ECP Milestone Report
Initial Integration of CEED Software in ECP Applications
WBS 1.2.5.3.04, Milestone CEED-MS8

Misun Min
Jed Brown
Veselin Dobrev
Paul Fischer
Tzanio Kolev
David Medina
Elia Merzari
Aleks Obabko
Scott Parker
Ron Rahaman
Stanimire Tomov
Vladimir Tomov
Tim Warburton

October 4, 2017

ECP Milestone Report
Initial Integration of CEED Software in ECP Applications
WBS 1.2.5.3.04, Milestone CEED-MS8

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

October 4, 2017

ECP Milestone Report
Initial Integration of CEED Software in ECP Applications
WBS 1.2.5.3.04, Milestone CEED-MS8

Approvals

Submitted by:

Tzanio Kolev, LLNL CEED PI	Date

Approval:

Douglas B. Kothe, Oak Ridge National Laboratory Director, Applications Development Exascale Computing Project	Date

Revision Log

Version	Creation Date	Description	Approval Date
1.0	October 4, 2017	Original	

EXECUTIVE SUMMARY

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

In this milestone we report on the initial integration of CEED software in the selected first-wave ECP/CEED apps and the continued exploration of other ECP applications as second-wave (milestone CEED-MS23) and third-wave (milestone CEED-MS35) ECP/CEED application candidates. An extensive evaluation of various approaches for GPU acceleration of the CEED computational motives is also a major focus of the report, both in the settings of CEED's Nekbone and Laghos miniapps, as well as versions of CEED's bake-off problems (introduced in milestone CEED-MS6).

The first-wave ECP applications targeted for coupling with CEED software include the ExaSMR and MARBL applications. We also report on near-term collaboration with the seed-funded Urban Systems project, as well as engagements with ACME, ExaAM, GEOS and the Application Assessment and Proxy App projects.

The ExaSMR project involves coupled thermal-hydraulics/neutronics, with the latter based on Monte Carlo methods and the former based on high-order discretizations of the Navier-Stokes and energy equations for accurate simulation of turbulent transport. The project requires fast and efficient turbulence simulation capabilities that will scale to exascale platforms. Work within CEED will ensure that all aspects of the Nek5000 workflow scale to the target problem space, which involves $> 10^{10}$ degrees of freedom in typical production runs. CEED's goal is to provide implementations that perform with maximum attainable efficiency and deliver low turn-around times. A deliverable for this milestone is beta-release of the GPU-based variant of Nek5000.

MARBL is targeting high-energy-density physics problems, including single fluid multi-material hydrodynamics and radiation/magnetic diffusion simulation, with applications in inertial confined fusion, pulsed power experiments, and equation of state/material strength analysis. The code uses high-order methods based arbitrary Lagrangian-Eulerian, direct Eulerian, and unstructured adaptive mesh refinement. A deliverable for this milestone is collaboration with MARBL for improved performance based on partially-assembled operators.

The Urban exascale project is planning to couple real-time data with high performance simulations to study particulate transport in cities. Nek5000 will be used for large eddy and RANS (Reynolds-averaged Navier-Stokes) simulations of turbulent transport in these domains. Areas where the CEED team is aiding the Urban Canyon group include creating high-order meshes for urban geometries and developing boundary conditions appropriate for simulations close to real-life urban conditions.

The software artifacts delivered as part of this milestone include GPU-enabled versions of the Nekbone and Laghos miniapps, which are provided through the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed`.

In addition to details and results from the above R&D efforts, in this document we are also reporting on other project-wide activities performed in Q4 of FY17, including: the CEED first annual meeting and participation of CEED researchers in a variety of outreach activities.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1. INTRODUCTION

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

In this milestone we report on the initial integration of CEED software in the selected first-wave ECP/CEED apps and the continued exploration of other ECP applications as second-wave (milestone CEED-MS23) and third-wave (milestone CEED-MS35) ECP/CEED application candidates. An extensive evaluation of various approaches for GPU acceleration of the CEED computational motives is also a major focus of the report, both in the settings of CEED's Nekbone and Laghos miniapps, as well as versions of CEED's bake-off problems (introduced in milestone CEED-MS6).

The software artifacts delivered as part of this milestone include GPU-enabled versions of the Nekbone and Laghos miniapps, which are provided through the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed`.

2. ECP APPLICATIONS

The first-wave ECP applications targeted for coupling with CEED software include the ExaSMR and MARBL applications. We also report on near-term collaboration with the seed-funded Urban Systems project, as well as engagements with ACME, ExaAM, GEOS and the Application Assessment and Proxy App projects.

2.1 MARBL

As discussed in previous reports, the MARBL application has two major components, namely BLAST and MIRANDA. The CEED efforts are concentrated exclusively on BLAST, which is an MFEM-based Arbitrary Lagrangian-Eulerian (ALE) hydrodynamics code that uses high-order finite elements. In this section we present the plan and timeline for integrating the CEED technologies into BLAST and report the CEED efforts related to this plan, including the CEED-developed Laghos miniapp, see Section 3.1.

2.1.1 Plan and Timeline for CEED Activities

The initial target of CEED is the optimization of the Lagrangian phase in BLAST. This part of the BLAST code is complicated, as it contains many physical manipulations and different options, making it difficult to work and optimize this code directly. As an initial step, it was decided to write a new MFEM miniapp, called Laghos (LAGrangian High-Order Solver), which resembles the main computational kernels without the additional overhead of physics-specific code. As both BLAST and Laghos are high-order finite element codes based on MFEM, improvements made in Laghos are easily extendable to BLAST.

Figure 1 shows a detailed plan and timeline for integrating CEED technologies into BLAST, derived in close collaboration between the CEED and BLAST teams. In this report we concentrate on the first three tasks that were planned for completion by the end of Q4 FY17. Details of Laghos, its recent improvements, and the related Open Concurrent Compute Abstraction (OCCA) GPU implementation efforts are discussed in Section 3.1. The refactoring of the BLAST Lagrange phase code is addressed in the next section.

As indicated in Figure 1, a main focus in FY18 will be the completion of the BLAST Lagrangian phase optimization, for both CPU and GPU performance. The exploration of different CPU and GPU optimization techniques in Laghos can make a big difference and will be invaluable in that regard, see e.g. Section 4.1 in this report. Beyond FY18, the CEED efforts will target the optimization of the full ALE algorithm, which includes methods for Continuous Galerkin (CG) advection, Discontinuous Galerkin (DG) monotone advection and high-order mesh optimization. After these tasks, CEED will concentrate on optimizing the radiation-hydrodynamics module of BLAST, which requires GPU implementation of operations on the $H(div)$ spaces, including partial assembly and matrix-free preconditioners for $H(div)$ linear systems.

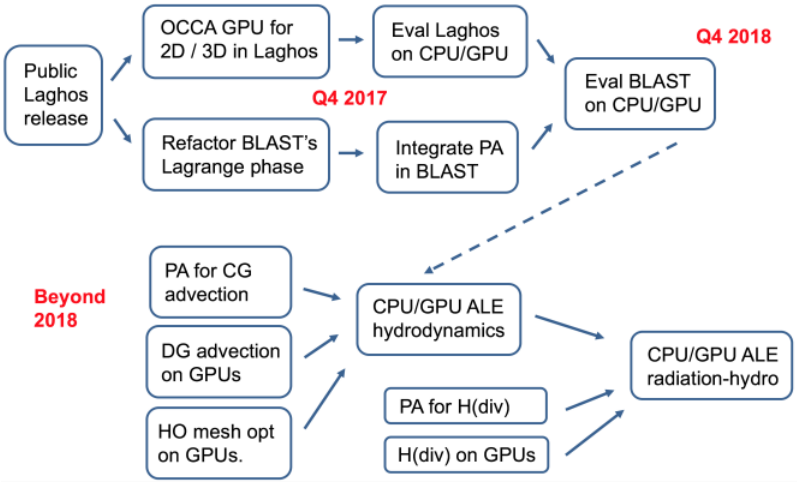

Figure 1: Integration plan and milestones for CEED/BLAST activities.

2.1.2 Refactoring of MARBL/BLAST's Lagrange phase

The BLAST code originated as a research tool, so adding new code to it has been mostly oriented towards a particular physical functionality and less towards performance. Therefore, the final code structure of its Lagrangian phase is not optimal in terms of amenability to optimization. To address this issue, the Laghos miniapp was prototyped with emphasis on the following features:

- Separation between finite element assembly and physics-related computations. In particular, pointwise computations for artificial viscosity, mesh size, equation of state (EOS) calls, etc. are clearly separated and independent of the chosen assembly algorithm.
- Batched calls to the EOS computations module. More specifically, EOS calls are not performed to estimate the result for one quadrature point at a time, but for all quadrature points in a given number of zones at once.

These features allow easier optimization of the assembly procedures and EOS calls. To be able to extend performance improvements from Laghos to BLAST, the BLAST code is being restructured in a similar manner. More specifically, the computation over all quadrature points in BLAST is being modeled on the Laghos structure as follows:

Algorithm 1 Improved Quadrature Evaluation for MARBL/BLAST

```
for e_batch ⊂ {e} – batches of elements do
    for e ∈ e_batch – elements in the current batch do
        Compute input data for all quadrature points and materials in the current element e
    end for
    Using the input of the current batch e_batch, execute a batched call to get
    the corresponding EOS output (pressure, sound speed, etc.)
    for e ∈ e_batch – elements in the current batch do
        Compute quadrature data for all quadrature points and materials in the current element e
    end for
end for
```

The major part of this restructuring is completed, and developers can run BLAST utilizing the new loop structure for standard single material test cases. Note that the above quadrature point computations are the critical numerical kernels for high performance and once they are performed, one can proceed in BLAST with finite element assembly (full or partial assembly) using the generated quadrature data structures.

2.2 ExaSMR

The ExaSMR project is focused on the exascale application of single and coupled MC and CFD physics. The application development objective is to optimize these applications for exascale execution of full core simulations. A CEED milestone, joint with ExaSMR, in Q4 FY17 focuses on initial GPU porting of Nek5000 for assembly-level simulations, provided with baseline performance. We report a successful GPU porting of Nek5000 with validation and demonstrate baseline performance for two benchmark problems: a subchannel problem (single-rod) to assess intranode performance and a larger full assembly problem with 17×17 rods, shown in Figure 2.

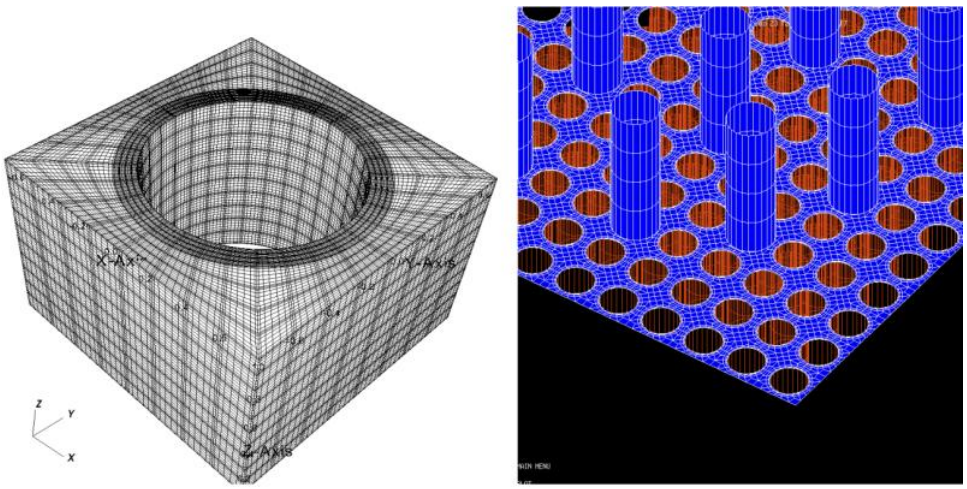

Figure 2: ExaSMR spectral element hexahedral meshes for subchannel and full assembly systems with 2×2 (left) and 17×17 (right) rods.

2.2.1 Nek5000 GPU Implementation

The initial GPU implementation is based on OpenACC pragmas and provides a basis for future accelerator-based development for Nek5000, extended from previous works[1, 2]. The principal focus has been on the linear system solves that constitute 70–80% of typical production runtimes. The port includes the full spectral element multigrid preconditioner and uses an extension of the Nek gather-scatter library *gslib*, `http://github.com/gslib`, that supports local (on-device) gather-scatters to minimize off-device communication.

We use OpenACC as a strategy for porting Nek5000 to GPU because of the relative ease of the pragma-based porting. OpenACC is a directive-based HPC parallel programming model, using host-directed execution with an attached accelerator device. The compiler maps the compute and data regions specified by the OpenACC directives to GPUs for higher performance. In contrast to other low-level GPU programming, such as CUDA and OpenCL, where more explicit compute and data management is necessary, porting of legacy CPU-based applications with OpenACC does not require significant structural changes in the original code, which allows considerable simplification and productivity improvement when hybridizing existing applications.

Before starting the Nek5000 time-advancement, each CPU executes data movement from the host CPU memory to GPU memory, and the majority of the computation is performed on GPUs during each time step. Within each step, the host CPU transfers GPU results back to the host when processing boundary conditions and the coarse-grid solver, neither of which are compute intensive. User-prescribed forcing functions and material properties are also implemented on the host. Inter-element (GPU-GPU) data exchanges are effected using GPU-direct communication, when available, or through a device-to-host transfer followed by MPI. Both modes are supported by *gslib*, which is called directly from the GPU.

While the directive-based approach of OpenACC greatly simplifies programming, it does not provide the performance flexibility of CUDA or OpenCL. For example, both CUDA and OpenCL provide fine-grained

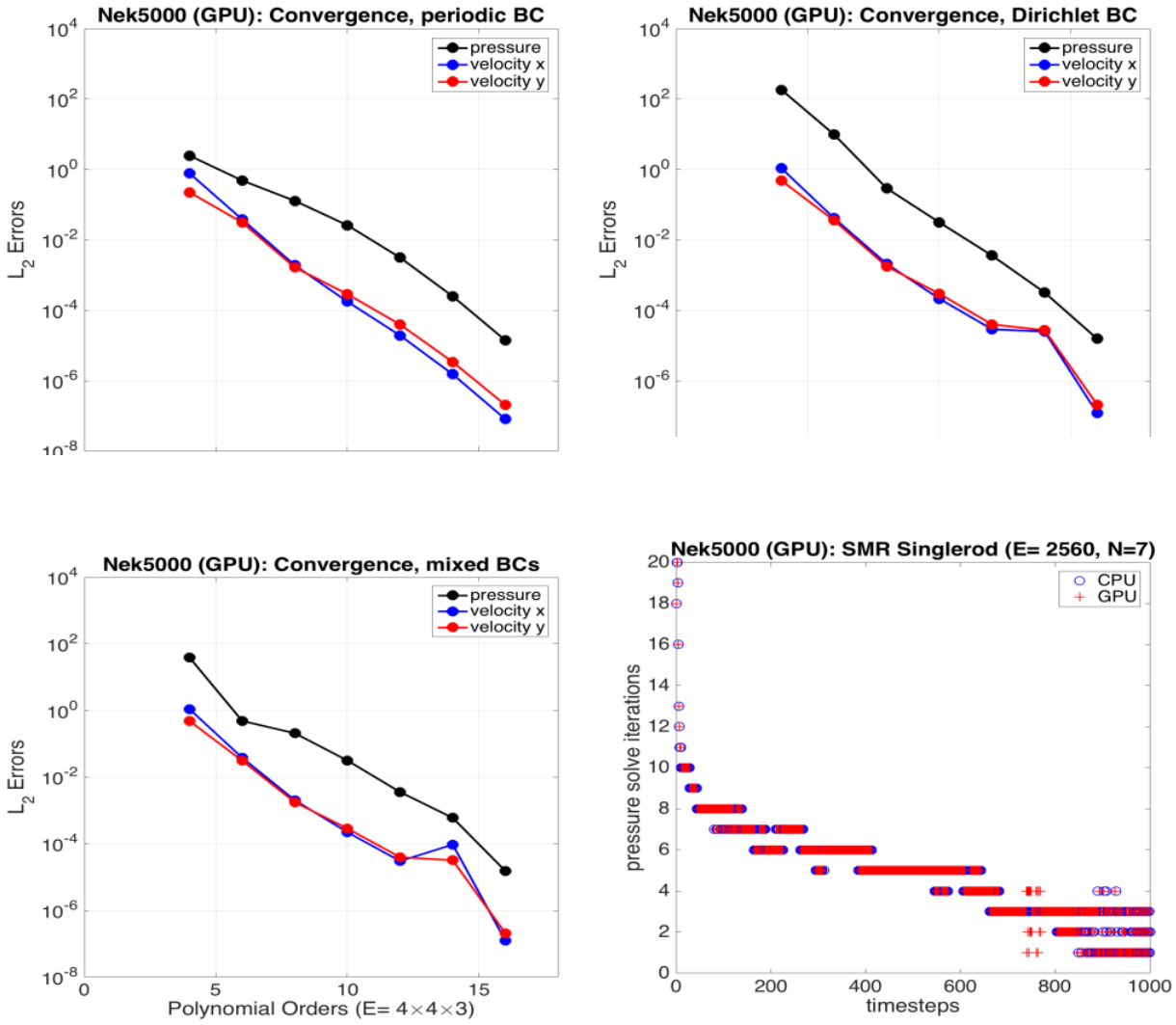

Figure 3: Nek5000 fully GPU-ported runs with validation for different boundary conditions for 3D eddy simulations, demonstrating spectral convergence as increasing $N = 3, 5, 7, 9, 11, 13, 15$ with $E = 4 \times 4 \times 3$ at time step 50; A single-rod mesh with $N = 7$ and $E = 2560$ demonstrating pressure solve iteration counts for 1000 step runs with validated results agreeing up to 12–14 digits between CPU vs. GPU.

synchronization primitives, such as thread synchronization and atomic operations, whereas OpenACC does not. Efficient implementations of applications may depend on the availability of software-managed on-chip memory, which can be used directly in CUDA and OpenCL, but not in OpenACC. These differences may prevent full use of the available architectural resources, potentially resulting in inferior performance when compared to highly tuned CUDA and OpenCL code. As a future strategy, we investigate highly optimized kernels based on the OCCA abstraction, which can produce low-level CUDA, OpenCL, or OpenMP code within a unified flexible framework. The potential for high performance using this strategy is discussed in detail in Section 4.1.

2.2.2 Nek5000 GPU/CPU Verification and Performance on ExaSMR Tests

The Nek5000 port has been developed on single-GPU platforms at ANL and extensively tested in multi-GPU settings on Titan and Summit-Dev at ORNL. Verification has included extensive comparisons to analytical solutions and to the baseline Nek5000 code for single- and multi-rod simulations relevant to the ExaSMR application. Figure 3 demonstrates verification of the GPU port for analytical solutions (3D extensions of Nek5000's *eddy* case) for the Navier-Stokes equations with Dirichlet, Neumann and mixed boundary

conditions. Spectral (exponential) convergence is observed at time step 50 with increasing N for a mesh with $E = 4 \times 4 \times 3$ elements. In Figure 3, we also demonstrate that the pressure solve iteration counts are in agreement between the GPU and CPU versions for 1000 time steps of a single-rod mesh with polynomial order $N = 7$ and $E = 2,560$ elements. In all cases, the GPU version agrees to within 12–14 digits with the standard Nek5000 baseline.

Table 1: Nek5000 single-rod run timings on P100 vs. Intel Xeon GHz (JLSE neddy/maud).

single-rod	Comparison	
	1 GPU	16 CPU Cores
total elapsed time	1.64117E+01 sec	1.73326E+01 sec
total solver time incl. I/O	1.06078E+01 sec	1.18436E+01 sec
time/timestep	2.12157E+00 sec	2.36871E+00 sec

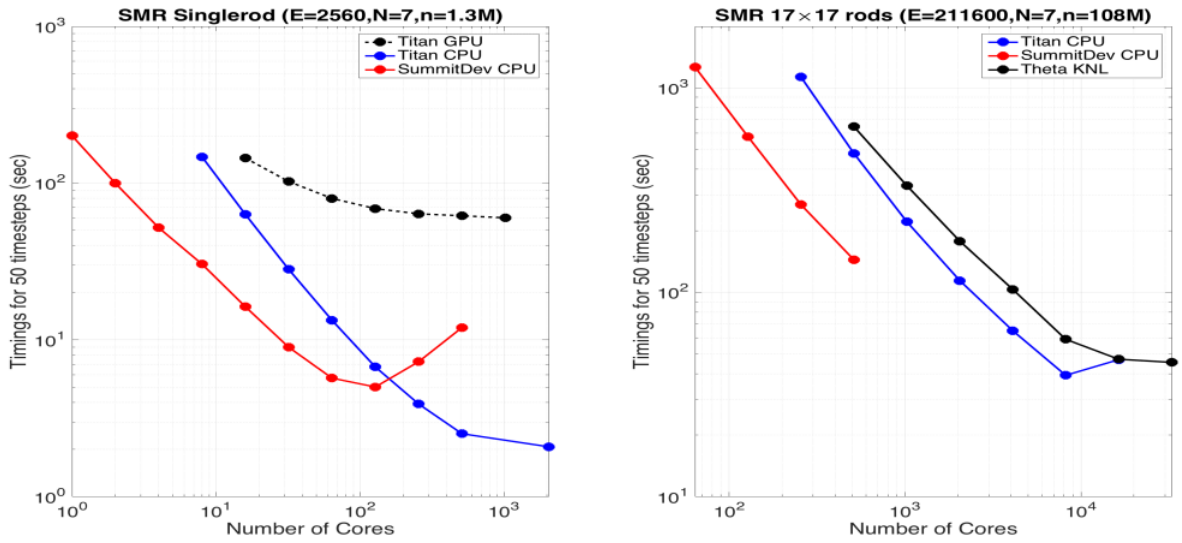

Figure 4: Nek5000 performance on SummitDev, Titan, and Theta.

Performance tuning of the GPU port is continuing on the platforms listed in Table 2. Current peak performance for the OpenACC kernels is a few hundred GFLOPS on an NVIDIA P100 (ALCF JLSE *maud*) demonstrating 90% of the speed of 16 Intel Xeon 2.2 GHz cores (ALCF JLSE), as shown in Table 1. By contrast, *highly-tuned* OCCA-driven CUDA kernels are exceeding 1 TFLOPS on the same GPU. (See Section 4.1). The next round of Nek5000-GPU development will involve using the OCCA-based kernels being developed by the group of Warburton at Virginia Tech under CEED.

Initial GPU+MPI simulation results with Nek5000 are shown in Figure 4, along with several CPU+MPI results. At this point, the GPU results are still slower than the parallel CPU runs, which have benefited from decades of development and tuning. We have identified several areas that need to be addressed in the Nek5000 GPU port which include: reductions in the number of host-device transfers, functional implementation of essentially all compute-intensive kernels (e.g., GPU-based advection is not properly functioning at this moment), and running sufficiently large (local) problems to keep each GPU busy and amortize MPI overhead. From Figure 4 (left), it is clear that there is not enough work to saturate the (current) GPU implementation in the single-rod case. (This latter point has been explored in the context of our earlier work with NekCEM on Titan.) Technical issues have kept the GPU version from functioning for the 17x17 rod case on SummitDev in GPU mode, so timings are not yet available for the larger cases.

In addition to the holistic approach of having a functional GPU-based version of Nek5000 that is consistent

(to very high precision) with the baseline version, we have pursued development and tests of highly performant kernels with several implementations of OpenACC. For example, one of the most important is the Poisson kernel (BP3.5, described in Section 4.1). Working with NVIDIA and ORNL engineers, we were able to get ∼200-300 GFLOPS for this kernel (GPU only, no MPI). A key factor in boosting performance was to lock in array dimensions at compile time, rather than leaving the (leading) array dimensions as run-time variables. This change was straightforward in Nek5000 because the polynomial order is fixed at compile time. For some kernels, this change yielded as much as a six-fold performance gain.

Despite these gains, it became clear that OpenACC might not be able to deliver the high fraction of peak that we are seeking on the GPU and some approach affording more fine-grained control is in order. Our choice will be OCCA, because it does not lock us into vendor-specific code (e.g., CUDA). Even with such fine-grained control, high-performance is not automatic. As described in Section 4.1, many iterations of development were required to yield TFLOPS/s performance.

2.3 Other ECP Applications

2.3.1 Urban

Urbanization is one of the great challenges and opportunities of this century, inextricably tied to global challenges ranging from climate change to sustainable use of energy and other natural resources, and from personal health and safety to accelerating innovation in metropolitan communities. Enabling science- and evidence-based urban design, policy and operation will require discovery, characterization and quantification of the inter-dependencies between major metropolitan sectors. Central to these inter-dependencies is human activity—decisions driven by social and economic factors—underpinning much of the use and development of cities.

The Urban team, in collaboration with the CEED team, explore the coupling of LES and building energy demand models, considering the building envelope as the boundary while current LES models operate at resolutions too coarse for use to examine the heat flow in urban canyons and how building designs and other factors related to urban form impact that flow. Using Nek5000 as the LES code, we have ongoing scoping simulations with grid sizes ranging from 1 to 100 meters square buildings, evaluating data flow characteristics and mesh requirements particularly with respect to further coupling with the building energy models that can be efficiently marched in time. In order to overcome the vast disparity of spatio-temporal scales for a target exascale simulation, we plan to mesh a sufficiently small but representative urban flow domain and compute a reference solution with maximum fidelity and resolution possible at a current leadership platforms. Then we can use this simulation to compare with a faster-turn-around but reduced fidelity models that can be further applied to larger urban landscape geometries.

As a first step toward acquiring the reference simulation, we have obtained preliminary simulation results with Nek5000 for airflow around the Lake Point Tower building geometry. Figure 5 shows velocity magnitude slices for a case where wind blowing from the lake shows regions of large sheer and velocity gradients which require further mesh improvements for an efficient computation at scale.

On a separate note relevant to target exascale simulations, the CEED team's effort in FY17 Q4 included significant performance improvement in I/O kernels, supporting large element counts ($> 15M$ elements).

Table 2: Compute systems overview for ExaSMR/Nek5000 runs

	SummitDev	Cray XK7 Titan	Cray XC40 Theta	JLSE neddy/maud
Processors	IBM Power9	AMD Opteron	Intel Xeon Phi	Intel Xeon
Nodes #	54	18,688	3,624	2
CPU cores #	1,080	299,008	231,935	32
	20 cores/node	16 cores/node	64 cores/node	16 cores/node
CPU clock rate	2.5–5 GHz	2.2 GHz	1.3 GHz	2.2 GHz
	NVIDIA P100 NVLink	NVIDIA K20X		NVIDIA P100 PCI-E
GPUs #	208	18,688	–	2
	4 GPUs/node	1 GPU/node	–	1 GPU/node
GPU clock rate	1.4 GHz	732 MHz	–	1.3 GHz

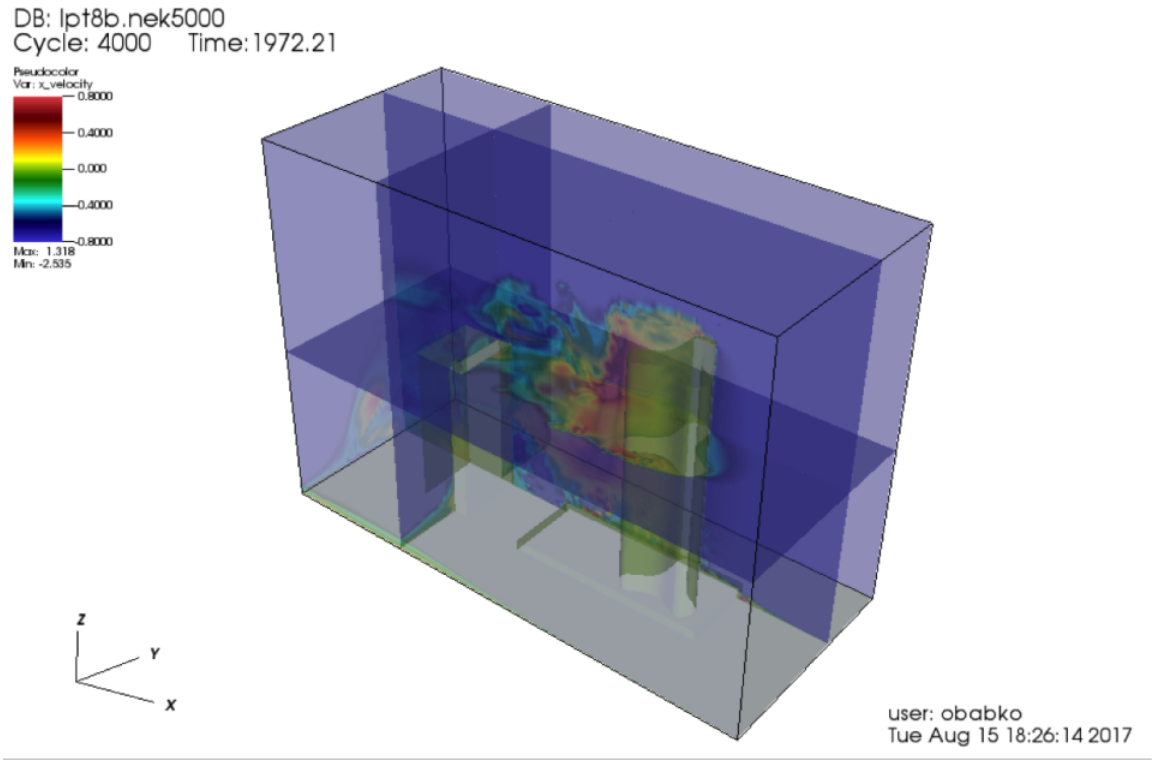

Figure 5: Preliminary LES results for a reference dataset of the flow over a small but representative landscape that includes the Lake Point Tower building geometry.

Further improvements and testing is ongoing.

2.3.2 ACME

The ACME-MMF project is developing a super-parameterized configuration of the DOE ACME Earth System Model, targeting the super-parametrization subcomponent modeling and simulations on future exascale architectures. One of the most compute expensive subcomponent includes the atmosphere dynamical core, HOMME, that is based on the spectral-element method, and ACME-MMF team plans to benefit by advances developed by the CEED project in high-order-element-based methods to achieve exascale performance and performance portability. Ongoing discussion identifying potential collaboration between ACME-MMF and CEED team include the following.

- Flow on a manifold. One of the most effective ways of dealing with spherical geometry in atmosphere models, and the approach taken by HOMME, is to use a 2D+1 approach. This approach separates the horizontal (surface of the sphere directions) from the vertical/radial direction and uses a 2D spectral element discretization of spherical shells, coupled to a 1D finite-difference discretization in the vertical. For a spectral element library, the key requirements are the ability to include metric terms (curvature) in the map from the physical element to the reference element while the physical element is a 2D element that lives in R^3, while the reference element lives in R^2.
- For efficiency, with a 2D+1 approach, ideally the vertical direction should be included as an array index for all state and field variables within each element, but not included with the various metric terms and horizontal derivatives that do not depend on the vertical coordinate. Vertical operators are relatively straightforward and could be implemented in the application or in the CEED library.

- Support for general curvilinear maps where the most natural map for spherical geometry is projection from the center of the sphere, which is non-orthogonal.
- Support efficient kernels for horizontal differential operators: HOMME relies on *div*, *grad*, and *curl* operators in strong form (nodal values), separate routines for their integrated-by-parts form and an integrated-by-parts scalar Laplacian and vector Laplacian.

2.3.3 ExaAM

The ECP Exascale Additive Manufacturing Application Project (ExaAM) couples multi-physics and multi-codes in an effort to use exascale computing to enable better qualification of additive manufacturing (AM) built parts. Fundamentally, this means coupling microsctructure development and local property analysis with process simulation. The ExaAM code with the physics required for the local property analysis is an export controlled code, and with the goal of creating an open source software platform for AM simulation, the ExaAM team explored the space of mini and proxy apps, and none were found suitable. The ExaAM team decided to create a new miniapp specifically for local property analysis in collaboration with the CEED team. The initial release (**ExaConstit**) is planned to remain ExaAM's miniapp for this application area and will become the foundation of a prototype application as required physics is added for local property analysis for AM materials. Significant integration between the ExaAM and CEED ECP activities is planned during this process. The new ExaConstit miniapp is currently available at `https://github.com/mfem/mfem/tree/exaconstit-dev/miniapps/exaconstit` and, after approval, will be part of the MFEM release on GitHub.

2.3.4 GEOS

The ECP Subsurface Simulator application (ADSE05) utilizes LLNL's GEOS code to model large scale deformation and flow, and LBNL's Chombo-Crunch code to pore scale flow and geochemistry in an effort to solve reservoir scale problems that also require direct simulation of pore scale physics. GEOS is a general purpose simulation framework that facilitates various numerical schemes such as the Finite Element Method and the Finite Volume Method, and has had success modeling the coupled flow, deformation, and fracture involved in the hydraulic stimulation of a tight shale reservoir. The approach to modeling fracture through mesh topology change has distinct advantages but also carries limitations, such as difficulties utilizing higher order discretizations. Although CEED is primarily an effort focused on higher order methods, the organization of various computational kernels that arise from the Finite Element Methods into a collection is potentially useful for any finite element code. As such, the GEOS-ECP team intends to utilize the CEED kernel library to organize a collection of low order *super-element* kernels that take advantage of the reduced memory demands of a higher order element while maintaining the topological flexibility of a low order discretization. The CEED and GEOS teams are also exploring the possibility of developing a simple MFEM-based miniapp for subsurface flow.

2.3.5 Combustion

DNS and LES of combustion problems is of major importance to the DOE mission. In collaboration with the Energy Systems Division at Argonne and Aerothermochemistry and Combustion Systems Laboratory at ETH Zurich, we have extended the arbitrary Lagrangian Eulerian (ALE) formulation in Nek5000 to support characteristics-based time-stepping that allows simulations to exceed standard Courant-Friedrichs-Lewy (CFL) stability constraints. In internal combustion simulations, the CFL constraint can be quite limiting during valve-open events, when the highest velocities are flowing through the intake or exhaust valve regions which typically have very fine mesh spacing. The characteristics scheme decouples advection from the pressure and viscous solves of the associated unsteady Stokes problem. Advection is effected through small CFL-stable time steps with no expensive pressure solves. The linear Stokes problems can be advanced with a time step that is an order of magnitude larger than the CFL-limited step size. Accounting for all the overhead, the net savings observed in recent (cold-flow) engine simulations is a factor of 3 to 4.

2.4 ECP Application Assessment

2.4.1 Nek5000 Full Application Weak and Strong Scaling

As part of the ECP Application Assessment Project with Kenny Roche (PNNL), we have performed a weak-scale study of Nek5000 on multipin geometries similar to those required for ExaSMR, as shown in Figure 6 (left). To simplify this study for a relatively large weak/strong-scale study, which involves generation of several meshes to cover the range of interest, we have eliminated the solid center of the pins in this multi-rod test case and thus focus only on the hydro plus thermal advection. (That is, we ignore diffusion in the additional elements that would make up the solid.) This simplification permits more transparent performance analysis across a wide range of processor counts and problem sizes because the algorithm is more homogeneous. Given that hydrodynamics typically dominates the solution time, this simplification can be made safely without undue compromise of the model problem's fidelity to the target exascale application.

The baseline mesh for single-rod consisted of $E_{xy} = 256$ elements in the $x - y$ plane, extrude in the z direction to yield $E = E_{xy} \times E_z = 2560$ elements. (Thus, $E_z = 10$.) For this larger study, we use anywhere up to $E = 524288$ elements, which can be realized through a combination of multiple pins and multiple layers in the axial flow direction. The 3D meshes are a tensor product of the 256-element 2D unit cell with E_z slabs in the z direction, and $C_x \times C_y$ cells in the x and y directions, respectively. In the present study, we fix the polynomial order to $N = 7$, which implies for the largest case, of $E = 2^{19}$, we have $n \approx EN^3 = 179.8$ million points. Note that we would expect a problem of this size to strong scale out to about $P = 100,000$ on BG/Q [3], where there are about n/P=2000 points per core. On other machines, strong-scale fall-off will probably be at $P < 100,000$.

Figure 6 (right) shows the weak-scale performance on BG/Q (Cetus) for several values of n/P. Two families of curves are shown. One uses the Nek5000 default fast direct XX^T-solver [4] that is used for standard small-scale production runs. The other uses algebraic multigrid [5] for the coarse-grid solver. XX^T is highly effective up to about 100,000 elements (in accord with analysis in [4], but AMG is superior beyond that point).

2.4.2 NekBench Utilities for Scaling Studies

The Nekbench repository `https://github.com/thilinarmtb/NekBench` provides scripts for benchmarking Nek5000. The user provides ranges for the important parameters (e.g., processor counts and local problem size ranges) and a test type (e.g., scaling or ping-pong test). Nekbench will run the given test in the given parameter space using a Nek5000 case file, which is also given by the user (in the ping-pong tests, the case file is optional). Nekbench is written using bash scripting language and runs any Unix-like operating system that supports bash. It has been successfully tested on Linux laptops/desktops, ALCF Theta, NERSC Cori (KNL and Haswell), and NERSC Edison machines for scaling tests.

Planned extensions for Nekbench include adding more machine types like Cetus, additional support for the ping-pong test type, and automated plot generation (e.g., scaling study graphs) for each test run.

3. MINIAPPS

3.1 Laghos

Laghos (LAGrangian High-Order Solver) is a new 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 built 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 [6]. The miniapp is available at `https://github.com/ceed/Laghos`.

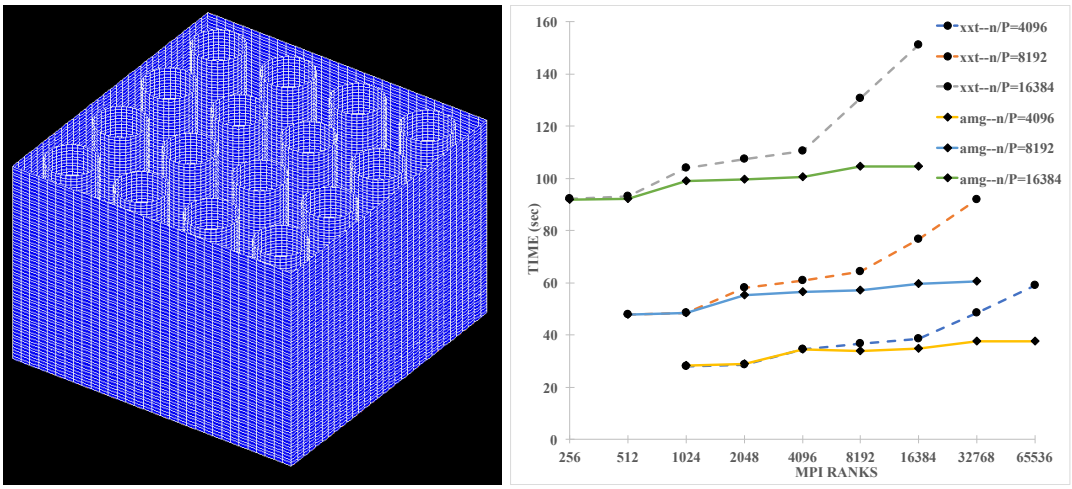

Figure 6: Nek5000 weak-scale study: (left) example of a scalable multi-rod mesh, (right) weak-scale performance on CETUS for varying problem sizes and coarse-grid solver choices.

Since it was introduced in milestone CEED-MS6, a number of additional features and improvements have been added to Laghos, including tensor-based computations of transformation Jacobians and function gradients, as well as batched calls to the equation-of-state (EOS) computation routines. More notably, a GPU port for both MFEM and Laghos was developed based on the Open Concurrent Compute Abstraction (OCCA). Some details and results from this work are discussed below. For more information about OCCA, see Section 4.1.

3.1.1 A GPU Version of MFEM Based on OCCA

Using OCCA is one of the approaches taken to enable GPU acceleration in MFEM. We were able to convert core components in the MFEM library to provide the computation flexibility from OCCA while keeping the API comparable to the original MFEM API (e.g. `Vector` → `OccaVector`). The set of features that can now be computed on devices through OCCA include:

- Vector operations (e.g. `+ - / *`) and reductions (e.g. `Min`, `l2Norm`)
- Setup of the bilinear form
- Integrators, including mass and diffusion
- Explicit solvers through templates
- Efficient implementations for elements with tensor-based basis
- Unstructured AMR and non-conforming meshes through prolongation and restriction operators
- Mixed finite element methods
- Distributed solves through MFEM's usage of Hypre
- User-defined coefficients

As an illustration of the small changes in the high-level user API, we compare the source of MFEM's original Example 1 (solving a Poisson problem):

```
FiniteElementSpace *fespace = new FiniteElementSpace(mesh, fec);
Array<int> ess_tdof_list;
Vector x, b;

...

Operator *A;
Vector B, X;

BilinearForm *a = new BilinearForm(ofespace);

a->AddDomainIntegrator(new DiffusionIntegrator(1.0));
a->FormLinearSystem(ess_tdof_list, x, b, A, X, B);

CG(*A, B, X, 1, 500, 1e-12, 0.0);
```

with the OCCA-ported version:

```
occa::setDevice("mode: CUDA, deviceID: 0")

OccaFiniteElementSpace *fespace = new OccaFiniteElementSpace(mesh, fec);
Array<int> ess_tdof_list;
OccaVector x, b;

...

Operator *A;
OccaVector B, X;

OccaBilinearForm *a = new OccaBilinearForm(ofespace);

a->AddDomainIntegrator(new OccaDiffusionIntegrator(1.0));
a->FormLinearSystem(ess_tdof_list, x, b, A, X, B);

CG(*A, B, X, 1, 500, 1e-12, 0.0);
```

In raw numbers, the OCCA port of MFEM required a change of 13k lines added and 1k lines removed, to be compared with the initial 120k lines of code in the MFEM library. Using `sloccount`, we see a change from 93k lines to 100k lines which include 2.5k lines from OCCA kernels. We were able to reduce the amount of code needed by leveraging OCCA's just in time (JIT) kernel compilation and templating. For example, `OccaVector` includes 25 methods computed in OCCA with a 2-5 line kernels such as the `OccaVector::operator +=` kernel.

```
makeCustomBuilder("vector_op_add", "v0[i] += c0;");
```

3.1.2 Laghos on OCCA

Laghos is the first miniapp to make use of the MFEM+OCCA API. Every time step of the computation found in this miniapp contains three major stages:

- Inversion of a global mass operator based on the H^1 space. This operation involves applications of the mass operator for every CG iteration.
- Application of a force operator, using both H^1 and L_2 spaces. The force operator is applied two times per time step.
- Computation of physical stresses at all quadrature points.

We leveraged the `OccaMassIntegrator` that was already ported in MFEM, but created custom kernels for the force operator and quadrature point updates. For the force operator and stress updates, distinct kernels were written for combinations of 2D and 3D elements, as well as for CPU and GPU architectures.

Below, in Figures 7–12, we present a preliminary comparison between the performance of the CPU Laghos code (original CPU version without OCCA) and the OCCA GPU implementation. For the CPU case, we test both the *full* and *partial* assembly options, denoted by FA and PA in the figures. We perform one time step of the Taylor-Green test case, which does not involve any discontinuities in the solution fields (test on the Sedov blast problem will be performed soon). Execution rates for all three major stages are presented, in 2D and 3D. Each label denotes the finite element spaces used for the corresponding result set, e.g., Q4Q3 means continuous kinematic space of type Q_4 and discontinuous thermodynamic set of type Q_3 on each cell.

All tests were performed on the Ray machine at LLNL, where each node has two IBM Power8 CPUs, each with 10 cores with 8 threads per core, and 4 Tesla P100 GPUs. All CPU runs were performed on one node using 16 and 64 MPI tasks in 2D and 3D, respectively. The GPU calculations use only one of the GPUs.

In Figures 7 and 8 we see that the CPU PA option is more beneficial than the CPU FA for orders 3 and above. It is important to keep in mind that the CPU FA applies a precomputed sparse matrix without any assembly operations (they are not included in the timing). The OCCA code is currently slower than the CPU partial assembly but, based on the BP2 results in CEED-MS6, we beleive that this performance can be improved significantly.

Figures 9 and 10 are related only to the application of the force operator; the computation of the data needed to form it is not included in the timing. The OCCA version is the best choice, performing better than the PA for every order and achieving about 800 MDOFs. Focusing on Figure 9, we see that we get about an order of magnitude improvement from a better algorithm (PA vs FA) and another order from the hardware (going from 16 cores to 1 GPU).

Figures 11 and 12 show that the OCCA code is also the best option for quadrature-based computations, exploiting the fact that these calculations do not involve any communication.

Figures 13 and 14 compare the total execution time of the three main kernels for the FA and PA options. The show that PA is the clearly preferred options for all orders except order 1 in 3D.

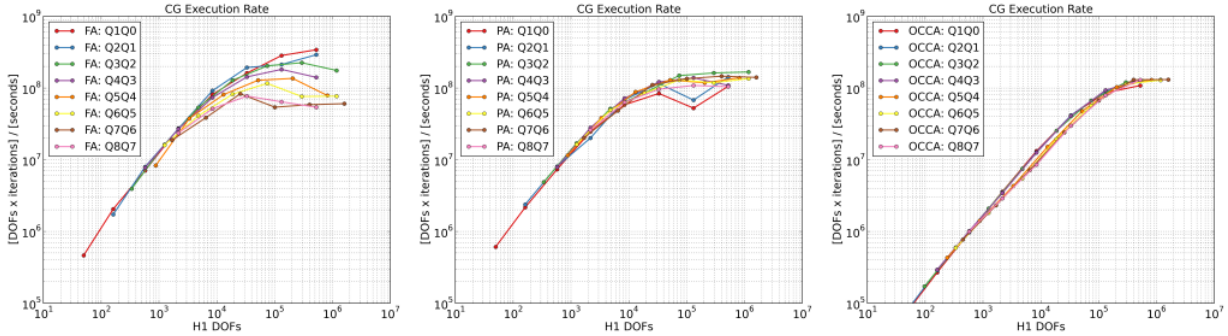

Figure 7: Laghos: 2D inversion of the global mass operator.

3.2 Nekbone

3.2.1 Introduction and Algorithm

Nekbone is a lightweight subset of Nek5000 that is intended to mimic the essential computational complexity of Nek5000, relevant to large eddy simulation (LES) and direct numerical simulation (DNS) of turbulence in complex domains, in relatively few lines of code. It allows software and hardware developers to understand the basic structure and computational costs of Nek5000 over a broad spectrum of architectures ranging from software-based simulators running at one ten-thousandth the speed of current processors to exascale platforms running millions of times faster than single-core platforms. Nekbone has weak-scaled to 6 million MPI ranks on the Blue Gene/Q Sequoia at LLNL while Nek5000 has strong-scaled to over a million ranks on the Blue Gene/Q Mira at ANL. Nekbone provides flexibility to adapt new programming approaches for scalability and performance studies on a variety of platforms without having to understand all the features of Nek5000,

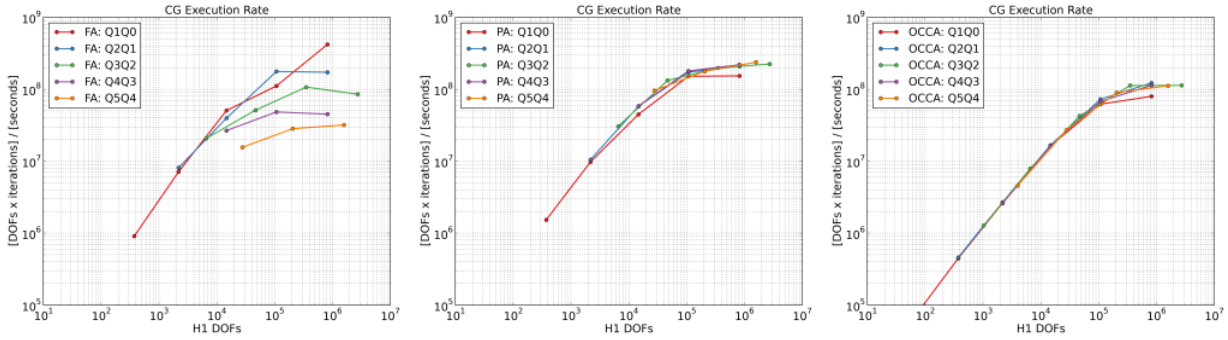

Figure 8: Laghos: 3D inversion of the global mass operator.

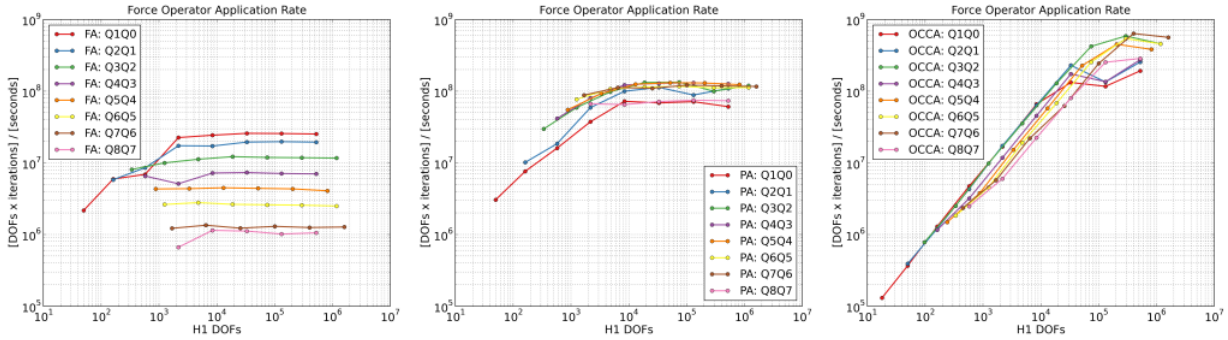

Figure 9: Laghos: 2D application of the force operator.

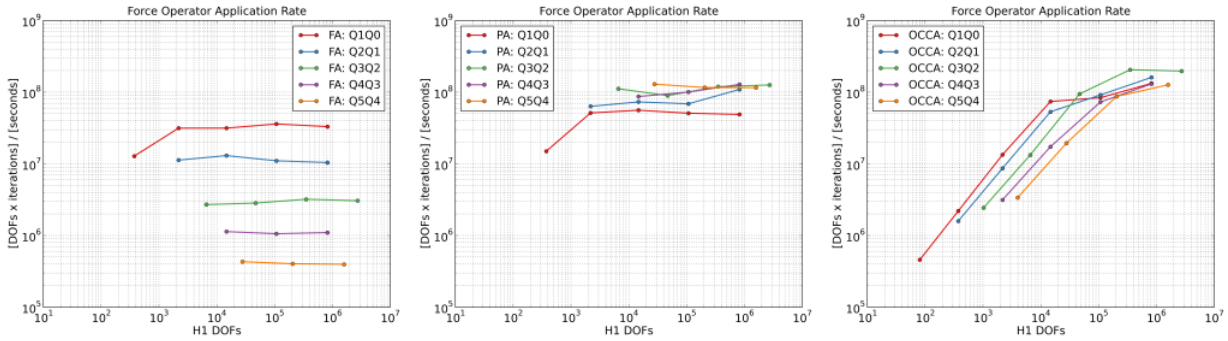

Figure 10: Laghos: 3D application of the force operator.

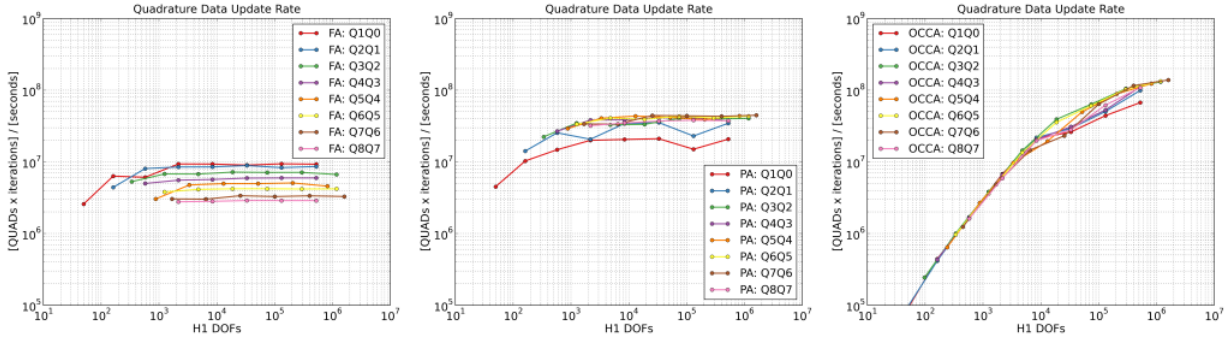

Figure 11: Laghos: 2D update of quadrature data.

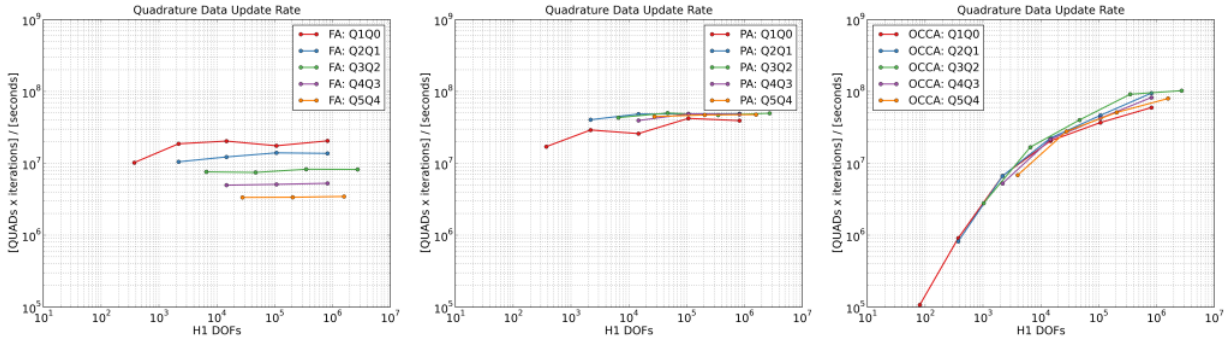

Figure 12: Laghos: 3D update of quadrature data.

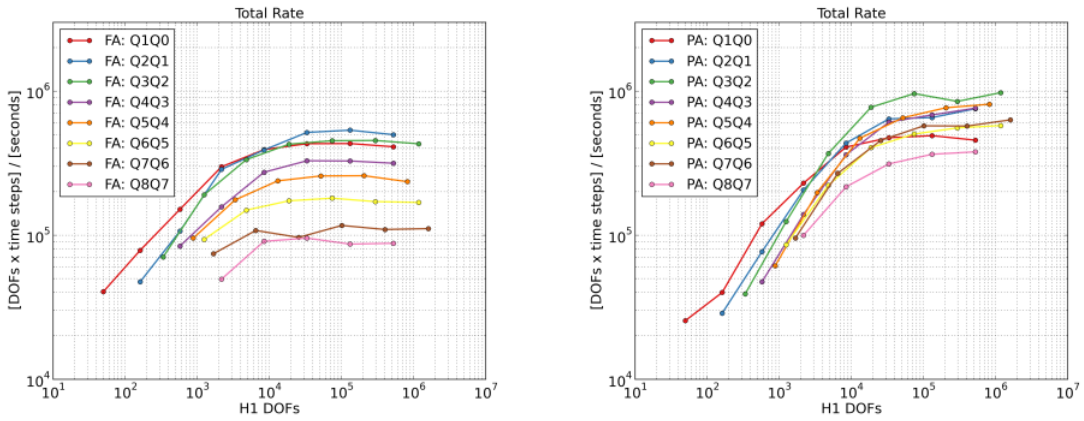

Figure 13: Laghos: 2D total execution rates.

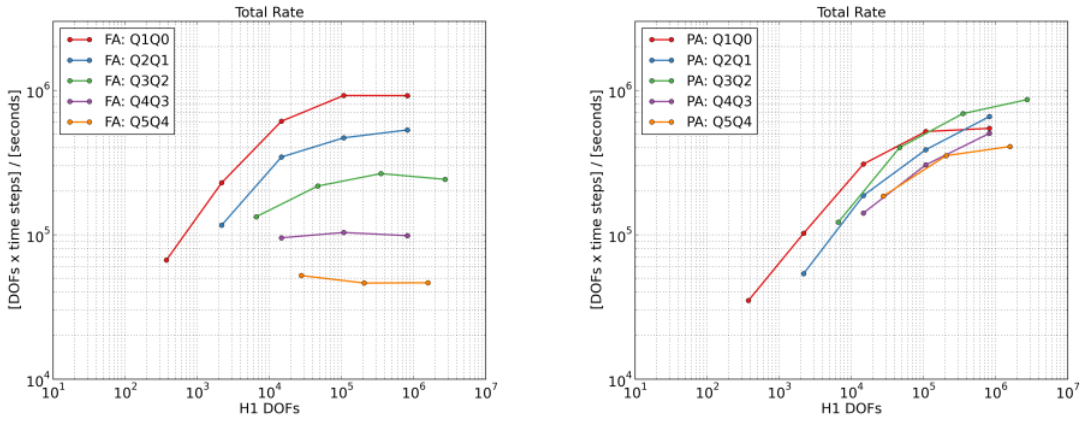

Figure 14: Laghos: 3D total execution rates.

currently supporting OpenACC/CUDA-based GPU variants. In CEED, Nekbone serves as a proxy for a sub-component of the ExaSMR application. It is also used as part of the CORAL and CORAL-2 acceptance suite. The Nekbone miniapp is available at `https://github.com/Nek5000/Nekbone` as well as at the CEED mirror `https://github.com/ceed/Nekbone`.

The core of Nekbone is the conjugate gradient (CG) solver, which includes domain decomposition and MPI communication for simple brick-like and linear arrangements of elements. The number of elements and polynomial order can be varied. While newer versions of Nekbone also represent a multigrid preconditioner, we present results from an established version without the multigrid preconditioner.

Table 3: Performance model of CPU Nekbone

Subroutine	DP Loads	DP Stores	DP Ops	q
glsc3	$3n$	0	$3n$	0.125
add2s1	$2n$	n	$2n$	0.083
local_grad3	n	0	$6nn_x$	$0.75n_x$
wr_ws_wt	$6n$	0	$15n$	0.26
local_grad3_t	0	n	$6nn_x + 2n$	$0.75 + 0.25n_x$
add2s2	$2n$	n	$2n$	0.083

Algorithm 2 summarizes the subroutines in Nekbone's CG solver. "Local" subroutines refer to operations performed on elements on a single subdomain (and hence a single MPI rank), and "global" subroutines refer to operations performed for all elements in the full domain (and hence across all MPI ranks). In both Nekbone and Nek5000, MPI communication is performed using the highly optimized *gslib* library, which outperforms many implementations of MPI collective operations.

Algorithm 2 Conjugate Gradient Solver in Nekbone

```
1:  for i ← niter do
2:      solveM                          ▷ Computer preconditioner
3:      glsc3                           ▷ Global inner product
4:      add2s1                          ▷ Local A = cA + B
5:      for e ← nelt do                 ▷ Local matrix-vector product
6:          local_grad3                 ▷ Local small matrix mults
7:          wr_ws_wt                    ▷ Local accumulation
8:          local_grad3_t               ▷ Local adds and small matrix mults
9:      end for
10:     dssum                           ▷ Global gather-scatter
11:     add2s2                          ▷ Local A = A + cB
12:     glsc3                           ▷ Global inner product
13:     add2s2                          ▷ Local A = A + cB
14:     add2s2                          ▷ Local A = A + cB
15:     glsc3                           ▷ Global inner product
16: end for
```

In the CPU version of Nekbone and Nek5000, local operations across elements in each MPI rank are done in serial. On traditional CPUs, this has been a beneficial implementation. Because MPI communication is minimal, running at the the strong-scaling limit (very few elements per rank) is the *modus operandi*, and parallelizing over so few elements per rank provides little or no performance benefit. However, on many-core architectures (such as Intel KNL and NVIDIA GPUs), running at the strong-scaling limit is no longer optimal and parallelizing local operations becomes essential.

Performance Model Table 3 shows hand-counted double-precision (DP) loads, stores, and arithmetic operations, based on standard assumptions of caching on a modern CPU. The quantity n_x is the polynomial order. The quantity n is typically degrees of freedom, but can vary depending on where the subroutine is called; regardless, it is factored out of the arithmetic intensity, q. Operations such as glsc3, add2s1, wr_ws_wt, and add2s2 have low arithmetic intensities ($q < 1$) and are generally expected to be bandwidth bound. Operations with matrix-matrix multiplication (mxm) components (such as local_grad3 and local_grad_3t) have arithmetic intensities that are dependent on the polynomial order, n_x, and we still expect these to be FLOP bound in all practical cases. For polynomial order $n_x = 2$, we expect $q = 1.5$ and $q = 1.25$ for local_grad3 and local_grad_3t. For $n_x = 16$, we likewise expect $q = 12$ and $q = 4.75$.

Referring to Table 4, we see that the machine balance (the ratio of FLOP rate and bandwidth) for KNL is 3.9, whereas the balance for P100 is 7.8. According to a naive roofline model, this means that an algorithm with arithmetic intensity of 3.9 or greater will reach peak FLOPs on KNL, and an algorithm with an arithmetic intensity of 7.9 or greater will reach peak FLOPs on P100. Considering this, the CPU

performance model in Table 5 demonstrates that in general, mxm operations (local_grad3 and local_grad_3t) will only approach peak FLOP/s for higher polynomial orders, since the arithmetic intensity of mxm is proportional to polynomial order. Finally, this model demonstrates that, on GPU compared to KNL, the polynomial must be higher to reach peak on-node FLOP/s.

This performance model alone does not offer insight into performance with respect to number of elements per node. The following tests provide some insight into this question for the particular case of KNL using OpenMP. Figure 15 shows the CG solve time with respect to degrees of freedom per node for 100-timestep problems with polynomial order 7 on an Intel KNL (one node of Theta). The implementations include:

- Serial/OpenMP execution on KNL
- Serial/OpenMP on KNL using a DGEMMs expressed as unrolled loops
- Serial/OpenMP on KNL using vendor-optimized libxsmm for matrix-vector products

From Figure 15, we see that at least $n = 2^{20}$ gridpoints are required before a single node of KNL, running 64 OpenMP threads, demonstrates linear scaling. Below this number, the runtime is flat, i.e., not decreasing with reduced problem size, because the resources are not being fully utilized. If we denote by P the number of threads (or MPI ranks in the all-MPI case), this value of $n/P = 16384$ is consistent with but larger than what is found to be required on BG/Q, where the performance roll-off is (conservatively) at $n/P \approx 8000$ [3]. More details on this analysis are presented here: `https://asc.llnl.gov/DOE-COE-Mtg-2016/talks/1-13_Parker.pdf`.

3.3 CORAL-2 Benchmarks and Vendor Interactions

The CEED miniapps are designed to be used as CEED's main vehicle in a variety of co-design activities with ECP vendors, software technologies projects and external partners.

A major highlight in this direction is that two of the CEED high-order miniapps: Nekbone, and the newly developed Laghos, were selected to be part of the CORAL-2 procurement benchmark suite, see `https://asc.llnl.gov/coral-2-benchmarks`. In addition, Nekbone, Laghos and HPGMG are now part of the proxy app list `https://exascaleproject.github.io/proxy-apps/all-apps` maintained by the ECP Proxy App project.

The CEED team has been using the Nekbone and Laghos miniapps, as well as the CEED benchmarks, `http://ceed.exascaleproject.org/bps` to engage researchers in Intel, Cray and AMD, as well as the international high-order community. We sent multiple representatives to Intel's first hackathon in Hudson, MA (where we ran kernels from these miniapps on their simulators) and will be represented in all of the Cray, Intel and AMD deep-dives coming up in October 2017. External researchers from the deal.ii project have picked up and run our proposed bake-off problems (BPs), see `https://github.com/kronbichler/ceed_benchmarks_dealii`.

Table 4: Comparison of KNL and P100 Statistics

Metric	KNL	P100
Cores / SMX	64	56
SIMD DP Width	8	32
Clock Speed [GHz]	1.1	1.3
Peak f32 [TFLOP/s]	4.5	9.3
Peak f64 [TFLOP/s]	2.3	4.7
Peak DGEMM [TFLOP/s]	1.9	4.5
Memory (IPM) [GB]	16	16
IPM Bandwidth [GB/s]	$\tilde{6}00$	732
STREAM Bandwidth [GB/s]	488	574
L1 Cache per core / SMX [KB]	32(D), 32(I)	64
L2 Cache [MB]	32	4
TDP[W]	215	250

Table 5: Performance Analysis of Nekbone on KNL

Subroutine	Time (s)	% Solve time	BW (GB/s)	% Peak BW	GFLOP/s	% Peak GFLOP/s
glsc3	1.56E-1	20.2	351.2	72	43.9	
wrwswt	1.28E-1	16.5	314.1	64	98.1	
add2s2	1.23E-2	16.02	480.3	98	40.0	
local_grad3_t	1.09E-1	14.1			754.4	40
gs_op	8.89E-2	11.5				
local_grad3	8.31E-2	10.7			969.6	51
add2s1	4.23E-2	5.4	475.6	97	39.6	

Table 6: Performance Analysis of Nekbone on GPU

Subroutine	Time (s)	% Solve time	BW (GB/s)	% Peak BW	GFLOP/s	% Peak GFLOP/s
local_grad3	3.35e-1	33.1	187.01		278.22	6
+ wrwswt						
local_grad3_t	2.96e-1	29.3	85.04		272.00	6
glsc3	9.23e-2	12.4	545.31	95	109.42	
add2s2	1.09e-1	10.8	512.36	89	46.20	
gs_op	5.56e-2	5.5				
add2s1	3.61e-2	3.6	512.81	89	46.52	

4. GPU PERFORMANCE OPTIMIZATION

4.1 OCCA

In this section, we analyze the implementation strategies for spectral element operators on GPUs. We use three benchmark problems taken from the CEED benchmark suite that are representative for a wide class of operators arising in spectral and high-order finite element methods. While there is no doubt that the GPUs are good accelerators, designing code that uses the GPU to its full potential requires deeper understanding of both the problem and the hardware architecture. We guide the reader through the benchmark problems, the challenges associated with each problem, and through the implementation details and fine-tuning of the code. using the Open Concurrent Compute Abstraction (OCCA) [7]. As a result of the code optimization process, our GPU code reaches an empirically defined memory roofline bounds for all benchmark problems.

The choice of OCCA for this initial exploration is motivated by the fact that it offers several important advantages in the context of CEED applications:

- It exposes all performance critical features of CUDA required for finite element calculations, allowing us to experiment with different kernel implementations and achieve CUDA-like performance (as illustrated in this section).
- It offers runtime compilation of compute kernels with JIT specialization and optimization, which is particularly important for high-order methods where innermost loops have bounds depending on the order.
- It allows developers to write kernels in a portable code language that is simple to understand
- Unifies and simplifies interacting with CUDA, OpenCL, and OpenMP backends.
- Can be called from both C++ (MFEM) and FORTRAN (Nek), and kernels can be shared between languages.
- It is lightweight and very portable (60k lines of C++) so it can be used only internally in CEED tools, without requiring changes in the application.

The rest of this section is organized as follows: in Section 4.2 we provide a review of past efforts in GPU optimization of key finite element kernels. Section 4.3 introduces the notation used in the benhcmark

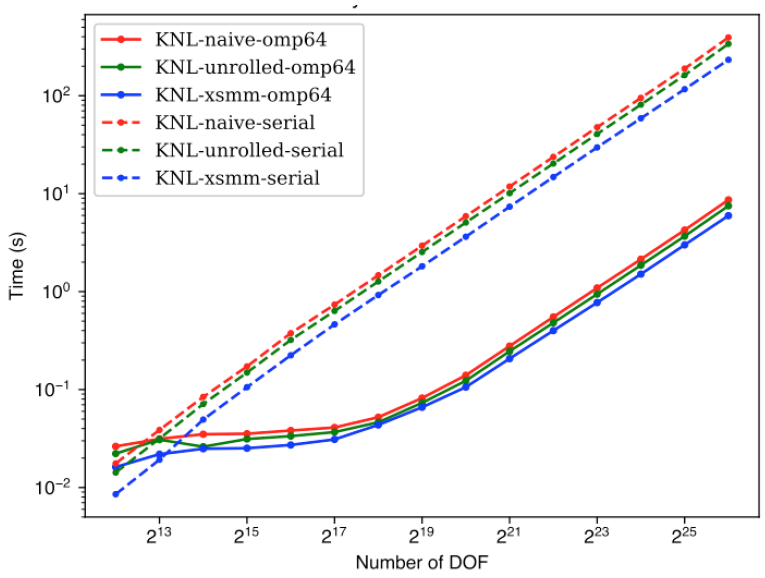

Figure 15: Nekbone OpenMP results on a single node of Theta (KNL).

problems described in the follow-on sections as follows: in Section 4.4, we introduce the first benchmark problem (mass matrix multiplication, BP1.0) and discuss the implementation and optimization choices; in Section 4.5, we discuss the second benchmark problem (collocation differentiation, BP3.5); and in Section 4.6, we analyze the third benchmark problem (differentiation involving interpolation and projection operations, BP3). We conclude the section by summarizing the GPU results and discussing future work in Section 4.7.

All kernel tuning was performed on an NVIDIA P100 PCI-E 12GB model. The fact that OCCA exposes a sufficiently comprehensive amount of CUDA functionality, allows us to fine tune the benchmark kernels for nearly ideal performance when compared with an empirical roofline model.

4.2 Background in GPU Acceleration of Finite Element Computations

Even before the era of easily programmable, general-purpose Graphic Processing Units, GPUs have been used to speed up finite element codes. As early as 2007, Göddeke had solved 2D elliptic PDEs using a cluster equipped with GPU accelerators [8, 9], and during the next ten years, there have been a lot advancements in implementation strategies.

Research generally focuses on accelerating specific parts of the computation and various aspects of the the solution process, such as global assembly or the solver. For example, Cecka et al [10] targets the global assembly phase of the finite element method (FEM), and shows how to optimize this phase for the GPU execution. Markall et al [11] also concentrates on the global assembly phase. The authors emphasize that the choice of the most efficient algorithm strongly depends on the available hardware and selected programming model. The authors consider both the GPU and the CPU hardware, and both OpenCL and CUDA parallel implementation. Some authors argue that making the code portable between different threading systems (such as many-core and multi-core architectures) requires a high-level language, see Markall et al [12, 11]. Other researchers focus mostly on the solver part; in the paper by Göddeke et al [13] the authors use GPUs to accelerate Navier-Stokes solver for (lid) driven cavity and laminar flow over cylinder.

Few authors target the entire FEM pipeline using a canonical problems, i.e., Fu et. al. [14] used elliptic Helmholtz problem to show how to port the code to the GPU. The paper discusses strategies for accelerating CG and multigrid solver. Another class of papers are those devoted to the GPU implementation of finite element discontinuous Galerkin (FEM-DG) methods. In [15], Klöckner et. al. applies FEM-DG to solve hyperbolic PDEs on the GPUs. The work is continued by Klöckner and coauthors in [16], where the details of implementation are provided. Remacle et. al. [17] shows a finite element scheme for solving elliptic equations for unstructured all-hex meshes on the GPUs. The authors also analyze the performance of the scheme using an off-the-shelf GPU. Chan et. al. [18] constructs similar analysis for wave equation using hybrid meshes.

The cost of the matrix-vector multiplication typically accounts for the highest cost of the elliptic FEM solver; see the work of Remacle et. al. [17, Table 2, Table 3]. Several authors target optimizing just this part of the solution process. For example, in the papers by Dehnavi et. al. [19] and Grigoraş et. al. [20] we find detailed description of implementation strategies for the matrix-vector product. Also, in [17], Remacle et. al. put high emphasis on optimizing matrix-free matrix-vector multiplication.

We have also concentrated on tuning GPU implementation of matrix-vector products arising in the CG solver for elliptic FEM. In our case, we perform this operation elementwise, and we never assemble the global matrix, which is known as *matrix-free matvec* in the literature. There are many reasons why this matrix-vector product is not easy to optimize on the GPU. First, by design we must perform the matrix-vector product in stages with thread synchronization in between, and synchronization is expensive on the GPU. Second, we need to allocate additional memory to store the intermediate results. For higher-order polynomials, doing so is troublesome because of the limited size of the on-chip memory. Third, we load and store a lot of data, especially if using geometric factors. Fourth, interpolation to a different set of nodes means that we are accessing memory in an irregular fashion, which might cause bank conflicts.

4.3 Notation

In this section, we use the following notation.

Symbol		
Code	**Math**	**Meaning**
`N`	N	degree of the polynomial used in the interpolation
`Nq`	N_q	number of Gauss-Lobatto-Legendre (GLL) nodes in 1 dimension. Note: $N_q = N + 1$
`Np`	N_p	number of nodes in an element: $N_p = N_q^3$
`glNq`	N_q^{GL}	number of Gauss-Jacobi (GJ) nodes in 1 dimension. Note: $N_q^{GL} = N + 2$
`gjNp`	N_p^{GL}	number of Gauss-Jacobi nodes in an element. $N_p^{GL} = \left(N_q^{GL}\right)^3$
`I`	$\mathbf{I}$	$(N+1) \times N$ interpolation matrix used to interpolate GLL nodes to GJ nodes
`D`	$\mathbf{D}$	$(N+1) \times (N+1)$ differentiation matrix used in BP1.0, see (3)
`gjD`	$\mathbf{D}^{GL}$	$(N+2) \times (N+2)$ differentiation matrix, used in BP3.0
`NElements`	N_{el}	Number of elements in the mesh

While listing parallel pseudocode, we use OCCA notation

- The **shared** quantifier denotes shared memory; shared variables are shared among all threads in the same thread block.
- The **exclusive** quantifier denotes a register variable; a register variable belongs to one and only one thread.
- The **occaBarrier** and **barrier** quantifiers are used to denote thread synchronization; the code after the barrier is executed only if all the threads have finished the computations preceding the barrier.
- **occaUnroll** or **Unroll** is used to denote loop unrolling.

To present our analysis and results, we use three benchmark problems that capture typical part of matrix-vector multiplication in FEM: interpolation, differentiation, and projection in various configurations. Our main focus here is on performance of these benchmark problems on the GPU. We discuss the choices one can make for the implementation, including the choices for parallel structure, memory layout, and data streaming.

4.4 BP1.0: Mass Matrix Multiplication

The first benchmark we consider consists of the matrix-vector product of a high-order finite-element mass matrix and corresponding vector.This operation is used in several kinds of implementations of finite element operators including, but not limited to, elliptic operators, projection operations, and some preconditioning strategies. This problem serves as a good initial performance benchmark as it consists of a very simple local matrix action. The operation also requires relatively little data transfers compared with more demanding differential operators which necessitates loading geometric factors from the elements.

4.4.1 BP1.0 Mathematical Formulation

Let us consider an unstructured mesh of a domain $\mathcal{D}$ into K hexahedral elements D^k, $k = 1, \ldots, K$ such that

$$\mathcal{D} = \bigcup_{k=1}^{K} D^k.$$

We construct a high-order finite element approximation of a function u on the domain $\mathcal{D}$ by forming a polynomial u^k on each element D^k. We express the polynomial u^k as a sum of basis interpolation polynomials by mapping the element D^k to a reference element $\hat{D}$ defined to be the bi-unit cube:

$$\hat{D} = \{(r, s, t) : 1 \leq r \leq 1, -1 \leq s \leq 1, -1 \leq t \leq 1\}.$$

We choose a polynomial basis on $\hat{D}$ by taking a tensor product of one-dimensional Lagrange interpolation polynomials l_j based on $N + 1$ Gauss-Lobatto-Legendre (GLL) nodes on the interval $[-1, 1]$. In this way we can write the polynomial u^k as

$$u^k(r, s, t) = \sum_{i=0}^{N} \sum_{j=0}^{N} \sum_{k=0}^{N} u^k_{n_{ijk}} l_i(r)\, l_j(s)\, l_k(t),$$

where

$$n_{ijk} = i + j(N + 1) + k(N + 1)^2,$$

so that $0 \leq n_{ijk} < N_p = (N + 1)^3$. We encode a polynomial q as a vector $\mathbf{q}$ of its polynomial coefficients, i.e. $\mathbf{q} = [q_0, q_1, \ldots, q_{N_p}]^T$. Using this notation, we define the local element mass matrix $\mathbf{M}^k$ to satisfy the following definition

$$\mathbf{v}^T \mathbf{M}^k \mathbf{q} = \int_{\hat{D}} |J^k(r, s, t)| v(r, s, t) q(r, s, t)\, dr ds dt.$$

where $|J^k|$ is determinant of the Jacobian of the mapping from D^k to $\hat{D}$, given by

$$J^k = \begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{\partial t} \end{pmatrix}.$$

We evaluate the integral in each dimension separately using a N_q node GL quadrature. We denote the quadrature weights and nodes in the r dimension as $\{w_a\}_{a=1}^{a=N_q}$ and $\{r_a\}_{a=1}^{a=N_q}$, respectively, and use an analogous notation for the s and t dimensions. We then write

$$\mathbf{v}^T \mathbf{M}^k \mathbf{q} = \sum_{a=1}^{N_q} \sum_{b=1}^{N_q} \sum_{c=1}^{N_q} w_a w_b w_c J^k(r_a, s_b, t_c) v(r_a, s_b, t_c) q(r_a, s_b, t_c).$$

Since the numerical quadrature is performed on the GL quadrature nodes instead of the GLL interpolation nodes, we need to interpolate u^k to the GL nodes first before performing the integration. We define the interpolation operators $\mathbf{I}^r$, $\mathbf{I}^s$, and $\mathbf{I}^t$ such that

$$\begin{aligned} \mathbf{I}^r_{qi} l_i(r) &= l_i(r_q), \\ \mathbf{I}^s_{qj} l_j(s) &= l_j(s_q), \\ \mathbf{I}^t_{qk} l_k(t) &= l_k(t_q), \end{aligned} \tag{1}$$

for all $i,j,k = 0,\dots,N$ and $q = 0,\dots,N_q$. Using these operators and letting $q = l_i(r)l_j(s)l_k(t)$ and $v = l_{i'}(r)l_{j'}(s)l_{k'}(t)$, we find

$$\mathbf{M}^k_{n_{i'j'k'},n_{ijk}} = \sum_{a=1}^{N_q}\sum_{b=1}^{N_q}\sum_{c=1}^{N_q} w_a w_b w_c J^k(r_a,s_b,t_c)\mathbf{I}^r_{ai'}\mathbf{I}^s_{bj'}\mathbf{I}^t_{ck'}\mathbf{I}^r_{ai}\mathbf{I}^s_{bj}\mathbf{I}^t_{ck},$$

Using matrix notation, we can write the action of the local mass matrix concisely as

$$\mathbf{M}^k = (\mathbf{I}^t)^T(\mathbf{I}^s)^T(\mathbf{I}^r)^T\mathbf{W}\mathbf{I}^r\mathbf{I}^s\mathbf{I}^t,$$

where $\mathbf{W}$ is a diagonal matrix of weights and geometric data, i.e. $\mathbf{W}$ has entries

$$\mathbf{W}_{n_{abc},n_{abc}} = w_a w_b w_c J^k(r_a,s_b,t_c).$$

Note that the interpolation operators $\mathbf{I}^r$, $\mathbf{I}^s$, and $\mathbf{I}^t$ are the same matrix operator acting of different dimensions. Hence the action of the local mass matrix consists of seven matrix-vector operations. Interpolation from the GLL interpolation nodes to the GL quadrature nodes comprises three matrix-vector products while the quadrature summation comprises a single matrix-vector product. Finally, the projection back to the GLL nodes via the transpose interpolation operator comprises three matrix-vector products.

We assemble the global mass matrix operator $\mathbf{M}$ by concatenating each of the element local mass matrices to form a block diagonal operator on the global vector of solution coefficients. Doing so, we write the action of the mass matrix $\mathbf{M}$ on a vector $\mathbf{q}$ as

$$\mathbf{w} = \mathbf{M}\mathbf{q}.$$

Due to its block-diagonal structure, the mass matrix operator can be applied in a matrix-free (element-wise) way and no communication is required between elements. We detail the full matrix-free action of the mass matrix in the pseudocode in Algorithm 3.

Algorithm 3 BP1.0: mass matrix multiplication

Data: (1) $\mathbf{q}$, size $N_{el}\cdot N_q^3$; (2) Interpolation matrix $\mathbf{I}$, size $N_q^{GL}\times N_q$; (3) weights, $\mathtt{G}$, size $N_q^{GL}\cdot N_q^{GL}\cdot N_q^{GL}$
Output: $\mathbf{Mq}$, size $N_{el}\cdot N_q^3\times 1$
for every element e **do**
  for $c,a = 1,\dots N_q$, $j = 1,\dots,N_q^{GL}$ **do**
    $\tilde{\mathbf{q}}^{(e)}_{cja} = \sum_{b=1}^{N_q}\mathbf{I}_{jb}\mathbf{q}^{(e)}_{cba}$ ▷ Interpolate in b direction
  end for
  for $c = 1,\dots N_q$, $i,j = 1,\dots,N_q^{GL}$ **do**
    $\hat{\mathbf{q}}^{(e)}_{cja} = \sum_{a=1}^{N_q}\mathbf{I}_{ia}\tilde{\mathbf{q}}^{(e)}_{cja}$ ▷ Interpolate in a direction
  end for
  for $k,i,j = 1,\dots,N_q^{GL}$ **do**
    $\mathbf{q}^{(e)}_{kji} = \mathtt{G}^{(e)}_{kji}\sum_{c=1}^{N_q}\mathbf{I}_{kc}\hat{\mathbf{q}}^{(e)}_{cji}$ ▷ Interpolate in c direction, integrate
  end for ▷ Scale by Jacobian and integration weights
  for $k,i = 1,\dots,N_q^{GL}, b = 1,\dots,N_q$ **do**
    $\tilde{\mathbf{q}}^{(e)}_{kbi} = \sum_{j=1}^{N_q^{GL}}\mathbf{I}_{jb}\mathbf{q}^{(e)}_{kji}$ ▷ Project back in b direction
  end for
  for $k = 1,\dots,N_q^{GL}, b,a = 1,\dots,N_q$ **do**
    $\hat{\mathbf{q}}^{(e)}_{kja} = \sum_{i=1}^{N_q^{GL}}\mathbf{I}_{ia}\tilde{\mathbf{q}}_{kbi}$ ▷ Project back in a direction
  end for
  for $c,b,a = 1,\dots,N_q$ **do**
    $\mathbf{Aq}^{(e)}_{cba} = \sum_{k=1}^{N_q^{GL}}\mathbf{I}_{kc}\hat{\mathbf{q}}_{kba}$ ▷ Project back in c direction, save
  end for
end for

4.4.2 BP1.0 Implementation

In our analysis, we measure performance (runtime and the number of FLOPS) of our GPU code. To assess the performance, we need a roofline bound, an upper limit to compare to in order to measure the performance of the implementation against a realistic metric. While theoretical performance bounds are generally not achievable, we create a model that produces an empirical bound.

Our empirical roofline model is based on an observation that the dominant cost for the benchmarks being considered is the cost of data movement [21]. In the three problems considered, we load and store a substantial amount of data. Based on Algorithm 3, in BP1.0 we load $(N_p + N_p^{GL}) \cdot N_{el}$ double-precision variables and write $N_p \cdot N_{el}$ variables. Even if our code performed no floating-point operations or performed the operations perfectly overlapped with the data movement, our code would not be faster than the time needed to transfer the data. Hence, we consider the cost of data movement an upper limit for the performance. More sophisticated models exist; see [22, 23, 24]. The advantage of our approach is its simplicity.

Let us assume that in some kernel, we load $\mathbf{d}_{in}$ bytes and store $\mathbf{d}_{out}$ bytes. The total data transfer is then $\mathbf{d}_{in} + \mathbf{d}_{out}$ bytes. Considering the two-way memory bus, we compare the performance of this kernel with copying $\frac{\mathbf{d}_{in}+\mathbf{d}_{out}}{2}$ bytes of data from device to device memory.

We measure the time needed to transfer the data; and based on the time, we estimate the bandwidth, b. Let us assume that our kernel executes w FLOPS. We estimate the roofline bound using the formula $\frac{b \cdot w}{d_{in}+d_{out}}$.

The performance of BP1.0 is limited by copying $(2N_p + N_p^{GL})/2$ bytes of data from device to device. Figure 16 shows the GFLOPS for two meshes (with 512 elements and with 4,096 elements) plotted against polynomial degree.

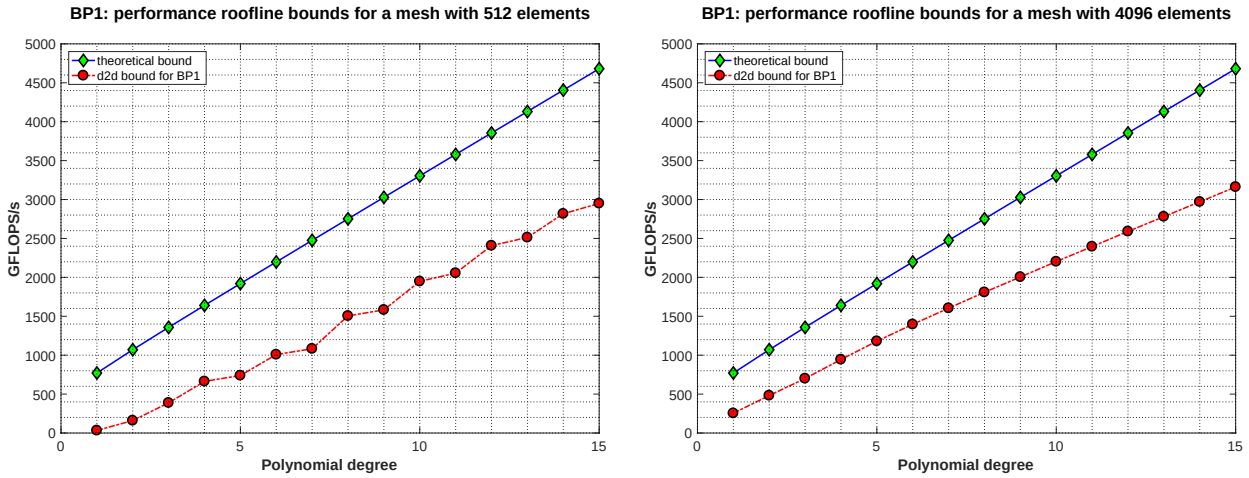

Figure 16: BP1.0: performance roofline bounds. The upper plot (line with diamond-shaped ticks) shows theoretical bound obtained by using a theoretical peak bandwidth of $549GB/s$ for the NVIDIA P100 PCI-E 12GB GPU. The lower plot (line with circle-shaped ticks) shows the empirical peak bandwidth bound obtained by using the measured bandwidth attained when performing a device memory to device memory copy. Left: performance bounds for cubical mesh with 512 hexahedral elements. Right: performance bounds for cubical mesh with $4,096$ hexahedral elements.

BP1.0 kernel optimization The problem comes with a natural parallel structure. We can assign an element to a block of threads. The first challenge in this kernel is that we cannot associate a thread with a node of an element as in [17] because with a high-order interpolation we easily exceed the maximum number of threads per block of threads. Thus, we need to find a way of assigning multiple nodes to a thread. We can use either a 3D thread structure or a 2D thread structure. Figure 17 shows two different approaches. For the BP1.0, we use a 2D thread structure since we found it more effective; we investigate the 3D thread structure for BP3.5 and BP3.0.

Another challenge comes with synchronization. The algorithm consists of 6 loops and in each loop we

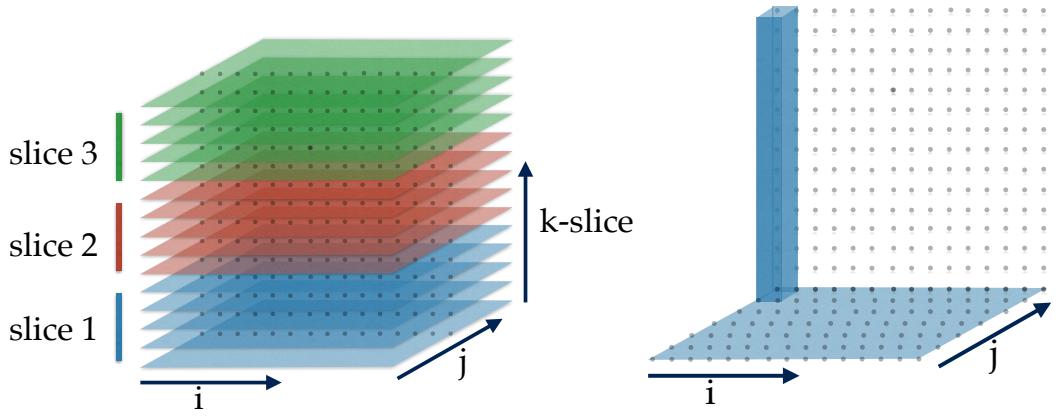

Figure 17: 3D vs 2D thread structure. On the left: 3D approach – each thread processes a "slice" of nodes. On the right: 2D approach – each thread processes a vertical "column" of nodes.

process the entire $\mathbf{q}^{(e)}$ in varying orders. Hence, we must ensure that all threads processing parts of $\mathbf{q}^{(e)}$ in the previous loop are done before we start the next loop, which means that we need at the minimum, synchronize the thread five times. Using a $2D$ thread structure might require more synchronizations because we process only a slice of a nodes at a time.

BP1.0 Thread Memory Optimization Since the matrix $\mathbf{I}$ is used by all threads in the block, we load it to the shared memory. We can either load $\mathbf{q}^{(\mathbf{e})}$ to shared memory or to registers or load it piece-by-piece from global memory when needed. We also need a placeholder for intermediate results (between the loops). We have several choices here; however, storing another shared memory array of size $\left(N_q^{GL}\right)^3$ is not feasible since for large N we exceed the limit of 48 KB shared memory for thread block for large N.

BP1.0 Thread Data Optimization We can use array padding to avoid bank conflicts. I.e., if $N_q = 8$ or $N_q = 16$, instead of using arrays of size `Nq` $\times$ `Nq`, we use arrays of size `Nq` $\times$ `Nq+1`. The same principle applied to cases when $N_q^{GL} = 8$ or $N_q^{GL} = 16$. In such cases, the arrays of size `glNq` $\times$ `glNq` are declared as `glNq` $\times$ `glNq+1`.

Results of BP1.0 Kernel Optimization We start with a reference kernel that uses 2D thread structure associated with horizontal $(r - s)$ slices. We declare two additional global variables for storing intermediate results. We use shared memory only for the interpolation matrix: in all the loops, we read from and write to global memory. This kernel is highly inefficient, reaching about 80 GFLOPS at the maximum. We optimize this kernel by introducing two shared-memory arrays as placeholder for partial results and getting rid of the auxiliary global arrays. The GFLOPS improve two times as much. Next, we declare all input variables (except the variable used for final result) as `const`, and we store some of the partial results in register (exclusive) variables . This action brings the GFLOPS to about 800 GFLOPS in the best case. In subsequent code versions, we pad the arrays, bringing the performance to over 1 TFLOP/s in the best case. Next, we unroll all the loops. Since we loop over (r-s) slices, unrolling this main loop makes a big difference. However, the performance is still far from our computed empirical limit. The performance bottleneck is the excessive thread synchronizations.

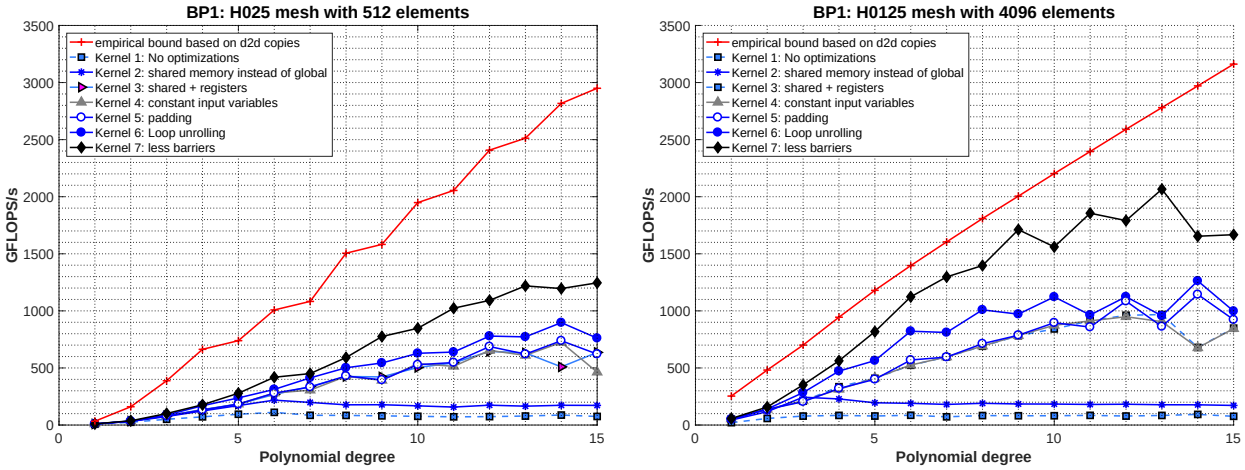

Figure 18: BP1.0: performance results for the code for BP1.0. Left: results obtained using cube-shaped mesh with 512 elements. Right: results obtained using cube-shaped mesh with 4,096 elements on a NVIDIA P100 PCI-E 12GB GPU.

Hence, we devise an alternative approach in which we use the same 2D thread structure but minimize the number of synchronizations to 5. To achieve this , however, we need to allocate more shared memory. In this approach we load the entire $\mathbf{q}^{(e)}$ to shared memory (not slicewise as in the previous versions) so use of shared memory increases. We call this approach "Kernel 9." Figure 18 illustrates the difference in performance in various stages of optimization on a cube hexagonal mesh with 512 elements and with $4,192$ elements. For the smaller mesh and $N = 13$ and $N = 15$, eliminating unnecessary barriers improves performance by about 33%. The second kernel achieves over 2 TFLOPS for the bigger mesh with $N = 13$.

We point out that our least-tuned kernel (kernel 0) barely achieves 80 GFLOPS in the best case, whereas our best kernel achieves 2 TFLOPS = 2000 GFLOPS, a 25-fold speedup. For BP1.0, the most effective steps in performance optimization were replacing all the global memory loads/stores with shared memory and registers and unrolling the loops. To get to 2 TFLOPS, however, we need to change the algorithm to account for the limitations intrinsic to the GPU (cost of many thread synchronizations). Figures 18 show that the performance of all the kernels is better for the bigger mesh; in the case of the smaller mesh, we underused the GPU because we do not have enough data to fully saturate it. For N up to 13 we are close to the empirical bound.

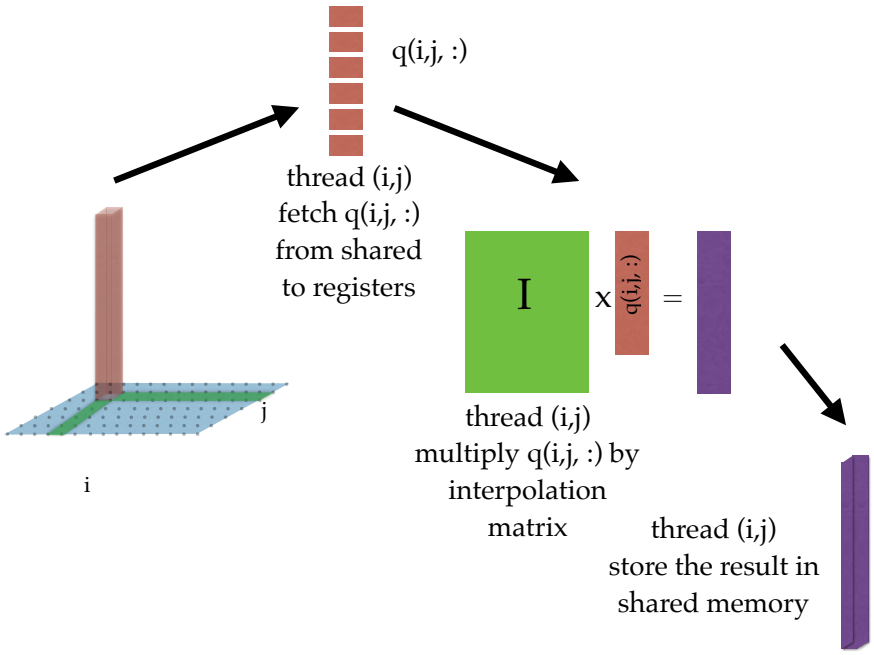

Figure 19: BP1.0: the idea behind reducing synchronizations in kernel 9. We fetch pieces of $\mathbf{q}^e$ to registers from shared memory and then write the result to shared memory. This action does not create race conditions because we do use a 2D thread structure and interpolate only in one direction at a time.

4.5 BP3.5: Stiffness Matrix with Collocation Differentiation

The second benchmark we consider is a matrix-vector product of a high-order finite-element stiffness matrix and corresponding vector. This operation is central to the many elliptic finite-element codes and is usually part of a discrete operator we wish to invert. For example, incompressible flow solvers such as Nek5000 [25] require solving a Poisson potential problem at each time step. Consequently, this matrix-vector product is potentially evaluated many times in each time step of a flow simulation, making its optimization a significant factor for good performance.

4.5.1 BP3.5 mathematical formulation

We consider the same discrete problem setup as in Section 4.4. Namely we discretize our domain $\mathcal{D}$ into K hexahedral elements D^k, $k = 1, \ldots, K$. We map each element to a bi-unit reference cube $\hat{D}$ and consider a high-order polynomial u^k on $\hat{D}$. We express the polynomial u^k as a sum of a tensor product basis of Lagrange interpolation polynomials at the GLL nodes.

Using the notation defined in Section 4.4, we define the local element stiffness matrix $\mathbf{A}^k$ to satisfy the following definition:

$$\mathbf{v}^T \mathbf{A}^k \mathbf{q} = \int_{\hat{D}} (\nabla v(r,s,t))^T \mathbf{G}^k(r,s,t) \nabla q(r,s,t)\, drdsdt, \tag{2}$$

where $\mathbf{G}^k$ is the scaled metric tensor on element D^k given by $\mathbf{G}^k = |J^k| \left(J^k\right)^T J^k$ which we write as the matrix

$$\mathbf{G}^k = \begin{pmatrix} \mathbf{G}^k_{rr} & \mathbf{G}^k_{rs} & \mathbf{G}^k_{rt} \\ \mathbf{G}^k_{sr} & \mathbf{G}^k_{ss} & \mathbf{G}^k_{st} \\ \mathbf{G}^k_{tr} & \mathbf{G}^k_{ts} & \mathbf{G}^k_{tt} \end{pmatrix}.$$

The ∇ operators are understood as operations on the (r,s,t) variables. We evaluate the integral in each dimension separately using the GLL interpolation nodes. We denote the quadrature weights and nodes in the r dimension as $\{w_a\}_{a=0}^{a=N}$ and $\{r_a\}_{a=0}^{a=N}$, respectively, and use an analogous notation for the s and t dimensions.

We use the fact that since these are the basis interpolation nodes we have $l_i(r_a) = \delta_{ia}$ for all $i, a = 0, \ldots, N$. We then obtain

$$\mathbf{v}^T \mathbf{A}^k \mathbf{q} = \sum_{a=0}^{N} \sum_{b=0}^{N} \sum_{c=0}^{N} w_a w_b w_c (\nabla v(r_a, s_b, t_c))^T \mathbf{G}^k(r, s, t) \nabla q(r_a, s_b, t_c).$$

To perform the differentiation operation, we introduce the derivative operators $\mathbf{D}^r$, $\mathbf{D}^s$, and $\mathbf{D}^t$, which satisfy

$$\begin{aligned} \mathbf{D}^r_{ai} l_i(r) &= l_i'(r_a), \\ \mathbf{D}^s_{bj} l_j(s) &= l_j'(s_b), \\ \mathbf{D}^t_{ck} l_k(t) &= l_k'(t_c), \end{aligned} \tag{3}$$

for all $i, j, k = 0, \ldots, N$ and $a, b, c = 0, \ldots, N$. Using these operators and letting $q = l_i(r) l_j(s) l_k(t)$ and $v = l_{i'}(r) l_{j'}(s) l_{k'}(t)$, we find

$$\begin{aligned} \mathbf{A}^k_{n_{i'j'k'}, n_{ijk}} = \sum_{a=0}^{N} \sum_{b=0}^{N} \sum_{c=0}^{N} w_a w_b w_c \Big(& \mathbf{D}^r_{ai'} \left(\mathbf{G}^k_{rr} \mathbf{D}^r_{ai} \delta_{jb} \delta_{kc} + \mathbf{G}^k_{rs} \mathbf{D}^s_{bj} \delta_{ck} + \mathbf{G}^k_{rt} \mathbf{D}^t_{ck} \delta_{bj} \right) \\ & \mathbf{D}^s_{bj'} \left(\mathbf{G}^k_{sr} \mathbf{D}^r_{ai} \delta_{kc} + \mathbf{G}^k_{ss} \mathbf{D}^s_{bj} \delta_{ai} \delta_{ck} + \mathbf{G}^k_{st} \mathbf{D}^t_{ck} \delta_{ai} \right) \\ & \mathbf{D}^t_{ck'} \left(\mathbf{G}^k_{tr} \mathbf{D}^r_{ai} \delta_{bj} + \mathbf{G}^k_{ts} \mathbf{D}^s_{bj} \delta_{ai} + \mathbf{G}^k_{tt} \mathbf{D}^t_{ck} \delta_{ai} \delta_{bj} \right) \Big). \end{aligned} \tag{4}$$

Absorbing the integration weights into the geometric factors $\mathbf{G}$ and collapsing the products involving the δ-functions we can write this compactly using matrix notation as

$$\begin{aligned} \mathbf{A}^k = & (\mathbf{D}^r)^T \left(\mathbf{G}^k_{rr} \mathbf{D}^r + \mathbf{G}^k_{rs} \mathbf{D}^s + \mathbf{G}^k_{rt} \mathbf{D}^t \right) \\ & (\mathbf{D}^s)^T \left(\mathbf{G}^k_{sr} \mathbf{D}^r + \mathbf{G}^k_{ss} \mathbf{D}^s + \mathbf{G}^k_{st} \mathbf{D}^t \right) \\ & (\mathbf{D}^t)^T \left(\mathbf{G}^k_{tr} \mathbf{D}^r + \mathbf{G}^k_{ts} \mathbf{D}^s + \mathbf{G}^k_{tt} \mathbf{D}^t \right). \end{aligned}$$

Therefore, we can apply the action of the local stiffness matrix using six matrix-vector operations. First, we differentiate along each dimension using three matrix-vector products and multiply by the necessary geofactors. Then we multiply by the transpose derivative operators with three additional matrix-vector products and sum the result.

We assemble the global stiffness matrix operator $\mathbf{A}$ by concatenating each of the element local stiffness matrices to form a block diagonal operator on the global vector of solution coefficients. Doing so, we write the action of the mass matrix $\mathbf{A}$ on a vector $\mathbf{q}$ as

$$\mathbf{w} = \mathbf{A}\mathbf{q}.$$

Due to its block diagonal structure, the stiffness matrix operator can be applied in a matrix-free (element-wise) way. We detail the full matrix-free action of the stiffness matrix in the pseudocode in Algorithm 4.

4.5.2 BP3.5 Implementation

For performance assessment, we use the same empirical model as in the BP1.0. Looking at Algorithm 4, in BP3.5 we load and store $2N_p + N_p^{GL}$ variables, therefore the performance is bounded by the cost of copying $(2N_p + N_p^{GL})/2$ bytes of data from device to device. Figure 20 shows the computed bound. The maximal expected GFLOPS/s are much lower than for BP1.0.

BP3.5 kernel optimization As in BP1.0, the problem comes with a natural parallel structure, since all the elements can be processed simultaneously. Hence we assign each element to a separate thread block. We investigate both 3D and 2D thread structure.

The biggest challenge in optimizing this kernel is hiding the cost of the data transfer. To differentiate, we need to multiply by geometric factors (Equation (4)). This means that we need to load 7 double precision variables for every node in an element (6 actual geometric factors and the quantity $|\mathbf{J}| w_i w_j w_k$); see lines 7, 8, and 21 in Algorithm 4). To hide global memory latency, we can overlap loading with computation.

Algorithm 4 BP3.5: collocation differentiation for 3D hexahedral mesh

Data: (1) Vector $\mathbf{q}$, size `Nelements` N_q^3, (2) differentiation matrix $\mathbf{D}$, size $N_q \times N_q$, (3) geometric factors $\mathbf{G}$, size `Nelements` $\cdot N_q^3 \times 7$, (4) parameter λ ;
Output: Vector $\mathbf{Aq}$, size `Nelements` $\cdot N_q^3 \times 1$;
for $e = 1, 2, \ldots N_{el}$ **do**
  for $i = 1, 2, \ldots N_q$ **do**
    for $j = 1, 2, \ldots N_q$ **do**
      for $k = 1, 2, \ldots N_q$ **do**
        ▷ Load geometric factors ↓
        `G00` $= \mathbf{G}^{(e)}_{1;kji}$, `G01` $= \mathbf{G}^{(e)}_{2;kji}$, `G02` $= \mathbf{G}^{(e)}_{3;kji}$;
        `G11` $= \mathbf{G}^{(e)}_{4;kji}$, `G12` $= \mathbf{G}^{(e)}_{5;kji}$, `G22` $= \mathbf{G}^{(e)}_{6;kji}$;
        ▷ Multiply by $\mathbf{D}$ ↓
        `qr` $= \sum_{n=1}^{N_q} \mathbf{D}_{in}\mathbf{q}^{(e)}_{kjn}$;
        `qs` $= \sum_{n=1}^{N_q} \mathbf{D}_{jn}\mathbf{q}^{(e)}_{kni}$;
        `qt` $= \sum_{n=1}^{N_q} \mathbf{D}_{kn}\mathbf{q}^{(e)}_{nji}$;
        ▷ Apply chain rule ↓
        $\texttt{rqr}^{(e)}_{ijk}$ = `G00*qr + G01*qs + G02*qt`;
        $\texttt{rqs}^{(e)}_{ijk}$ = `G01*qr + G11*qs + G12*qt`;
        $\texttt{rqt}^{(e)}_{ijk}$ = `G02*qr + G12*qs + G22*qt`;
      end for
    end for
  end for
  for $i = 1, 2, \ldots N_q$ **do**
    for $j = 1, 2, \ldots N_q$ **do**
      for $k = 1, 2, \ldots N_q$ **do**
        `GwJ` $= \mathbf{G}^{(e)}_{7;kji}$
        $\mathbf{Aq}^{(e)}_{ijk} = \lambda\texttt{GwJ}\mathbf{q}^{(e)}_{kji} + \sum_{n=1}^{N_q} \mathbf{D}_{ni}\texttt{rqr}^{(e)}_{kjn} + \mathbf{D}_{nj}\texttt{rqs}^{(e)}_{kin} + \mathbf{D}_{nk}\texttt{rqr}^{(e)}_{nji}$;
      end for
    end for
  end for
end for

The extra load increases both the number of.global memory reads and the number of registers needed per thread. Another idea is to compute geometric factors "on the fly" inside the kernel instead of loading them from global memory. This reduces the number of global loads to $18 + 2N_q$ per block but increases the number of registers needed in a kernel, which turned out to be impractical.

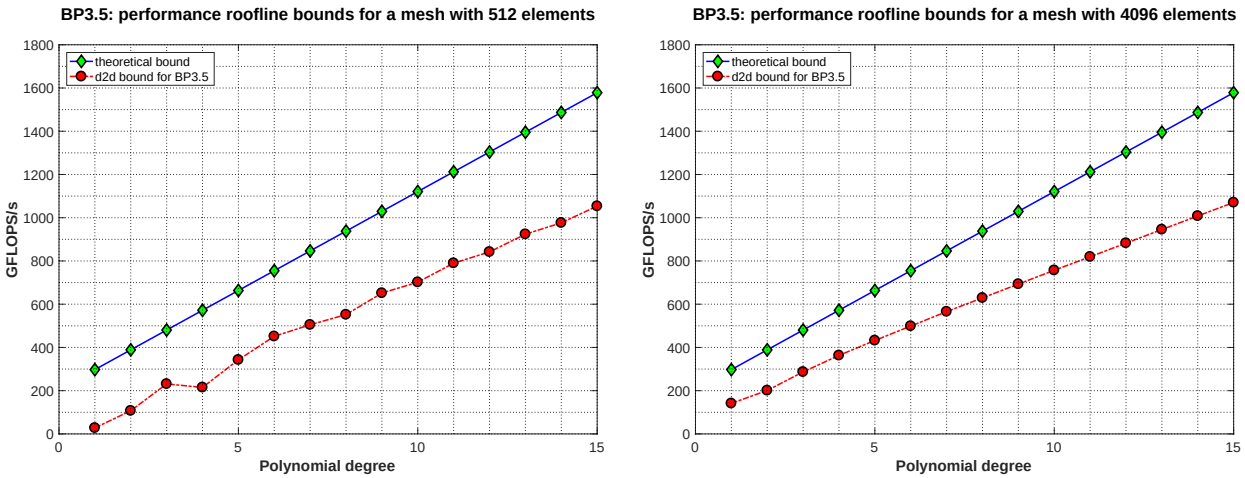

Figure 20: BP3.5: performance roofline bounds. The upper plot (line with diamond-shaped ticks) shows theoretical bound obtained using theoretical peak bandwidth of $549GB/s$ on a single NVIDIA P100 PCI-E 12GB GPU. The lower plot (line with circle-shaped ticks) shows the bound obtained using bandwidth for device to device copy. Left: performance bounds for cubical mesh with 512 elements. Right: performance bounds for cubical mesh with $4,096$ elements.

2D thread structure. We investigate a sequence of kernels. We start from a kernel that is a direct, straightforward implementation of Algorithm 5.

Algorithm 5 BP3.5: starting point of the implementation (2D thread structure)

```
1: for All elements e, in parallel do
2:    for each thread i,j do
3:       Load D to shared memory variable s_D
4:       Allocate one shared auxiliary array s_tmps[Nq][Nq][Nq] and three register arrays: r_Aq[Nq],
   r_qt[Nq] and r_tmpt[Nq] for storing intermediate results;
5:       Synchronize threads;
6:       for k = 1,...,Nq do
7:          base = i + j*p_Nq + e*p_Np;
8:          for n = 1,...,Nq do
9:             qtk = s_D[k][n]*q[base + n*Nq*Nq];
10:         end for
11:         r_qt[k] = qtk;
12:      end for
13:      for k = 1,...,Nq do
14:         Load geometric factors to local variables G00, G01, G02, G11, G12, G12, G22, GwJ
15:         Declare variables qr and qs and set them to 0.
16:         for n = 1,...,Nq do
17:            qr = qr + s_D[i][n]*q[n + j*Nq + k*Nq*Nq+e*Np];
18:            qs = qs + s_D[j][n]*q[i + n*Nq + k*Nq*Nq+e*Np];
19:         end for
20:         base = i + j*p_Nq + e*p_Np;
21:         Aqtemp[base + k*Nq*Nq] = G00*qr + G01*qs + G02*r_qt[k];
22:         s_tmps[k][j][i] = G01*qr + G11*qs + G12*r_qt[k];
23:         r_tmpt[k] = G02*qr + G12*qs + G22*r_qt[k];
24:      end for
25:      Aq[base + k*Nq*Nq] = Aq[base + k*Nq*Nq]
   +GwJ*lambda*q[i + j*Nq + k*Nq*Nq+e*Np]
26:      for n = 1,...,Nq do
27:         Aq[base + n*Nq*Nq] =Aq[base + n*Nq*Nq+s_D[k][n]*r_tmpt[k];
28:      end for
29:      Synchronize threads;
30:      for k = 1,...,Nq do
31:          base = i + j*Nq + e*Np;
32:         Declare variables Aq1 and Aq2 and set them to 0.
33:         for n = 1,...,Nq do
34:            Aq1 = Aq1 + s_D[n][i]*Aqtemp[k*Nq*Nq + j*Nq + e*Np +n];
35:            Aq2 = Aq2 + s_D[n][j]*s_tmps[k][n][i];
36:         end for
37:          Aq[base + k*Nq*Nq] +=  Aq1 + Aq2;
38:      end for
39:   end for
40: end for
```

Many of the speedups are due to subtle changes in the code: declaring variables as constant, adding padding, or unrolling the loops. Storage type has the biggest impact on the performance; once we reduce the use of global memory (starting in Kernel 6), the performance improves substantially, especially for bigger N. Even with all application of these strategies, however, obtaining a kernel performance close to the roofline is difficult. Performance results are shown in Figure 21.

In the first version of the code, the performance barely reached 200 GFLOPS and the best version achieved up to 1 TFLOP/s. Although it is only five fold improvement, for the 4096 mesh we reached the peak performance. This means that the best kernel is achieving throughput comparable with just streaming the minimally necessary data.

Kernel #	Type of Optimization
Kernel 0	As shown in the pseudocode - no optimization.
Kernel 1	All input variables declared as `const`.
Kernel 2	All loops involving `k` and `n` unrolled.
Kernel 3	The internal loop (lines 390–404) becomes external (i.e., in the new setup, `k` is the slowest changing index).
Kernel 4	Replace a cube of shared memory (`s_tmps[Nq][Nq][Nq]`) with two shared arrays of size `Nq*Nq` each.
Kernel 5	The variable `q` is loaded to shared memory.
Kernel 6	Eliminate any unnecessary use of global memory; `q` is read once and stored in shared memory. We write to `Aq` only once at the very end of the kernel.
Kernel 7	We add padding to all shared arrays in case `Nq=7` and `Nq=15`; this eliminates bank conflicts.
Kernel 8	Slightly different approach; in this kernel we copy `q` from shared memory to registers before differentiating (before main `k`-loop.
Kernel 9	We use three shared memory variables of size `Nq*Nq` to store partial results. The variable `q` is copied to registers, not to shared;.

Table 7: Optimization strategies applied to a baseline kernel with 2D thread structure for BP3.5 problem

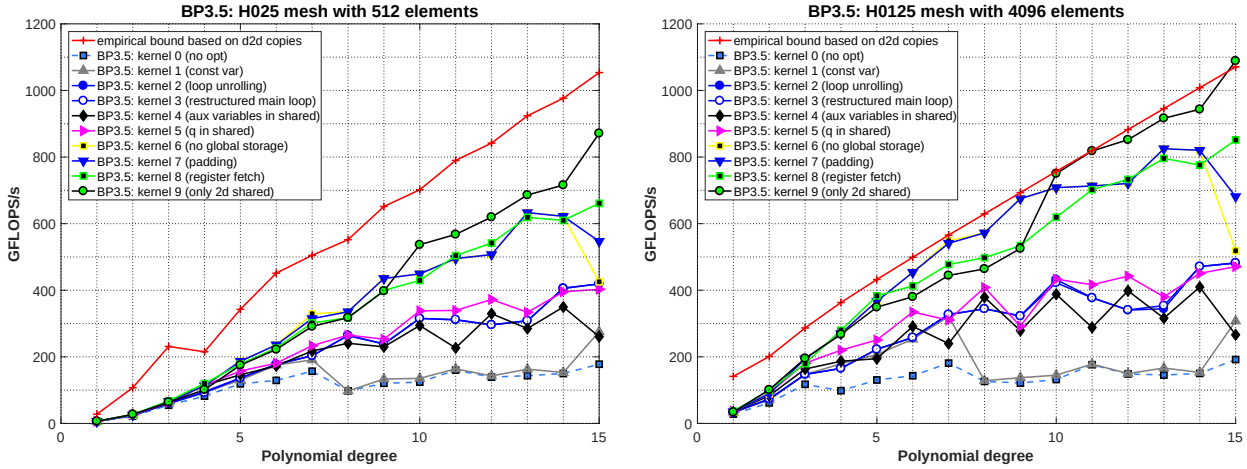

Figure 21: BP3.5: Performance of 2D kernels in various stages of optimization. The red line marked with crosses is the empirically determined roofline based on optimal achievable device to device memory copies on an NVIDIA P100 PCI-E 12GB GPU. Left: GFLOPS for cubical mesh with 512 elements. Right: GFLOPS for cubical mesh with $4,096$ elements.

For the BP3.5 kernel treatment using a 3D thread structure, we start with the pseudocode in Algorithm 6.

We optimize this kernel only up to $N = 9$, since it will require more than the maximum of 1,024 threads for higher degree. Note also that `Aqtemp` is a global array used as a placeholder. Table 8 shows steps in the optimization process. The results are shown in Figure 22.

In the case of 3D kernels with $N \leq 9$, we have relatively little data per thread (we associate one node in an element with one thread). Hence, we expect good performance. Due to optimization, the most tuned

kernel achieves twice the performance of an untuned baseline kernel. The most important tune-up is putting `q` into shared memory. This brings us very close to the peak for the bigger mesh.

Kernel #	Type of optimization
Kernel 0	As shown in the pseudocode - no optimization.
Kernel 1	All input variables declared as `const`; loops with `n` iterator unrolled.
Kernel 2	Geometric factors fetched only once and stored in registers.
Kernel 3	Register variables for partial results, instead of storing them directly in `Aq`.
Kernel 4	Load `q` to shared memory.
Kernel 5	Allocate three extra shared memory arrays to store partial results.

Table 8: Optimization strategies applied to a baseline kernel with 3D thread structure for BP3.5 problem

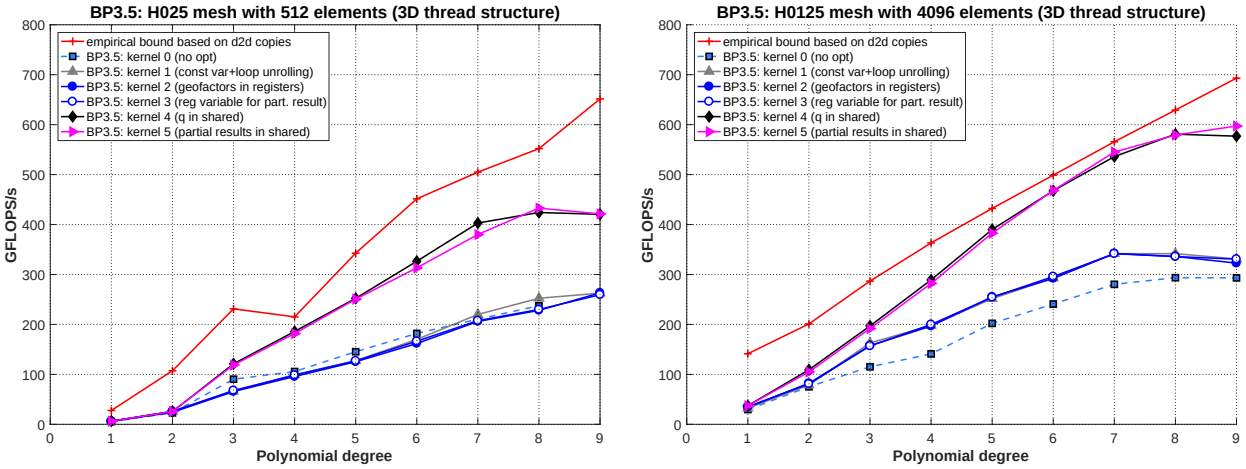

Figure 22: BP3.5: Performance of 3D kernels in various stages of optimization. The red line marked with crosses is the roofline computed based on device to device copies measured on an NVIDIA P100 PCI-E 12GB GPU. Left: GFLOPS/s for cubical mesh with 512 elements. Right: GFLOPS/s for cubical mesh with 4,096 elements.

4.6 BP3.0: Stiffness Matrix A Evaluated with Quadrature

The final benchmark we consider is the same matrix-vector product of the high-order finite element stiffness matrix as in BP3.5, but this time the inner products in the definition of $\mathbf{A}$ in (2) are computed by using a GL quadrature. This benchmark combines computational elements from BP1.0 and BP3.5, resulting in an arithmetically intense kernel and making performance maximization on GPUs challenging.

4.6.1 BP3.0 Mathematical Formulation

Beginning with the definition of the high-order finite-element stiffness matrix in (2), we evaluate the integral in each dimension separately using a GL quadrature. We denote the quadrature weights and nodes in the r dimension as $\{w_a\}_{a=0}^{a=N_q}$ and $\{r_a\}_{a=0}^{a=N_q}$, respectively, and use an analogous notation for the s and t dimensions. We then obtain

$$\mathbf{v}^T\mathbf{A}^k\mathbf{q} = \sum_{a=0}^{N}\sum_{b=0}^{N}\sum_{c=0}^{N} w_a w_b w_c (\nabla v(r_a,s_b,t_c))^T \mathbf{G}^k(r,s,t)\nabla q(r_a,s_b,t_c).$$

Note that the quadrature requires the gradients of q and v to be evaluated at the GL quadrature nodes. Using our interpolation operators defined in (1) and the derivative operators on the Lagrange interpolation polynomials defined in (3), we would require nine matrix-vector products to evaluate this quantity. Specifically, for each of the r, s, and t derivatives of q, we would require an operation that combines differentiation and interpolation to the GL quadrature along one dimension and only interpolation to the GL quadrature along the remaining two dimensions.

We can, however, reduce the number of required matrix-vector operations by considering the Lagrange interpolating polynomials $\hat{l}_n(r)$ associated with the set of GL quadrature nodes. We can then define the derivative operators one this set of polynomials as follows:

$$\begin{aligned}\hat{\mathbf{D}}^r_{ai}\hat{l}_i(r) &= \hat{l}'_i(r_a),\\ \hat{\mathbf{D}}^s_{bj}\hat{l}_j(s) &= \hat{l}'_j(s_b),\\ \hat{\mathbf{D}}^t_{ck}\hat{l}_k(t) &= \hat{l}'_k(t_c).\end{aligned}$$

Thus, if we view the interpolation operators defined in (1) as transforming a polynomial from the basis of Lagrange interpolating polynomials on the GLL nodes to the basis of Lagrange interpolating polynomials on the GL quadrature, we can use the operators $\hat{\mathbf{D}}^r$, $\hat{\mathbf{D}}^s$, and $\hat{\mathbf{D}}^t$ to evaluate the derivatives of this polynomial on the GL node basis.

With these derivative operators, we continue the derivation of the stiffness matrix $\mathbf{A}^k$ analogously to Section 4.5 for BP3.5 and introduce the interpolation operators as done in Section 4.4 for BP1.0 to find that

$$\begin{aligned}\mathbf{A}^k = (\mathbf{I}^t)^T(\mathbf{I}^s)^T(\mathbf{I}^r)^T\Big((\hat{\mathbf{D}}^r)^T\left(\mathbf{G}^k_{rr}\hat{\mathbf{D}}^r + \mathbf{G}^k_{rs}\hat{\mathbf{D}}^s + \mathbf{G}^k_{rt}\hat{\mathbf{D}}^t\right)\\ (\hat{\mathbf{D}}^s)^T\left(\mathbf{G}^k_{sr}\hat{\mathbf{D}}^r + \mathbf{G}^k_{ss}\hat{\mathbf{D}}^s + \mathbf{G}^k_{st}\hat{\mathbf{D}}^t\right)\\ (\hat{\mathbf{D}}^t)^T\left(\mathbf{G}^k_{tr}\hat{\mathbf{D}}^r + \mathbf{G}^k_{ts}\hat{\mathbf{D}}^s + \mathbf{G}^k_{tt}\hat{\mathbf{D}}^t\right)\Big)\mathbf{I}^r\mathbf{I}^s\mathbf{I}^t\end{aligned}$$

Therefore, we can apply the action of this quadrature version of the local stiffness matrix using twelve matrix-vector operations. We first interpolate to the GL quadrature nodes along each dimension using three matrix-vector products. We then differentiate along each dimension using three matrix-vector products and multiply by the necessary geofactors. We multiply by the transpose derivative operators with three additional matrix-vector products and sum the result. We then multiply by the transpose interpolation operators in each dimension to project the result back to the GLL interpolation nodes.

We assemble the global stiffness matrix operator $\mathbf{A}$ as before by concatenating each of the element local stiffness matrices to form a block diagonal operator on the global vector of solution coefficients. Doing so, we again write the action of the mass matrix $\mathbf{A}$ on a vector $\mathbf{q}$ as

$$\mathbf{w} = \mathbf{A}\mathbf{q}.$$

We detail the full matrix-free action of the stiffness matrix evaluated with the numerical quadrature in pseudocode in Algorithm 7.

BP3.0 Kernel Design Considerations BP3.0 can be considered a fusion of BP1.0 and BP3.5. It shares the interpolation/projection part with BP1.0 and the integration part with BP3.5, and hence it comes with all the challenges inherited from BP1.0 and BP3.5 i.e., we need to synchronize threads multiple times, and we transfer a large quantity of data. The additional issue is that now we need even more memory to store partial results (shared and/or registers). We also transfer more data per block compared with BP3.5 because we differentiate on a denser grid. Figure 23 shows theoretical performance bounds. Since in this kernel we load and store $(2N_p + 7N_p^{GL})$ bytes of data, we compute the bounds by copying $(2N_p + 7N_p^{GL})/2$ from device to device memory.

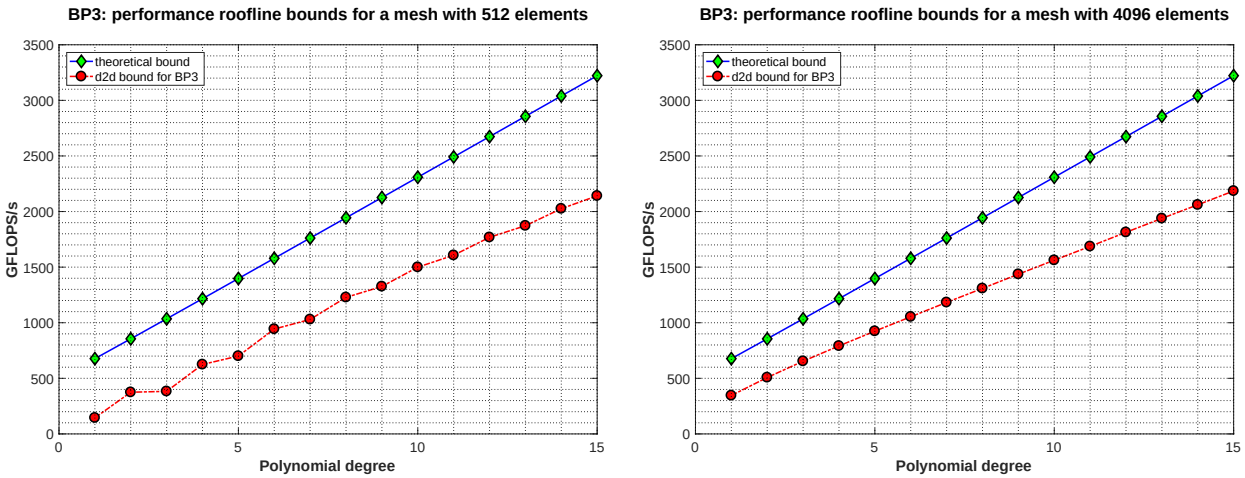

Figure 23: BP3.0: performance roofline bounds. The upper plot (line with diamond-shaped ticks) shows theoretical bound obtained using theoretical peack bandwidth of $549GB/s$ on an NVIDIA P100 PCI-E 12GB GPU. The lower plot (line with circle-shaped ticks) shows the bound obtained using bandwidth for device to device copy. Left: performance bounds for cubical mesh with 512 elements. Right: performance bounds for cubical mesh with $4,096$ elements.

We associate an element with a tread block, and we have a choice between using 2D or 3D thread structure. With $3D$ structure, if $N > 8$, we cannot associate a node with a thread because we do not have enough threads.

Since BP3.0 has three distinct parts, we might consider splitting the problem into three kernels. This strategy reduces the memory requirements per kernel and makes the code more readable. One of the advanced features of OCCA, which we use for implementation, is that the OCCA language lets us easily split one kernel into multiple kernels. The kernels can all be written as one kernel in the OCCA language (the splitting happens during compilation), and we investigate this possibility.

BP3.0 2D Thread Structure. We performed a sequence of optimization strategies on this version (see Table 9). The results are shown in Figure 24. While the more basic strategies bring some performance improvements, we need a more substantial changes in algorithm to see more gains.

Since the first kernel reads `q` from the global memory multiple times, we expect its performance to be poor. Adding `const` does not bring noticeable improvement. Unrolling the loops has more influence on the performance on the bigger grid. Loading **q** to register makes a difference only for $N < 5$ and $N \geq 13$ on bigger mesh. A surprising result is that what makes the biggest difference for GFLOPS/s is using the interpolation/projection as developed in the best kernel in BP1.0; changing the impact of using more efficient differentiation is marginal. Reducing the use of shared memory (Kernel 6) leads to a better performance for big mesh with $N = 11$ and $N = 13$.

In this case, the most-tuned kernel performs 4x as fast as the least tuned kernel. Although we did not hit the peak performance, for the H0125 mesh (4096 elements) and $N \leq 11$ we are very close.

BP3.0 Kernel with 3D Thread Structure

We perform an analogous tuning process using a 3D thread structure for $N < 9$. In this case, we start with an unoptimized version with three separate kernels. That is, in Kernel 0 we split the launch into three separate CUDA kernels, and we execute Kernel 1 as one CUDA kernel. Barely any difference exists between these two kernels. Thus, the cost of addition kernel launches is small compared with the cost of data transfer. Next we add quantifier `const` and unroll the loops, and we fetch geometric factors to register variables declared as `exclusive`. The first major speedup is the result of explicitly caching `q` in shared memory during the differentiation part. Eliminating more and more global reads and then replacing them with shared variables and registers brings us close to the peak for the mesh with $4,096$ elements.

The sequence of progressive optimizations is shown in Table 9, and the results are shown in Figure 25.

Kernel #	Type of optimization
Kernel 0	As shown in the pseudocode - no optimization.
Kernel 1	All input variables declared as `const`. We load `q` to register array to eliminate global accesses.
Kernel 2	All loops involving `k` and `n` unrolled.
Kernel 3	We add padding to all shared-memory arrays.
Kernel 4	We change the interpolation and projection to what we developed in the most optimized kernel for BP1.0. As a result, we reduce the number of synchronizations.
Kernel 5	We change the differentiation strategy.
Kernel 6	We eliminate one of the shared-memory arrays used as placeholder for partial result on the cost of extra synchronization.
Kernel 7	Both `D` and its transpose `DT` are stored in shared memory.

Table 9: Optimization strategies applied to a baseline kernel with 2D thread structure for BP3.0 problem

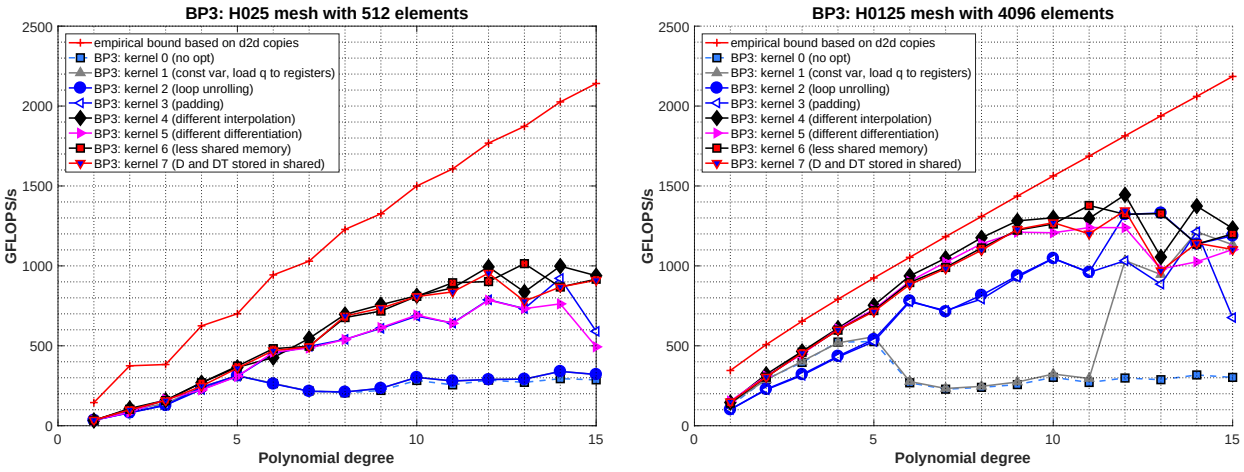

Figure 24: BP3.0: Performance of 2D kernels in various stages of optimization. The red line marked with crosses is the roofline computed based on device to device copies on a single NVIDIA P100 PCI-E 12GB GPU. Left: GFLOPS/s for cubical mesh with 512 elements. Right: GFLOPS/s for cubical mesh with $4,096$ elements.

4.7 Summary and Future Work

We have demonstrated that one can develop kernels expressed in the OCCA OKL kernel language that achieve close to empirical peak performance for all CEED benchmark problems considered (BP1.0, BP3.0, and BP3.5) up to a reasonable polynomial order on the NVIDIA P100 PCI-E 12GB GPU. The kernel optimization task was simplified by several OCCA features, not least the ability to access shared memory, control register usage, and easily combine or split kernels.

We focused in particular on BP3.5 given its direct relevance as the most time intensive component of the Nek5000 simulation code. We obtained a true empirical peack for BP3.5 on a mesh of 4,096 elements. Our performance model does, however, need further work. One of our future goals is to improve the model to account for additional performance limiters, which will be able to provide the true peak performance with

Kernel #	Type of Optimization
Kernel 0	Naive kernel: intentionally unoptimized.
Kernel 1	One-kernel version.
Kernel 2	All loops unrolled; input variables declared as `const`.
Kernel 3	Geometric factors loaded to registers once (eliminated redundant reads).
Kernel 4	`q` loaded to shared memory in the differentiation part.
Kernel 5	Partial results stored in registers, not in global memory, in the differentiation part.
Kernel 6	All the global storage eliminated except between three kernel parts.
Kernel 7	All the unnecessary global storage eliminated.
Kernel 8	Partial results stored in shared variables

Table 10: Optimization strategies applied to a baseline kernel with 3D thread structure for BP3.0 problem

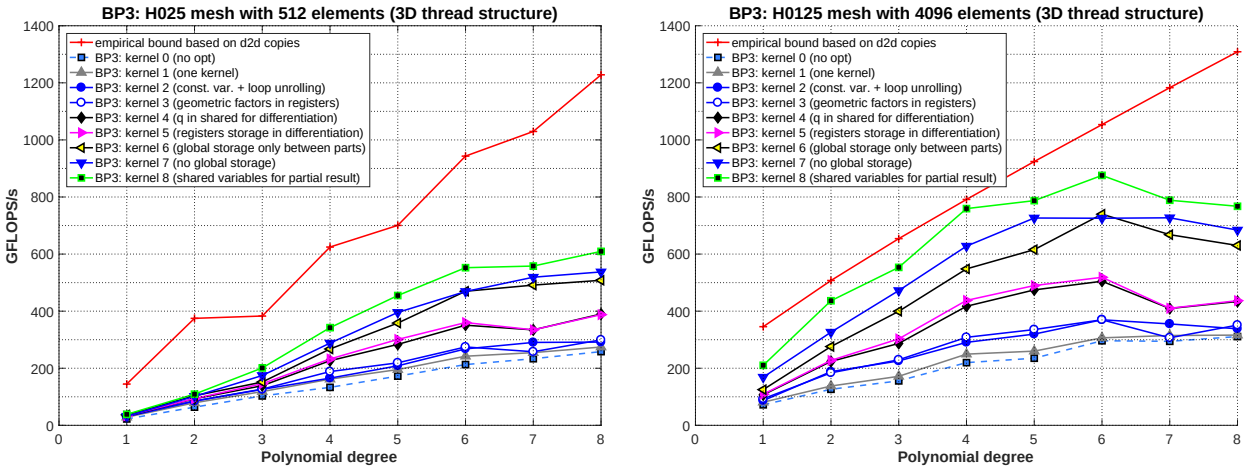

Figure 25: BP3.0: Performance of 3D kernels in various stages of optimization. The red line marked with crosses is the roofline computed based on device to device copies on a single NVIDIA P100 PCI-E 12GB GPU. Left: GFLOPS/s for cubical mesh with 512 elements. Right: GFLOPS/s for cubical mesh with 4096 elements.

more confidence at all polynomial orders targeted for optimization.

Our plans involve using different threading models for testing our approaches (OpenCL, OpenMP). These will provide insight into how fine-tuning the code for different threading systems differs from one threading system to the next.

5. OTHER PROJECT ACTIVITIES

5.1 CEED First Annual Meeting

The first annual meeting of the CEED co-design center took place August 15–17 at the HPC Innovation Center of Lawrence Livermore National Laboratory. Participants reported on the progress in the center, deepened existing and established new connections with ECP hardware vendors, ECP software technologies projects and other collaborators, planned project activities and brainstormed / worked as a group to make technical progress.

The meeting was very successful and was attended by 50+ participants from several DOE labs, academia and industry. Presentations and other meeting documents are available at `https://tinyurl.com/ceed1am`.

5.2 Outreach

CEED researchers were involved in a number of outreach activities, including 8 presentations at the Argonne Training Program on Extreme-Scale Computing (ATPESC17), as well as participation in the 8th International Conference on Numerical Methods for Multi-Material Fluid Flow (MultiMat 2017), the ASCR Applied Math PI meeting and the US Congress on Computational Mechanics. We are also organizing a 3-part minisimposium at the upcoming International Conference in Spectral and High-Order Methods (ICOSAHOM18) centered around the work in CEED and including key representatives of the international high-order community.

6. CONCLUSION

In this milestone we worked closely with the ExaSMR and MARBL teams to help them integrate and use the currently available CEED discretization technologies (Nek5000 and MFEM, respectively). We also helped the ExaAM team to develop a new MFEM-based miniapp for their project and engaged and planned future activities with Urban, ACME, GEOS, the Application Assessment and Proxy App projects. We delivered GPU-enabled versions of the Nekbone and Laghos miniapp, which are available through the CEED website, `http://ceed.exascaleproject.org` and the CEED GitHub organization, `http://github.com/ceed`. In this report, we also described additional CEED activities performed in Q4 of FY17, including: the CEED first annual meeting and participation of CEED researchers in a variety of 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-739479.

REFERENCES

[1] Matthew Otten, Jing Gong, Azamat Mametjanov, Aaron Vose, John Levesque, Paul Fischer, and Misun Min. An MPI/OpenACC implementation of a high order electromagnetics solver with GPUDirect communication. *The International Journal of High Performance Computing Application*, 30(3):320–334, 2016.

[2] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Jing gong, stefano markidis, erwin laure, matthew otten, paul fischer, misun min, nekbone performance on gpus with openacc and cuda fortran implementations, special issue on sustainability on ultrascale computing systems and applications. *Journal of Supercomputing*, 72:4047–4068, 2016.

[3] P. Fischer, K. Heisey, and M. Min. Scaling limits for pde-based simulation (invited). In *22nd AIAA Computational Fluid Dynamics Conference, AIAA Aviation*. AIAA 2015-3049, 2015.

[4] H.M. Tufo and P.F. Fischer. Fast parallel direct solvers for coarse-grid problems. *J. Parallel Distrib. Comput.*, 61:151–177, 2001.

[5] P.F. Fischer, J. Lottes, W.D. Pointer, and A. Siegel. Petascale algorithms for reactor hydrodynamics. *J. Phys. Conf. Series*, 125:012076, 2008.

[6] 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.

[7] David S. Medina, Amik St.-Cyr, and Timothy Warburton. OCCA: A unified approach to multi-threading languages. *CoRR*, abs/1403.0968, 2014.

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

[9] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven H. M. Buijssen, Matthias Grajewski, and Stefan Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Comput.*, 33(10-11):685–699, November 2007.

[10] Cris Cecka, Adrian J Lew, and Eric Darve. Assembly of finite element methods on graphics processors. *International journal for numerical methods in engineering*, 85(5):640–669, 2011.

[11] GR Markall, A Slemmer, DA Ham, PHJ Kelly, CD Cantwell, and SJ Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1):80–97, 2013.

[12] Graham R Markall, David A Ham, and Paul HJ Kelly. Towards generating optimised finite element solvers for gpus from high-level specifications. *Procedia Computer Science*, 1(1):1815–1823, 2010.

[13] Dominik Göddeke, Sven HM Buijssen, Hilmar Wobker, and Stefan Turek. Gpu acceleration of an unmodified parallel finite element navier-stokes solver. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 12–21. IEEE, 2009.

[14] Zhisong Fu, T James Lewis, Robert M Kirby, and Ross T Whitaker. Architecting the finite element method pipeline for the gpu. *Journal of computational and applied mathematics*, 257:195–211, 2014.

[15] Andreas Klöckner, Tim Warburton, Jeff Bridge, and Jan S Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[16] Andreas Klöckner, Timothy Warburton, and Jan S Hesthaven. High-order discontinuous galerkin methods by gpu metaprogramming. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 353–374. Springer, 2013.

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

[18] Jesse Chan, Zheng Wang, Axel Modave, Jean-Francois Remacle, and Tim Warburton. GPU-accelerated discontinuous Galerkin methods on hybrid meshes. *Journal of Computational Physics*, 318:142–168, 2016.

[19] Maryam Mehri Dehnavi, David M Fernández, and Dennis Giannacopoulos. Finite-element sparse matrix vector multiplication on graphic processing units. *IEEE Transactions on Magnetics*, 46(8):2982–2985, 2010.

[20] Paul Grigoraş, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. Optimising sparse matrix vector multiplication for large scale fem problems on fpga. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–9. IEEE, 2016.

[21] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[22] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. *Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis*, pages 129–148. Springer International Publishing, Cham, 2015.

[23] James Stevens and Andreas Klöckner. A unified, hardware-fitted, cross-GPU performance model. *arXiv preprint arXiv:1604.04997*, 2016.

[24] David H Bailey, Robert F Lucas, and Samuel Williams. *Performance tuning of scientific applications*. CRC Press, 2010.

[25] P.F. Fischer. Nek5000: Fast & scalable spectral element CFD solver. https://nek5000.mcs.anl.gov/. Accessed: 2017-09-26.

Algorithm 6 BP3.5: collocation differentiation for 3D hexahedral mesh (3D thread structure)

```
 1: for All elements e, in parallel do
 2:    for each thread i,j, k do
 3:       If k=0, load D to shared memory variable s_D;
 4:       Declare register variables r_qr, r_qs, and r_qt;
 5:       Synchronize threads
 6:       Load GwJ;
 7:       Declare variables qr, qs, qt and set them to 0.
 8:       for n = 1,...,Nq do
 9:          qr = qr + s_D[i][n]*q[n + j*Nq + k*Nq*Nq + e*Np];
10:          qs = qs + s_D[j][n]*q[i + n*Nq + k*Nq*Nq + e*Np];
11:          qt = qt + s_D[k][n]*q[i + j*Nq + n*Nq*Nq + e*Np];
12:       end for
13:       Set r_qr = qr, r_qs = qs and r_qt = qt;
14:       Aq[i + j*Nq + k*Nq*Nq + e*Np]
   =GwJ*lambda*q[i + j*Nq + k*Nq*Nq + e*Np];
15:    end for
16:    for each thread i,j, k do
17:       Load G00, G01, G02;
18:       Aqtemp[i + j*Nq + k*Nq*p_Nq + e*Np;
   = G00*r_qr + G01*r_qs + G02*r_qt;
19:    end for
20:    for each thread i,j, k do
21:       Declare variable tmp and set them to 0.
22:       for n = 1,...,Nq do
23:          tmp = tmp+ s_D[n][i]* Aqtemp[n + j*Nq + k*Nq*Nq + e*Np];
24:       end for
25:       Aq[i + j*Nq + k*Nq*Nq + e*Np]
   = Aq[i + j*Nq + k*Nq*Nq + e*N]+ tmp;
26:    end for
27:    for each thread i,j, k do
28:       Load G10, G11, G12;
29:       Aqtemp[i + j*Nq + k*Nq*p_Nq + e*Np;
   = G10*r_qr + G11*r_qs + G12*r_qt;
30:    end for
31:    for each thread i,j, k do
32:       Declare variable tmp and set them to 0.
33:       for n = 1,...,Nq do
34:          tmp = tmp+ s_D[n][j]* Aqtemp[i + n*Nq + k*Nq*p_Nq + e*Np];
35:       end for
36:       Aq[i + j*Nq + k*Nq*Nq + e*Np]
   = Aq[i + j*Nq + k*Nq*Nq + e*N]+ tmp;
37:    end for
38:    for each thread i,j, k do
39:       Load G10, G11, G12;
40:       Aqtemp[i + j*Nq + k*Nq*p_Nq + e*Np;
   = G20*r_qr + G21*r_qs + G22*r_qt;
41:    end for
42:    for each thread i,j, k do
43:       Declare variable tmp and set them to 0.
44:       for n = 1,...,Nq do
45:          tmp = tmp+ s_D[n][k]* Aqtemp[i + j*Nq + n*Nq*p_Nq + e*Np];
46:       end for
47:       Aq[i + j*Nq + k*Nq*Nq + e*Np]
   = Aq[i + j*Nq + k*Nq*Nq + e*N]+ tmp;
48:    end for
49: end for
```

Algorithm 7 BP3.0: differentiation for 3D hexahedral elements

Data: (1) Vector $\mathbf{q}$, size `Nelements` N_q^3, (2) differentiation matrix $\mathbf{D}$, size $gjN_q \times gjN_q$, (3) Interpolation matrix $\mathbf{I}$, size $glN_q \times N_q$, (4) geometric factors $\mathbf{G}$, size `Nelements` $\cdot gjN_q^3 \times 7$, (5) parameter λ ;
Output: Vector $\mathbf{Aq}$, size `Nelements` $\cdot N_q^3 \times 1$;
for $e = 1, 2, \ldots$ `NElements` **do**
  Interpolate $\mathbf{q}^{(\mathbf{e})}$ to $\mathbf{q}^{(\mathbf{e},\mathbf{GJ})}$, size $glNq^3 \times 1$
  for $i = 1, 2, \ldots gjN_q$ **do**
    for $j = 1, 2, \ldots gjN_q$ **do**
      for $k = 1, 2, \ldots gjN_q$ **do**
        ▷ Load geometric factors ↓
        `G00` $= \mathbf{G}^{(e)}_{1;kji}$, `G01` $= \mathbf{G}^{(e)}_{2;kji}$, `G02` $= \mathbf{G}^{(e)}_{3;kji}$;
        `G11` $= \mathbf{G}^{(e)}_{4;kji}$, `G12` $= \mathbf{G}^{(e)}_{5;kji}$, `G22` $= \mathbf{G}^{(e)}_{6;kji}$;
        ▷ Multiply by $\mathbf{D}$ ↓
        `qr` $= \sum_{n=1}^{N_{gjq}} \mathbf{D}_{in}\mathbf{q}^{(e,GJ)}_{kjn}$;
        `qs` $= \sum_{n=1}^{N_{gjq}} \mathbf{D}_{jn}\mathbf{q}^{(e,GJ)}_{kni}$;
        `qt` $= \sum_{n=1}^{N_{gjq}} \mathbf{D}_{kn}\mathbf{q}^{(e,GJ)}_{nji}$;
        ▷ Apply chain rule ↓
        `rqr`$^{(e)}_{ijk}$ = `G00*qr + G01*qs + G02*qt`;
        `rqs`$^{(e)}_{ijk}$ = `G01*qr + G11*qs + G12*qt`;
        `rqt`$^{(e)}_{ijk}$ = `G02*qr + G12*qs + G22*qt`;
      end for
    end for
  end for
  for $i = 1, 2, \ldots gjN_q$ **do**
    for $j = 1, 2, \ldots gjN_q$ **do**
      for $k = 1, 2, \ldots N_q$ **do**
        `GwJ` $= \mathbf{G}^{(e)}_{7;kji}$
        $\mathbf{Aq}^{(e,GJ)}_{ijk} = \lambda$`GwJ`$\mathbf{q}^{(e,GJ)}_{kji} + \sum_{n=1}^{gjN_q} \mathbf{D}_{ni}$`rqr`$^{(e)}_{kjn} + \mathbf{D}_{nj}$`rqs`$^{(e)}_{kin} + \mathbf{D}_{nk}$`rqr`$^{(e)}_{nji}$;
      end for
    end for
  end for
  Project $\mathbf{Aq}^{(e,GJ)}_{ijk}$ to $\mathbf{Aq}^{(e)}_{ijk}$;
end for