The Great CEED Bake-off DPC++ Edition

Kris Rowe
Argonne Leadership Computing Facility

Saumil Patel
Computational Science Division

oneAPI Developer Summit at SC21 | November 14, 2021

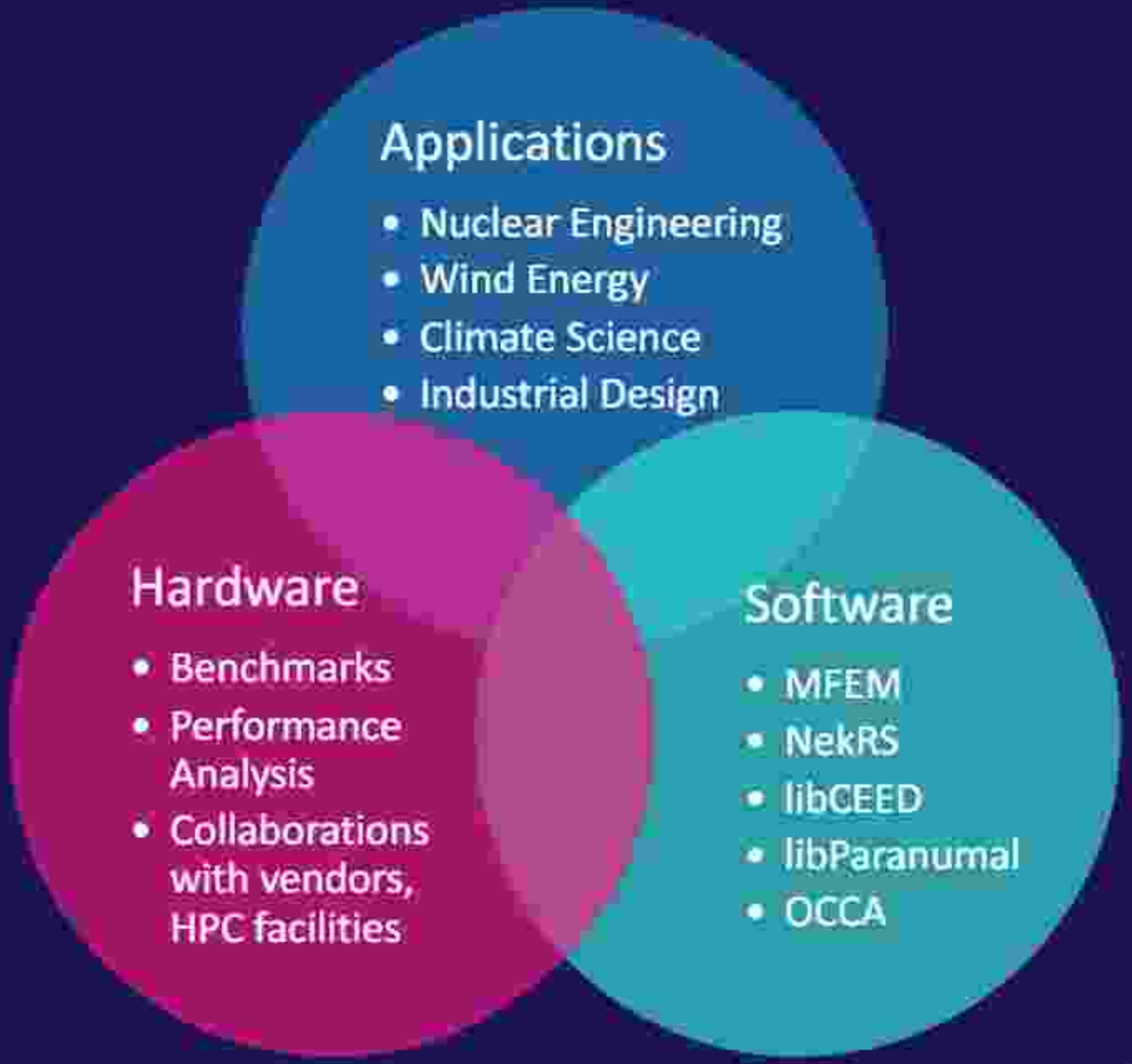
Shout-outs

This work was supported by Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357 and by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

The Big Picture

What is CEED?

- Centre for **E**fficient **E**xascale **D**iscretizations
- ECP Co-design centre
- Focused on high-order finite and spectral element methods for PDEs on unstructured meshes
- 30+ researchers from 2 U.S. DOE labs and 5 universities



A 60 Second Course on Element Methods

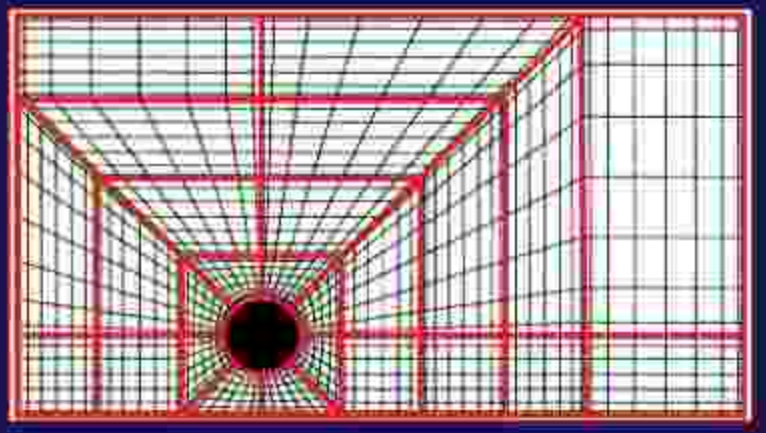
Partition the problem domain into disjoint (hexahedral) sub-domains, called elements

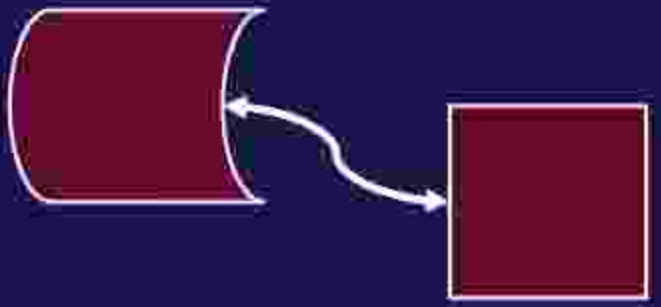
Map the tensor-product of 1D Gaussian quadrature nodes on the unit square/cube to each element to form a local mesh

Approximate the PDE's weak form using a polynomial basis in each element; enforce continuity at element interfaces

Decompose integrals as the sum of local integrals over each element. Use Gaussian quadrature to compute local integrals

Approximate derivatives on each element by differentiating local interpolating polynomials





$$\int_{\Omega} dx = \sum_{e} \int_{\Omega_e} dx$$

The CEED Bake-off Problems

- Collection of related matrix equations arising from FEM/SEM discretizations
- The *p*th order Gauss-Legendre-Lobatto (GLL) nodes are used for interpolation in each case
- The operator and number of components (e.g., scalar or vector field) change

Bake-off Kernels (BKs)

- The **local component** of an operator
- Calculation for each element is independent
- Values at element boundaries are computed redundantly

Bake-off Problems (BPs)

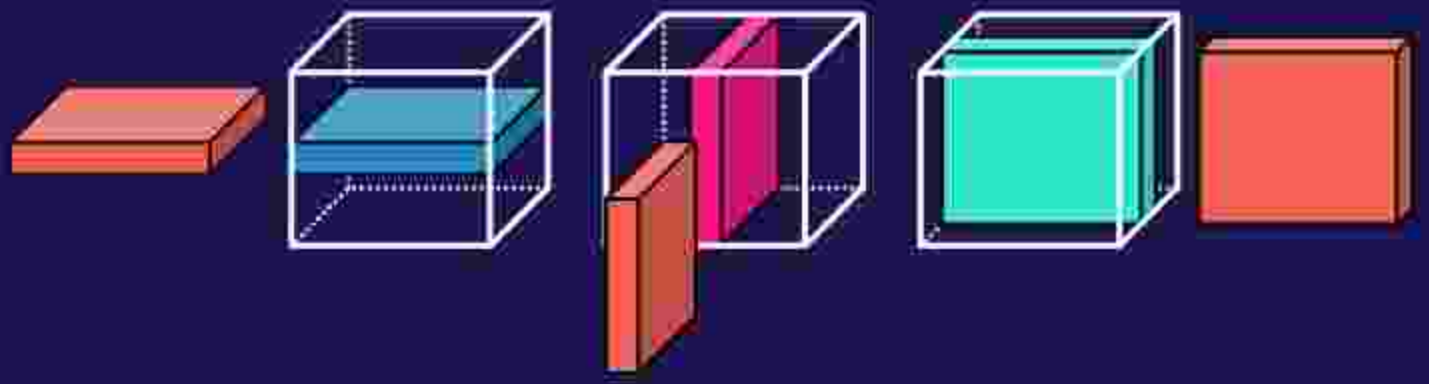
- Iterative solvers using **global operators**
- Preconditioned Krylov methods (e.g., PCG)
- Computation involves:
  1. The local component of an operator
  2. Summation of values along element boundaries (reduce-broadcast)

- Geometric information—the Jacobian and metric tensor—is precomputed for each element
- Implementations cannot use details about the mesh topology to speed-up computation

The CEED Bake-off Problems

	BP1	BP2	BP3	BP4	BP5	BP6
# of Components	Scalar	Vector	Scalar	Vector	Scalar	Vector
Global Operator	Mass Matrix		Stiffness (Poisson) Matrix			
Local 1D Operator	Interpolation Matrix		Differentiation Matrix		Differentiation Matrix	
Interpolation Nodes	$p+1$ Gauss-Legendre-Lobatto (GLL)					
Quadrature Nodes	$p+2$ Gauss-Legendre				$p+1$ GLL	

Tensor product structure means local operators are equivalent to multiplying slices of a 3-array by the same matrix.



Why do we

Care about the CEED Bake-off Problems?

Want to have a DPC++ implementation?

Preparing CSE applications for Aurora

Baseline for apps like NekRS on Intel CPUs/GPUs

Comparison with OCCA framework DPC++ backend

Sandbox to test benefits of new DPC++ features, libraries

Common DPC++ Implementation

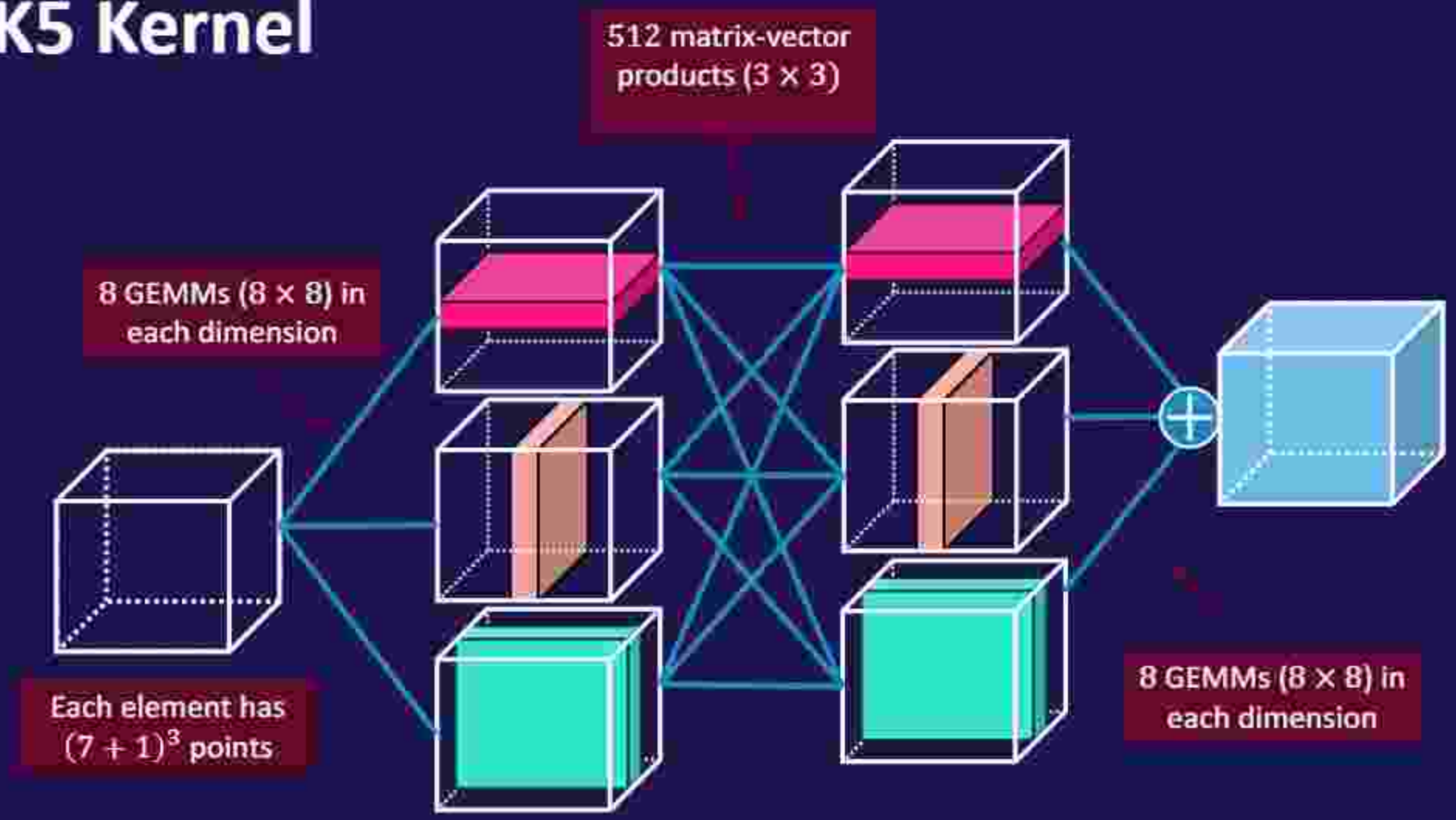
- Benchmarks share a core set of functions for
  - Gaussian quadrature nodes and weights
  - 1D interpolation, derivative matrices
  - Constructing 2D/3D meshes
  - Verification of kernel accuracy
  - Timing and statistics
- Memory
  - DPC++ USM is used
  - All memory is allocated on the device
  - Host-to-device `memcpy` occurs before kernel calls, isn't included in timings
  - Events are used to explicitly manage memory dependencies between kernels
- Kernels
  - Directly programmed DPC++ kernels are defined using lambdas
  - Kernels are called once for accuracy verification before any measurement
  - This should trigger JIT compilation so that it is not included in timings
- Verification
  - A serial C++ version of each kernel is used to calculate a reference solution on the host
- Measurement
  - Each kernel is run multiple times
  - Event profiling is used for kernel timings
  - Execution time is recorded *for each* kernel call

Direct Programming

- A C++ (host) function is used to wrap each of the DPC++ kernels (lambdas)
  - First argument is a queue
  - Last argument is a vector of event dependencies
- Choose function arguments to match oneMKL routines where equivalent ones exists
- BLAS-1 type kernels use the largest 1D work-group size supported by a device.
  - copy
  - scal
  - axpy
  - dotw*
  - normw*
  - Fused axpy-dotw

```
auto e = q.submit(
  [&](sycl::handler & handler_) {
    handler_.depends_on(dependencies);
    auto reduce_xdoty = sycl::reduction(
      xdoty,
      sycl::plus<>(),
      {sycl::property::reduction::initialize_to_identity()}
    );

    handler_.parallel_for<class Dotw>(
      kernel_range,
      reduce_xdoty,
      [=](auto item_, auto& xdoty_) {
        const auto global_index = item_.get_global_linear_id();
        const auto local_index = item_.get_local_linear_id();

        auto & work = *(
          sycl::group_local_memory_for_overwrite<double[block_size]>(item_.get_group())
        );

        work[local_index] = x[global_index] * y[global_index] * w[global_index];
        item_.barrier(sycl::access::fence_space::local_space);

        reduce::work_group_reduce<block_size>(item_,work);

        if (0 == local_index)
          xdoty_ += work[0];
      });
```

BK5 Kernel



Replacing Kernels with oneMKL

- Characterized by tensor-product forms, the CEED BPs are highly dependent on fast matrix-matrix multiplication routines
- Consider the BK5 kernels:

$$K\underline{u} := \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}^T \begin{pmatrix} G_{11} & G_{12} & G_{13} \\ G_{21} & G_{22} & G_{23} \\ G_{31} & G_{32} & G_{33} \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix} \underline{u}$$

$$\underline{u} := \{\underline{u^e}\}_{e=1}^{E}, \underline{u^e} := u^e(i,j,k)$$

- Array elements in u^e are arranged with consecutive entries in memory advancing with leading *i* index
  - u^e can be referenced in any way that is convenient for the given operation
- The application of D_i and D_i^T requires many mat-mat multiplications *per element*. Typically runs require O(10^3) elements per rank (or GPU)

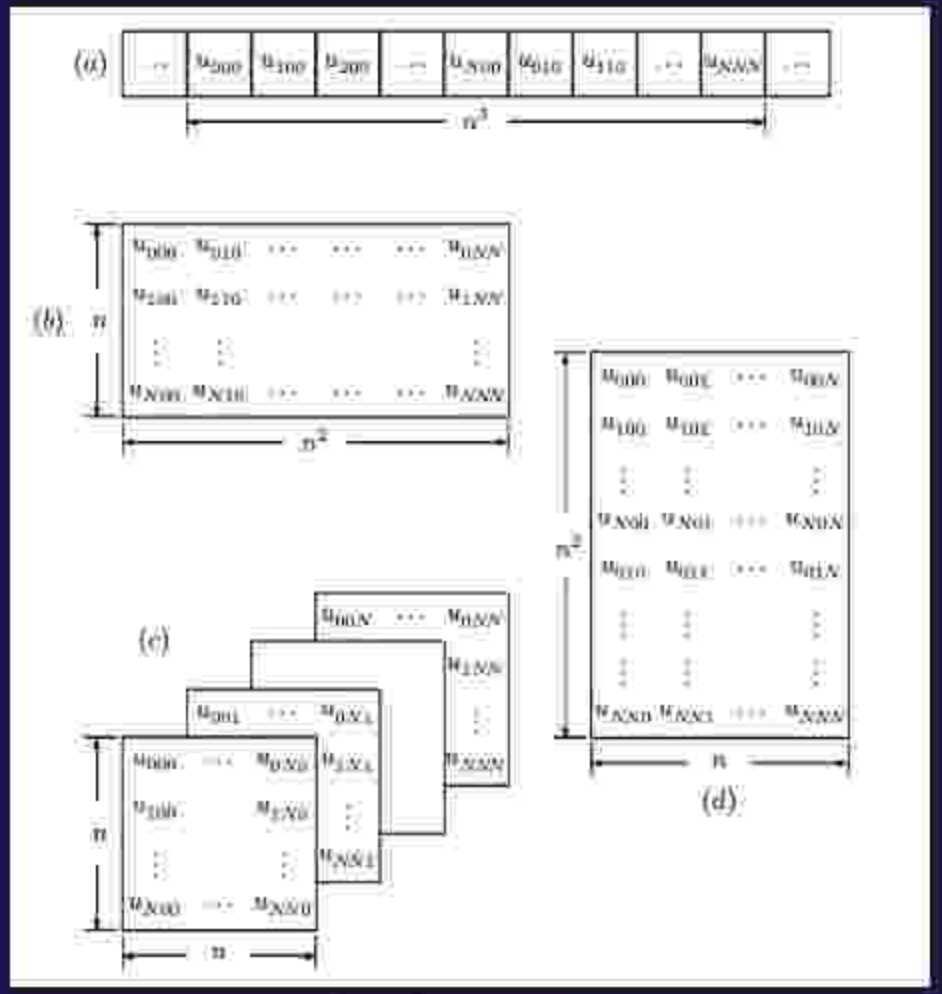


Fig: Interpretation of 3D data array
Source: Deville, Fischer, Mund (2002)

Replacing Kernels with oneMKL

- Often, the BK5 kernel (a.k.a. "axhelm"), is performed in a loop over all elements local to the rank (or device)
- The many mat-mat multiplications in BP5 lend themselves well to being called by the oneMKL API through `gemm_batch` calls

$D_1\,\underline{u}$

$D_2\,\underline{u}$

$D_3\,\underline{u}$

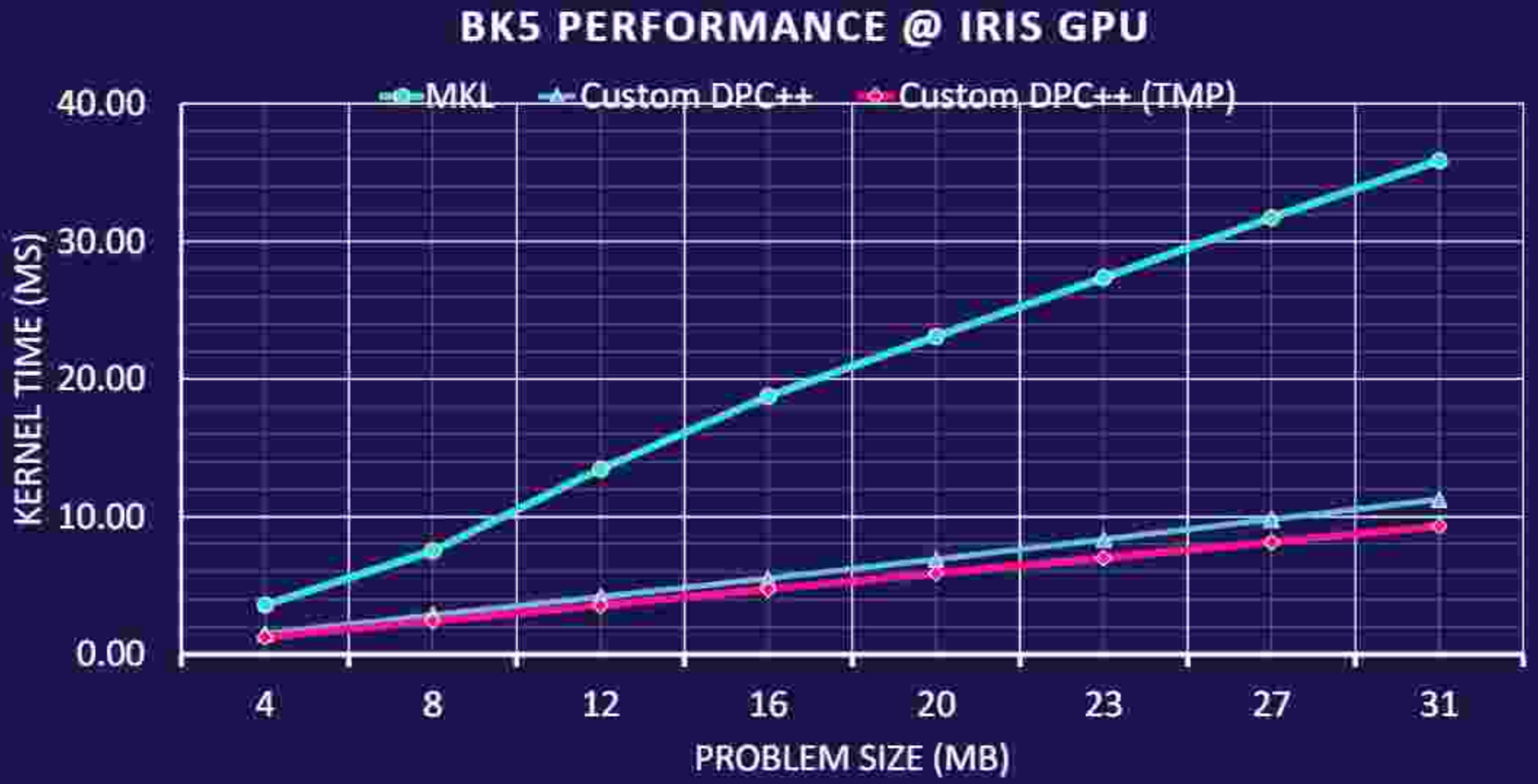
```
//Partial Derivatives
gemm_batch_e1 = oneapi::mkl::blas::gemm_batch(main_queue, ta,
                                              tb, m_r, n_r, k_r, alpha_r,
                                              {(const fp**) D_r, (size_t) E_l}, lda_r,
                                              {(const fp**) UVr, (size_t) E_l}, ldb_r, beta_r,
                                              Urb, ldc_r, 1, group_size_r, gemm_batch_dependencies);

gemm_batch_e2 = oneapi::mkl::blas::gemm_batch(main_queue, ta,
                                              tbt, m_s, n_s, k_s, alpha_s,
                                              {(const fp**) UVs, (size_t) E_l*Nq}, lda_s,
                                              {(const fp**) D_s, (size_t) E_l*Nq}, ldb_s, beta_s,
                                              Usb, ldc_s, 1, group_size_s, gemm_batch_dependencies);

gemm_batch_e3 = oneapi::mkl::blas::gemm_batch(main_queue, ta,
                                              tbt, m_t, n_t, k_t, alpha_t,
                                              {(const fp**) UVt, (size_t) E_l}, lda_t,
                                              {(const fp**) D_t, (size_t) E_l}, ldb_t, beta_t,
                                              Utb, ldc_t, 1, group_size_t, gemm_batch_dependencies);
```

- Gemm_batch performs the tensor-product-based, local partial derivatives over *all* elements

Setup

- We compare the performance of three implementations
  - oneMKL – using GEMM_BATCH
  - Custom DPC++
  - Templated Custom DPC++
- Performance measure is mean kernel execution time (ms) over 4000 trials
  - Execution time is recorded *for each* kernel call
  - Event profiling is used for kernel timings
- Problem size is measured in MB for the input array on which the Laplacian is operated.
  - Varies by increasing the total number of elements from 1000 to 8000 @ a fixed polynomials order (7).

Results



Results

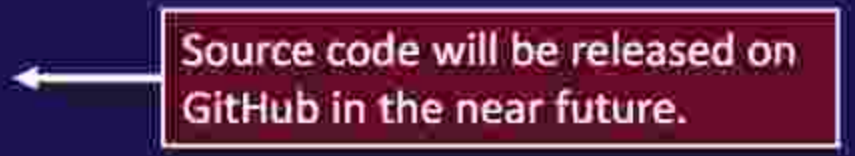
Next time on *The Great CEED Bake-off*

- *Outline some of the next steps for this project …*

Links

Checkout the following links for more info

1. CEED Bake-off Problems
2. The Great CEED Bake-off on Intel DevMesh



Takeaways

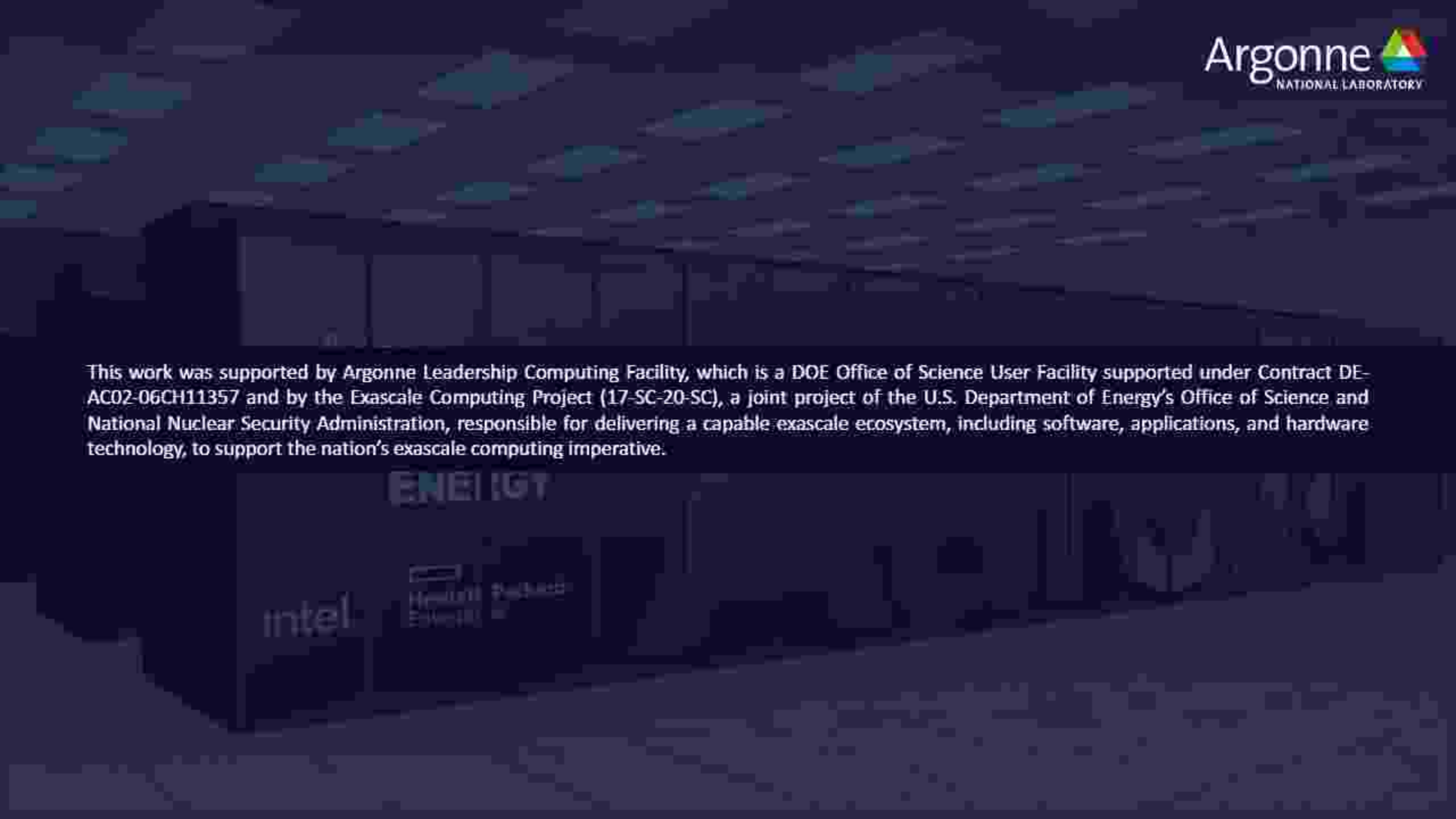
The CEED Bake-off Problems are performance critical kernels for high-order finite and spectral element applications

Coarse grain parallelism comes from decomposition of problem domain into elements; fine grain parallelism is due to tensor product grids

Replacement of directly programmed kernels with oneMKL routines gave similar performance for BLAS-1 type functions

A directly programmed kernel which utilized the specialized structure of tensor product matrices gave better performance than using batched gemm calls

DPC++ features, such as work-group collectives and asynchronous global-to-local copies, will be explored to improve performance in the future



This work was supported by Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357 and by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.