ECP Milestone Report

Initial CEED API

WBS 2.2.6.06, Milestone CEED-MS10

Jed Brown
Jean-Sylvain Camier
Veselin Dobrev
Paul Fischer
Tzanio Kolev
Thilina Ratnayaka
Mark Shephard
Jeremy Thompson
Vladimir Tomov

December 22, 2017

ECP Milestone Report
Initial CEED API
WBS 2.2.6.06, Milestone CEED-MS10

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

December 22, 2017

ECP Milestone Report
Initial CEED API
WBS 2.2.6.06, Milestone CEED-MS10

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	December 22, 2017	Original	

EXECUTIVE SUMMARY

Key components of CEED software development involve fast finite element operator storage and evaluation, architecture optimizations, performant algorithms for all orders, global kernels for finite element operators, efficient use of the memory sub-system, optimal data locality and motion, enhanced scalability and parallelism, and fast tensor contractions. As ultimate goals of the project, CEED is exploring and identifying the best algorithms for the full range of discretizations and applying algorithmic and software development to support ECP applications' needs.

In this milestone, we initiated the development and implementation of a common CEED API consisting of two main components: low-level and high-level APIs. We accomplished the goal of having the low-level component ready for the release of CEED v1.0 in the next quarter, see CEED-MS13.

In this report we also provide a highlight from CEED's integration with the MARBL app and the corresponding Laghos miniapp. A main theme in this work has been the systematic transition of advanced algorithms, developed and tested in Laghos, to the large-scale multi-physics settings of the MARBL code.

The low-level CEED API provides a set of FE kernels and components for writing new low-level kernels. Examples include: vector and sparse linear algebra, element matrix assembly over a batch of elements, partial assembly and action for efficient high-order operators like mass, diffusion, advection, etc. The main goal of the low-level API is to establish the basis for the high-level API. Also, identifying such low-level kernels and providing a reference implementation for them serves as the basis for collaboration with vendors and STs. Another goal for the low-level API is to make it easy for applications with their own discretization infrastructure, who do not want to switch directly to using the high-level CEED API, to still evaluate and use the core operations provided by the low-level API.

One of the challenges with high-order methods is that a global sparse matrix is no longer a good representation of a high-order linear operator, both with respect to the FLOPs needed for its evaluation, as well as the memory transfer needed for a matvec. Thus, another important question we tried to address while developing the low-level API is: is it possible to abstract out in a natural and efficient way the different implementations and data structures that arise when dealing with various computational device types: CPUs, GPUs, etc. Fully answering this question is beyond the scope of this particular milestone, since that requires longer time as we develop and expand the devices we support.

The future high-level API will be the frontend provided by CEED consisting of high-level abstractions such as meshes, finite element spaces and solutions, bilinear forms, solvers, etc. At this high-level, an important aspect of the API will be to provide a uniform abstraction layer for writing code in a portable manner. To achieve that, we plan to utilize the MPI programming model combined with a device abstraction layer (DAL) for finite element computations that we will develop. The DAL will serve as the bridge between the low- and the high-level APIs, allowing a variety of programming models to be used for the low-level kernel implementations.

The software artifacts delivered as part of this milestone include an initial CEED API software implementation (libCEED) provided through the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed`.

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 Q1 of FY18, including: the Nek5000 v17.0 MFEM v3.3.2 and PETSc v3.8 releases, collaborations with ECP/ST and SciDAC projects, the inaugural Nek5000 hackathon and various outreach activities.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In this milestone, we initiated the development and implementation of a common CEED API consisting of two main components: low-level and high-level APIs. We accomplished the goal of having the low-level component ready for the release of CEED v1.0 in the next quarter, see CEED-MS13.

In this report we also provide a highlight from CEED's integration with the MARBL app and the corresponding Laghos miniapp. A main theme in this work has been the systematic transition of advanced algorithms, developed and tested in Laghos, to the large-scale multi-physics settings of the MARBL code.

The low-level CEED API provides a set of FE kernels and components for writing new low-level kernels. Examples include: vector and sparse linear algebra, element matrix assembly over a batch of elements, partial assembly and action for efficient high-order operators like mass, diffusion, advection, etc. The main goal of the low-level API is to establish the basis for the high-level API. Also, identifying such low-level kernels and providing a reference implementation for them serves as the basis for collaboration with vendors and STs. Another goal for the low-level API is to make it easy for applications with their own discretization infrastructure, who do not want to switch directly to using the high-level CEED API, to still evaluate and use the core operations provided by the low-level API.

One of the challenges with high-order methods is that a global sparse matrix is no longer a good representation of a high-order linear operator, both with respect to the FLOPs needed for its evaluation, as well as the memory transfer needed for a matvec. Thus, another important question we tried to address while developing the low-level API is: is it possible to abstract out in a natural and efficient way the different implementations and data structures that arise when dealing with various computational device types: CPUs, GPUs, etc. Fully answering this question is beyond the scope of this particular milestone, since that requires longer time as we develop and expand the devices we support.

The future high-level API will be the frontend provided by CEED consisting of high-level abstractions such as meshes, finite element spaces and solutions, bilinear forms, solvers, etc. At this high-level, an important aspect of the API will be to provide a uniform abstraction layer for writing code in a portable manner. To achieve that, we plan to utilize the MPI programming model combined with a device abstraction layer (DAL) for finite element computations that we will develop. The DAL will serve as the bridge between the low- and the high-level APIs, allowing a variety of programming models to be used for the low-level kernel implementations.

The software artifacts delivered as part of this milestone include an initial CEED API software implementation (libCEED) provided through the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed`.

2. THE CEED API LIBRARY: LIBCEED

This section contains a description of the initial low-level API library for the efficient high-order discretization methods developed under CEED. While our focus is on high-order finite elements, the approach is algebraic and thus applicable to other discretizations in factored form.

One of the challenges with high-order methods is that a global sparse matrix is no longer a good representation of a high-order linear operator, both with respect to the FLOPs needed for its evaluation, as well as the memory transfer needed for a matvec. Thus, high-order methods require a new "format" that still represents a linear (or more generally, non-linear) operator, but not through a sparse matrix.

The goal of libCEED is to propose such a format, as well as supporting implementations and data structures, that enable efficient operator evaluation on a variety of computational device types (CPUs, GPUs, etc.). This new operator description is based on an algebraically factored form, which is easy to incorporate in a wide variety of applications without significant refactoring of their own discretization infrastructure.

2.1 Finite Element Operator Decomposition

Finite element operators are typically defined through weak formulations of partial differential equations that involve integration over a computational mesh. The required integrals are computed by splitting them as a

sum over the mesh elements, mapping each element to a simple reference element (e.g. the unit square) and applying a quadrature rule in reference space.

This sequence of operations highlights an inherent hierarchical structure present in all finite element operators where the evaluation starts on *global (trial) degrees of freedom (dofs)* on the whole mesh, restricts to *degrees of freedom on subdomains* (groups of elements), then moves to independent *degrees of freedom on each element*, transitions to independent *quadrature points* in reference space, performs the integration, and then goes back in reverse order to global (test) degrees of freedom on the whole mesh.

This is illustrated in Figure 1 for the simple case of symmetric linear operator on third order (Q_3) scalar continuous (H^1) elements, where we use the notions **T-vector**, **L-vector**, **E-vector** and **Q-vector** to represent the sets corresponding to the (true) degrees of freedom on the global mesh, the split local degrees of freedom on the subdomains, the split degrees of freedom on the mesh elements, and the values at quadrature points, respectively.

We refer to the operators that connect the different types of vectors as:

- Subdomain restriction P
- Element restriction G
- Basis (Dofs-to-Qpts) evaluator B
- Operator at quadrature points D

More generally, when the test and trial space differ, they get their own versions of P, G and B.

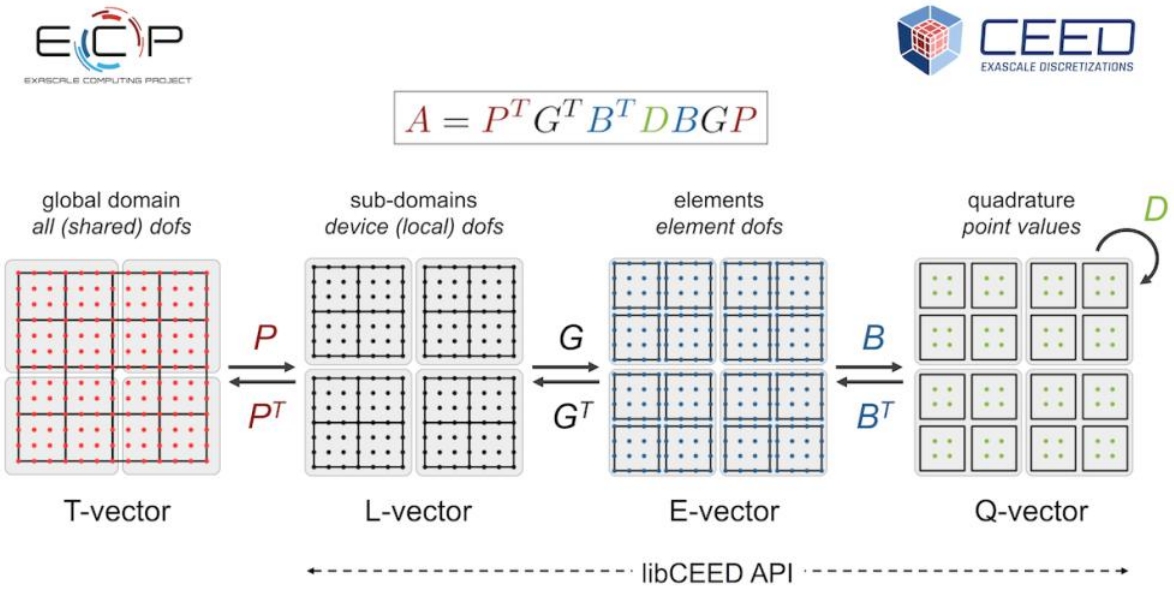

Figure 1: Fundamental finite element operator decomposition.

Note that in the case of adaptive mesh refinement (AMR), the restrictions P and G will involve not just extracting sub-vectors, but evaluating values at constrained degrees of freedom through the AMR interpolation. There can also be several levels of subdomains (P_1, P_2, etc.), and it may be convenient to split D as the product of several operators (D_1, D_2, etc.).

2.1.1 Partial Assembly

Since the global operator A is just a series of variational restrictions with B, G and P, starting from its point-wise kernel D, a "matvec" with A can be performed by evaluating and storing some of the innermost variational restriction matrices, and applying the rest of the operators "on-the-fly". For example, one can

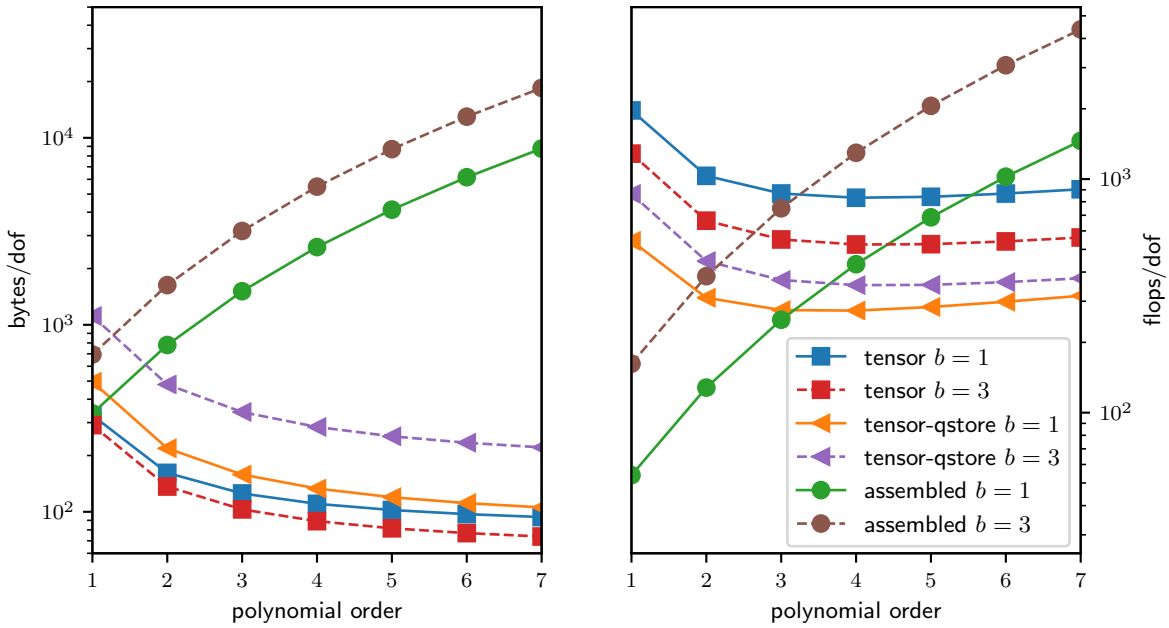

Figure 2: Comparison of memory transfer and floating point operations per degree of freedom for different representations of a linear operator for a PDE in 3D with b components and variable coefficients arising due to Newton linearization of a material nonlinearity. The "tensor" representation computes metric terms on the fly and stores a compact representation of the linearization at quadrature points. The "tensor-qstore" representation pulls the metric terms into the stored representation. The "assembled" representation uses a (block) CSR format.

compute and store a global matrix on **T-vector** level. Alternatively, one can compute and store only the subdomain (**L-vector**) or element (**E-vector**) matrices and perform the action of A using matvecs with P or P and G. While these options are natural for low-order discretizations, they are not a good fit for high-order methods due to the amount of FLOPs needed for their evaluation, as well as the memory transfer needed for a matvec.

Our focus in libCEED, instead, is on partial assembly, where we compute and store only D (or portions of it) and evaluate the actions of P, G and B on-the-fly. Critically for performance, we take advantage of the tensor-product structure of the degrees of freedom and quadrature points on quad and hex elements to perform the action of B without storing it as a matrix.

Implemented properly, the partial assembly algorithm requires optimal amount of memory transfers (with respect to the polynomial order) and near-optimal FLOPs for operator evaluation, see Figure 2. It consists of an operator *setup* phase, that evaluates and stores D and an operator *apply* (evaluation) phase that computes the action of A on an input vector. When desired, the setup phase may be done as a side-effect of evaluating a different operator, such as a nonlinear residual. The relative costs of the setup and apply phases are different depending on the physics being expressed and the representation of D.

2.1.2 Parallel Decomposition

After the application of each of the first three transition operators, P, G and B, the operator evaluation is decoupled on their ranges, so P, G and B allow us to "zoom-in" to subdomain, element and quadrature point level, ignoring the coupling at higher levels.

Thus, a natural mapping of A on a parallel computer is to split the **T-vector** over MPI ranks (a non-overlapping decomposition, as is typically used for sparse matrices), and then split the rest of the vector

types over computational devices (CPUs, GPUs, etc.) as indicated by the shaded regions in Figure 1.

One of the advantages of the decomposition perspective in these settings is that the operators P, G, B and D clearly separate the MPI parallelism in the operator (P) from the unstructured mesh topology (G), the choice of the finite element space/basis (B) and the geometry and point-wise physics D. These components also naturally fall in different classes of numerical algorithms – parallel (multi-device) linear algebra for P, sparse (on-device) linear algebra for G, dense/structured linear algebra (tensor contractions) for B and parallel point-wise evaluations for D.

Currently in libCEED, it is assumed that the host application manages the global **T-vector**s and the required communications among devices (which are generally on different compute nodes) with P. Our API is thus focused on the **L-vector** level, where the logical devices, which in the library are represented by the `Ceed` object, are independent. Each MPI rank can use one or more `Ceed`s, and each `Ceed`, in turn, can represent one or more physical devices, as long as libCEED backends support such configurations. The idea is that every MPI rank can use any logical device it is assigned at runtime. For example, on a node with 2 CPU sockets and 4 GPUs, one may decide to use 6 MPI ranks (each using a single `Ceed` object): 2 ranks using 1 CPU socket each, and 4 using 1 GPU each. Another choice could be to run 1 MPI rank on the whole node and use 5 `Ceed` objects: 1 managing all CPU cores on the 2 sockets and 4 managing 1 GPU each. The communications among the devices, e.g. required for applying the action of P, are currently out of scope of libCEED. The interface is non-blocking for all operations involving more than $O(1)$ data, allowing operations performed on a coprocessor or worker threads to overlap with operations on the host.

2.2 API Description

The libCEED API takes an algebraic approach, where the user essentially describes in the frontend the operators G, B and D and the library provides backend implementations and coordinates their action to the original operator on **L-vector** level (i.e. independently on each device / MPI task).

One of the advantages of this purely algebraic description is that it already includes all the finite element information, so the backends can operate on linear algebra level without explicit finite element code. The frontend description is general enough to support a wide variety of finite element algorithms, as well as some other types algorithms such as spectral finite differences. The separation of the front- and backends enables applications to easily switch/try different backends. It also enables backend developers to impact many applications from a single implementation.

Our long-term vision is to include a variety of backend implementations in libCEED, ranging from reference kernels to highly optimized kernels targeting specific devices (e.g. GPUs) or specific polynomial orders. A simple reference backend implementation is provided in the file `backends/ref/ceed-ref.c`.

On the frontend, the mapping between the decomposition concepts and the code implementation is as follows:

- **L-**, **E-** and **Q-vector** are represented as variables of type `CeedVector`. (A backend may choose to operate incrementally without forming explicit **E-** or **Q-vectors**.)
- G is represented as variable of type `CeedElemRestriction`.
- B is represented as variable of type `CeedBasis`.
- The action of D is represented as variable of type `CeedQFunction`.
- The overall operator $G^T B^T D B G$ is represented as variable of type `CeedOperator` and its action is accessible through `CeedOperatorApply()`.

To clarify these concepts and illustrate how they are combined in the API, consider the implementation of the action of a simple 1D mass matrix (see also file `tests/t30-operator.c`).

```
#include <ceed.h>

static int setup(void *ctx, void *qdata, CeedInt Q, const CeedScalar *const *u,
                 CeedScalar *const *v) {
  CeedScalar *w = qdata;
```

```
  for (CeedInt i=0; i<Q; i++) {
    w[i] = u[1][i]*u[4][i];
  }
  return 0;
}

static int mass(void *ctx, void *qdata, CeedInt Q, const CeedScalar *const *u,
                CeedScalar *const *v) {
  const CeedScalar *w = qdata;
  for (CeedInt i=0; i<Q; i++) {
    v[0][i] = w[i] * u[0][i];
  }
  return 0;
}

int main(int argc, char **argv) {
  Ceed ceed;
  CeedElemRestriction Erestrictx, Erestrictu;
  CeedBasis bx, bu;
  CeedQFunction qf_setup, qf_mass;
  CeedOperator op_setup, op_mass;
  CeedVector qdata, X, U, V;
  CeedInt nelem = 5, P = 5, Q = 8;
  CeedInt Nx = nelem+1, Nu = nelem*(P-1)+1;
  CeedInt indx[nelem*2], indu[nelem*P];
  CeedScalar x[Nx];

  CeedInit(argv[1], &ceed);
  for (CeedInt i=0; i<Nx; i++) x[i] = i / (Nx - 1);
  for (CeedInt i=0; i<nelem; i++) {
    indx[2*i+0] = i;
    indx[2*i+1] = i+1;
  }
  CeedElemRestrictionCreate(ceed, nelem, 2, Nx, CEED_MEM_HOST, CEED_USE_POINTER,
                            indx, &Erestrictx);

  for (CeedInt i=0; i<nelem; i++) {
    for (CeedInt j=0; j<P; j++) {
      indu[P*i+j] = i*(P-1) + j;
    }
  }
  CeedElemRestrictionCreate(ceed, nelem, P, Nu, CEED_MEM_HOST, CEED_USE_POINTER,
                            indu, &Erestrictu);

  CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, 2, Q, CEED_GAUSS, &bx);
  CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, P, Q, CEED_GAUSS, &bu);

  CeedQFunctionCreateInterior(ceed, 1, 1, sizeof(CeedScalar),
                              (CeedEvalMode)(CEED_EVAL_GRAD|CEED_EVAL_WEIGHT),
                              CEED_EVAL_NONE, setup, __FILE__ ":setup", &qf_setup);
  CeedQFunctionCreateInterior(ceed, 1, 1, sizeof(CeedScalar),
                              CEED_EVAL_INTERP, CEED_EVAL_INTERP,
                              mass, __FILE__ ":mass", &qf_mass);

  CeedOperatorCreate(ceed, Erestrictx, bx, qf_setup, NULL, NULL, &op_setup);
  CeedOperatorCreate(ceed, Erestrictu, bu, qf_mass, NULL, NULL, &op_mass);

  CeedVectorCreate(ceed, Nx, &X);
  CeedVectorSetArray(X, CEED_MEM_HOST, CEED_USE_POINTER, x);
  CeedOperatorGetQData(op_setup, &qdata);
  CeedOperatorApply(op_setup, qdata, X, NULL, CEED_REQUEST_IMMEDIATE);

  CeedVectorCreate(ceed, Nu, &U);
  CeedVectorCreate(ceed, Nu, &V);
  CeedOperatorApply(op_mass, qdata, U, V, CEED_REQUEST_IMMEDIATE);

  CeedQFunctionDestroy(&qf_setup);
  CeedQFunctionDestroy(&qf_mass);
```

```
  CeedOperatorDestroy(&op_setup);
  CeedOperatorDestroy(&op_mass);
  CeedElemRestrictionDestroy(&Erestrictu);
  CeedElemRestrictionDestroy(&Erestrictx);
  CeedBasisDestroy(&bu);
  CeedBasisDestroy(&bx);
  CeedVectorDestroy(&X);
  CeedVectorDestroy(&U);
  CeedVectorDestroy(&V);
  CeedDestroy(&ceed);
  return 0;
}
```

The constructor

```
  CeedInit(argv[1], &ceed);
```

creates a logical device `ceed` on the specified *resource*, which could also be a coprocessor such as `"/nvidia/0"`. There can be any number of such devices, including multiple logical devices driving the same resource (though performance may suffer in case of oversubscription). The resource is used to locate a suitable backend which will have discretion over the implementations of all objects created with this logical device.

The `setup` routine above computes and stores D, in this case a scalar value in each quadrature point, while `mass` uses these saved values to perform the action of D. These functions are turned into the `CeedQFunction` variables `qf_setup` and `qf_mass` in the `CeedQFunctionCreateInterior()` calls:

```
int setup(void *ctx, void *qdata, CeedInt Q,
          const CeedScalar *const *u, CeedScalar *const *v);
int mass(void *ctx, void *qdata, CeedInt Q,
         const CeedScalar *const *u, CeedScalar *const *v);

{
  CeedQFunction qf_setup, qf_mass;

  CeedQFunctionCreateInterior(ceed, 1, 1, sizeof(CeedScalar),
                              (CeedEvalMode)(CEED_EVAL_GRAD|CEED_EVAL_WEIGHT),
                              CEED_EVAL_NONE, setup, __FILE__ ":setup", &qf_setup);
  CeedQFunctionCreateInterior(ceed, 1, 1, sizeof(CeedScalar),
                              CEED_EVAL_INTERP, CEED_EVAL_INTERP,
                              mass, __FILE__ ":mass", &qf_mass);
}
```

A `CeedQFunction` performs independent operations at each quadrature point and the interface is intended to facilitate vectorization. The second argument is an expected vector length. If greater than 1, the caller must ensure that the number of quadrature points `Q` is divisible by the vector length. This is often satisfied automatically due to the element size or by batching elements together to facilitate vectorization in other stages, and can always be ensured by padding. The data at quadrature points, `qdata`, is opaque to the library and can be of any type; it is of type `CeedScalar` here because it simply stores a weight. The evaluation mode `CEED_EVAL_INTERP` for both inputs and outputs indicates that the mass operator only contains terms of the form

$$\int_\Omega v f_0(u)$$

where v are test functions. More general operators, such as those of the form

$$\int_\Omega v f_0(u, \nabla u) + \nabla v \cdot f_1(u, \nabla u)$$

can be expressed using a bitwise or `CEED_EVAL_INTERP | CEED_EVAL_GRAD`, in which case the callback will receive multiple inputs (outputs).

In addition to the function pointers (`setup` and `mass`), `CeedQFunction` constructors take a string representation specifying where the source for the implementation is found. This is used by backends that support Just-In-Time (JIT) compilation (i.e., OCCA) to compile for coprocessors.

The B operators for the mesh nodes, `bx`, and the unknown field, `bu`, are defined in the calls to the function `CeedBasisCreateTensorH1Lagrange`. In this example, both the mesh and the unknown field use H^1

Lagrange finite elements of order 1 and 4 respectively (the `P` argument represents the number of 1D degrees of freedom on each element). Both basis operators use the same integration rule, which is Gauss-Legendre with 8 points (the `Q` argument).

```
CeedBasis bx, bu;

CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, 2, Q, CEED_GAUSS, &bx);
CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, P, Q, CEED_GAUSS, &bu);
```

Other elements with this structure can be specified in terms of the $\texttt{Q} \times \texttt{P}$ matrices that evaluate values and gradients at quadrature points in one dimension using `CeedBasisCreateTensorH1`. Elements that do not have tensor product structure, such as symmetric elements on simplices, will be created using different constructors.

The G operators for the mesh nodes, `Erestrictx`, and the unknown field, `Erestrictu`, are specified in the `CeedElemRestrictionCreate()`. Both of these specify directly the dof indices for each element in the `indx` and `indu` arrays:

```
CeedInt indx[nelem*2], indu[nelem*P];

/* indx[i] = ...; indu[i] = ...; */

CeedElemRestrictionCreate(ceed, nelem, 2, Nx, CEED_MEM_HOST, CEED_USE_POINTER,
                          indx, &Erestrictx);
CeedElemRestrictionCreate(ceed, nelem, P, Nu, CEED_MEM_HOST, CEED_USE_POINTER,
                          indu, &Erestrictu);
```

If the user has arrays available on a device, they can be provided using `CEED_MEM_DEVICE`. This technique is used to provide no-copy interfaces in all contexts that involve problem-sized data.

For discontinuous Galerkin and for applications such as Nek5000 that only explicitly store **E-vectors** (inter-element continuity has been subsumed by the parallel restriction P), the element restriction G is the identity so the explicit indices can be elided (`NULL`). We plan to support other structured representations of G which will be added according to demand. In the case of non-conforming mesh elements, G needs a more general representation that expresses values at slave nodes (which do not appear in **L-vectors**) as linear combinations of the degrees of freedom at master nodes.

With partial assembly, we first perform a setup stage where D is evaluated and stored. This is accomplished by the operator `op_setup` and its application to `X`, the nodes of the mesh (these are needed to compute Jacobians at quadrature points). Note that the corresponding `CeedOperatorApply` has only input (the output is `NULL`):

```
CeedVectorCreate(ceed, Nx, &X);
CeedVectorSetArray(X, CEED_MEM_HOST, CEED_USE_POINTER, x);
CeedOperatorGetQData(op_setup, &qdata);
CeedOperatorApply(op_setup, qdata, X, NULL, CEED_REQUEST_IMMEDIATE);
```

The action of the operator is then represented by operator `op_mass` and its `CeedOperatorApply` to the input **L-vector** `U` with output in `V`:

```
CeedVectorCreate(ceed, Nu, &U);
CeedVectorCreate(ceed, Nu, &V);
CeedOperatorApply(op_mass, qdata, U, V, CEED_REQUEST_IMMEDIATE);
```

A number of function calls in the interface, such as `CeedOperatorApply`, are intended to support asynchronous execution via their last argument, `CeedRequest*`. The specific (pointer) value used in the above example, `CEED_REQUEST_IMMEDIATE`, is used to express the request (from the user) for the operation to complete before returning from the function call, i.e. to make sure that the result of the operation is available in the output parameters immediately after the call. For a true asynchronous call, one needs to provide the address of a user defined variable. Such a variable can be used later to explicitly wait for the completion of the operation.

2.3 Interface Principles and Evolution

LibCEED is intended to be extensible via backends that are packaged with the library and packaged separately (possibly as a binary containing proprietary code). Backends are registered by calling `CeedRegister(prefix,`

`init_function)`, typically in a library initializer or "constructor" that runs automatically. `CeedInit` uses this prefix to find an appropriate backend for the resource.

Source (API) and binary (ABI) stability are important to libCEED. LibCEED is evolving rapidly at present, but we expect it to stabilize soon at which point we will adopt semantic versioning. User code, including libraries of `CeedQFunctions`, will not need to be recompiled except between major releases. The backends currently have some dependence beyond the public user interface, but we intent to remove that dependence and will prioritize if anyone expresses interest in distributing a backend outside the libCEED repository.

3. UPDATES IN THE CEED MINIAPPS AND ECP APPLICATIONS

In this section we provide brief highlights from CEED's recent work with ECP applications and software technologies projects.

Some updates from integration with the MARBL app are discussed in Section 3.2. A main theme in this work has been the systematic transition of advanced algorithms, developed and tested in the Laghos miniapp, to the large-scale multi-physics settings of the MARBL code.

On the miniapp side, both the Laghos and Nekbone miniapps are now part of the ECP Proxy Applications Suite v1.0. Both miniapps were also picked to be CORAL-2 benchmarks, and Laghos was selected as one of LLNL's ASC co-design miniapps. As part of the preparation of the the CORAL-2 benchmark problem, Laghos was scaled up to the full Vulcan BG/Q machine at LLNL (384K MPI tasks, 200M mesh elements, 63B total unknowns). The results of these runs are described in the following Section 3.1.

3.1 Laghos Tests on BG/Q

Here we present the baseline results obtained for Laghos on the Vulcan BG/Q machine at LLNL. We ran the 3D Sedov shock test case and documented results for three different finite element space configurations, namely, Q_2Q_1, Q_3Q_2 and Q_4Q_3. All computations were performed using the partial assembly (PA, see Section 2.1.1) option, which is much more efficient than the full assembly in 3D, as shown in CEED-MS8. Every test executed three RK4 time steps with exactly 50 CG iterations per Runge-Kutta stage, resulting in a total of 12×50 CG iterations. The computational mesh was always distributed between the MPI tasks by a Cartesian partition, so that each MPI task had the same number of zones. We used the `mpixlcxx_r-fastmpi` compiler on Vulcan to build the required *hypre* and MFEM libraries.

In Figures 3–5 we show execution rates of the major Laghos kernels, namely, (i) CG inversion of the global $H1$ mass operator, (ii) application of the global force operator, and (iii) update of the physics-based quadrature data, along with the total execution rate in Figure 6. We observe that most execution rates have similar values for Q_2Q_1 and Q_3Q_2, which are always better than Q_4Q_3. All rates do not seem to be influenced by the number of utilized nodes.

Figures 7 and 8 display the weak scaling of the CG iterations and all combined kernels, respectively. The other major kernels are not shown as they are less communication-intensive than the conjugate gradient one. We observe almost ideal weak scaling for the performed tests.

The strong scaling study for CG and all combined kernels is shown in Figures 9 and 10, respectively. We observe some slope deterioration as the limit of one zone per MPI task is approached, but the behavior improves when the finite element order is increased. These tests use at most 64K Vulcan cores (4K nodes).

3.2 Updates in MARBL

The MARBL-related activities during this reporting period were focused on the following tasks:

1. **Full completion of the refactoring of MARBL/BLAST's Lagrangian phase.** As discussed in our prior milestone report, CEED-MS8, this involved rearranging the quadrature-based computations in BLAST. This task was completed in a feature branch to be merged into BLAST's master in Q2/FY18, allowing the kernels from Laghos to be moved into MARBL.

2. **Integration of partial assembly (PA, see Section 2.1.1) kernels into BLAST's Lagrange phase.** The PA kernels for the action of the force operator and action of the mass matrices were taken

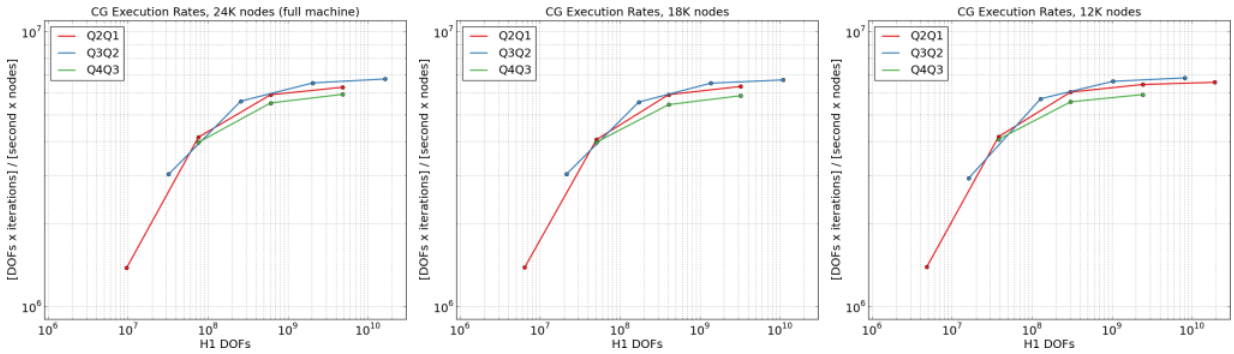

Figure 3: Laghos: Inversion of the global mass operator.

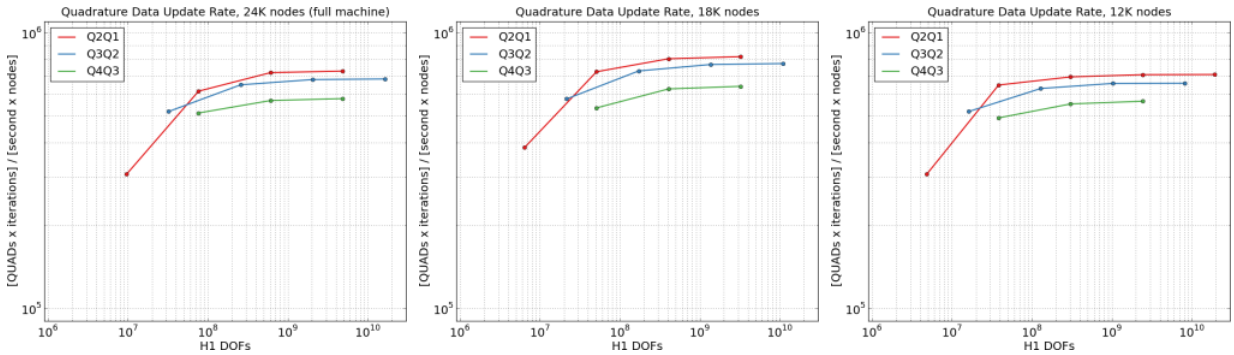

Figure 4: Laghos: Application of the force operator.

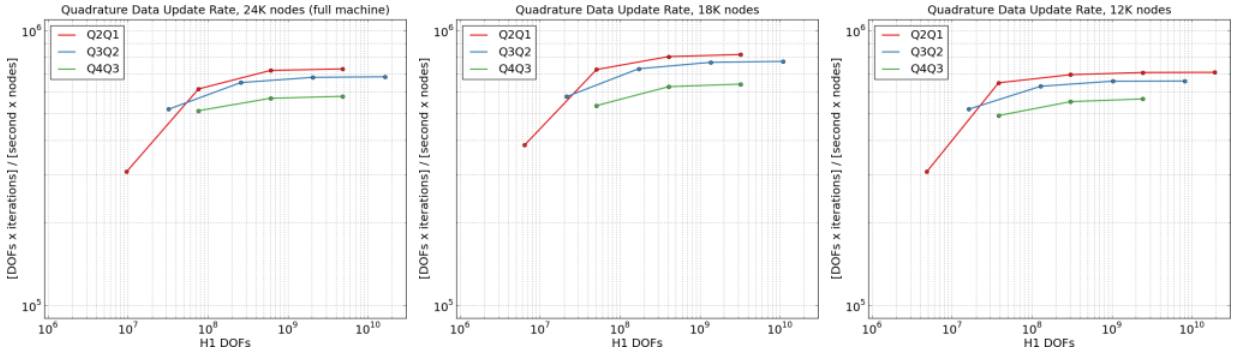

Figure 5: Laghos: Update of quadrature data.

from Laghos and successfully used in BLAST. The new capabilities were tested, and the results obtained by the fully and partially assembled operators agreed, in both 2D and 3D.

3. **Physics-related extensions.** As Laghos is a single-material miniapp with simplified physics, a set of extensions is needed to support all of BLAST's capabilities. During this period, we extended the PA mass operators with capability to perform axisymmetric calculations, and a similar extension for the force operators is in progress. In addition, all operators were extended to the case of multi-material calculations.

Topics of near future work include: (i) adding PA operators for fast evaluation of gradients and mesh Jacobians during the quadrature data update, (ii) support for multi-material closure model calculations, (iii) support for calculations related to material strength models and magnetohydrodynamics coupling.

Once the above steps are completed, MARBL/BLAST will maintain all of its established functionality, while using the same kernels as in Laghos. This new structure will allow quick inclusion of any optimizations

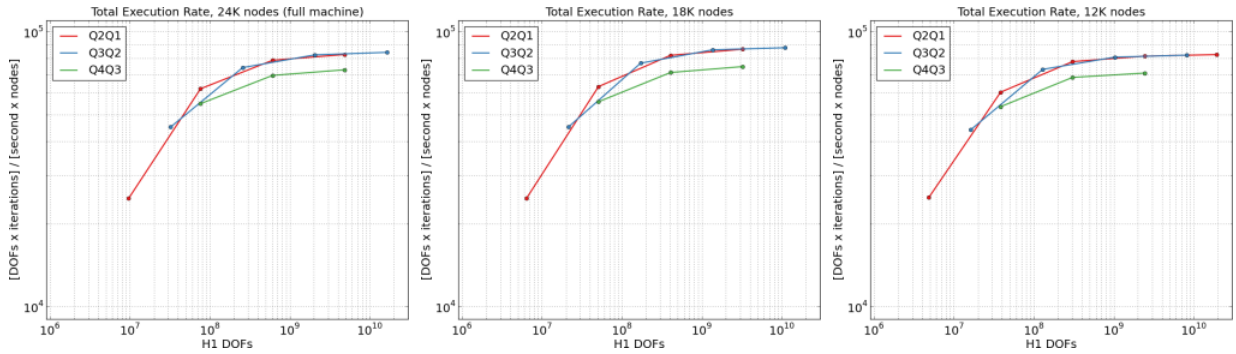

Figure 6: Laghos: Total execution rates.

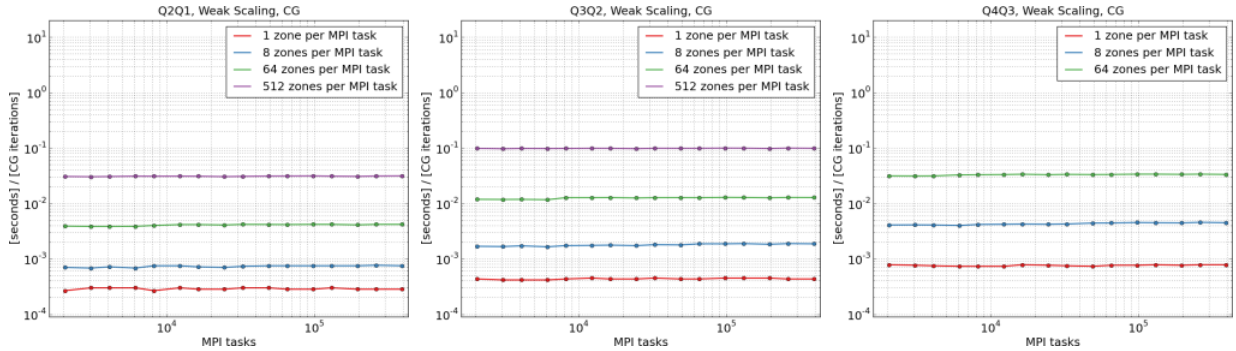

Figure 7: Laghos: Weak scaling of the CG iterations for different spaces.

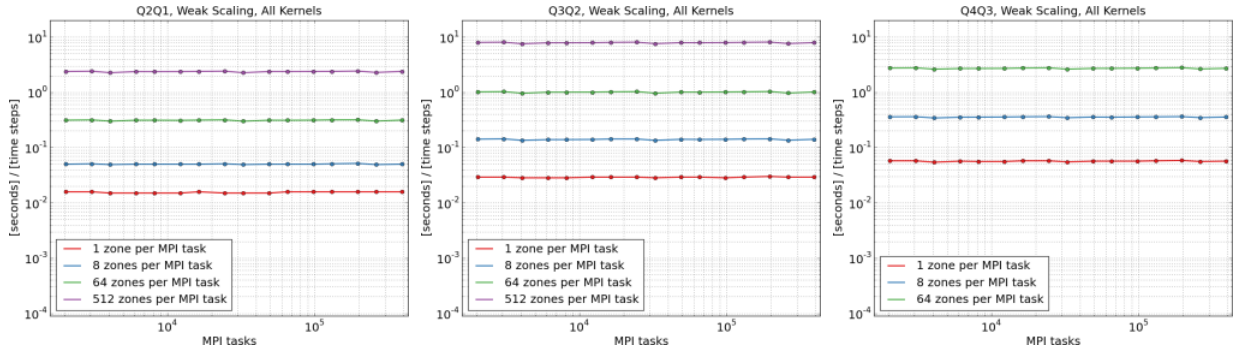

Figure 8: Laghos: Weak scaling of all major kernels for different spaces.

done in Laghos, and give us the ability to compare the performance resulting from the different CPU/GPU optimization approaches.

4. OTHER PROJECT ACTIVITIES

4.1 Nek5000 v17.0 Release

Nek5000 v17.0 was released as a major upgrade to Nek5000. Major features improvements include:

- Refactored build system.
- New user-input parameter file format (*.par* replacing *.rea*).
- Characteristics (large time-step) support for moving mesh problems.

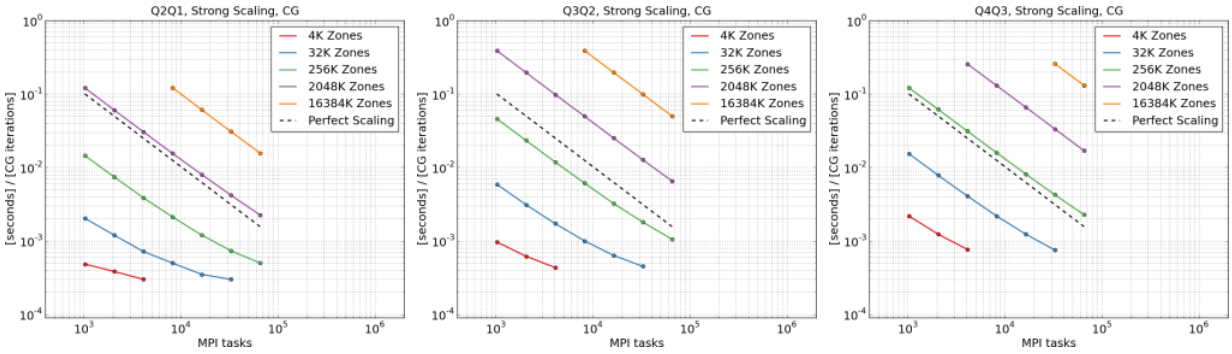

Figure 9: Laghos: Strong scaling of the CG iterations for different spaces.

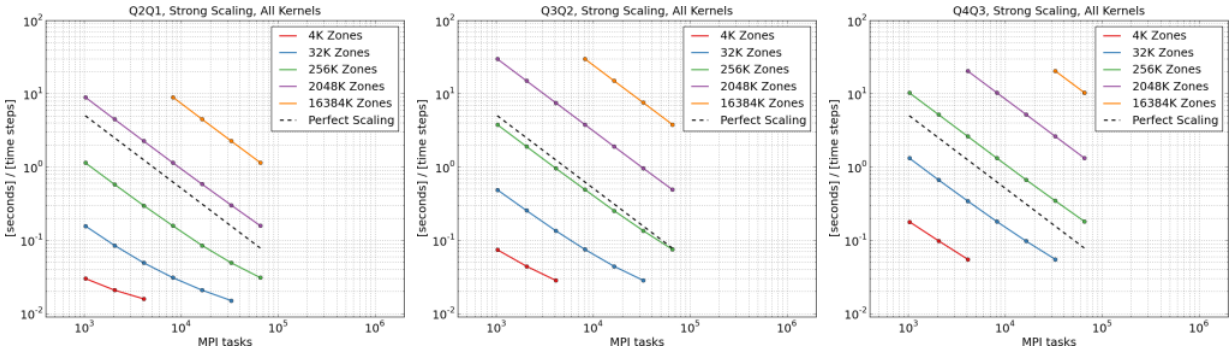

Figure 10: Laghos: Strong scaling of all major kernels for different spaces.

- Moving mesh support for the $PN - PN$ formulation.
- Improved stability for $PN - PN$ with variable viscosity.
- Support for mixed *Helmholtz/CVODE* solves.
- New fast *AMG setup* tool based on HYPRE.
- New *EXODUSII* mesh converter.
- New interface to *libxsmm* (fast MATMUL library).
- Extended *lowMach* solver for time varying thermodynamic pressure.
- Added DG for scalars.
- Reduced solver initialization time (parallel binary reader for all input files).
- Automatic general mesh-to-mesh transfer for restarts.
- Refactored support for overlapping domains (NekNek).
- Added high-pass filter relaxation (alternative to explicit filter).
- Refactored residual projection including support for coupled Helmholtz solves.

4.2 MFEM v3.3.2 Release

MFEM v3.3.2 was released with many new features, including:

- Support for high-order mesh optimization based on the target-matrix optimization paradigm.

- Implementation of the xSDK community policies.
- Integration with STRUMPACK.
- New linear interpolators, examples and miniapps.
- Various memory, performance, discretization and solver improvements.
- Continuous integration testing on Linux, Mac and Windows.
- And many more...

See `http://mfem.org` for more details.

4.3 PETSc v3.8 Release

PETSc v3.8 was released with many new features, including:

- A revamped Fortran interface.
- New FETI-DP preconditioners.
- Improved unstructured and adaptive mesh support.
- New adaptive controllers for time integration.
- Support for discrete and continuous adjoints of time-dependent models.

4.4 ECP/ST and SciDAC Collaboration

MFEM joined the xSDK project in ECP/ST as of release xSDK-0.3.0, see `https://xsdk.info/packages`. MFEM and PUMI are also part of the FASTMath institute in SciDAC. One example of collaboration in this area is the Center for Integrated Simulation of Fusion Relevant RF Actuators FES SciDAC partnership, where the PUMI team has initiated efforts to go from complex antenna CAD models to MFEM simulations.

4.5 Nek5000 Hackathon

The inaugural Nek5000 Hackathon held at University of Illinois, Urbana-Champaign was attended by researchers and Nek5000 developers to promote the application of Nek5000 to new problems from industry, national laboratories, and academia. Twenty-five participants spent three days working on setting up new examples, developing new features, and helping one another to get maximum performance on their applications. Some of the more prominent exchanges of ideas included standardization of synthetic turbulent inflow techniques, use of CVODE for pure advection-diffusion problems, and the use of the characteristics methods for moving geometry applications.

4.6 Outreach

CEED researchers were involved in a number of outreach activities, including vendor deep-dives: Cray (18-19), AMD (24-25), Intel (31-2) and a week-long PathForward reviews at LLNL. CEED researchers (P. Fischer, E. Merzari, A. Obabko) won a Best Paper Award at the 17th International Topical Meeting on Nuclear Reactor Thermal Hydraulics (NURETH-17) and a CEED-organized 3-part minisymposium has been accepted at the 2018 International Conference on Spectral and High-Order Methods (ICOSAHOM18). The project was highlighted in an article on the ECP website (“Co-Design Center Develops Next-Generation Simulation Tools”), HPCwire, Twitter and LLNL’s CASC newsletter. Nine CEED researchers attended the SC17 conference.

5. CONCLUSION

In this milestone we initiated the development and implementation of a common CEED API, focusing on its low-level API component that it makes it easy for applications that have their own discretization to take advantage of the high-performance algorithms developed under CEED. This work is in preparation for the release of CEED v1.0 in the next quarter, see CEED-MS13. We delivered a initial CEED API software implementation (libCEED) which is available through the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed`. In this report, we also provided a highlight from CEED's integration with the MARBL app and the corresponding Laghos miniapp and described additional CEED activities performed in Q1 of FY18, including: the Nek5000 v17.0 MFEM v3.3.2 and PETSc v3.8 releases, collaborations with ECP/ST and SciDAC projects, the inaugural Nek5000 hackathon and various outreach activities.

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-743647.