



ECP Milestone Report
Public release of CEED 1.0
WBS 2.2.6.06, Milestone CEED-MS13

Jed Brown
Ahmad Abdelfata
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
Tim Warburton

March 31, 2018

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via US Department of Energy (DOE) SciTech Connect.

Website <http://www.osti.gov/scitech/>

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service

5285 Port Royal Road

Springfield, VA 22161

Telephone 703-605-6000 (1-800-553-6847)

TDD 703-487-4639

Fax 703-605-6900

E-mail info@ntis.gov

Website <http://www.ntis.gov/help/ordermethods.aspx>

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange representatives, and International Nuclear Information System representatives from the following source:

Office of Scientific and Technical Information

PO Box 62

Oak Ridge, TN 37831

Telephone 865-576-8401

Fax 865-576-5728

E-mail reports@osti.gov

Website <http://www.osti.gov/contact.html>

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ECP Milestone Report
Public release of CEED 1.0
WBS 2.2.6.06, Milestone CEED-MS13

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

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

March 31, 2018

ECP Milestone Report
Public release of CEED 1.0
WBS 2.2.6.06, Milestone CEED-MS13

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	March 31, 2018	Original	

EXECUTIVE SUMMARY

In this milestone, we created and made publicly available the first full CEED software distribution, release CEED 1.0, consisting of software components such as MFEM, Nek5000, PETSc, MAGMA, OCCA, etc., treated as dependencies of CEED. The release consists of 12 integrated Spack packages for libCEED, mfem, nek5000, nekcem, laghos, nekbone, hpgmg, occa, magma, gslib, petsc and pumi plus a new CEED *meta-package*. We choose to use the Spack package manager to provide a common, easy-to-use build environment, where the user can build the CEED distribution with all dependencies.

The artifacts delivered include a consistent build system based on the above 13 Spack packages, documentation and verification of the build process, as well as improvements in the integration between different CEED components. As part of CEED 1.0, we also released the next version of libCEED, which contains major improvements in the OCCA backend and a new MAGMA backend. See the CEED website, <http://ceed.exascaleproject.org/ceed-1.0/> 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 Q2 of FY18, including: benchmark release by the Paranumal team, collaboration with SciDAC projects, organization of batched BLAS mini-symposium at SIAM PP18, collaboration with Zfp, the OCCA 1.0-alpha pre-release, and other outreach efforts.

TABLE OF CONTENTS

Executive Summary	vi
List of Figures	viii
1 Introduction	1
2 The CEED 1.0 Distribution	1
2.1 Spack packages	1
2.2 Testing and documentation	2
2.3 libCEED improvements	4
2.3.1 OCCA backend	4
2.3.2 New MAGMA backend	5
2.4 Interoperability	6
2.4.1 Parallel conforming mesh adaptation	6
3 CEED Kernels and Benchmarks	9
3.1 Bake-off kernels (BKs)	9
3.2 New fused GPU kernels using register shuffling	9
3.2.1 Overview	9
3.2.2 Preliminary results	10
3.3 Optimizing finite element stiffness operations on the NVIDIA V100 SXM2 GPU	11
3.3.1 Micro-benchmarking the NVIDIA V100 SXM2 GPU	12
3.3.2 Revisiting the VT CEED hexahedral bake-off kernels on the NVIDIA V100	13
3.3.3 Optimizing finite element stiffness action kernels for affine tetrahedral elements	13
3.3.4 Appendix: Code listing for ellipticAxTet3DK10.okl	17
3.4 MAGMA batched kernels	18
4 CEED Applications and Miniapps	22
4.1 ExaSMR	22
4.2 MARBL	22
4.3 ExaWind	23
4.4 Laghos	23
4.4.1 Laghos developments	23
4.4.2 Initial RAJA and pure CPU, GPU ports	23
4.4.3 First wave results	25
4.4.4 Second wave results	27
4.4.5 Third wave results	27
5 Other Project Activities	29
5.1 Benchmark release by Paranumal team	29
5.2 SciDAC collaborations	29
5.3 Batched BLAS minisymposium at SIAM PP18	31
5.4 Collaboration with Zfp	31
5.5 OCCA 1.0-alpha pre-release	31
5.6 Outreach	31
6 Conclusion	31

LIST OF FIGURES

1	Machines used to verify the CEED 1.0 distribution	3
2	libCEED allows different ECP applications to share highly optimized discretization kernels	4
3	Connections between the CEED software components created during the project (before: left, after: right). Discretization libraries are in blue, miniapps in green, performance libraries in orange, and discretization tools in purple. Note the central role of libCEED.	7
4	An example of order elevation on complex geometry: linear element (left) and quadratic element (right)	8
5	Curved mesh adaptation. 3D CAD model of the pillbox (top) with the loaded surface colored, side view of the initial mesh (bottom left) and final adapted mesh after deformation (bottom right).	8
6	Performance on Pascal P100 for BK 0.5	10
7	Performance on Pascal P100 for BK 1.0	11
8	Left: Roofline model for the NVIDIA V100 SXM2 16GB Volta class GPU. Each shaded region corresponds to the target band for a kernel with a given number of shared+L1 bytes accessed per flop. Right: occaBench benchmark performance results for mixed streaming and compute on NVIDIA V100 SXM2 16GB Volta class GPU.	12
9	VT experiments on an NVIDIA V100 SXM2 GPU with 4096 hexes. Left: BK1 with $(N+2)^3$ intermediate Gauss Legendre quadrature nodes. Middle: BK3 with $(N+2)^3$ intermediate Gauss Legendre quadrature nodes. Right: BK3 with $(N+1)^3$ Gauss-Lobatto-Legendre quadrature nodes.	13
10	Performance results for the BK3 bake-off kernel on a mesh of 320,000 affine tetrahedral elements on Volta Titan V GPU. Left: FP64 floating point performance for kernels K0:10 for polynomial degrees 1:8. Right: FP64 throughput measured in giga-nodes per second.	15
11	Left: FP64 GFLOPS/s performance of the optimal kernel from K0:10 for each polynomial degree on a V100. Right: throughput measured in billion nodes per second (GNODES/s) on a single V100.	15
12	Left: FP64 GFLOPS/s performance of the cuBLAS implementation on a V100. Right: throughput measured in billion nodes per second (GNODES/s) on V100.	17
13	Memory hierarchies of the experimental CPU and GPU hardware targeted for development and optimization of Batched BLAS in MAGMA.	19
14	Batched DGEMM on P100 GPU (Left) and V100 GPU (Right).	19
15	Performance comparison (in Gflop/s) of fused batched DGEMMs on P100 GPU (Left) vs. non-fused batched DGEMMs on V100 GPU (Right).	20
16	Batched DGEMM in cuBLAS and MAGMA vs. Batched fused DGEMM in MAGMA on matrices of sizes coming from real applications (in the MFEM library). The speedups reported are compared to the cuBLAS non-fused batched DGEMMs.	21
17	Laghos P0,2D serial speedup : RAJA/Master and OCCA/Master	25
18	Laghos P0,2D test case: CUDA Speedup: OCCA/RAJA	27
19	Laghos Kernels serial speedup on Ray with templated arguments	28
20	Laghos P1,2D CUDA speedup : OCCA/templated-RAJA on GeForce GTX 1070	28
21	Laghos P0,2D CUDA speedup : OCCA/templated-RAJA on Ray	29
22	Laghos P1,3D CUDA NVVP Kernels profile	29
23	Laghos P1,3D CUDA Kernels speedup vs. OCCA	30

1. INTRODUCTION

In this milestone, we created and made publicly available the first full CEED software distribution, release CEED 1.0, consisting of software components such as MFEM, Nek5000, PETSc, MAGMA, OCCA, etc., treated as dependencies of CEED. The release consists of 12 integrated Spack packages for libCEED, mfem, nek5000, nekcem, laghos, nekbone, hpgmg, occa, magma, gslib, petsc and pumi plus a new CEED *meta-package*. We choose to use the Spack package manager to provide a common, easy-to-use build environment, where the user can build the CEED distribution with all dependencies.

The artifacts delivered include a consistent build system based on the above 13 Spack packages, documentation and verification of the build process, as well as improvements in the integration between different CEED components. As part of CEED 1.0, we also released the next version of libCEED, which contains major improvements in the OCCA backend and a new MAGMA backend. See the CEED website, <http://ceed.exascaleproject.org/ceed-1.0/> and the CEED GitHub organization, <http://github.com/ceed> for more details.

2. THE CEED 1.0 DISTRIBUTION

The CEED distribution is a collection of software packages that integrate to enable efficient discretization algorithms for high-order PDE-based applications on unstructured grids.

CEED is using the Spack package manager to enable the compatible building and installation of its software components. Spack is a package manager for scientific software that supports multiple versions, configurations, platforms, and compilers. While Spack does not change the build system that already exists in each CEED component, it coordinates the dependencies between these components and enables them to be build with the same compilers and options.

2.1 Spack packages

In its initial version, CEED 1.0, the CEED software suite consists of the following 12 packages, plus a new CEED *meta-package* in Spack:

- GSLIB
- HPGMG
- Laghos
- libCEED
- MAGMA
- MFEM
- Nek5000
- Nekbone
- NekCEM
- PETSc
- PUMI
- OCCA

If Spack is already installed on your system and is part of your PATH, you can install the CEED software simply with:

```
1 spack install -v ceed
```

To enable package testing during the build process, use instead:

```
1 spack install -v --test=all ceed
```

If you don't have Spack, you can download it and install CEED with the following commands:

```
1 git clone https://github.com/spack/spack.git
2 cd spack
3 ./bin/spack install -v ceed
```

Spack will install the CEED packages (and the libraries they depend on) in a subtree of `./opt/spack/` that is specific for the architecture and compiler used (multiple compiler and/or architecture builds can coexist in a single Spack directory). You can then use this installation to build for example MFEM-codes as follows:

```
1 git clone git@github.com:mfem/mfem.git
2 cd mfem; git checkout v3.3.2
3 cd examples
4 make CONFIG_MK='spack location -i mfem'/share/mfem/config.mk
5 cd ../miniapps/electromagnetics
6 make CONFIG_MK='spack location -i mfem'/share/mfem/config.mk
```

Similarly, libCEED-based applications can be build with

```
1 git clone git@github.com:CEED/libCEED.git
2 cd libCEED/examples/ceed
3 make CEED_DIR='spack location -i libceed'
4 ./ex1 -ceed /cpu/self
```

2.2 Testing and documentation

The build process has been documented on the CEED project website at to make it easier for application scientist that may be new to Spack to install and use our software, see <http://ceed.exascaleproject.org/ceed-1.0>. This documentation includes a comprehensive collection of machine-specific settings for the leadership computing facilities at ALCF, OLCF, NERSC and LLNL. Such configuration files can significantly speed-up the Spack installation by providing the locations of common build tools (e.g. MPI). One example is the `packages.yaml` file for the TOSS3 system type at Livermore Computing.

```
1 packages:
2   all:
3     compiler: [intel, gcc, clang, pgi]
4     providers:
5       mpi: [mvapich2, mpich, openmpi]
6       blas: [intel-mkl, openblas]
7       lapack: [intel-mkl, openblas]
8   intel-mkl:
9     paths:
10       intel-mkl@2018.0.128: /usr/tce/packages/mkl/mkl-2018.0
11     buildable: False
12   mvapich2:
13     paths:
14       mvapich2@2.2%intel@18.0.1: /usr/tce/packages/mvapich2/mvapich2-2.2-intel-18.0.1
15       mvapich2@2.2%gcc@4.9.3: /usr/tce/packages/mvapich2/mvapich2-2.2-gcc-4.9.3
16       mvapich2@2.2%gcc@7.1.0: /usr/tce/packages/mvapich2/mvapich2-2.2-gcc-7.1.0
17     buildable: False
18
19   cmake:
20     paths:
21       cmake@3.8.2: /usr/tce/packages/cmake/cmake-3.8.2
22     buildable: False
23   python:
24     paths:
25       python@2.7.14: /usr/tce/packages/python/python-2.7.14
26     buildable: False
27   zlib:
28     paths:
29       zlib@1.2.7: /usr
30     buildable: False
```

Another example is the `packages.yaml` file for ALCF's Theta machine:

```

1 packages:
2   all:
3     compiler: [intel@16.0.3.210, gcc@5.3.0]
4     providers:
5       mpi: [mpich]
6   intel-mkl:
7     paths:
8       intel-mkl@16.0.3.210%intel@16.0.3.210 arch=cray-CNL-mic_knl: /opt/intel
9     buildable: False
10  mpich:
11    modules:
12      mpich@7.6.3%gcc@5.3.0 arch=cray-CNL-mic_knl: cray-mpich/7.6.3
13      mpich@7.6.3%intel@16.0.3.210 arch=cray-CNL-mic_knl: cray-mpich/7.6.3
14    buildable: False
15
16  cmake:
17    paths:
18      cmake@3.5.2%gcc@5.3.0 arch=cray-CNL-mic_knl: /usr
19      cmake@3.5.2%intel@16.0.3.210 arch=cray-CNL-mic_knl: /usr
20    buildable: False
21  libx11:
22    paths:
23      libx11@system: /usr
24    version: [system]
25    buildable: False
26  libxt:
27    paths:
28      libxt@system: /usr
29    version: [system]
30    buildable: False
31  python:
32    paths:
33      python@2.7.13%gcc@5.3.0 arch=cray-CNL-mic_knl: /usr
34      python@2.7.13%intel@16.0.3.210 arch=cray-CNL-mic_knl: /usr
35    buildable: False

```

We developed and tested machine-specific configurations for Spack packages and compilers on a variety of other machines. The full list is presented in Figure 1.











Platform	Architecture	Spack Configuration
Mac	darwin-x86_64	packages 
Linux (RHEL7)	linux-rhel7-x86_64	packages 
Cori (NERSC)	cray-CNL-haswell	packages 
Edison (NERSC)	cray-CNL-ivybridge	packages 
Theta (ALCF)	cray-CNL-mic_knl	packages 
Titan (OLCF)	cray-cn15-interlagos	packages 
CORAL-EA (LLNL)	blueos_3_ppc64le_ib	packages  compilers 
TOSS3 (LLNL)	toss_3_x86_64_ib	packages  compilers 

Figure 1: Machines used to verify the CEED 1.0 distribution

2.3 libCEED improvements

The CEED API library, libCEED, was released in milestone CEED-MS10 as a lightweight portable library that allows a wide variety of ECP applications to share highly optimized discretization kernels. This central role of libCEED is illustrated in Figure 2.

A main component of the CEED 1.0 effort, was the continued improvement of libCEED and specifically the release of its next version, libCEED-0.2. A lot of that work was focused on adding new backends and extending the existing ones as described below.

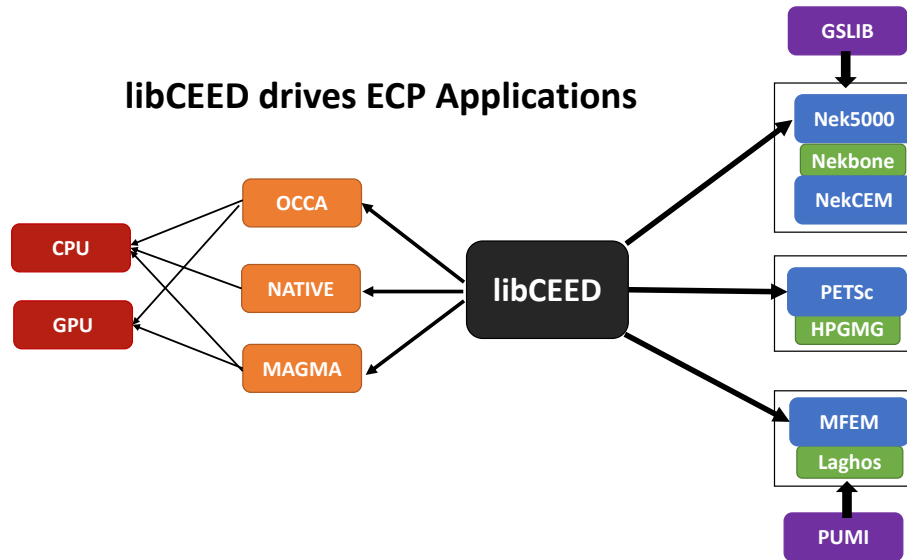


Figure 2: libCEED allows different ECP applications to share highly optimized discretization kernels

2.3.1 OCCA backend

The OCCA backend saw significant improvements from libCEED-0.1 to libCEED-0.2. With the latest version, four different OCCA backends are available to libCEED applications:

- `/cpu/occa` – serial backend
- `/gpu/occa` – CUDA backend
- `/omp/occa` – OpenMP backend
- `/ocl/occa` – OpenCL backend

We emphasize that a single backend implementation can be shared between very different frontend applications. To illustrate this point we provide below a complete demo for installing CEED 1.0 and *using the same /gpu/occa libCEED kernels in MFEM, PETSc and Nek example codes.*

```

1 # Install CEED 1.0 distribution via Spack
2 git clone git@github.com:spack/spack.git
3 cd spack
4 spack install ceed+cuda
5
6 # Setup CEED component directories
7 setenv CEED_DIR 'spack location -i libceed'
8 setenv MFEM_DIR 'spack location -i mfem'
9 setenv PETSC_DIR 'spack location -i petsc'
10 setenv NEK5K_DIR 'spack location -i nek5000'

```

```

11
12 # Clone libCEED examples directory as proxy for libCEED-based codes
13 git clone git@github.com:CEED/libCEED.git
14 mv libCEED/examples ceed-examples
15 rm -rf libCEED
16
17 # libCEED examples
18 cd ceed-examples/ceed
19 make
20 ./ex1 -ceed /gpu/occa
21 cd ../..
22
23 # MFEM+libCEED examples
24 cd ceed-examples/mfem
25 make
26 ./bp1 -ceed /gpu/occa -no-vis
27 cd ../..
28
29 # PETSc+libCEED examples
30 cd ceed-examples/petsc
31 make
32 ./bp1 -ceed /gpu/occa
33 cd ../..
34
35 # Nek+libCEED examples
36 cd ceed-examples/nek5000
37 ./make-nek-examples.sh
38 ./run-nek-example.sh -ceed /gpu/occa -b 3
39 cd ../..

```

2.3.2 New MAGMA backend

As part of CEED 1.0 we integrated a MAGMA backend in libCEED. This is our initial integration that sets up the framework of using MAGMA and provides the libCEED functionality through MAGMA kernels as one of libCEED's computational backends. As any other backend, the MAGMA backend provides extended basic data structures for `CeedVector`, `CeedElemRestriction`, and `CeedOperator`, and implements the fundamental CEED building blocks to work with the new data structures.

In general, the MAGMA-specific data structures keep the libCEED pointers to CPU data but also add corresponding device (e.g., GPU) pointers to the data. Coherency is handled internally, and thus seamlessly to the user, through the functions/methods that are provided to support them. These functions are specified and follow the libCEED API. For example, `CeedVector_Magma` and `CeedElemRestriction_Magma` are given as follows:

```

1 typedef struct {
2     CeedScalar *array, *darray;
3     int own_, down_;
4 } CeedVector_Magma;
5
6 typedef struct {
7     CeedInt *indices, *dindices;
8     int own_, down_;
9 } CeedElemRestriction_Magma;

```

Here `array` is pointer to CPU/Host memory and `darray` is the corresponding GPU/Device pointer. The added 'd' superscript always indicates that certain data/pointer is on the device. Either of the CPU or device data, pointed to by these pointers, can be owned (memory has to be freed when destroying the structure) or not, which is controlled by the `own_` and `down_` fields for the host and device, respectively. This allows us to run certain parts of the code on CPU or on the devices, or both, and subsequently add some run-time scheduling mechanism, if needed.

There are two ways that MAGMA kernels can be invoked in this backend. The first one is if the functionality needed is already implemented in MAGMA. Examples are the tensor contractions that we are developing, e.g., as described in Section 3.4. The second way is for kernels/computations that are not available. Often, these are not computationally intensive kernels, and therefore has not been need to port

them in high-performance numerical libraries, but still for completeness and in order to avoid data transfers between device and CPU is best to have them available on the device as well. For these cases we provide a MAGMA code generator that translates code from a MAGMA domain specific language (DSL) to CUDA. We have identified a number of templates/motifs to support and can add others as needed. We keep this port for now with a code generator, vs. including static code, in order to allow easily to modify and tune the templates later without changing the libCEED codes.

Here is an example of using the generator and the MAGMA DSL API. The following code is part of the CPU `CeedElemRestrictionApply` routine in libCEED.

```

1  const CeedScalar *uu;
2  CeedScalar *vv;
3  ierr = CeedVectorGetArrayRead(u, CEED_MEM_HOST, &uu); CeedChk(ierr);
4  ierr = CeedVectorGetArray(v, CEED_MEM_HOST, &vv); CeedChk(ierr);
5  ...
6  for (CeedInt e = 0; e < nelelem; e++)
7    for (CeedInt d = 0; d < ncomp; d++)
8      for (CeedInt i=0; i < elemsize; i++)
9        vv[i + elemsize*(d+ncomp*e)] = uu[indices[i+elemsize*e]+ndof*d];
10 ...
11 for (CeedInt e = 0; e < nelelem; e++)
12   for (CeedInt d = 0; d < ncomp; d++)
13     for (CeedInt i=0; i < elemsize; i++)
14       vv[indices[i + elemsize*e]+ndof*d] += uu[i + elemsize*(d+e*ncomp)];

```

Here the `uu` and `vv` data pointers are obtained from the `u` and `v` `CeedVectors`, respectively, and subsequently `vv` is modified on the CPU. The corresponding device code, implemented using MAGMA's DSL, is given as follows:

```

1  const CeedScalar *uu;
2  CeedScalar *vv;
3  ierr = CeedVectorGetArrayRead(u, CEED_MEM_DEVICE, &uu); CeedChk(ierr);
4  ierr = CeedVectorGetArray(v, CEED_MEM_DEVICE, &vv); CeedChk(ierr);
5  ...
6  magma_template<<e=0:nelem, d=0:ncomp, i=0:elemsize>>
7    (const CeedScalar *uu, CeedScalar *vv, CeedInt *dindices, int ndof) {
8      vv[i + iend*(d+dend*e)] = uu[dindices[i+iend*e]+ndof*d];
9    }
10 ...
11 magma_template<<e=0:nelem, d=0:ncomp, i=0:elemsize>>
12    (const CeedScalar *uu, CeedScalar *vv, CeedInt *dindices, CeedInt ndof) {
13      magmblas_datomic_add( &vv[dindices[i+iend*e]+ndof*d], uu[i+iend*(d+e*dend)]);
14    }

```

Currently, the tensor contractions needed are called by sending the data needed to the device, performing the computations using MAGMA, and sending back the results. This is just a functionality port for now to verify and pass all error/correctness checks the tests. Performance will be targeted next, when we minimize the data transfers and add further optimizations.

2.4 Interoperability

In addition to improving each of the software components in CEED, the project has also been working on improving the interoperability between them, with libCEED, the common API we are developing, a prime example. There are, however, many other synergistic connections that we have developed inside CEED, as presented in Figure 3. The rest of this section is devoted on providing a short description of one of these connections: between PUMI and MFEM.

2.4.1 Parallel conforming mesh adaptation

The Parallel Unstructured Mesh Infrastructure (PUMI) [11, 5, 4] and mesh adaptation component MeshAdapt [8, 9] have been integrated into the MFEM to support the conforming adaptation of high-order curved meshes. PUMI, provides management of distributed partitioned meshes by providing a full range of services from supporting mesh entity adjacencies, inter-part communications, read only copies, mesh migration, mesh entity creation and deletion, relating field information to mesh entities and maintaining the association of mesh

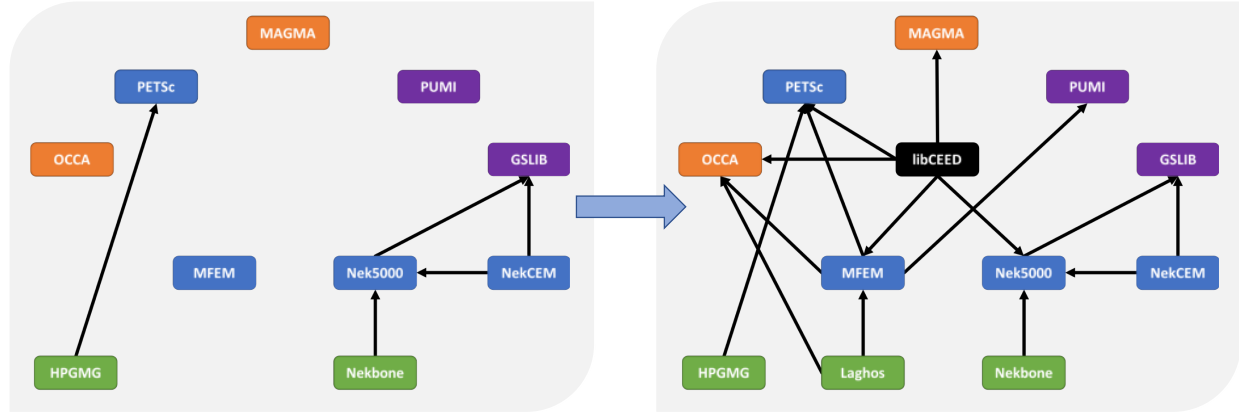


Figure 3: Connections between the CEED software components created during the project (before: left, after: right). Discretization libraries are in blue, miniapps in green, performance libraries in orange, and discretization tools in purple. Note the central role of libCEED.

entities to the original domain topological entities. PUMI is capable of handling general non-manifold models and effectively supporting automated adaptive analysis with a full range of operations on massively parallel computers. It consists of five libraries responsible for, phased message passing and thread management, geometric model interface, unstructured mesh representation, mesh partitioning and field management.

PUMI’s MeshAdapt module provides high order conforming unstructured curved mesh adaptation on general 3D domains. The current starting point for an adaptive simulation can be a linear or quadratic geometry mesh. In this case the geometric representation of the mesh is increased to meet the required geometric accuracy as dictated by the order of finite element basis functions (currently mesh geometry up to order 6 is supported). After the order elevation, load balancing [12] is performed to maintain balanced distribution of computation among processes. The finite element solver is called, and the solution field is obtained. To adaptively improve solution accuracy PUMI APIs are used to extract the field information needed to perform error estimation and correction indication. The output of the correction process is mesh size field which can be either an isotropic size field or an anisotropic mesh metric field. The mesh adaptation process employs local curved mesh modification operators that check for unsatisfactory element shape quality, and if there is any, improve the shape quality using proper local mesh modification operation such as edge/face swap, and/or curved element reshape operations [8, 9]. The adapted mesh is dynamically load balanced and the next analysis step is executed.

Currently the PUMI/MFEM integration in CEED supports conforming adaptation of unstructured tetrahedral meshes. PUMI APIs are used to load the parallel mesh along with the model that can be either a CAD model or a mesh model. If a CAD model, native or Parasolid, is provided, then the full range of mesh geometric interrogations is available through the PUMI APIs.

In our MFEM implementation the desired geometric order is indicated by the `-go` argument at the command line and the curvature is increased using a Bezier representation of the model (see Figure 4). Afterwards the mesh is loaded into the MFEM parallel mesh data structures. Examples of this process with explanation are provided in MFEM examples under the `pumi` subdirectory.

As the PUMI mesh is classified [2] on the CAD model, attributes can be directly assigned to the topological entities of the geometric model. Once the mesh is loaded, classification is used to modify the attribute of each boundary element. The same applies after each mesh modification process as the boundary conditions are assigned to the CAD model and are not affected by the mesh. Figure 5 shows an example of the mesh adaptation procedure applied to a pillbox model of an accelerator cavity. The model is considered as a multi-material cantilever beam and is solved for linear elasticity equation. A vertical Loading is applied to the middle section that has the nominal stiffness of $K=10$ while the lower and upper section have the stiffness of 100 and 1, respectively. A uniform first order tetrahedral mesh is used as the input. After loading, the mesh geometric approximation is increased to second order and then the boundary conditions are assigned

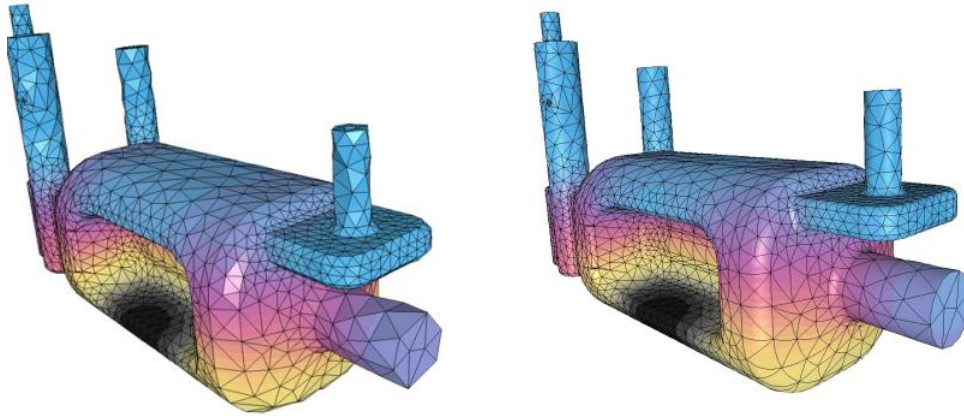


Figure 4: An example of order elevation on complex geometry: linear element (left) and quadratic element (right)

using the classification, finally the elasticity problem is solved. Once the solution is obtained, SPR error estimation is performed to provide the size field for each element and MeshAdapt is called to perform the curved mesh adaptation.

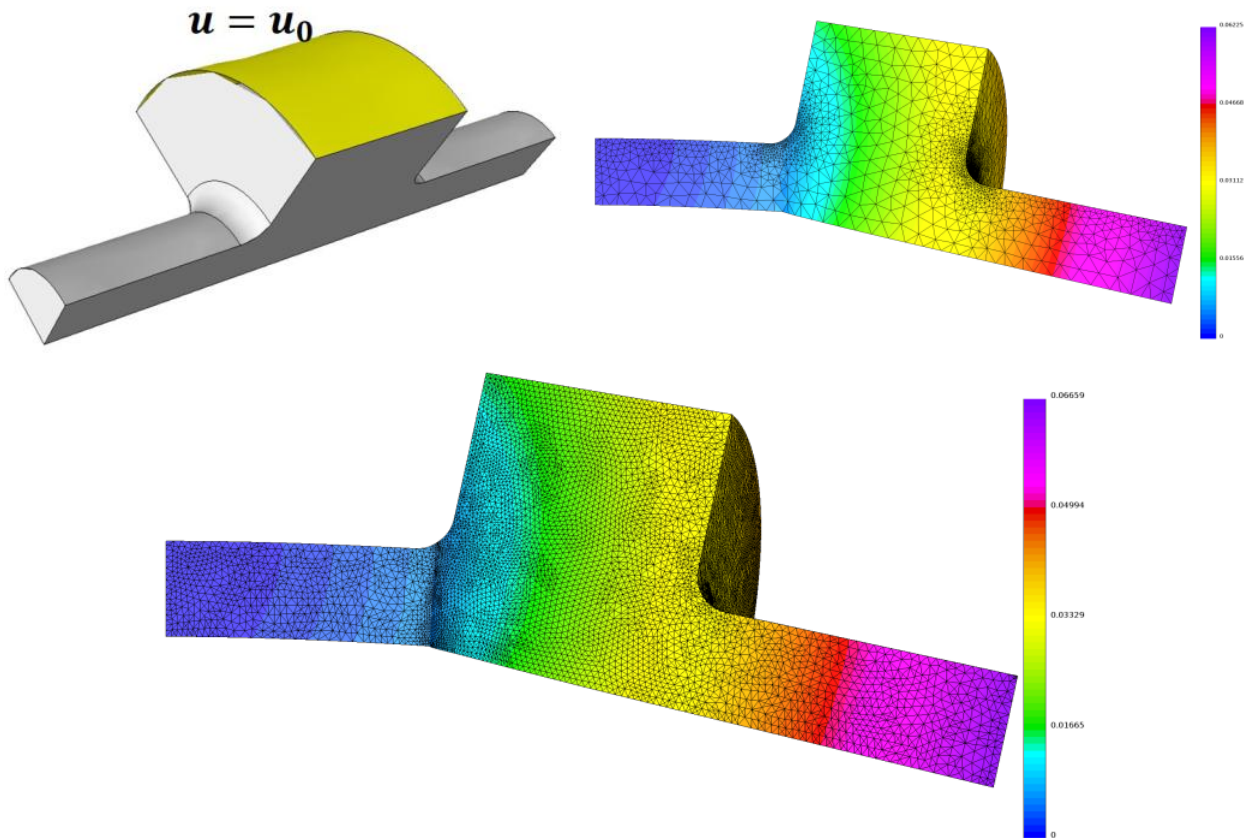


Figure 5: Curved mesh adaptation. 3D CAD model of the pillbox (top) with the loaded surface colored, side view of the initial mesh (bottom left) and final adapted mesh after deformation (bottom right).

3. CEED KERNELS AND BENCHMARKS

3.1 Bake-off kernels (BKs)

In addition to the CEED Bake-off Problems (BPs) introduced in CEED-MS6 and described on our website, <http://ceed.exascaleproject.org/bps>. The CEED team has also introduced companion *bake-off kernels* that share the BP numbering but work on element (**E-vector**) level, i.e. they focus on the dense linear algebra kernels, ignoring the parallel and element scatter/gather (the actions of P and G and their transposes in the BP notation). A short summary of our benchmark and kernels is as follows:

Bake-off Problems (BPs)

- BP1: scalar PCG with mass matrix, $q = p + 2$
- BP2: vector PCG with mass matrix, $q = p + 2$
- BP3: scalar PCG with stiffness matrix, $q = p + 2$
- BP3.5: scalar PCG with stiffness matrix, $q = p + 1$
- BP4: vector PCG with stiffness matrix, $q = p + 2$

These are all **T-vector-to-T-vector** and consist of parallel scatter + element scatter + element evaluation kernel + element gather + parallel gather. The case $q = p + 1$ correspond to using the same quadrature points as the degrees of freedom.

Bake-off Kernels (BKs)

- BK1: scalar **E-vector-to-E-vector** evaluation of mass matrix, $q = p + 2$
- BK2: vector **E-vector-to-E-vector** evaluation of mass matrix, $q = p + 2$
- BK3: scalar **E-vector-to-E-vector** evaluation of stiffness matrix, $q = p + 2$
- BK3.5: scalar **E-vector-to-E-vector** evaluation of stiffness matrix, $q = p + 1$
- BK4: vector **E-vector-to-E-vector** evaluation of stiffness matrix, $q = p + 2$

The BKs are parallel to the BPs, except they do not include parallel and element scatter/gather (the actions of P and G and their transposes).

3.2 New fused GPU kernels using register shuffling

3.2.1 Overview

In this section we describe new BK and BP algorithms that take advantage of the `__shfl_sync()` intrinsic to avoid using shared memory for the computation. Using `__shfl_sync()` requires algorithms to be designed at the warp level. This constraint results in a maximum block size of 32 threads on the NVidia GPUs considered here. Therefore, no more than 32 threads can be used to compute the tensor contractions for a single element. Ideally, we want to use as close as possible to 32 threads to use all available computational power. Each thread is also limited by a maximum of 255 registers on the current GPU architecture. Using the smallest number of registers can also improve GPU utilization, and thereby also increase performance. The following paragraphs describe the different approaches explored so far for performing the tensor contraction on an input tensor (degrees of freedom) of size N^3 .

3Dreg: The simplest approach is to use one thread to perform the entire fused tensor contraction for one element. Since no communication occurs between threads, this approach does not require `__shfl_sync()`. This is highly efficient, but is also highly limited by the maximum number of registers per thread. For even moderate orders, the algorithm requires more registers than are available per thread and spills to local cache. This algorithm becomes inefficient as soon as it spills.

2Dreg 1Dshfl: To overcome the limitation of the previous algorithm, this approach uses N threads per element storing a plane of N^2 values of the input tensor. The contractions inside each plane do not require communication between threads, and `__shfl_sync()` is used for the remaining contractions between planes. Theoretically up to $N = 32$ planes should be supported, but the register limitation occurs for substantially smaller N . Since only N threads work on an element, the reads and writes cannot be coalesced unless the input arrays are interleaved. It is not currently assumed that the input data can be easily stored in this format, so the performance numbers below will be slightly lower due to this effect.

1Dreg 2Dshfl: The register limitation of the 2Dreg 1Dshfl algorithm can be overcome by using N^2 threads, each storing one column of N values of the input tensor. Therefore, `__shfl_sync()` is now used to perform contractions inside planes of threads. This results in more `__shfl_sync()` contractions than the previous algorithm. Unfortunately, this algorithm is quickly limited by the warp size (32 threads). $N = 6$ requires 36 threads communicating, thus this approach is limited to $N < 6$. Despite this drawback, there are numerous benefits: the algorithm can be extended to BP3 because the number of registers is low.

1Dreg 2Dshfl 2cols: Instead of storing only one column per thread, this approach uses $N^2/2$ threads, each storing two columns of N values. Thus, it achieves a good balance between thread and register usage, in fact up to $N = 8$ fits into the registers available on the GPU. This approach also works for BP3 up to $N = 8$.

3.2.2 Preliminary results

We show here preliminary results obtained with a NVidia Pascal P100 GPU. All of our different algorithms – written directly in CUDA – are compared to the existing unoptimized baseline GPU implementation in MFEM. This baseline kernel happens to be implemented with OCCA, but is not highly optimized and not representative of OCCA performance in general.

Figures 7 and 6 show GFlops and Bandwidth performance obtained for BK1 and BK0.5 – a version of BK1 with $q = p + 1$ respectively. We computed the maximum GFlops performance by considering the Flops/Byte achievable at maximum bandwidth. Since these algorithms are bandwidth limited, we see that the bandwidth usage directly drive the GFlops performance. The kernels achieve maximum performance when data can be perfectly read and written. Thus, reading and writing in a so-called coalesced way is a major limiting factor for these algorithms. Because of this, we see that the performance of BK1 is not as good as the performance of BK0.5. Indeed, for certain orders the performance is impacted by the fact that not all threads of a warp are reading and writing. This effect is amplified when degrees of freedom and quadrature points are of different sizes, as is the case in BK1. For this reason, we are exploring different algorithms to read and write perfectly independently of the number of degrees of freedom and quadrature points.

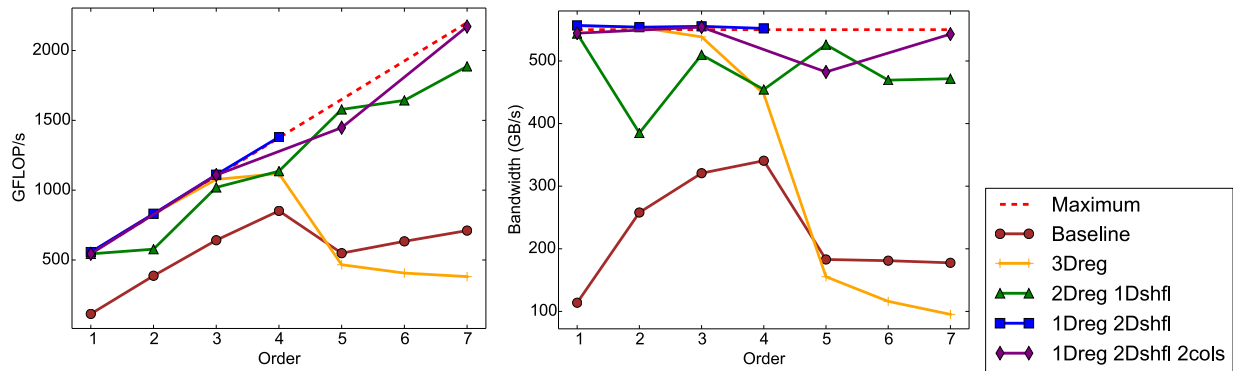


Figure 6: Performance on Pascal P100 for BK 0.5

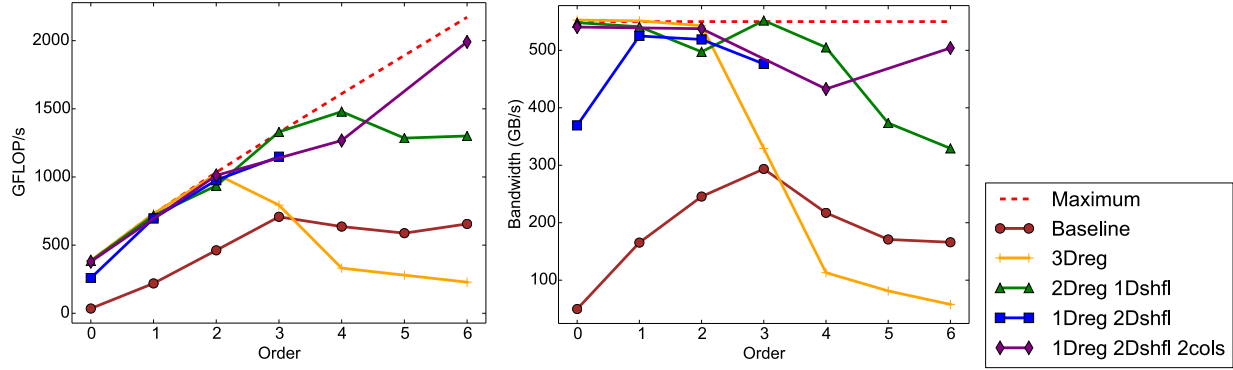


Figure 7: Performance on Pascal P100 for BK 1.0

Table 1 shows a comparison of the performance of BP1 using 1Dreg 2Dshfl 2cols compared to the unoptimized baseline implementation. These tables show the percentage of time spent in the most critical parts of BP1. The scatter and gather algorithms are common for the baseline and optimized algorithms. The Mult percentage corresponds to the time spent in the element level batched tensor contractions.

We see that even though the speedup obtained by our algorithm is significant over the baseline algorithm, the overall speedup is limited due to the scatter and gather. This shows that fusing the scattering and the gathering of degrees of freedom with the fused tensor contraction kernels will be critical.

Order	Method	Scatter (%)	Mult (%)	Gather (%)	MDOF/S
2	baseline	44.49	24.8	17.15	447.85
	optimized	52.16	11.68	20.14	511.07
4	baseline	21.20	47.72	10.21	715.97
	optimized	33.59	16.46	16.14	1010.88
6	baseline	14.83	48.44	8.73	812.133
	optimized	22.83	17.78	13.43	1001.57

Order	Kernel Speedup	Overall Speedup
2	$\times 2.5$	$\times 1.14$
4	$\times 4.6$	$\times 1.41$
6	$\times 4.2$	$\times 1.23$

Table 1: Performance for BP 1.0 using 1Dreg 2Dshfl 2cols vs baseline OCCA implementation

3.3 Optimizing finite element stiffness operations on the NVIDIA V100 SXM2 GPU

The CEED team at Virginia Tech (VT) performed a detailed analysis of the NVIDIA V100 16GB SXM2 GPU (referred to V100 henceforth). Understanding the performance characteristics of this device was of paramount performance in optimizing finite element stiffness matrix action kernels. In particular determining performance limiters related to accessing global device memory, shared memory, as well as the nature of the V100 shared memory and L1 cache was critical in guiding the choice of kernel optimizations. In Section 3.3.1 we describe steps taken to benchmark and model the performance of the V100.

In Section 3.3.2 we benchmark the performance of the hand tuned bake-off kernels on the V100. In Section 3.3.3 we describe a sequence of progressively optimized finite element kernels developed for evaluating the action of the stiffness matrix for affine geometry tetrahedral elements with Lagrange Warp & Blend basis up to degree 8. The performance of these kernels was then compared against the performance of a new CUDA &

cuBLAS based implementation discussed in Section 3.3.3. Both of these implementations will be released during the next project period.

Finally, analysis of the compute kernels highlight how heavily the block dense elemental operator matrices are compute bound so going forward we are investigating alternative polynomial basis choices that induces sparse elemental stiffness matrices.

3.3.1 Micro-benchmarking the NVIDIA V100 SXM2 GPU

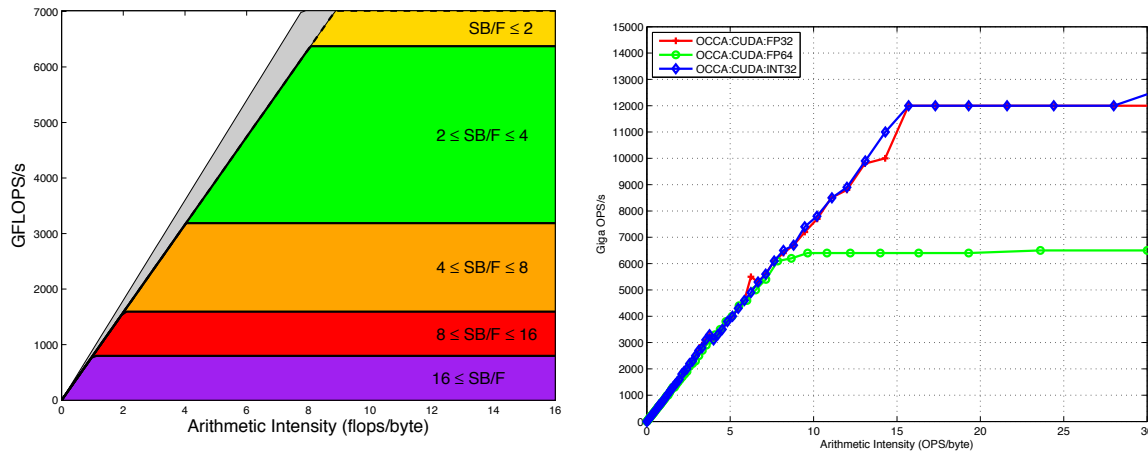


Figure 8: Left: Roofline model for the NVIDIA V100 SXM2 16GB Volta class GPU. Each shaded region corresponds to the target band for a kernel with a given number of shared+L1 bytes accessed per flop. Right: occaBench benchmark performance results for mixed streaming and compute on NVIDIA V100 SXM2 16GB Volta class GPU.

Starting from public performance data, the card has:

- Theoretical device memory bandwidth of 900GB/s. Using `cudaMemcpy` we measure achievable memory bandwidth of 790GB/s.
- Combined shared memory & L1 cache with which we guesstimate to have throughput: $(SH + L1) \text{ GB/s} = 80 \text{ (cores)} \times 32 \text{ (SIMD width)} \times 4 \text{ (word bytes)} \times 1.245 \text{ (base clock)} \approx 12.748 \text{ TB/s}$
- Theoretical peak flops of 7TFLOPS/s (FP64).

Combining the V100 performance characteristics we visualize the FP64 performance as a graded roofline model in Figure 8(top). Interpreting this diagram: each band shows the maximum performance we can expect for a given range of combined shared & L1 memory accesses where we have assumed the same memory bus is used for both caches. We note that a kernel may drop below the performance band suggested by a naive estimate of arithmetic intensity and local cache accesses if there is one or more additional performance limiter. For instance: use of special function units, extremely low occupancy caused by excess registers or shared memory usage, or spilling to local (i.e. L1) memory may impact performance.

The VT team added the mixbench mixed workload microbenchmarking code [6, 7] to the occaBench portable benchmarking suite. Initially, the default settings for occaBench gave erroneously high throughput estimates on the V100, likely due to a different cache configuration than earlier GPUs. We modified the workload for the benchmarking kernel and were able to obtain performance that did not exceed the manufacturer spec. The occaBench code running on the V100 in OCCA:CUDA mode on a vector of length 10,240,000 achieves the performance shown in Figure 8(bottom). It is notable that the peak measured FP64 performance of the V100 is very close to the manufacturer spec. To achieve near peak requires a kernel with arithmetic intensity in excess of eight flops per byte of data accessed in global device memory.

3.3.2 Revisiting the VT CEED hexahedral bake-off kernels on the NVIDIA V100

To warm up on the V100 we benchmarked the BK1, BK3, and BK3 collocation kernels using a mesh of 4096 hexahedral elements. The performance results in Figure 9 show that:

BK1: achieves close to streaming bandwidth up to degree 9 on the V100.

BK3: (GL integration) performance is limited by shared + L1 memory accesses and by the requirement to stream seven geometric factors per Gauss-Legendre quadrature node.

BK3: (GLL integration) performance is limited by capacity of the global device memory to stream seven geometric factors per Gauss-Lobatto-Legendre quadrature node.

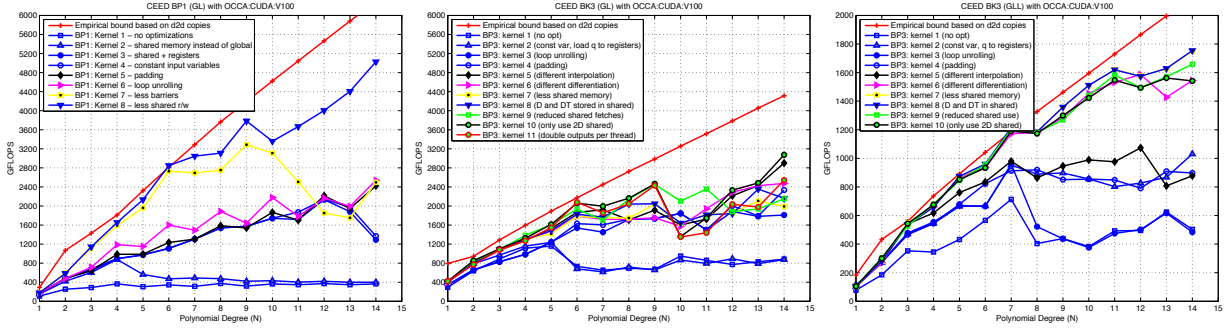


Figure 9: VT experiments on an NVIDIA V100 SXM2 GPU with 4096 hexes. Left: BK1 with $(N + 2)^3$ intermediate Gauss Legendre quadrature nodes. Middle: BK3 with $(N + 2)^3$ intermediate Gauss Legendre quadrature nodes. Right: BK3 with $(N + 1)^3$ Gauss-Lobatto-Legendre quadrature nodes.

Overall the performance of the VT BK kernels on the V100 is comparable to the performance on the NVIDIA P100 with performance gains proportional to improvements in the global device memory bandwidth and the increased shared memory throughput. No additional kernel optimizations were performed in this study, i.e. the kernels were tuned for the P100 architecture and yet still delivered strong performance on the V100.

3.3.3 Optimizing finite element stiffness action kernels for affine tetrahedral elements

The target of this work is the optimization of GPU kernels for the BK3 benchmark kernel problem posed on a mesh of tetrahedral finite elements that are affine images of a reference element. We took two separate approaches:

- V1: We constructed a sequence of progressively optimized elliptic operator action kernels for nodal tetrahedra using the Open Concurrent Compute Abstraction (OCCA) in the OCCA Kernel Language (OKL). The performance of these kernels was compared to a roofline model that neglects the movement of reference element matrices.
- V2: We repeated V1 but using a CUDA kernel based implementation combined with calls to the cuBLAS library. The elemental stiffness matrices are block dense with $\mathcal{O}(N^6)$ entries thus the operations are compute bound, and it was reasonable to compare the custom built kernels of V1 with the performance of generic cuBLAS kernels.

V1: nodal affine tetrahedra implemented using OCCA/OKL. First we developed a sequence of progressively optimized monolithic BK3 kernels that compute the expression

$$r_n^e = \sum_{m=0}^{m=N_p-1} S_{nm}^e q_m^e,$$

where each elemental stiffness matrix is given by

$$S_{nm}^e := \int_{D^e} \nabla l_n \cdot \nabla l_m + \lambda l_n l_m,$$

where l_n is the Lagrange interpolant function associated with node n of the nodes in a degree N Warp & Blend family of interpolation nodes for the tetrahedra [13]. Here D^e refers to element e . The stiffness matrix is constructed on the fly via the affine map assumption

$$\begin{aligned} S^e &= G_{rr}^e S^{rr} + G_{rs}^e (S^{rs} + S^{sr}) + G_{rt}^e (S^{rt} + S^{tr}) + G_{ss}^e S^{ss} + G_{st}^e (S^{st} + S^{ts}) + G_{tt}^e S^{tt} + J^e M, \\ &= G_{rr}^e S^{rr} + G_{rs}^e \hat{S}^{rs} + G_{rt}^e \hat{S}^{rt} + G_{ss}^e S^{ss} + G_{st}^e \hat{S}^{st} + G_{tt}^e S^{tt} + J^e M, \end{aligned}$$

where the geometric factors are entries from the symmetric Jacobian matrix

$$G_{rr}^e, G_{rs}^e, G_{rt}^e, G_{ss}^e, G_{st}^e, G_{tt}^e.$$

The six reference stiffness matrices and one reference mass matrix have entries given by

$$\begin{aligned} S_{nm}^{rr} &:= \int_{\hat{D}} \frac{\partial l_n}{\partial r} \frac{\partial l_m}{\partial r}, S_{nm}^{ss} := \int_{\hat{D}} \frac{\partial l_n}{\partial s} \frac{\partial l_m}{\partial s}, S_{nm}^{tt} := \int_{\hat{D}} \frac{\partial l_n}{\partial t} \frac{\partial l_m}{\partial t}, M := \int_{\hat{D}} l_n l_m, \\ \hat{S}_{nm}^{rs} &:= \int_{\hat{D}} \frac{\partial l_n}{\partial r} \frac{\partial l_m}{\partial s} + \frac{\partial l_n}{\partial s} \frac{\partial l_m}{\partial r}, \hat{S}_{nm}^{rt} := \int_{\hat{D}} \frac{\partial l_n}{\partial r} \frac{\partial l_m}{\partial t} + \frac{\partial l_n}{\partial t} \frac{\partial l_m}{\partial r}, \hat{S}_{nm}^{st} := \int_{\hat{D}} \frac{\partial l_n}{\partial s} \frac{\partial l_m}{\partial t} + \frac{\partial l_n}{\partial t} \frac{\partial l_m}{\partial s}, \end{aligned}$$

where \hat{D} is the reference bi-unit tetrahedron with local coordinate system $-1 \leq r, s, t; r + s + t \leq 1$.

With this notation in hand we see that the action of the stiffness matrix consists of seven matrix-vector multiplications per element. We started with an initial kernel implementation and then constructed ten progressively optimized kernels from the base kernel with optimizations rationalized as follows:

- K0: Baseline kernel with minimal optimizations, while avoiding most pitfalls like poor memory layout.
- K1: Annotate pointers as `const` where appropriate to help the compiler perform safe optimizations where possible.
- K2: Unroll innermost serial loops.
- K3: Reduce number of reads of q^e from global device memory.
- K4: Prefetch q^e to shared memory. This is less important on Volta class GPUs because of the unified shared and L1 caches on each core.
- K5: Prefetch the seven geometric factors to shared memory. Again this is less important on Volta class GPUs.
- K6: We reduce from ten matrix-vector multiplies to seven matrix multiplies (and we read only seven matrices). This potentially reduces the kernel execution time by 30%.
- K7: Each thread processes node n for `p_Ne` elements. The goal is to reuse entries from the references stiffness matrices and reduce traffic to the L1 (and possibly L2) cache.
- K8: Each thread-block uses `p_Nb` times `p_Np` threads. The goal is to better align the number of threads per thread-block with the number of SIMD lanes on a GPU core. The `p_Nb` parameter was co-optimized with the `p_Ne` parameter.
- K9: Replace the use of a list of elements to process with a single integer offset into a list of elements that is pre-sorted.
- K10: Revert to not pre-fetching the geometric factors into shared memory in response to profiling information.

Each optimization is applied in addition to prior optimizations, with the exception of kernel K10 where an unsuccessful optimization was removed. The optimizations for kernels K7 & K8 were guided by assumptions we made about the underlying Volta streaming multiprocessor architecture. In Figure 8 we show a graded roofline model where the color coded regions indicate the maximum expected performance based on the average number of FP64 floating point operations performed per byte accessed from the unified L1+shared memory cache. It is notable that to achieve optimal FP64 throughput on the V100 GPU it is necessary to access less than two bytes from shared/L1 cache on average per floating point operation. This is particularly challenging to achieve. More details about the optimization process can be found in this [blog entry](#). For reference we include kernel K10 at the end of this document in Appendix 3.3.4.

In Figure 10 we show performance results for K0:10 on a V100. The left chart shows that

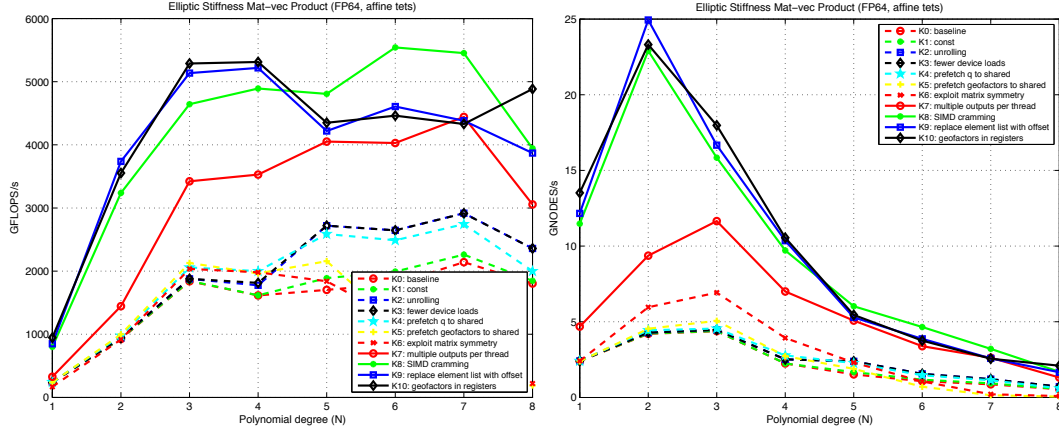


Figure 10: Performance results for the BK3 bake-off kernel on a mesh of 320,000 affine tetrahedral elements on Volta Titan V GPU. Left: FP64 floating point performance for kernels K0:10 for polynomial degrees 1:8. Right: FP64 throughput measured in giga-nodes per second.

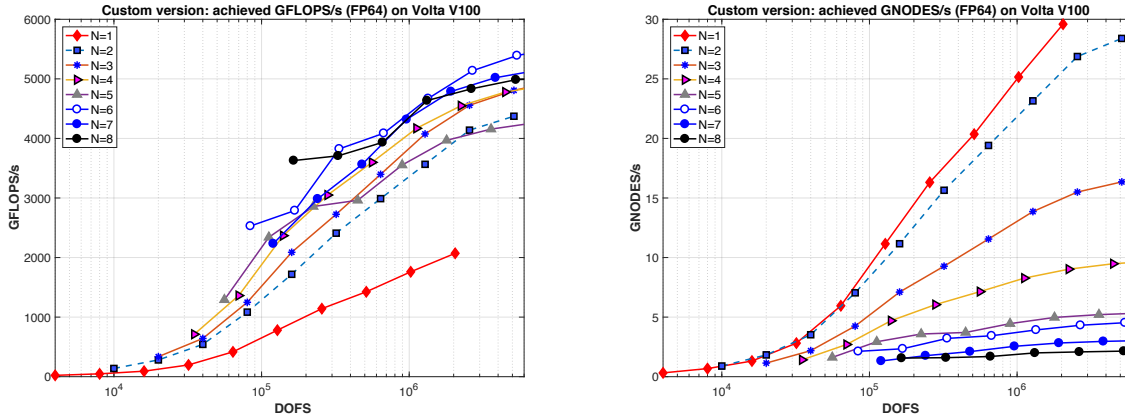


Figure 11: Left: FP64 GFLOPS/s performance of the optimal kernel from K0:10 for each polynomial degree on a V100. Right: throughput measured in billion nodes per second (GNODES/s) on a single V100.

- The kernels achieve up to a peak of approximately 5.5 TFLOPS/s which is a little shy of the 6.5 TFLOPS/s peak performance measured with the occaBench benchmark (see Figure 8).
- There is a large spread between the lowest performing and best performing kernel at each order.
- K8:10 are the best performing kernels. These kernels require input parameters p_Ne and p_Nb . These are calibrated through an exhaustive computational study. The results shown in the figure were obtained using the best parameters.
- The K5 optimization was performance negative, rectified by the K10 optimization.

The right chart of Figure 10 shows that although the best kernels are well tuned the overall throughput strongly drops as polynomial degree increases. This is expected because the arithmetic intensity increase is cubic with N and the higher order kernels are strongly compute bound.

In Figure 11 we explore the dependence of the best kernel at each polynomial order on the number of tetrahedral elements. It is clear that in general it takes in excess of $\mathcal{O}(10^5)$ FEM elemental degrees of freedom to hit the asymptotic best performance.

V2: CUDA+cuBLAS Implementation for Nodal Tetrahedra. After the first phase of this effort we decided to offload the arithmetically intense matrix-multiplication to cuBLAS. We constructed a CUDA implementation that first calls the cuBLAS version of the BLAS double precision general matrix-matrix multiplication subroutine (`dgemm`) to compute the eight matrix products all-at-once by stacking the mass matrix and six reference stiffness matrices into a single matrix `d_Dcombined`, i.e.,

```

1 void gpuBlasGemm(cublasHandle_t &handle, const dfloat *A, const dfloat *B, dfloat *C, const
   int m, const int k, const int n) {
2
3     const int lda=m,ldb=k,ldc=m;
4     const dfloat alf = 1, bet = 0;
5     const dfloat *alpha = &alf, *beta = &bet;
6
7     // Do the actual multiplication
8     if(sizeof(dfloat)==8)
9         cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, (double*)alpha, (double*)A, lda,
10            (double*)B, ldb, (double*)beta, (double*)C, ldc);
11     else
12         cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, (float*)alpha, (float*)A, lda, (
13            float*)B, ldb, (float*)beta, (float*)C, ldc);
14 }
15 ...
16
17 gpuBlasGemm(handle, d_Dcombined, d_q, d_Dq, 7*p_Np, E, p_Np, 0);

```

The above code places the result in the array `d_Dq` with `Nggeo*Np*E` entries, where `Nggeo` is the number of geometric factors, `Np` is the number of nodes per element and `E` is the number of elements in mesh. This array is further reduced to an array with `Np*E` entries using a CUDA kernel listed below.

```

1 __global__ void geofactorsKernelv1(const int Nelements, const dfloat * __restrict__ ggeo,
   const dfloat * __restrict__ q, const dfloat lambda, dfloat * __restrict__ Aq ){
2
3     const int p_Np = blockDim.x; // define this
4     const int e = blockIdx.x;
5     const int t = threadIdx.x;
6
7     const dfloat Grr = ggeo[e*p_Nggeo + p_G00ID];
8     const dfloat Grs = ggeo[e*p_Nggeo + p_G01ID];
9     const dfloat Grt = ggeo[e*p_Nggeo + p_G02ID];
10    const dfloat Gss = ggeo[e*p_Nggeo + p_G11ID];
11    const dfloat Gst = ggeo[e*p_Nggeo + p_G12ID];
12    const dfloat Gtt = ggeo[e*p_Nggeo + p_G22ID];
13    const dfloat J = ggeo[e*p_Nggeo + p_GWJID];
14    const int base = e*p_Nggeo*p_Np + t;
15
16    Aq[t+p_Np*e] =
17        Grr*q[base + 0*p_Np] + Grs*q[base + 1*p_Np] + Grt*q[base + 2*p_Np] + Gss*q[base + 3*p_Np]
18        + Gst*q[base + 4*p_Np] + Gtt*q[base + 5*p_Np] + J*lambda*q[base + 6*p_Np];
19 }

```

We note that in this approach we move more data compared to V1, as we need to both, allocate memory for `d_Dcombined` and for `d_Dq`. We do not use these data structures in V1. In addition, the execution consists of two stages and requires two kernel launches unlike the V1 implementation where we launch only one kernel.

In Figure 12 we show throughput performance. Comparing with the performance of the V1 kernels in Figure 11 several things are apparent:

- The extra streaming steps for the intermediate results produced by `cublasDgemm` and consumed in `geofactorsKernelv1` reduces the $N = 1$ throughput from 30 GNODES/s to 2 GNODES/s. At this low degree the computation is bandwidth limited and the performance ratio is driven by the need to write and then read 14 extra sets of node data.
- The cuBLAS version achieves similar performance at the highest order.

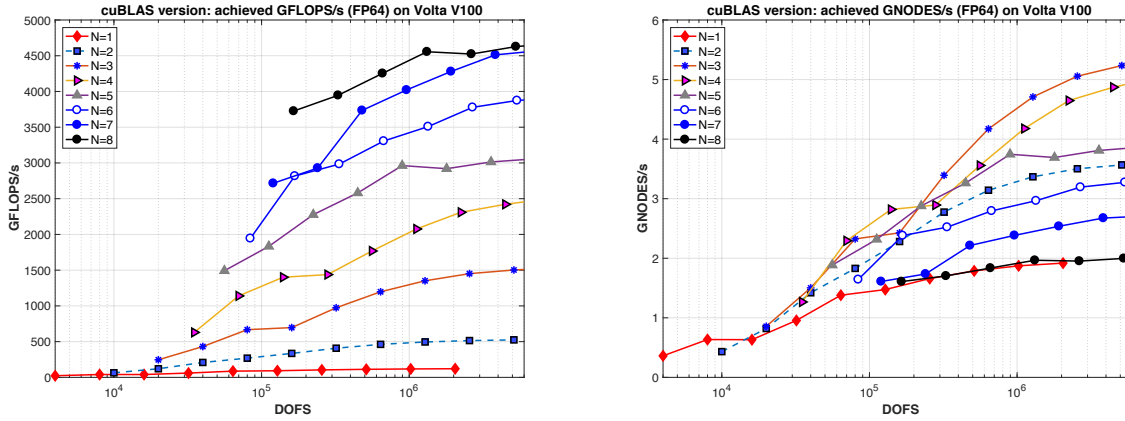


Figure 12: Left: FP64 GFLOPS/s performance of the cuBLAS implementation on a V100. Right: throughput measured in billion nodes per second (GNODES/s) on V100.

Although the cuBLAS implementation exercise was 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 OKL kernels deliver strong performance at all orders in contrast to cuBLAS which underperformed until degree $N = 7$.

3.3.4 Appendix: Code listing for *ellipticAxTet3DK10.okl*

```

1 kernel void ellipticPartialAxTet3D_Ref10(const int Nelements,
2     const int elementOffset,
3     const datafloat * restrict ggeo,
4     const datafloat * restrict SrrT, const datafloat * restrict SrsT, const datafloat *
5     restrict SrtT,
6     const datafloat * restrict SsrT, const datafloat * restrict SssT, const datafloat *
7     restrict SstT,
8     const datafloat * restrict StrT, const datafloat * restrict StsT, const datafloat *
9     restrict SttT,
10    const datafloat * restrict MM,
11    const datafloat lambda, const datafloat * restrict q, datafloat * restrict Aq){
12
13    for(int eo=0;eo<Nelements;eo+=p_Ne*p_Nb;outer0){
14
15        shared datafloat s_q[p_Ne][p_Nb][p_Np];
16
17        for(int b=0;b<p_Nb;++b;inner1){
18            for(int n=0;n<p_Np;++n;inner0){
19
20                occaUnroll(p_Ne)
21                for(int et=0;et<p_Ne;++et){
22                    const int ebase = eo + b + p_Nb*et;
23                    const int e = elementOffset + ebase;
24
25                    if(ebase<Nelements){
26                        const int id = n + e*p_Np;
27                        s_q[et][b][n] = q[id];
28                    }
29                }
30            }
31        }
32
33        barrier(localMemFence);
34    }
35}

```

```

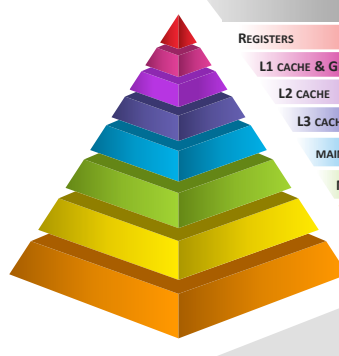
32   for(int b=0;b<p_Nb;++b;inner1){
33       for(int n=0;n<p_Np;++n;inner0){
34
35           datafloat qrr[p_Ne], qrs[p_Ne], qrt[p_Ne], qss[p_Ne], qst[p_Ne], qtt[p_Ne], qM[p_Ne
36   ];
37
38           occaUnroll(p_Ne)
39               for(int et=0;et<p_Ne;++et){
40               qrr[et] = 0; qrs[et] = 0; qrt[et] = 0; qss[et] = 0; qst[et] = 0; qtt[et] = 0;
41               qM[et] = 0;
42           }
43
44           // overall this does p_Ne*14 flops for (7+p_Ne)*|datafloat| L1+shared accesse
45           // arithmetic intensity is (p_Ne*14/((7+p_Ne)*8)) flops per byte
46           occaUnroll(p_Np)
47               for (int k=0;k<p_Np;k++) {
48
49                   const datafloat Srr_nk = SrrT[n+k*p_Np], Srs_nk = SrsT[n+k*p_Np], Srt_nk = SrtT[
50   n+k*p_Np];
51                   const datafloat Sss_nk = SssT[n+k*p_Np], Sst_nk = SstT[n+k*p_Np], Stt_nk = SttT[
52   n+k*p_Np];
53                   const datafloat MM_nk = MM[n+k*p_Np];
54
55                   occaUnroll(p_Ne)
56                       for(int et=0;et<p_Ne;++et){
57
58                           const datafloat qk = s_q[et][b][k];
59                           qrr[et] += Srr_nk*qk; qrs[et] += Srs_nk*qk; qrt[et] += Srt_nk*qk;
60                           qss[et] += Sss_nk*qk; qst[et] += Sst_nk*qk; qtt[et] += Stt_nk*qk;
61                           qM[et] += MM_nk*qk;
62                       }
63
64                   }
65
66           occaUnroll(p_Ne)
67               for(int et=0;et<p_Ne;++et){
68                   const int ebase = eo + b + p_Nb*et;
69                   const int e = elementOffset + ebase;
70                   if(ebase<Nelements){
71
72                       int gid = e*p_Nggeo;
73
74                       datafloat Grr = ggeo[gid + p_G00ID], Grs = ggeo[gid + p_G01ID], Grt = ggeo[gid
75   + p_G02ID], Gss = ggeo[gid + p_G11ID];
76                       datafloat Gst = ggeo[gid + p_G12ID], Gtt = ggeo[gid + p_G22ID], J = ggeo[gid
77   + p_GWJID];
78
79                       const int id = n + e*p_Np;
80                       Aq[id] = Grr*qrr[et] + Grs*qrs[et] + Grt*qrt[et] + Gss*qss[et] + Gst*qst[et] +
81   Gtt*qtt[et] + J*lambda*qM[et];
82                   }
83               }
84           }
85       }
86   }
87   }
88   }
89   }

```

3.4 MAGMA batched kernels

The direct use of generic BLAS in the CEED applications has performance shortcomings. As reported in this section above, one way to address these shortcomings is through custom-built OCCA kernels, which has been a success on targeted applications and cases. Another aspect on the work in CEED is to further extend and co-design these kernels with application developers, hardware vendors, and ECP software technologies projects and to make them available through general interfaces. In particular, our efforts here have been to overcome the above mentioned shortcomings of BLAS by extending it with Batched BLAS kernels. Subsequently, the CEED libraries will leverage the software sustainability and performance portability benefits that get associated with the use of the BLAS standard.

To this end, we organize the tensor contractions in CEED in terms of Batched BLAS calls, and in particular, batched general matrix-matrix multiplications (GEMMs), and concentrate on the development of these kernels for various architectures. The architectures targeted so far and some of their memory hierarchy characteristics are given in Figure 13. We developed a number of algorithms and techniques that allow us to have a single source for the cross-architecture support for highly efficient, small size matrix-matrix products [10]. The code is also very simple to extend. For example, adding support for IBM processors with AltiVec SIMD instructions only required us to add an overload for each SIMD functions we needed.



Memory hierarchies	Intel Haswell E5-2650 v3	Intel KNL 7250 DDR5 MCDRAM	IBM Power 8	ARM Cortex A57	Nvidia P100	Nvidia V100
	10 cores 368 Gflop/s 105 Watts	68 cores 2662 Gflop/s 215 Watts	10 cores 296 Gflop/s 190 Watts	4 cores 32 Gflop/s 7 Watts	56 SM 64 cores 4700 Gflop/s 250 Watts	80 SM 64 cores 7500 Gflop/s 300 Watts
REGISTERS	16/core AVX2	32/core AVX-512	32/core	32/core	256 KB/SM	256 KB/SM
L1 CACHE & GPU SHARED MEMORY	32 KB/core	32 KB/core	64 KB/core	32 KB/core	64 KB/SM	96 KB/SM
L2 CACHE	256 KB/core	1024 KB/2cores	512 KB/core	2 MB	4 MB	6 MB
L3 CACHE	25 MB	0...16 GB	8 MB/core	N/A	N/A	N/A
MAIN MEMORY	64 GB	384 16 GB	32 GB	4 GB	16 GB	16 GB
MAIN MEMORY BANDWIDTH	68 GB/s	115 421 GB/s	85 GB/s	26 GB/s	720 GB/s	900 GB/s
PCI EXPRESS GEN3 x16 NVLINK	16 GB/s	16 GB/s	16 GB/s	16 GB/s	16 GB/s	300 GB/s (NVL)
INTERCONNECT INFINIBAND EDR	12 GB/s	12 GB/s	12 GB/s	12 GB/s	12 GB/s	12 GB/s

Memory hierarchies for different type of architectures

Figure 13: Memory hierarchies of the experimental CPU and GPU hardware targeted for development and optimization of Batched BLAS in MAGMA.

Figure 14 summarizes our performance results on Batched DGEMMs of square matrices of small sizes on the P100 GPU (Left) vs. V100 GPU (Right). The two GPUs used are both PCIe versions, which are slightly slower from their NVLink counterparts that are in systems like the SummitDev at ORNL. The V100 for NVLink has double precision peak of 7.5 TeraFlops vs 7 TeraFlop for PCIe.

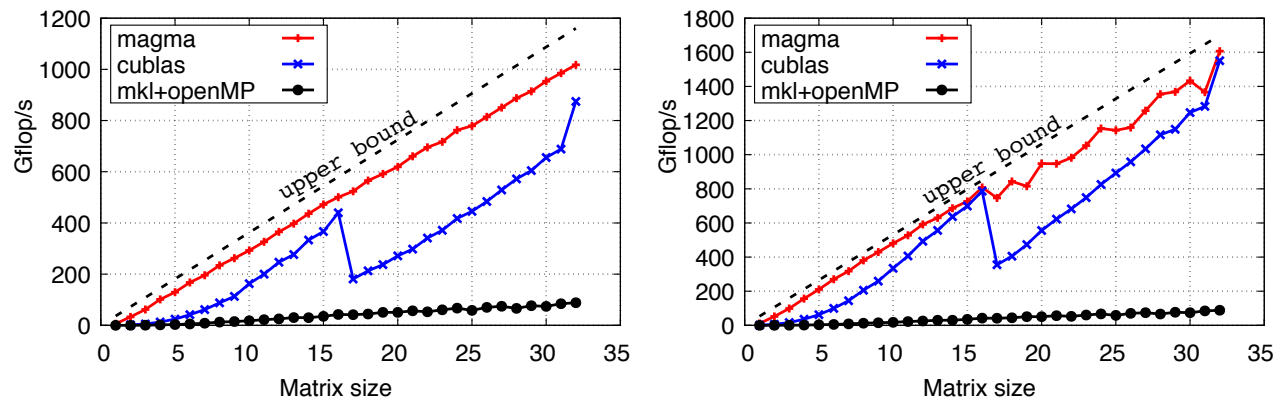


Figure 14: Batched DGEMM on P100 GPU (Left) and V100 GPU (Right).

The performance target is to get close to the theoretical peaks. In this case, as sizes are small, the kernels are memory bound and the peaks are given by the dashed lines of the roof-line models on Figure 14. The

dashed lines are computed based on achievable maximum bandwidth of 580 GB/s for the P100, and 850 GB/s for the V100 GPU. In either case we are close (within 90+%) to these peaks. The difference between the V100 and P100 is as expected about 50%, which is similar to the differences in their bandwidth and compute peak specifications.

While one can use directly these MAGMA batched BLAS kernels to accelerate the computations, the tensor contractions in CEED, if properly organized, can have higher computational intensity (computation vs. data needed) than that of a single GEMM. In particular, the application of a CEED differential operator can be expressed as a batch over the finite elements e , e.g., $\text{batch}\langle e=0..\text{nelems}\rangle\{ B_e^T D_e * (B_e A_e B_e^T) B_e \}$, where the matrices involved are small, dense, and depend on the particular element e . This computation has much larger computational intensity than the one that splits it into a number of batched GEMM calls, e.g.,

```

batch<e=0..nelems> {  $C_e = A_e B_e^T$  };
batch<e=0..nelems> {  $C_e = B_e C_e$  };
batch<e=0..nelems> {  $C_e = D_e * C_e$  };
batch<e=0..nelems> {  $C_e = C_e B_e$  };
batch<e=0..nelems> {  $C_e = B_e^T C_e$  };

```

Figure 15 compares the performances of fused vs. non-fused batched DGEMMs in MAGMA on small square matrices of sizes up to 16. In both cases the computation is for $B_e^T D_e * (B_e A_e B_e^T) B_e$.

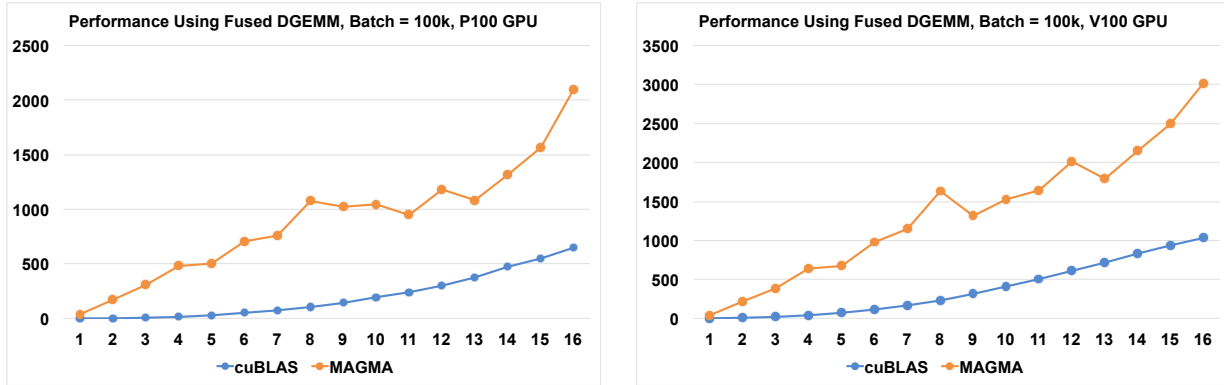


Figure 15: Performance comparison (in Gflop/s) of fused batched DGEMMs on P100 GPU (Left) vs. non-fused batched DGEMMs on V100 GPU (Right).

The effect on performance of the two approaches (fused GEMMs vs. non-fused) for realistic matrix sizes, taken from the MFEM code, is illustrated in Figure 16. Thus, although both approaches reach performances close to their theoretical peaks, the fused one benefits significantly from its reduced data transfers.

To achieve this fusion of the GEMMs, we developed device functions that can be called in a sequence, as illustrated in Listing 1. The data used by the kernel is first loaded into shared memory and registers, subsequently used by the kernels through the shared memory or register shuffling (when needed), and finally the result is written back to the main memory only once.

```

1 template<int M, int N, int DIMX, int DIMY, int sM, int sN> __global__ void
2 zgemm_batched_tensor_kernel(const magmaDoubleComplex alpha,
3                             magmaDoubleComplex const * const * dA_array, int ldda,
4                             magmaDoubleComplex const * dB, int lddb,
5                             magmaDoubleComplex const * dD, int lddd,
6                             const magmaDoubleComplex beta,
7                             magmaDoubleComplex** dC_array, int lddc,
8                             const int batchSize) {
9     const int tx = threadIdx.x, ty = threadIdx.y, tz = threadIdx.z, bx = blockIdx.x;
10
11     const int batchid = bx * blockDim.z + tz;
12     if(batchid >= batchSize) return;
13
14     const magmaDoubleComplex* dA = dA_array[batchid];

```

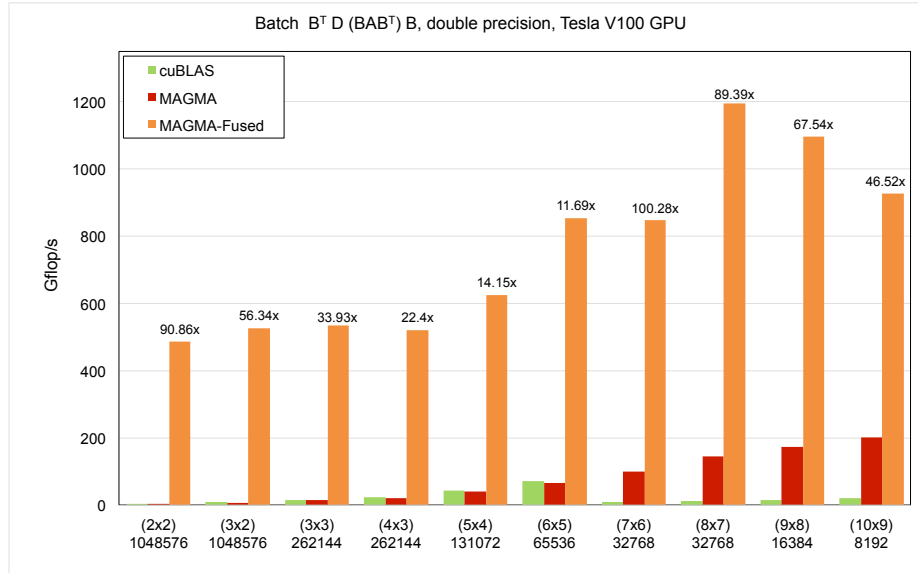


Figure 16: Batched DGEMM in cuBLAS and MAGMA vs. Batched fused DGEMM in MAGMA on matrices of sizes coming from real applications (in the MFEM library). The speedups reported are compared to the cuBLAS non-fused batched DGEMMs.

```

15     magmaDoubleComplex* dC = dC_array[batchid];
16
17     const int wdim = max(sM, sN);
18     const int slda = SLDA(wdim), sldb = SLDA(sM), sldc = SLDA(wdim);
19     const magmaDoubleComplex c_one = MAGMA_Z_ONE, c_zero = MAGMA_Z_ZERO;
20     magmaDoubleComplex* sA = (magmaDoubleComplex*)(zdata);
21     magmaDoubleComplex* sC = (magmaDoubleComplex*)(sA + blockDim.z * slda * wdim);
22     magmaDoubleComplex* sB = (magmaDoubleComplex*)(sC + blockDim.z * sldc * wdim);
23     magmaDoubleComplex* sD = (magmaDoubleComplex*)(sB + sldb * sN);
24
25     sA += tz * slda * wdim;
26     sC += tz * sldc * wdim;
27
28     // read
29     zread<sN, sN, DIMX, DIMY>(N, N, dA, ldda, sA, slda, tx, ty);
30     if(tz == 0){
31         zread<sM, sN, DIMX, DIMY>(M, N, dB, lddb, sB, sldb, tx, ty);
32         zread_diagonal<sM, DIMX>(M, dD, lddd, sD, tx, ty);
33     }
34     __syncthreads();
35
36     /**** computing B' D (B A B') B ****/
37     // compute sC = AB'
38     zgemm_device_nt<sN, sM, N, DIMX, DIMY>(c_one, sA, slda, sB, sldb, c_zero, sC, sldc, tx, ty);
39     __syncthreads();
40
41     // compute sA = sB (B) x sC (AB')
42     zgemm_device_nn<sM, sM, N, DIMX, DIMY>(c_one, sB, sldb, sC, sldc, c_zero, sA, slda, tx, ty);
43     __syncthreads();
44
45     // compute sC = sA (B A B') x sB (B)
46     zgemm_device_nn<sM, sN, M, DIMX, DIMY>(c_one, sA, slda, sB, sldb, c_zero, sC, sldc, tx, ty);
47     __syncthreads();
48
49     // compute sC = sD (D) x sC (BAB' B)
50     zgemm_device_prediag_nn<sM, sN, DIMX, DIMY>(sD, sC, sldc, tx, ty);
51     __syncthreads();

```

```

52
53 // compute sA = sB' (B') x sC (D B A B' B)
54 zgemm_device_tn<sN, sN, M, DIMX, DIMY>(c_one, sB,sldb, sC,sldc, c_zero, sA,slda, tx,ty);
55 __syncthreads();
56
57 // write
58 zwrite<sN, sN, DIMX, DIMY>(N, N, sA, slda, dC, lddc, tx, ty);
59 }

```

Listing 1: Design for tensor contractions through fused Batched BLAS in MAGMA. BLAS routines, e.g., GEMMs are defined as device functions and called in a sequence (as needed for a particular tensor contraction) from the same kernel.

Further details and summary of our results on the batched fused BLAS can be found in our poster, submitted and accepted for presentation at GTC'18 [1].

4. CEED APPLICATIONS AND MINIAPPS

4.1 ExaSMR

Multiscale Control in Time. ExaSMR's RANS model requires efficient simulations for the flow and heat evolution involving time-scale difference at several orders of difference. CEED team has developed a steady-state solver for solving nonlinear convection-diffusion equation based on a pseudo-timestepping approach that allows freezing the velocity while heat transfer occurs insignificant amount. CEED team implemented a robust algorithm with solid studies on the choices of parameters that can assure efficient and reliable performance. Our approach is based on the Jacobian-free Newton Krylov method during the pseudo-timestepping using the backward difference formula for linearization at a local time step and solves the linearized system by GMRES with our spectral element multigrid preconditioner. Then a Jacobian-Free Newton Krylov method for solving the pseudo time-transient step is applied, which allows increasing timestep size as the Newton iteration evolves to reach a steady state solution.

4.2 MARBL

As discussed in prior milestone reports, we have been rewriting the Lagrangian phase of BLAST (the ALE component of MARBL). The objective of this refactoring is to execute batched EOS and material constitutive model calls and provide separation between finite element assembly and physics-related calculations. The new structures closely resemble those of the Laghos miniapp, which is expected to allow straightforward inclusion of any optimizations done in Laghos and libCEED functionality. The MARBL-related activities during this reporting period were focused on the following tasks:

1. **Achieved a fully functional refactored Lagrangian phase.** The refactoring of the key computational component of the BLAST Lagrangian hydrodynamics solver was recently completed. The complete version now includes support for multi-material closure model calculations, elasto-plastic flow (material strength) models, magneto-hydrodynamics and radiation-hydrodynamics coupling.
2. **Integrated all Laghos partial assembly (PA) kernels into BLAST.** The remaining Laghos kernels were moved into BLAST. In this period we added (i) the PA operators for tensor-based evaluation of gradients, Jacobians and interpolated function values at quadrature points, and (ii) a diagonal preconditioner for the partially assembled velocity mass matrix.
3. **Derived BLAST-specific modifications of the PA operators.** As Laghos is a single-material miniapp with simplified physics, a set of extensions was needed to support all of BLAST's capabilities. All PA operators were extended by functionality for axisymmetric computations and multi-material simulations.
4. **Extensive testing and debugging of the new functionality and comparisons to the BLAST's established baseline version.** Because the new code provides a completely new execution path, we performed comprehensive comparisons to the established results. We achieved complete agreement

between the baseline code and the refactored code. We are still analyzing differences that appear when some of the PA kernels are used.

4.3 ExaWind

McAllister-Takahashi Blade Validation Problem. We have engaged with ExaWind project milestone on *single blade-resolved simulation in non-rotating turbulent flow*. ExaWind milestone is to establish baseline capabilities for blade-resolved simulations in terms of turbulence modeling, towards full wind plant simulations, for predicting the formation and evolution of blade-tip vortices that are important in analyzing turbine wakes. CEED and ExaWind teams are currently engaged with a finite-length NACA0015 blade simulations with the experimental data performed in a wind tunnel by McAlister and Takahashi. ExaWind team uses the low-Mach-number Navier-Stokes solver, *Nalu* on unstructured grids, utilizing Trilinos and the Sierra Toolkit for parallelization and I/O. CEED team will conduct simulations with Nek5000 and provide its performance and the detail analysis of flow measurement on the tip vortex evolution for flat- and rounded-tip configurations that would leverage the validation process of ExaWind results.

4.4 Laghos

4.4.1 Laghos developments

Laghos (LAGrangian High-Order Solver) is a miniapp developed in CEED that solves the time-dependent Euler equations of compressible gas dynamics in a moving Lagrangian frame using unstructured high-order finite element spatial discretization and explicit high-order time-stepping. In CEED, Laghos serves as a proxy for a sub-component of the MARBL/LLNLApp application. Laghos captures the basic structure and user interface of many other compressible shock hydrocodes, including the BLAST code at LLNL. The miniapp is build on top of a general discretization library (MFEM), thus separating the pointwise physics from finite element and meshing concerns. It exposes the principal computational kernels of explicit time-dependent shock-capturing compressible flow, including the FLOP-intensive definition of artificial viscosity at quadrature points. The implementation is based on the numerical algorithm from [3].

Since it was introduced in milestone CEED-MS6 and ported to the Open Concurrent Compute Abstraction (OCCA) for CEED-MS8, a number of additional features and improvements have been added to Laghos, including a **GPU Direct**-capable MPI communicator that now allows all the data to stay on the device during the communication phases. We conducted detailed performance comparison of RAJA, OCCA and pure programming models to evaluate the cost of portability abstractions. This involved extracting the OCCA kernels, simplifying code and creating different OCCA-agnostic versions to conduct head-to-head performance comparisons. Here are the different stand-alone ports for Laghos that have been developed so far:

- pure **CPU**: with or without the use of lambda statements (C++ λ -functions), with or without using templates ($\langle \rangle$) to allow fair comparison with OCCA's Just-In-Time (JIT) capabilities,
- pure **GPU**: with or without the λ -functions or the templates $\langle \rangle$,
- pure **RAJA**: CPU or GPU versions, only through λ for-loop bodies.

These ports target the **partial assembly** kernels, where the amount of data storage, memory transfers, and FLOPs are lower, especially for higher orders. Table 2 presents the number of source lines of code; all of these ports represent less than 8k lines of code (kloc), more than a half for the kernels and 1kloc for the **GPU Direct**-capable MPI communicator.

4.4.2 Initial RAJA and pure CPU, GPU ports

In order to have a fair comparison between the different versions, we started with OCCA kernels from CEED-MS8. Currently, there are two flavors of OCCA kernels: one version is for *CPU* and another for *GPU-High-Order*. The *GPU* ones use **share** memory, are launched with a topology that matches their nested for-loops ranges, which are features that require some recent or work-in-progress developments for the RAJA abstraction layer.

SLOC (cpp)	Directory
4775	kernels
1277	fem
772	linalg
651	general
332	config
210	tests
36	top_dir

Table 2: SLOC

Thus, the pure *CPU* and *GPU* versions are very useful to compare and measure the overhead of compilers, impact of UVM and the λ functions, or just the *GPU-High-Order* kernels versus the *CPU* ones,

The first step has been to extract all of these kernels to some C-style ones:

- *in* and *out* data arguments types are turned to `double*`,
- offset computation are hidden in macros $X(q, e)$ vs. $X[ijN(q, e, NUM_QUAD)]$
- *outer*, *inner*, *share* OCCA keywords have been removed. The first two can be handled by some macros hiding the way the for loop is going to iterate, but the *share* is still a challenge,
- most of the OCCA properties (`bool`, `int`) that are set while parsing the kernel have been added as template parameters. The templates parameters allows to do some static (not yet dynamic) *partial evaluation*, however reducing the arguments that are known at compile time, in order to *specialize* the kernel.

Table 3 presents an example of such a port. `NUM_DOFS` and `NUM_QUAD` are passed as templates parameters and used for the for-loop range and offset computation.

<pre> typedef double* DofToQuadD1D_t @dim(NUM_QUAD, NUM_DOFS); typedef double* Local1D_t @dim(1, NUM_DOFS, numElements); typedef double* Jacobian1D_t @dim(NUM_QUAD, numElements); typedef double* QLocal_t @dim(NUM_QUAD, numElements); kernel void InitGeometryInfo1D(const int numElements, const DofToQuadD1D_t restrict dofToQuadD, const Local1D_t restrict nodes, Jacobian1D_t restrict J, Jacobian1D_t restrict invJ, QLocal_t restrict detJ) { for (int e = 0; e < numElements; ++e; outer) { shared double s_nodes[NUM_DOFS]; for (int q = 0; q < NUM_QUAD; ++q; inner) for (int d = q; d < NUM_DOFS; d += NUM_QUAD) s_nodes[d] = nodes(0, d, e); for (int q = 0; q < NUM_QUAD; ++q; inner) { double J11 = 0; for (int d = 0; d < NUM_DOFS; ++d) J11 += dofToQuadD[ijN(q, d, NUM_DOFS)] * s_nodes[d]; J(q, e) = J11; invJ(q, e) = 1.0 / J11; detJ(q, e) = J11; } } } </pre>	<pre> template<const int NUM_DOFS, const int NUM_QUAD> kernel void rlniGeom1D(const int numElements, const double* restrict dofToQuadD, const double* restrict nodes, double* restrict J, double* restrict invJ, double* restrict detJ) { forall(eI, nzones, { shared double s_nodes[NUM_DOFS]; for (int q = 0; q < NUM_QUAD; ++q) for (int d = q; d < NUM_DOFS; d += NUM_QUAD) s_nodes[d] = nodes[ijkN(0, d, e, NUM_QUAD)]; for (int q = 0; q < NUM_QUAD; ++q) { double J11 = 0; for (int d = 0; d < NUM_DOFS; ++d) J11 += dofToQuadD[ijN(q, d, NUM_DOFS)] * s_nodes[d]; J[ijN(q, e, NUM_QUAD)] = J11; invJ[ijN(q, e, NUM_QUAD)] = 1.0 / J11; detJ[ijN(q, e, NUM_QUAD)] = J11; } }); } </pre>
OCCA Kernel Language kernel	C-style kernel

Table 3: OCCA to C-kernel example

4.4.3 First wave results

Sources, remotes and branches are set as follow:

- **Master** version comes from <https://github.com/CEED/Laghos/tree/master>.
- **Kernels** version is from <https://github.com/CEED/Laghos/tree/raja-dev>: compiled with the `cpu` makefile target, this version uses the kernels ported from OCCA and produces an executable for CPU and GPU, each of them with or without λ functions and with or without the template parameters.
- **Raja** version is from <https://github.com/CEED/Laghos/tree/raja-dev> too: compiled with the `raja` makefile target, this version uses the `nvcc` compiler from cuda-9.0 SDK and uses **RAJA** from <https://github.com/LLNL/RAJA>. The produced executable can use a serial CPU policy or a CUDA GPU one, that can be chosen at runtime on the command line.
- **Occa** version is from <https://github.com/dmed256/Laghos/tree/occa-dev>, uses MFEM's <https://github.com/mfem/mfem/tree/occa-dev> branch, and **OCCA** from <https://github.com/libocca/occa>. The executable can target multiple backends at runtime: serial CPU, OpenMP, OpenCL and GPU.
- The hardware used for these first results was on one Intel(R) Xeon(R) CPU E5-1620v4@3.50GHz for the serial results and one GeForceGTX (Pascal) device or the Ray machine at LLNL (a CORAL early access architecture) for the CUDA results.

Table 4 shows the first results we obtained: **Master** is our reference implementation, that we are comparing to the three others (**Kernels**, **Raja** and **Occa**), that now use the same *underlying* kernels, except for the CUDA OCCA ones. These kernels are however called differently (*directly* or *through RAJA* and *through OCCA*), with additional *properties* set accordingly to possibilities of each layer.

The **Kernels** and **Raja** versions are similar, which is what was expected, slightly better than the **Master** version. The **Occa** version is behind, which is in agreement with what was presented in the CEED MS8 report (3.1.2 Laghos on OCCA).

Table 5 compares the two CUDA versions, showing an order of magnitude for **Occa** over **Raja**. But again, it is no more a head-to-head comparison, because of **Occa**'s high-order CUDA kernels.

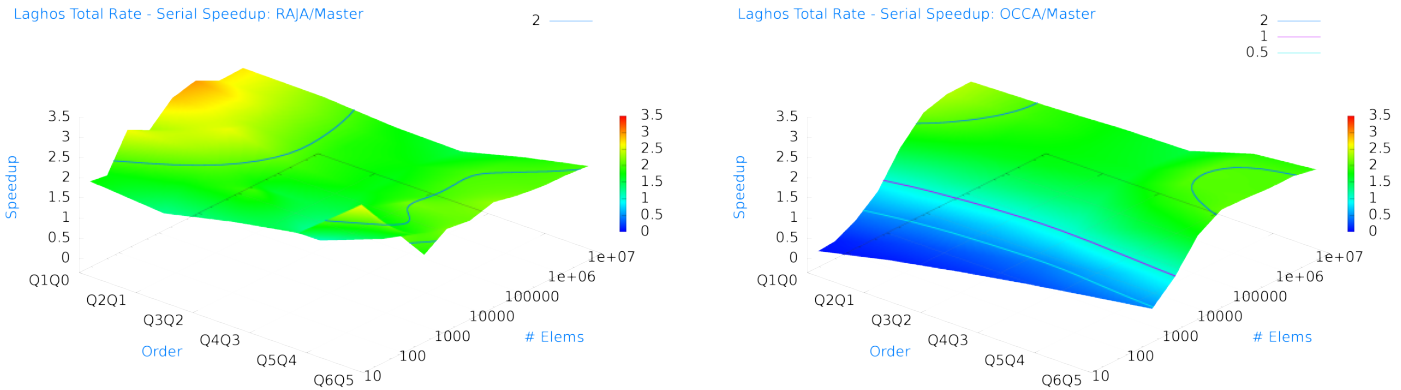


Figure 17: Laghos P0,2D serial speedup: RAJA/Master and OCCA/Master

Figure 17 provides another way to see the relative performances of two versions. The surface shows the speedup of the **Raja** and **Occa** versions compared to the **Master** one, sweeping different orders and different number of elements per node. The **Raja** version is about twice as fast as the **Master** one, and **Occa** exposes the zone where it is less efficient.

Figure 18 presents the OCCA/RAJA speedup of one 2D problem (P0) Laghos test case. For high orders and fewer degrees of freedom per node, OCCA's CUDA kernels are showing their efficiency.

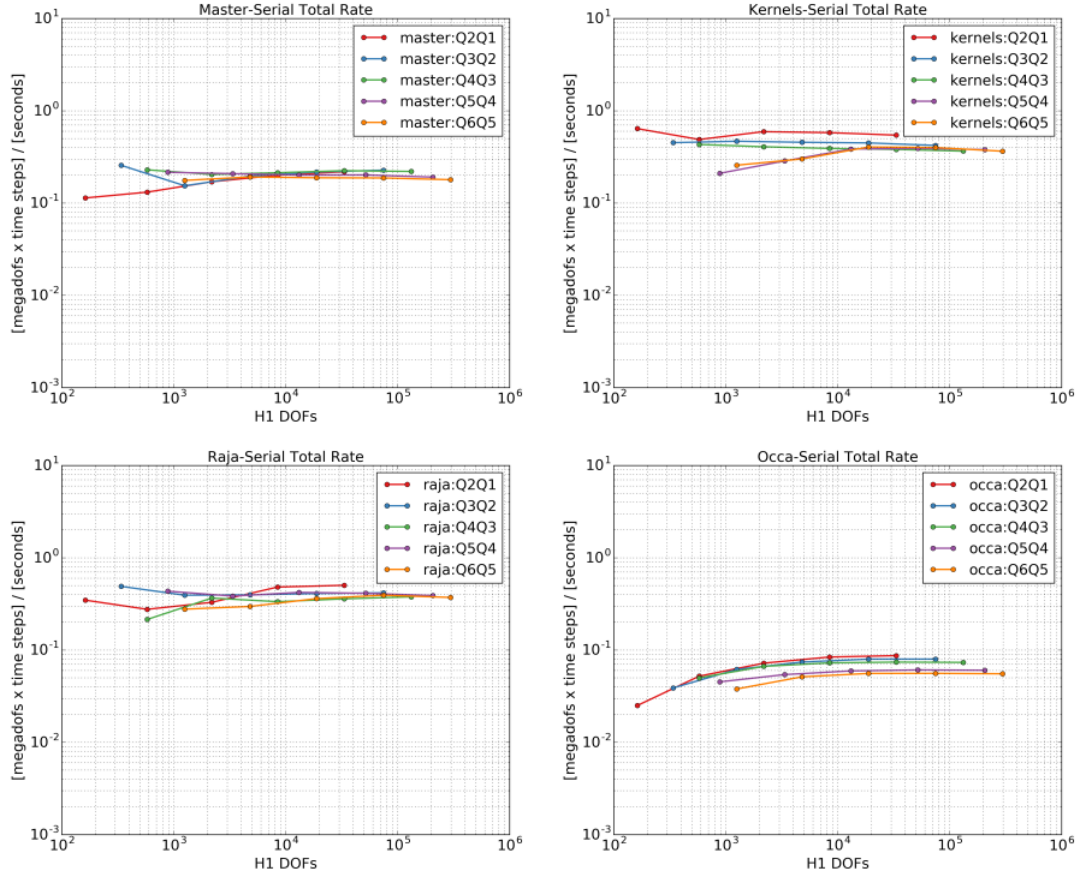


Table 4: Laghos serial results: Master, Kernels, Raja and Occa

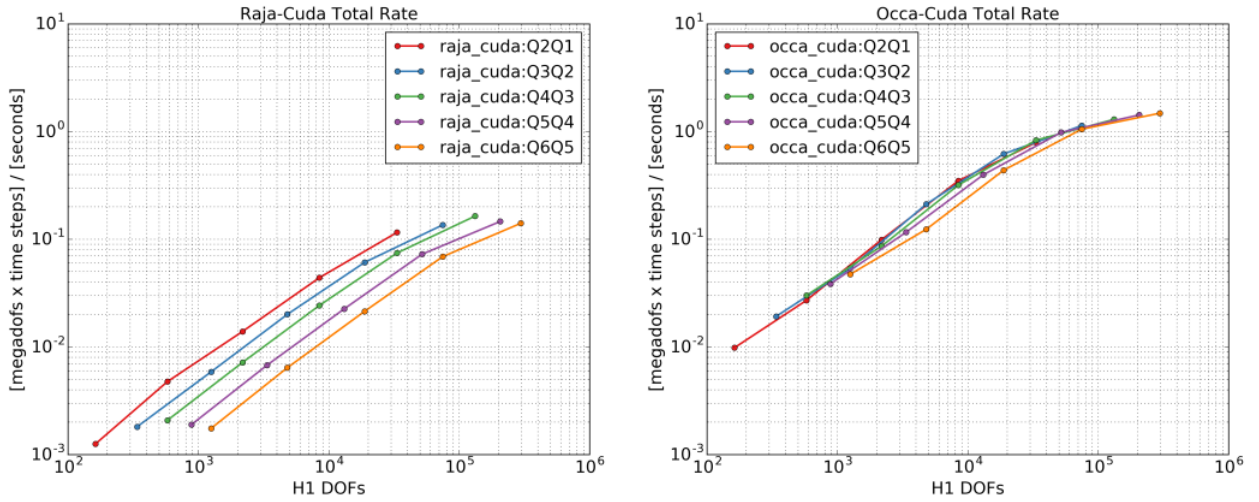


Table 5: Laghos CUDA results: Raja and Occa

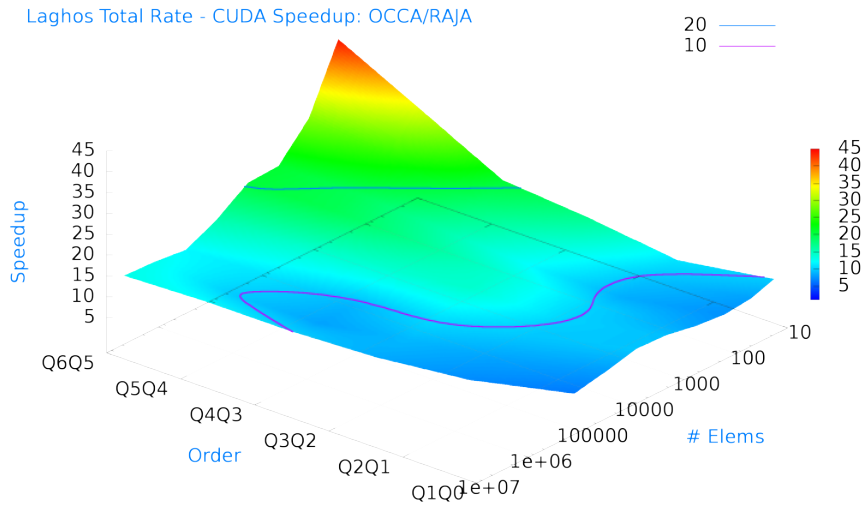


Figure 18: Laghos P0,2D test case: CUDA Speedup: OCCA/RAJA

4.4.4 Second wave results

The second part of the optimization consisted in:

1. Switching to the CUDA driver (vs. runtime) API, similar to OCCA.
2. Adding a way for the **Kernels** to be launched *with* or *without* the lambda-body capture mechanism.
3. Adding a template feature to the **Kernels**, in order to take advantage at compile time of these static inputs (some of the main for-loop ranges). This feature allows that option to be comparable with the Just-In-Time (JIT) feature of OCCA.
4. Being able to run *with* or *without* **UVM**, adopting the simple convention that all MFEM data resides on the host, and the RAJA one on the device, with explicit movements through simple type conversions.
5. Working on *reducers* and the **dot product**, to be competitive with the ones used by **OCCA**.

Figure 19 shows the gain in performances when adding the template feature that mimics OCCA's JIT capability. A gain of two to four is obtained on this test case on Ray's Power processors.

Figures 20 and 21 are the speedups of OCCA versus templated-RAJA with the second wave of optimizations. There is no more an order of magnitude between the two versions, but still an efficient zone that OCCA exploits by using his high-order kernels.

4.4.5 Third wave results

The third part of the optimization process continued by:

1. Optimizing the data movement: most of the kernels now have zero data transfers between the host and the device.
2. Cleaning all (re)allocations that are not required during the time steps: each allocation is done during the setup phase, no more during solver iterations or for local vectors declarations and references.
3. Porting the high-order kernels to the **Kernels** GPU executable (<gpuK-share>), to be able to test these kernels while RAJA implements nested for-loops with in-middle code, as well as shared memory local array declarations.

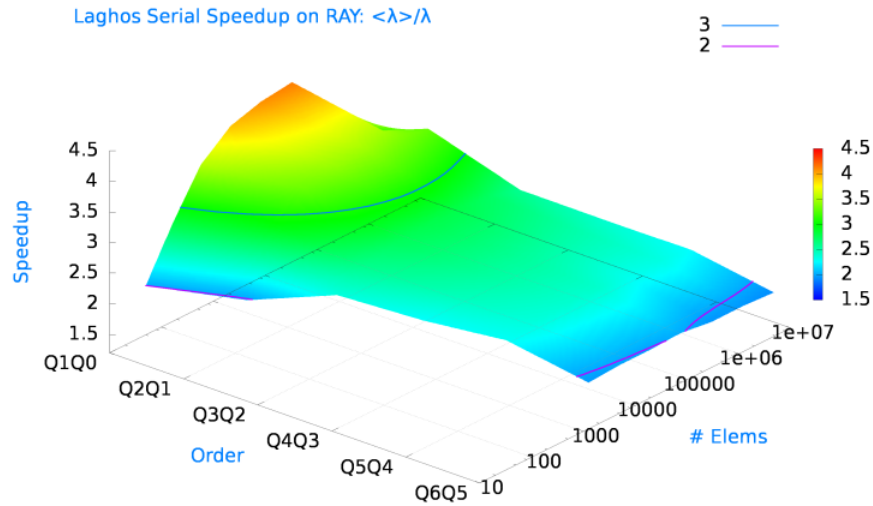


Figure 19: Laghos **Kernels** serial speedup on Ray with templated arguments

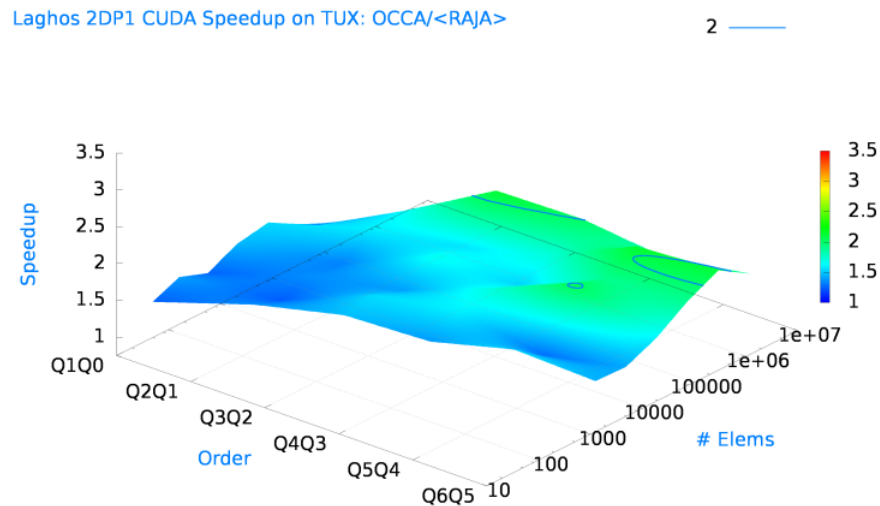


Figure 20: Laghos P1,2D CUDA **speedup**: OCCA/templated-RAJA on GeForce GTX 1070

4. Porting some of MFEM's communication operators to the device: **Restriction** and **ConformingProlongation**, allowing when possible direct communication from device memory between MPI ranks.

Figure 22 is an example of NVIDIA's NVVP profiler output for Laghos. Even if the MemCpy DtoH, DtoD and HtoD seem heavily used, only 8 bytes for the results of the reduction flies from the device to the host to pinned memory. The main of the time is now spent in the mass kernel, which represents here up to 80% of the computation.

Figure 23 is the latest performance results obtained on Ray, showing now a speedup up to four times compared to the **Occa** version. The high-order kernels are the same now, but the Laghos miniapp has been optimized:

Laghos 2DP0 CUDA Speedup on RAY: OCCA/<RAJA>

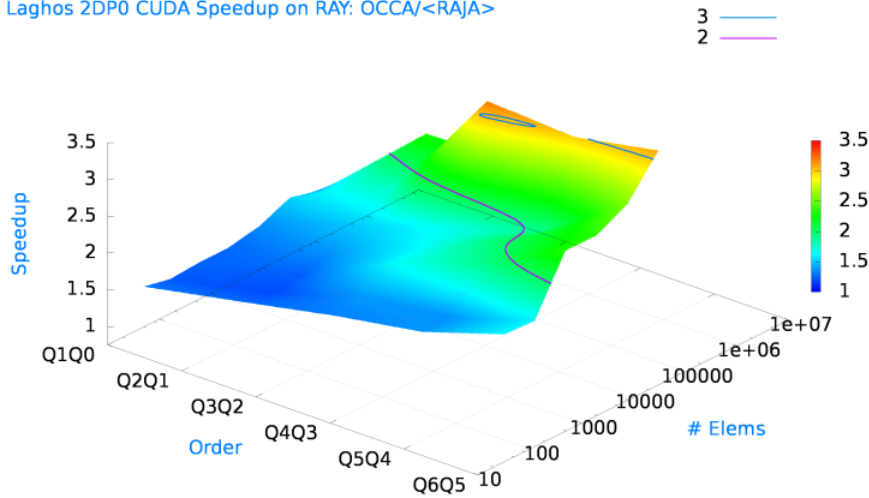


Figure 21: Laghos P0,2D CUDA speedup: OCCA/templated-RAJA on Ray

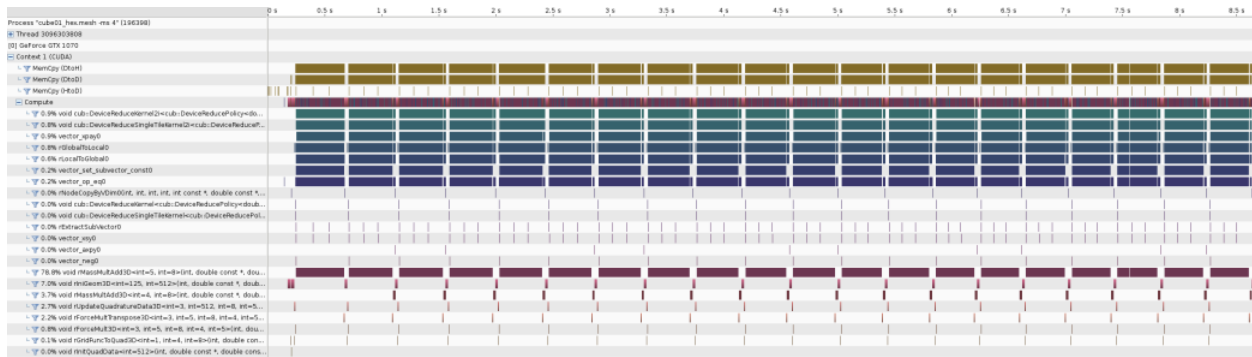


Figure 22: Laghos P1,3D CUDA NVVP Kernels profile

data movement and memory allocations are now optimal, such optimizations could be pushed to the OCCA branch.

5. OTHER PROJECT ACTIVITIES

5.1 Benchmark release by Paranumal team

The CEED group at Virginia Tech released standalone implementations of CEED's BP1.0, BP3.0, and BP3.5 benchmarks at <https://github.com/kswirydo/CEED-Ax>. For results and discussion, see their new Paranumal blog at www.paranumal.com.

5.2 SciDAC collaborations

Several CEED researchers have been also in collaboration with domain and application scientist under the SciDAC program The FASTMath applied math SciDAC institute addresses the needs of DOE applications in the following eight areas:

- Structured mesh spatial discretization

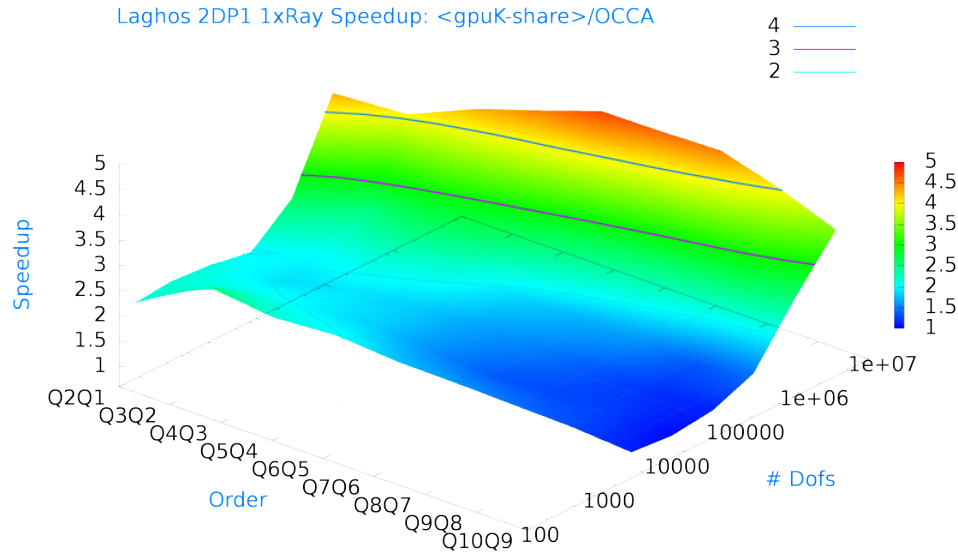


Figure 23: Laghos P1,3D CUDA **Kernels** speedup vs. OCCA

- Unstructured mesh spatial discretization
- Time integrators
- Linear systems solvers
- Solution of eigenvalue problems
- Numerical optimization
- Uncertainty quantification
- Data analytics

The primary interactions between CEED and FASTMath relate to unstructured mesh spatial discretization where Dobrev, Kolev, Shephard and Smith are members of both CEED and the FASTMath unstructured mesh teams. Unstructured mesh areas of interaction where CEED will interact with, and take advantage of, FASTMath developments include:

- Non-conforming parallel mesh adaptation
- Conforming parallel mesh adaptation
- Support for the representation and adaptation of high-order curved meshes for complex domains.
- Support of solution fields and discretization error estimators.
- Dynamic load balancing.

CEED will also track FASTMath developments in linear system solvers and data analytics (the in situ visualization work) and take advantage of all appropriate advances.

5.3 Batched BLAS minisymposium at SIAM PP18

CEED's UTK team organized a two-session minisymposium at the SIAM Conference on Parallel Processing and Scientific Computing (SIAM PP'18) in Tokyo, Japan from March 7-10, 2018, devoted on the "Batched BLAS Standardization". This is part of our efforts on standardization and co-design of exascale discretization APIs with application developers, hardware vendors and ECP software technologies projects. The goal is to extend the BLAS standard to include batched BLAS computational patterns/application motifs that are essential for representing and implementing tensor contractions. Besides participation from the CEED project, we invited stakeholders from ORNL, Sandia, NVidia, Intel, IBM, and Universities [14].

5.4 Collaboration with Zfp

CEED researchers begun a collaboration with the Zfp team on compression algorithms that are specifically tailored to high-order finite element data. Our initial goal was to apply Zfp as an HDF5 filter to MFEM-generated data. An HDF5 DataCollection class was developed in MFEM and the mesh-explorer miniapp was enhanced to write to the new format. Investigation is ongoing, but the initial results are promising.

5.5 OCCA 1.0-alpha pre-release

Several pre-releases of OCCA 1.0, featuring new parser and many other improvements, were published on GitHub at <https://github.com/libocca/occa>. A number of CEED packages, including libCEED and Paranalum are being updated to use the new features.

5.6 Outreach

CEED researchers were involved in a number of outreach activities, including a successful breakout session on high-order methods and application at the ECP second annual meeting in Knoxville, a talk on Laghos at JOWOG 34 at SNL and a poster accepted at GTC18 [1]. The MFEM project was highlighted in the Jan/Feb issue of LLNL's Science and Technology Review magazine (including the cover of the magazine). The team also finalized the upcoming CEED-organized minisymposium, "Efficient High-Order Finite Element Discretizations at Large Scale", at the International Conference in Spectral and High-Order Methods (ICOSAHOM18).

6. CONCLUSION

In this milestone, we created and made publicly available the first CEED software release consisting of software components such as MFEM, Nek5000, PETSc, MAGMA, OCCA, etc., treated as dependencies of CEED. The artifacts delivered include a consistent build system based on the 12 integrated Spack packages, plus a new CEED *meta-package*, documentation and verification of the build process, as well as improvements in the integration between different CEED components, see the CEED website, <http://ceed.exascaleproject.org/ceed-1.0/> and the CEED GitHub organization, <http://github.com/ceed> for more information. . In this report, we also described additional CEED activities performed in Q2 of FY18, including: benchmark release by the Paranalum team, collaboration with SciDAC projects, organization of batched BLAS mini-symposium at SIAM PP18, collaboration with Zfp, the OCCA 1.0-alpha pre-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-748602.

REFERENCES

- [1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Tensor contractions using optimized batch gemm routines. http://icl.cs.utk.edu/projectsfiles/magma/pubs/72-gtc18_tensor_poster.pdf. GPU Technology Conference (GTC), Poster, March 26-29, 2018, San Jose, CA.
- [2] M.W. Beall, J. Walsh, and M.S. Shephard. A comparison of techniques for geometry access related to mesh generation. *Engineering with Computers*, 20:210–221, 2004.
- [3] V. A. Dobrev, T. V. Kolev, and R. N. Rieben. High-order curvilinear finite element methods for Lagrangian hydrodynamics. *SIAM J. Sci. Comp.*, 34(5):B606–B641, 2012.
- [4] Dan Ibanez and Mark S Shephard. Modifiable array data structures for mesh topology. *SIAM Journal on Scientific Computing*, 39(2):C144–C161, 2017.
- [5] Daniel A. Ibanez, E. Seogyoun Seol, Cameron W. Smith, and Mark S. Shephard. Pumi: Parallel unstructured mesh infrastructure. *ACM Transactions on Mathematical Software (TOMS)*, 42(3):17, 2016.
- [6] Elias Konstantinidis and Yiannis Cotronis. A practical performance model for compute and memory bound gpu kernels. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 651–658. IEEE, 2015.
- [7] Elias Konstantinidis and Yiannis Cotronis. A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37–56, 2017.
- [8] Qiukai Lu, Mark S. Shephard, Saurabh Tendulkar, and Mark W. Beall. Parallel mesh adaptation for high-order finite element methods with curved element geometry. *Engineering with Computers*, 30(2):271–286, 2014.
- [9] Xiao-Juan Luo, Mark S. Shephard, Lie-Quan Lee, Lixin Ge, and Cho Ng. Moving curved mesh adaptation for higher-order finite element simulations. *Engineering with Computers*, 27(1):41–50, 2011.
- [10] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcoub, and J. Dongarra. Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices. *Parallel Comput. Syst. Appl.*, 2018. (submitted).
- [11] PUMI: Parallel unstructured mesh infrastructure, 2016. <http://www.scorec.rpi.edu/pumi>.
- [12] Michel Rasquin, Cameron Smith, Kedar Chitale, E Seogyoun Seol, Benjamin A Matthews, Jeffrey L Martin, Onkar Sahni, Raymond M Loy, Mark S Shephard, and Kenneth E Jansen. Scalable Implicit Flow Solver for Realistic Wing Simulations with Flow Control. *Computing in Science & Engineering*, (6):13–21, 2014.
- [13] T Warburton. An explicit construction of interpolation nodes on the simplex. *Journal of engineering mathematics*, 56(3):247–262, 2006.
- [14] Mawussi Zounon, Azzam Haidar, and Siva Rajamanickam (Organizers). On Batched BLAS Standardization. http://meetings.siam.org/sess/dsp_programsess.cfm?SESSIONCODE=63546, March 7-10 2018. Mini-symposium at SIAM PP’18, Tokyo, Japan.