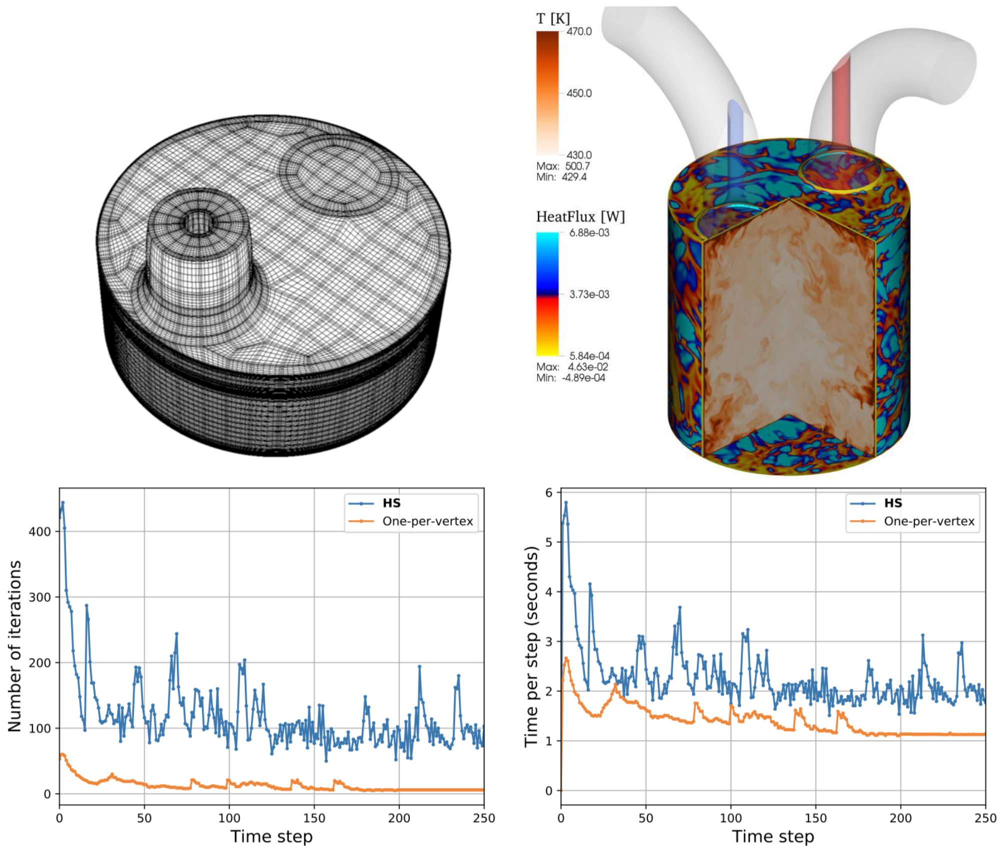


Figure 21: A mesh for TTC-III engine (top left) and temperature distribution and surface thermal flux in the TCC-III engine model during the compression stroke (top right). GMRES iteration counts (bottom left) and simulation time per time step (bottom right) with $(E, N) = (6784, 7)$.

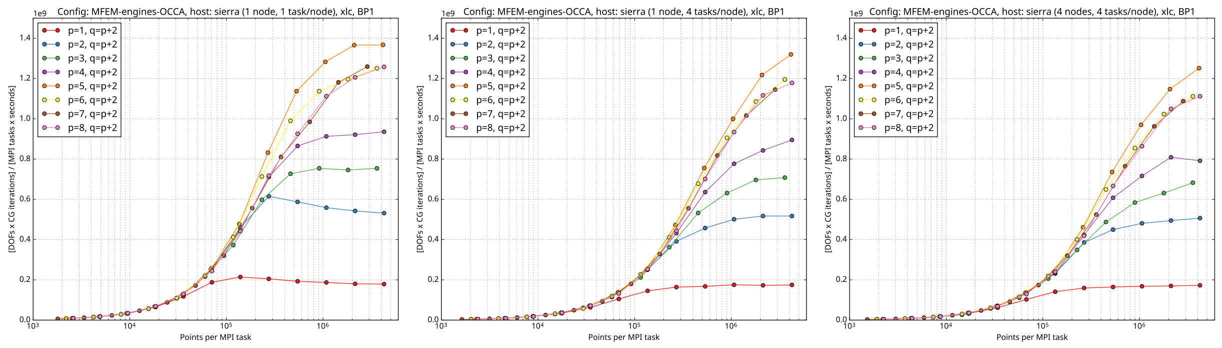


Figure 22: BP1 on Sierra using (left to right) 1, 4, and 16 GPUs. Each plot shows number of DOFs per GPU versus performance per GPU, measured as (DOFs $\times$ CG iterations)/(CG time $\times$ GPUs).

The results for BP1 and BP3 are presented on Figures 22 and 23, respectively. We summarize our observations regarding the results:

- For both BP1 and BP3, performance drops at relatively high number of DOFs. For example, we see $\sim 50\%$ drop around 200k–300k DOFs.

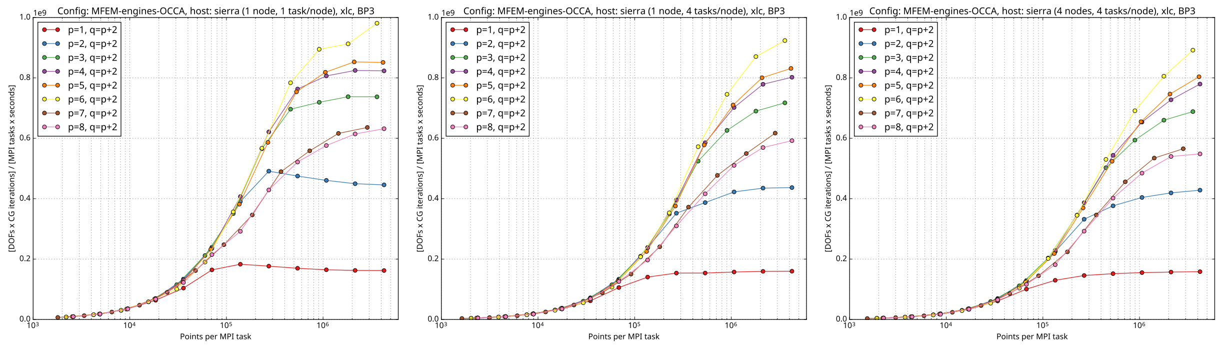
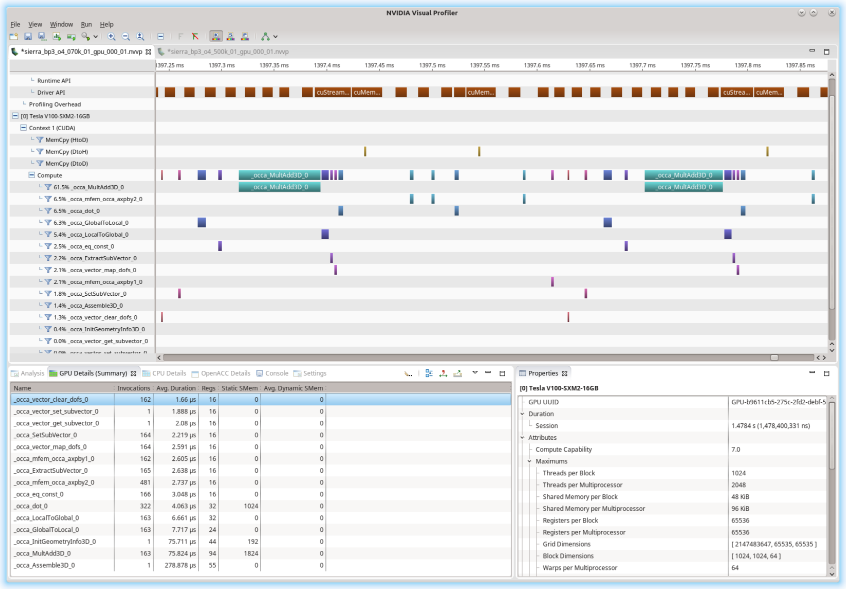


Figure 23: BP3 on Sierra using (left to right) 1, 4, and 16 GPUs. Each plot shows number of DOFs per GPU versus performance per GPU, measured as (DOFs $\times$ CG iterations)/(CG time $\times$ GPUs).


NVIDIA Visual Profiler
File View Window Run Help
*sierra_bp3_o4_070k_01_gpu_000_01.nvvp
*sierra_bp3_o4_500k_01_gpu_000_01.nvvp
Runtime API
Driver API
Profiling Overhead
[0] Tesla V100-SXM2-16GB
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
61.5% _occa_MultAdd3D_0
6.5% _occa_mfem_occa_axpby2_0
6.5% _occa_dot_0
6.3% _occa_GlobalToLocal_0
5.4% _occa_LocalToGlobal_0
2.5% _occa_eq_const_0
2.2% _occa_ExtractSubVector_0
2.1% _occa_vector_map_dofs_0
2.1% _occa_mfem_occa_axpby1_0
1.8% _occa_SetSubVector_0
1.4% _occa_Assemble3D_0
1.3% _occa_vector_clear_dofs_0
0.4% _occa_InitGeometryInfo3D_0
0.0% _occa_vector_get_subvector_0
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
Console
Settings
Properties

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
_occa_vector_clear_dofs_0	162	1.66 μs	16	0	0
_occa_vector_set_subvector_0	1	1.888 μs	16	0	0
_occa_vector_get_subvector_0	1	2.08 μs	16	0	0
_occa_SetSubVector_0	164	2.219 μs	16	0	0
_occa_vector_map_dofs_0	164	2.591 μs	16	0	0
_occa_mfem_occa_axpby1_0	162	2.605 μs	16	0	0
_occa_ExtractSubVector_0	165	2.638 μs	16	0	0
_occa_mfem_occa_axpby2_0	481	2.737 μs	16	0	0
_occa_eq_const_0	166	3.048 μs	16	0	0
_occa_dot_0	322	4.063 μs	32	1024	0
_occa_LocalToGlobal_0	163	6.661 μs	32	0	0
_occa_GlobalToLocal_0	163	7.717 μs	24	0	0
_occa_InitGeometryInfo3D_0	1	75.711 μs	44	192	0
_occa_MultAdd3D_0	163	75.824 μs	94	1824	0
_occa_Assemble3D_0	1	278.878 μs	55	0	0

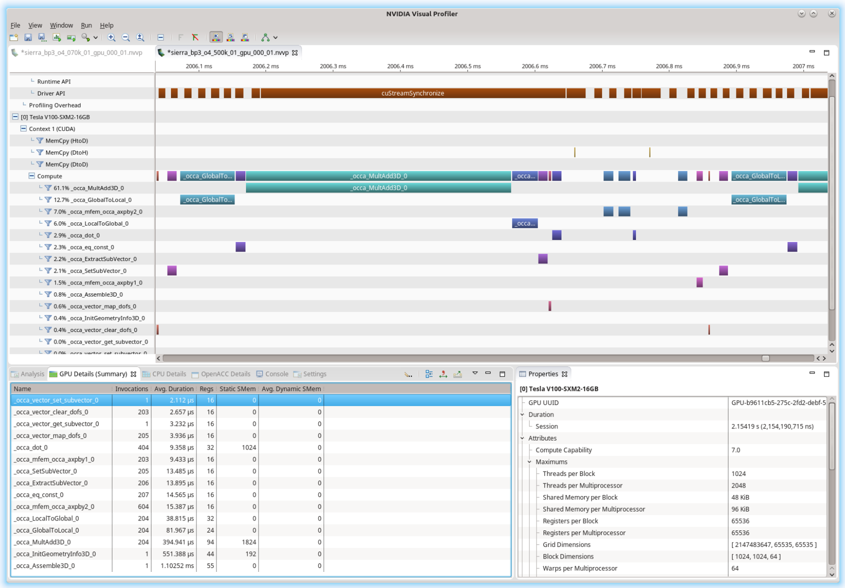
[0] Tesla V100-SXM2-16GB

GPU UUID	GPU-b9611cb5-275c-2fd2-debf-5
Duration	
Session	1.4784 s (1,478,400,331 ns)
Attributes	
Compute Capability	7.0
Maximums	
Threads per Block	1024
Threads per Multiprocessor	2048
Shared Memory per Block	48 KiB
Shared Memory per Multiprocessor	96 KiB
Registers per Block	65536
Registers per Multiprocessor	65536
Grid Dimensions	[2147483647, 65535, 65535]
Block Dimensions	[1024, 1024, 64]
Warps per Multiprocessor	64

Figure 24: Profiling result from BP3 on 1 GPU with $\sim$ 70k DOFs. The time-line is zoomed-in around one of the CG iterations.

- The kernel used in BP1 performs best for orders 5–8, whereas the kernel used in BP3 performs best for orders 3–6.
- For both BP1 and BP3, going from 1 to 4 GPUs (within the same compute node), the medium size problems are visibly affected by the addition of on-node MPI communication.
- Going from 4 to 16 GPUs (from 4 GPUs on one node, to 4 GPUs/node on 4 nodes), the inter-node MPI communication overhead is on the order of the on-node overhead, so the performance decrease is relatively small. However, scaling to larger number of nodes will be needed to fully assess the scaling properties of the machine and the BP implementations.

In order to investigate the reasons behind the first bullet above (the performance drop at a relatively high number of DOFs), we ran the NVIDIA profiler on two representative cases. We chose BP3, run on 1 GPU, with order $p = 4$, and problem sizes of $\sim$ 70k and $\sim$ 500k DOFs. The results we present show screenshots of the profiling tool with a view of the execution time-line zoomed-in to show the computations done during one CG iteration — this is representative for the overall problem (BP3), consisting of multiple CG iterations



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
_occa_vector_set_subvector_0	1	2.112 µs	16	0	0
_occa_vector_clear_dofs_0	203	2.657 µs	16	0	0
_occa_vector_get_subvector_0	1	3.232 µs	16	0	0
_occa_vector_map_dofs_0	205	3.936 µs	16	0	0
_occa_dot_0	404	9.358 µs	32	1024	0
_occa_mfem_occa_axpby1_0	203	9.433 µs	16	0	0
_occa_SetSubVector_0	205	13.485 µs	16	0	0
_occa_ExtractSubVector_0	206	13.895 µs	16	0	0
_occa_eq_const_0	207	14.565 µs	16	0	0
_occa_mfem_occa_axpby2_0	604	15.387 µs	16	0	0
_occa_LocalToGlobal_0	204	38.815 µs	32	0	0
_occa_GlobalToLocal_0	204	81.967 µs	24	0	0
_occa_MultAdd3D_0	204	394.941 µs	94	1824	0
_occa_InitGeometryInfo3D_0	1	551.388 µs	44	192	0
_occa_Assemble3D_0	1	1.10252 ms	55	0	0

[0] Tesla V100-SXM2-16GB

GPU UUID	GPU-b9611cb5-275c-2fd2-debf-5
Duration	
Session	2.15419 s (2,154,190,715 ns)
Attributes	
Compute Capability	7.0
Maximums	
Threads per Block	1024
Threads per Multiprocessor	2048
Shared Memory per Block	48 KiB
Shared Memory per Multiprocessor	96 KiB
Registers per Block	65536
Registers per Multiprocessor	65536
Grid Dimensions	[2147483647, 65535, 65535]
Block Dimensions	[1024, 1024, 64]
Warps per Multiprocessor	64

Figure 25: Profiling result from BP3 on 1 GPU with $\sim$ 500k DOFs. The time-line is zoomed-in around one of the CG iterations.

which are almost identical. The screenshots are shown on Figure 24, for the smaller problem, and on Figure 25, for the larger problem.

The main observation from these results is that the percentage of time during which the GPU is busy is quite different in the two cases. For the larger problem, the white gaps in the "Compute" line are much smaller compared to the same gaps for the smaller problem. The actual GPU kernels scale relatively well, as seen by the timings in the lower-left corner of the screenshots. For example, the main kernel, `MultAdd3D`, takes $395\mu s$ and $76\mu s$, respectively, a ratio of ~ 5.20. On the other hand, the CG time per iteration, as seen in Figure 23, is roughly 0.75×10^9 DOFs/s or $670\mu s$, for the larger problem, and 0.25×10^9 DOFs/s or $280\mu s$, for the smaller problem. This gives a ratio of about 2.4 speedup of the CG time per iteration (going from the larger to the smaller problem) which is worse compared to the speedup of 5.20 for the `MultAdd3D` kernel. This clearly emphasizes the need for reducing the time when the GPU is not utilized.

3.3.2 *MARBL/BLAST*

As discussed in previous reports, the CEED team targets the optimization of the MARBL simulation code by improving its BLAST component, which is a multi-material multi-physics code based on the MFEM library. Details of the targeted computational kernels in BLAST are discussed in CEED-MS1. The main tool to impact BLAST is the Laghos miniapp, developed by CEED, which resembles the main computational kernels of BLAST's Lagrangian phase. Overview of Laghos is given in the CEED-MS6 report, and numerous performance results for Laghos on CPUs and GPUs are presented in the CEED-MS8, CEED-MS10 and CEED-MS13 reports. See also Section 4.4 for description of the new Laghos-1.1 release.

The developments in Laghos have already provided several benefits to BLAST, as described in the CEED-MS8 and CEED-MS10 reports. These include partial assembly kernels for the four major computations of BLAST's Lagrangian phase, which were originally written in Laghos and then transferred to BLAST with several modifications (e.g. for multi-material and axisymmetric terms). Furthermore, based on the Laghos code structure, extensive parts of the BLAST code were refactored to allow separation between physics and finite element assembly, allowing batched equation of state (EOS) calls and better compatibility with partial-assembly based calculations. These additions and modifications allowed partial assembly simulations in BLAST, which lead to faster CPU simulations for higher order finite element spaces, see Figure 26.

For the past period the CEED researchers have continued improving BLAST through improvements in Laghos and MFEM. The pure CUDA implementation of the Laghos kernels (see CEED-MS13 for details)

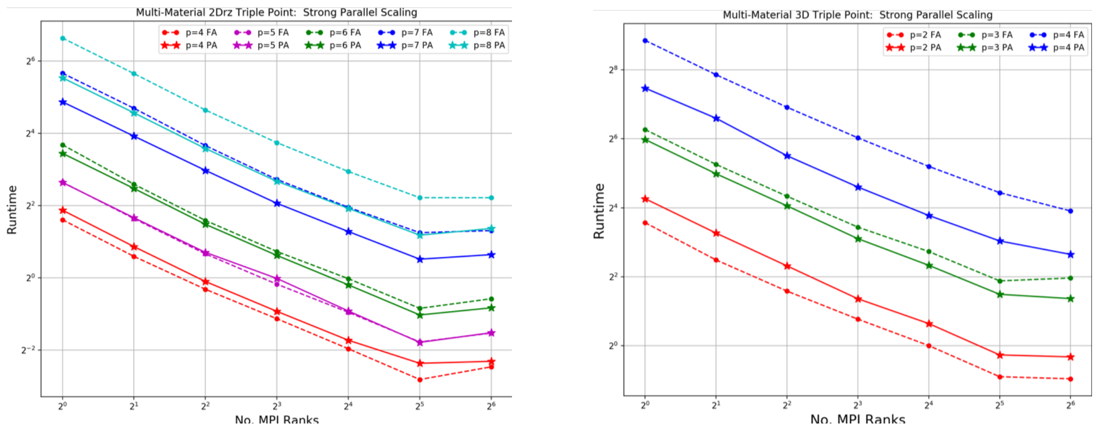


Figure 26: Strong scaling in 2D (left) and 3D (right) for the Triple Point problem in BLAST. Both plots compare partial assembly (PA) vs full assembly (FA). In 2D, PA wins for orders above 5. In 3D, PA wins for orders above 2.

was tested on up to 1024 GPUs of the Sierra machine at LLNL. Weak and strong scaling of the pure CUDA version are presented in Figures 27 and 28, respectively. Figure 29 compares execution rates for various orders between the full Vulcan machine and 1024 GPUs of Sierra (6.25% of the machine). The above plots imply, as observed in other applications, that the GPU implementation is faster as long as there is enough work per GPU. One straightforward method to optimize BLAST is to transfer the pure CUDA kernels directly from Laghos, as done with the CPU partial assembly kernels. A more flexible approach is described in the following paragraph.

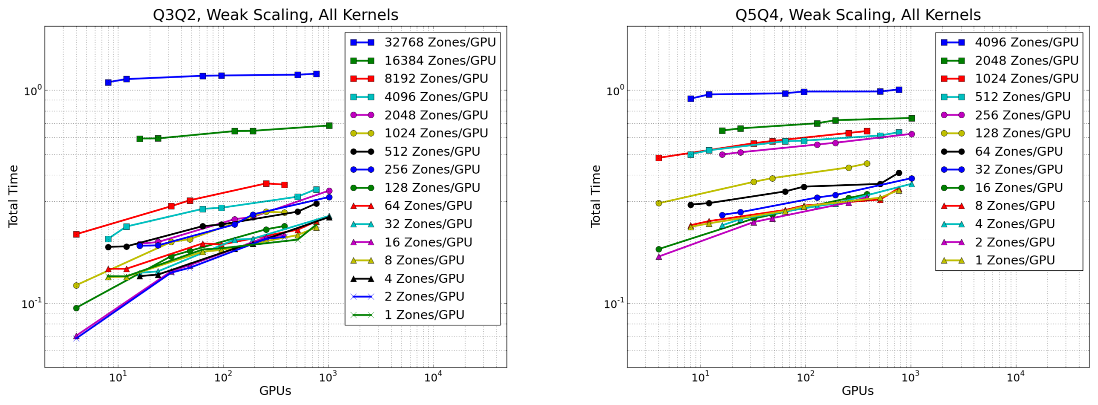


Figure 27: Weak scaling for Laghos on Sierra, pure CUDA version, Q3Q2 (left) and Q5Q4 (right) finite element spaces.

CEED researchers have been actively working on defining an additional abstraction layer in MFEM. This new layer provides a method to utilize heterogeneous systems through a common interface. The new interface allows MFEM to connect to an arbitrary execution *engine*. Examples of such engines are OCCA and libCEED. Laghos has been connected to this interface, meaning that it can be executed through any engine, as long as its custom kernels are defined for that engine. The CEED team has also defined the *Kernels* engine, which can be used to switch between the pure CUDA, CPU and RAJA versions of the Laghos kernels. Thus, as the needed kernels for Laghos already exist, the new interface allows to switch easily (at compile-time or run-time) between the OCCA (CPU or GPU), RAJA (CPU or GPU), and pure CUDA versions of Laghos. The pure CUDA version of Laghos will be used as a benchmark for performance, which all of the above abstractions should not deteriorate. This new engines interface is currently being incorporated in BLAST, following the approach that's used in Laghos. Once completed, BLAST would have a way to connect, through MFEM, to any of OCCA, libCEED, the Kernels engine, and other future engines.

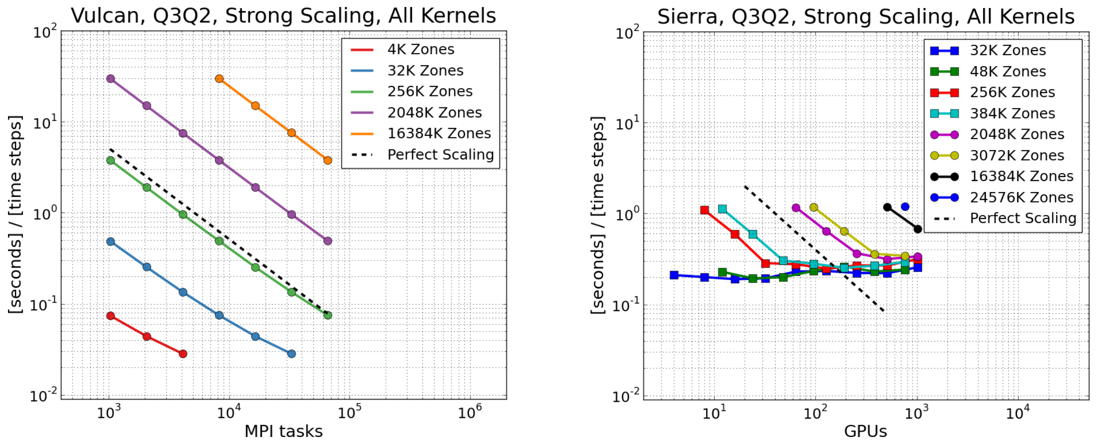


Figure 28: Strong scaling comparison for Laghos on Vulcan (left) and Sierra (right), pure CUDA version, Q3Q2 finite element spaces.

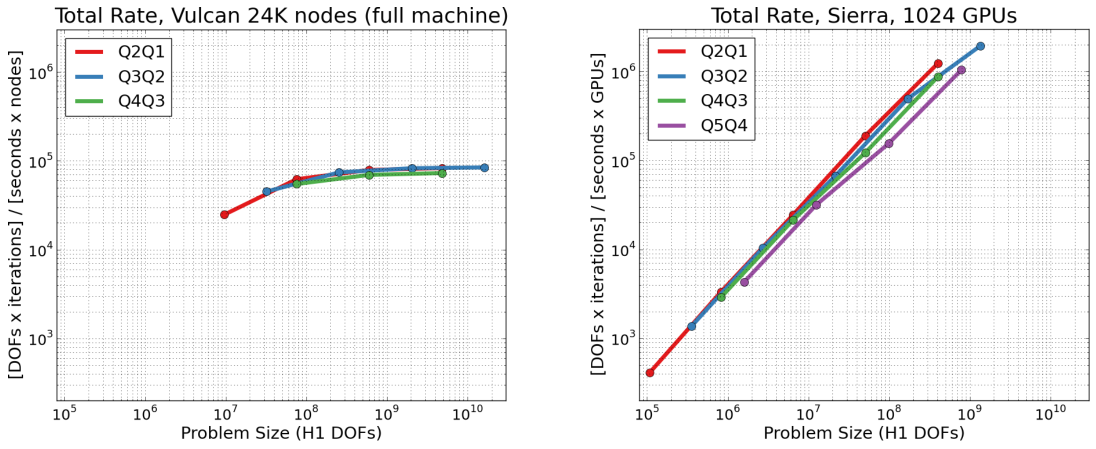


Figure 29: Strong scaling comparison for Laghos on Vulcan (left) and Sierra (right), pure CUDA version, Q3Q2 finite element spaces.

4. OTHER PROJECT ACTIVITIES

4.1 CEED Second Annual Meeting

The second annual meeting of the CEED co-design center took place August 8–10 at the University of Colorado at Boulder. Participants reported on the progress in the center, deepened existing and established new connections with ECP hardware vendors, ECP software technologies projects and other collaborators, planned project activities and brainstormed / worked as a group to make technical progress. The meeting was very successful and was attended by 40+ participants from several DOE labs, academia and industry.

4.2 ICOSAHOM18 Minisymposium

The CEED team organized a minisymposium on “Efficient High-Order Finite Element Discretizations at Large Scale” at the premier conference on high-order methods, the International Conference in Spectral and High-Order Methods (ICOSAHOM18) in London. The minisymposium featured 12 participants from universities and labs around the globe representing the leading edge of the international high-order community.

4.3 CEED Reports Publicly Available

The ECP/CEED milestone reports are now publicly available on the CEED publications page at `http://ceed.exascaleproject.org/pubs/#ceed-reports`. The currently available reports are:

- CEED-MS1: Engage first wave ECP/CEED applications.
- CEED-MS6: Identify initial kernels, bake-off (benchmark) problems and miniapps.
- CEED-MS8: Initial integration of CEED software in ECP applications.
- CEED-MS10: Initial CEED API.
- CEED-MS13: Public release of CEED 1.0.
- CEED-MS18: Propose high-order mesh/data format.

4.4 Laghos-1.1

Version 1.1 of the Laghos miniapp was released at `https://github.com/CEED/Laghos/releases`. The new release includes diagonal preconditioning, energy-conserving time integration, a new example problem (Gresho vortex) and instructions for building CUDA and RAJA versions in the `raja-dev` branch. Laghos-1.1 will be part of the 2.0 release of ECP's Proxy Applications Suite, see `https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/`.

4.5 Outreach

CEED researchers were involved in a number of outreach activities, including 7 presentations at the Argonne Training Program on Extreme-Scale Computing (ATPESC18), presentations at the Smoky Mountains Computational Sciences & Engineering Conference (SMC18) and the SIAM Annual Meeting. A new paper, "OpenACC acceleration for the Pn/Pn-2 algorithm in Nek5000" was submitted to the Journal of Parallel and Distributed Computing. Article about CEED appeared in Krell's institute DEIXIS magazine, which was further highlighted on the ECP's website. CEED researchers are also organizing two minisymposium at SIAM CSE19, "Exascale Software for High-Order Methods" and "Exascale Applications with High-Order Methods", centered around the work in CEED and including key representatives of the international high-order community (16 talks total).

5. CONCLUSION

In this milestone, we developed optimization techniques and tuned for performance the CEED software. The focus was on developing and tuning fast finite element operator storage and evaluation, architecture optimizations, and global kernels for finite element operators. We also worked on performance tuning of CEED's first-wave ECP applications, including the ExaSMR and MARBL applications.

The artifacts delivered include performance improvements in CEED's 1st wave of applications, and tuned CEED software for various architectures through a number of backends, freely available in the CEED's repository on GitHub. See the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed` for more details.

In addition to details and results from the above R&D efforts, in this document we are also reporting on other project-wide activities performed in Q4 of FY18 including: CEED's second annual meeting, a successful minisymposium at the premier international conference on high-order methods (ICOSAHOM), making the CEED milestone reports publicly available, new Laghos release, and other outreach efforts.

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-758990.

REFERENCES

[1] Pedro D. Bello-Maldonado and Paul Fischer. Scalable low-order finite element preconditioner for spectral element poisson solvers. 2018. submitted.

[2] Jed Brown, Ahmad Abdelfatah, Jean-Sylvain Camier, Veselin Dobrev, Jack Dongarra, Paul Fischer, Aaron Fisher, Yohann Dudouit, Azzam Haidar, Kazem Kamran, Tzanio Kolev, Misun Min, Thilina Ratnayaka, Mark Shephard, Cameron Smith, Stanimire Tomov, Vladimir Tomov, and Tim Warburton. ECP Milestone Report CEED-MS13: Public release of CEED 1.0, April 2, 2018.

[3] C. Canuto, P. Gervasio, and A. Quarteroni. Finite-element preconditioning of G-NI spectral methods. *SIAM J. Sci. Comput.*, 31:4422–44251, 2010.

[4] P.F. Fischer. An overlapping Schwarz method for spectral element solution of the incompressible Navier-Stokes equations. *J. Comput. Phys.*, 133:84–101, 1997.

[5] P.F. Fischer. Projection techniques for iterative solution of $A\underline{x} = \underline{b}$ with successive right-hand sides. *Comput. Methods Appl. Mech. Engrg.*, 163:193–204, 1998.

[6] P.F. Fischer and J.W. Lottes. Hybrid Schwarz-multigrid methods for the spectral element method: Extensions to Navier-Stokes. In R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund, and J. Xu, editors, *Domain Decomposition Methods in Science and Engineering Series*. Springer, Berlin, 2004.

[7] P.F. Fischer, N.I. Miller, and H.M. Tufo. An overlapping Schwarz method for spectral element simulation of three-dimensional incompressible flows. In P. Bjørstad and M. Luskin, editors, *Parallel Solution of Partial Differential Equations*, pages 158–180, Berlin, 2000. Springer.

[8] A Karakus, N Chalmers, Jan S Hesthaven, and T Warburton. Discontinuous galerkin discretizations of the boltzmann equations in 2d: semi-analytic time stepping and absorbing boundary layers. *arXiv preprint arXiv:1805.02082*, 2018.

[9] Ali Karakus, Noel Chalmers, Kasia Swirydowicz, and Timothy Warburton. Gpu acceleration of a high-order discontinuous galerkin incompressible flow solver. *arXiv preprint arXiv:1801.00246*, 2017.

[10] Dana A Knoll and David E Keyes. Jacobian-free newton-krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397, 2004.

[11] J. W. Lottes and P. F. Fischer. Hybrid multigrid/Schwarz algorithms for the spectral element method. *J. Sci. Comput.*, 24:45–78, 2005.

[12] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joel Falcou, and Jack Dongarra. Algorithms and Optimization Techniques for High-Performance Matrix-Matrix Multiplications of Very Small Matrices. Technical Report ICL-UT-18-09, 09-2018 2018.

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

[14] S. Patel, P. Fischer, M. Min, and A. Tomboulides. A characteristic-based, spectral element method for moving-domain problems. *Under Review*, 2018.

[15] Will Pazner and Per-Olof Persson. Approximate tensor-product preconditioners for very high order discontinuous galerkin methods. *Journal of Computational Physics*, 354:344–369.

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

[17] A. Tomboulidesa, S. M. Aithal, P. F. Fischer, E. Merzari, A. V. Obabkob, and D. R. Shaver. A novel numerical treatment of the near-wall regions in the k-ω class of RANS models. *International Journal of Heat and Fluid Flow*, 2018.

[18] Ping-Hsuan Tsai, Yu-Hsiang Lan, Misun Min, and Paul Fischer. Jacobi-free Newton Krylov method for Poisson-Nernst-Planck equations. *to be submitted*, 2018.

[19] Warburton, Timothy. Finite element stiffness matrix action: monolithic kernel optimization on titan v, 2018. `https://www.paranumal.com/single-post/2018/03/02/Finite-Element-Stiffness-Matrix-Action-monolithic-kernel-optimization-on-Titan-V`.

[20] Warburton, Timothy. Finite element stiffness matrix action: to blas or not to blas, that is the question, 2018. `https://www.paranumal.com/single-post/2018/03/13/Finite-Element-Stiffness-Matrix-Action-to-BLAS-or-not-to-BLAS-that-is-the-question`.

[21] Warburton, Timothy. Finite element stiffness matrix action: to precompute or not to precompute, 2018. `https://www.paranumal.com/single-post/2018/03/14/Finite-Element-Stiffness-Matrix-Action-to-precompute-or-not-to-precompute`.

[22] Warburton, Timothy. libparanumal: Library for parallel numerical algorithms, 2018. `https://github.com/paranumal/libparanumal`.