High-Order Finite Element Method using Standard and Device-Level Batch GEMM on GPUs

Natalie Beams, Ahmad Abdelfattah, Stan Tomov, Jack Dongarra
Innovative Computing Laboratory
University of Tennessee, USA
{nbeams,ahmad,tomov,dongarra}@icl.utk.edu

Tzanio Kolev, Yohann Dudouit
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, USA
{kolev1,dudouit1}@llnl.gov

***Abstract*—We present new GPU implementations of the tensor contractions arising from basis-related computations for high-order finite element methods. We consider both tensor and non-tensor bases. In the case of tensor bases, we introduce new kernels based on a series of fused device-level matrix multiplications (`GEMMs`), specifically designed to utilize the fast memory of the GPU. For non-tensor bases, we develop a tuned framework for choosing standard batch-BLAS `GEMMs` that will maximize performance across groups of elements. The implementations are included in a backend of the libCEED library. We present benchmark results for the diffusion and mass operators using libCEED integration through the MFEM finite element library and compare to those of the previously best-performing GPU backends for stand-alone basis computations. In tensor cases, we see improvements of approximately 10-30% for some cases, particularly for higher basis orders. For the non-tensor tests, the new batch-`GEMMs` implementation is twice as fast as what was previously available for basis function order greater than five and greater than approximately 10^5 degrees of freedom in the mesh; up to ten times speedup is seen for eighth-order basis functions.**

***Index Terms*—Tensor contractions, finite elements, high-order methods, matrix-free FEM, GPU, batched linear algebra**

I. Introduction

The Center for Efficient Exascale Discretizations (CEED) [1] is a co-design center of the Exascale Computing Project, with the goal of providing scientific application software with tools for incorporating effective and accurate high-order discretization methods that fully utilize current and future high-performance computing hardware. A key focus of the CEED project is high-order finite element methods with matrix-free evaluation, which requires less memory and fewer FLOPs per "matrix"-vector application than standard methods involving the assembly of sparse matrices for finite element operators [2], [3]. In addition to high-level finite element libraries MFEM [4], Nek5000/NekRS [5], and libParanumal [6], the CEED project is developing libCEED [7], which gives applications a flexible, versatile, low-level API for defining high-order operators for matrix-free evaluation.

This research was supported by NVIDIA and 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, in support of the nation's exascale computing imperative.

libCEED has multiple backends to provide performance for a variety of use cases across many CPU and GPU architectures. The work presented here is developed within the framework of improving the performance of the libCEED MAGMA [8], [9] backend, specifically the basis computations for both tensor and non-tensor bases. We describe new algorithms for device-level batch-`GEMM`-type operations for tensor bases, and present examples of tuning standard batch `GEMM` for non-tensor bases. Though the algorithms presented are designed for high-order finite element methods, the tensor basis kernels could further be adapted for applications with tensor contractions of similar sizes.

II. Related Work

Many physical systems of interest can be entirely or partially written in terms of tensor contractions [10]. Accordingly, there has been much interest recently in efficient algorithms and code generation for general tensor contractions [11]–[13], as well as tensor contraction implementations for GPUs [14], [15]. For tensor contractions arising from high-order finite element methods, some of the authors have previously investigated the use of batch-`GEMMs` to perform tensor contractions [16]. Here, we improve the performance through formulating a series of tensor contractions to share the same execution context, thus increasing the memory bandwidth by maximizing data reuse. Świrydowicz et al. demonstrated highly-tuned optimizations of the specific computations required for the CEED bake-off problems as part of a corresponding "bake-off kernel" study [17]; however, these kernels are optimized for each finite element operator, and cannot be used within libCEED's more general framework, which is explained further in Section III. Instead, we focus solely on the "basis actions" of a fully-compliant libCEED backend. Our series of fused device-level batch-`GEMM` actions is similar to the approach of Springer and Bientinesi for general tensor contractions on CPUs [18].

III. Design Overview of a libCEED Backend

libCEED aims to define and provide an interface to a general format representing the operators from high-order discretizations, for which building a sparse matrix is not the most efficient choice for storage or use [7]. This format is based on an algebraic factorization that can be written for a general finite element operator A as

$$A = P^{\mathrm{T}} \underbrace{G^{\mathrm{T}} B^{\mathrm{T}} D B G}_{\text{libCEED operations}} P. \tag{1}$$

The P operator is related to the management of a distributed parallel mesh and is handled by the high-level application code. libCEED handles local computations, with its functionality accessed through a `Ceed` struct instance created by a distributed process. The $B^{\mathrm{T}} D B$ sequence of operations represents, for each local process, the transformation of its elements to the reference element's quadrature points and the subsequent numerical integration of terms in the weak form of the equation to be solved.

A. libCEED Operators

1) The G Operator – Element Restrictions: The libCEED element restrictions are mappings between what libCEED calls "L-vectors," containing all the degrees of freedom that are local to a process, and "E-vectors," with degrees of freedom ordered by element. In an E-vector, nodes on element boundaries will be repeated in multiple elements for continuous finite element spaces. G is defined as the action that takes an L-vector and returns an E-vector, and G^{T} provides the corresponding reverse mapping.

2) The B Operator – Basis Actions: The B operator takes the E-vector produced by the element restriction and computes values for each basis function at the quadrature points of each element. The specific action of B on the basis functions is determined by the operator (e.g. interpolation, gradient). In the tensor case, this action is represented through a series of one-dimensional tensor contractions for each element, as detailed in Section IV-A; for non-tensor bases, we structure the basis actions as standard dense matrix-matrix multiplications (see Section V). The transpose operator, B^{T} is handled similarly, with an additional sum over the dimensional component in the case of the transpose gradient action. The improvement of these basis operators is the emphasis of this work.

3) The D Operator – QFunctions: A key difference between libCEED's factorized approach and other high-order finite element implementations is the use of the general user-defined "QFunction." This function operates solely on the quadrature points. It involves computations related to the mesh transformations and the physics of the equation. This approach provides greater flexibility and ease of implementing new operators, at the cost of reducing some opportunities for optimization. The general interface for providing a user-defined QFunction also allows the use of QFunctions from other sources, such as automatic differentiation libraries or the output from functions in outside software.

B. libCEED Backends and Interoperability

Each libCEED backend implements actions related to the three main libCEED operators, plus a high-level operator that combines the libCEED actions as listed in equation 1, and supplemental functions related to memory management. The libCEED backend structure aims to minimize code duplication through delegation. The MAGMA backend, for example, delegates QFunction application to the non-fused CUDA backends.

At the time of this work, only two CUDA backends implemented non-tensor basis actions: `cuda-ref`, the reference CUDA backend, and MAGMA. For tensor bases, the best performance is achieved by operator fusion with runtime compilation in the `cuda-gen` backend. Here "operator fusion" refers to creating one kernel to perform the entire high-level operation, rather than applying each sub-operator (G, B, D, B^{T}, and G^{T}) separately. Prior to the work detailed here, the best non-fused CUDA backend was `cuda-shared`, so named because it utilizes the GPU's shared memory to increase performance. In cases where fusion is not possible (e.g. not enough GPU memory available for the fused kernel or the need for a QFunction provided through an external library or source, which cannot be converted to code for runtime compilation), it is important to also have fast "stand-alone" kernels for the computationally-intensive basis actions.

IV. Tensor Basis Computations

We begin with some general definitions related to the kernel design:

- p: Number of nodes in one direction of the tensor basis. It is equal to $(\hat{p}+1)$, where $\hat{p}$ is the order of basis functions.
- q: Number of nodes in one direction of the tensor quadrature rule. It is usually equal to $(\hat{p}+2)$ or so, but could be $= \hat{p}$ or $< \hat{p}$.
- $\bar{P}$: Total number of nodes in each component in an element. For the tensor case, it is equal to p^{dim}.
- $\bar{Q}$: Total number of quadrature nodes in an element. For the tensor case, it is equal to q^{dim}.

The MAGMA backend currently provides optimized GPU kernels for three basis actions: `interp`, `grad`, and `weight`, We will now describe these actions.

A. Tensor Basis Actions

The `interp` action interpolates the basis functions to the quadrature points on the reference element. In the three-dimensional tensor-grid case, we can write the interpolation operator as a six-dimensional tensor, $\mathcal{J}_{lmnijk}$, where $l, m, n \in [1, q]$ are indices corresponding to the quadrature points and $i, j, k \in [1, p]$ are indices for the basis nodes. This $\mathcal{J}$ operator is the tensor product of its one-dimensional equivalent, $\hat{\mathcal{J}}$, a rank-two tensor of size $q \times p$:

$$\mathcal{J}_{lmnijk} = \hat{\mathcal{J}}_{li} \otimes \hat{\mathcal{J}}_{mj} \otimes \hat{\mathcal{J}}_{nk}. \tag{2}$$

The `interp` action takes an input u_{ijk} containing values of a function at the basis nodes of an element and returns v_{lmn}, the interpolation of this function at the quadrature points.

The `grad` action evaluates the gradient of the basis functions at the quadrature points. We represent it as three separate series of tensor contractions, $\mathcal{D}^x$, $\mathcal{D}^y$, and $\mathcal{D}^z$,

$$\begin{aligned}\mathcal{D}^x_{lmnijk} &= \hat{\mathcal{D}}_{li} \otimes \hat{\mathcal{J}}_{mj} \otimes \hat{\mathcal{J}}_{nk}\\ \mathcal{D}^y_{lmnijk} &= \hat{\mathcal{J}}_{li} \otimes \hat{\mathcal{D}}_{mj} \otimes \hat{\mathcal{J}}_{nk}\\ \mathcal{D}^z_{lmnijk} &= \hat{\mathcal{J}}_{li} \otimes \hat{\mathcal{J}}_{mj} \otimes \hat{\mathcal{D}}_{nk}, \qquad (3)\end{aligned}$$

where $\hat{\mathcal{D}}$ is again a rank-two tensor of size $q \times p$, corresponding to the evaluation of the derivative of the one-dimensional basis functions at each quadrature point on the reference line. The `grad` action takes the same input of values, u_{ijk}, and produces v_{dlmn}, the gradient of the input function at the quadrature points, with the d index referring to the dimension of the partial derivative.

Unlike the `interp` and `grad` actions, the `weight` action takes no input, but merely builds the tensor product of the one-dimensional quadrature weights $\hat{w}$ to create the three-dimensional weights, $\mathcal{W}_{lmn}$, as

$$\mathcal{W}_{lmn} = \hat{w}_l \otimes \hat{w}_m \otimes \hat{w}_n. \qquad (4)$$

The `weight` action does not have a transpose action, as it only computes something for the quadrature points, and has no correlation with the basis nodes.

B. Design Outlines

The GPU kernels for the basis actions share some common design outlines. **First**, the core computational work of each action is implemented using a GPU device routine instead of a kernel. This enables calling the routine in different kernels that perform a certain action. For example, a 3D `grad` action reads a 1D vector and writes a 3D vector in a non-transposed mode, and reduces a 3D vector to a 1D vector in the transposed mode. However, both kernels call the same device routines. **Second**, the device-level basis action operates only on the shared memory or the register file. Any global memory transactions are handled separately in other device routines. **Third**, apart from temporary scalar variables, no device routine allocates shared memory buffers or register arrays. These are usually defined at the kernel level, and passed to the device routines. **Fourth**, all device routines assume the same thread configuration. Such a property would allow the MAGMA backend to fuse multiple actions into one kernel in future developments.

C. Device-level Arguments

The `interp` and `grad` basis actions accept three main arguments. **The first** is an input vector u of size p^{dim} per each component. Due to data layout considerations in libCEED, we will make a distinction between the components of the field u (i.e., whether it is scalar- or vector-valued) and the components *added* through the non-transpose `grad` action. These gradient components will be referred to with `dim`. The vector is read-only. We use p^{dim-1} threads to read the input vector in a 3D register array `rU[dim][ncomp][p]`. **The second argument** is an input/output vector v of size q^{dim} per component. We use q^{dim-1} threads to read/write the vector using another register array `rV[dim][ncomp][q]`. **The third argument** is one or more constant basis matrices. These are the $\hat{\mathcal{J}}$ and $\hat{\mathcal{D}}$ basis matrices defined in IV-A. Regardless of the transposition mode of the basis action, constant basis matrices are always stored in $p \times q$ buffers in the shared memory of the GPU.

As mentioned previously, the `weight` action does not operate on an input vector u, and uses a constant vector of the quadrature weights ($\hat{w}$) in place of a basis matrix. It has an output v.

D. Kernel Configurations

Each kernel in the MAGMA backend performs one basis action by allocating the necessary register arrays and shared memory, reading the inputs, performing the action, and writing the output using a sequence of calls to the appropriate device routines. The developed kernels are batched across independent elements, with a default configuration of one thread-block per element. The thread configuration is $max(p,q)^{dim-1}$. However, sometimes this can lead to inefficient use of warps (e.g., 1D operators would need 1 thread per thread-block). This is why we allow one thread block to process multiple elements using parallel groups of threads. The sizes `dim`, `ncomp`, `p`, and `q` are compile-time constants that are passed as C++ template parameters for the kernels. As an example, consider a 3D `interp` action with $(p,q) = (2,3)$. This means $\bar{P} = 2^3 = 8$, and $\bar{Q} = 3^3 = 27$, and so thread-blocks would use $3^{3-1} = 9$ threads per element. Figure 1 is a high-level representation of such a basis action.

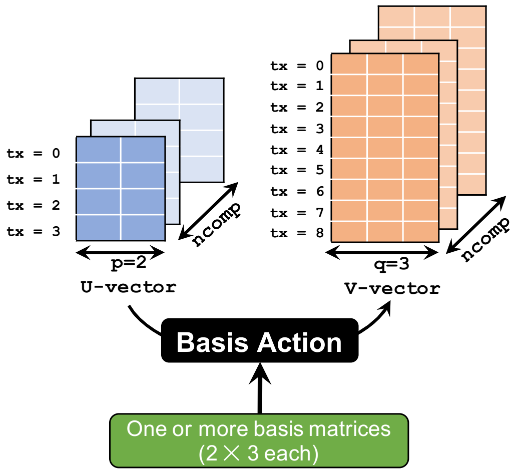

Fig. 1. A high-level view of a 3D basis action kernel for $(p,q) = (2,3)$. The u-vector is read using 4 threads, while the v-vector is read/written using 9 threads.

E. Tensor Contraction as Batch GEMM

The `interp` and `grad` basis actions call a tensor contraction function at their core. The reference CPU backend in libCEED implements the tensor contraction as follows:

```
int tensor_contract(
        int A, int B, int C, int J,
        const double* s, CeedTransposeMode tmode,
        int Add,
        const double *x, double* y)
{
  int tstride0 = B, tstride1 = 1;
  if (tmode == CEED_TRANSPOSE) {
    tstride0 = 1; tstride1 = J;
  }

  if (!Add)
    for (int q=0; q<A*J*C; q++)
      y[q] = (CeedScalar) 0.0;

  for (int a=0; a<A; a++)
    for (int b=0; b<B; b++)
      for (int j=0; j<J; j++) {
        CeedScalar tq = s[j*tstride0 + b*tstride1];
        for (int c=0; c<C; c++)
          y[(a*J+j)*C+c] += tq * x[(a*B+b)*C+c];
      }
  return 0;
}
```

This serial CPU code can be interpreted as a batch GEMM operation. Recall that GEMM is defined as ($\bar{C}_{m\times n} = \alpha\bar{A}_{m\times k} \times \bar{B}_{k\times n} + \beta\bar{C}_{m\times n}$). Figure 2 is a GEMM -like interpretation of the code above. Each of the x and y vectors can be represented using an array of independent matrices, of sizes C×B and C×J, respectively. The constant B×J matrix s represents the constant basis matrix for such contraction. The variables stride0 and stride1 handle the transposition of the s matrix. The Add option can be handled through the scalar β in the GEMM equation. At all times, the scalar α is set to one.

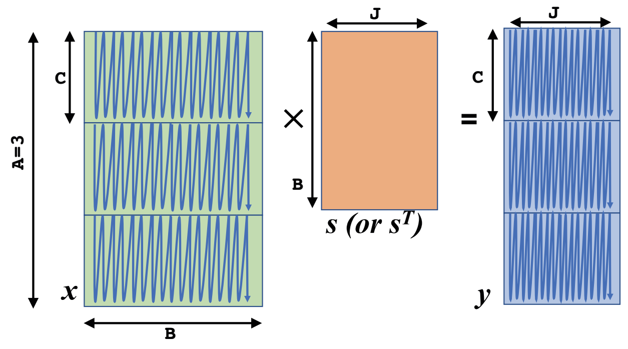

Fig. 2. Batch GEMM representation of a single tensor contraction in libCEED .

F. Example: 3D interp and grad Basis Actions

The interp and grad basis actions are represented as a sequence of tensor contractions. The 3D interp action performs three tensor contractions (i.e. three batch GEMMs), while the grad action requires nine batch GEMMs (three for each dimension). We consider an example of a 3D interp basis action for (p, q) = (2, 3), and for one component only. The three batch GEMMs are shown in Figures 3 through 5. As the computation progresses, the batch size becomes smaller (divided by p), but the individual matrix size becomes larger (rows multiplied by q, and columns fixed at p, except for the final output). The intermediate outputs are transformed in shared memory as a pre-processing step before the next product.

The first product uses the u-vector as an input, which is represented as a batch of single-row matrices (size $1\times p$). Each thread possesses one row of the u-vector, and independently computes the corresponding single-row output matrix (size $1\times q$).

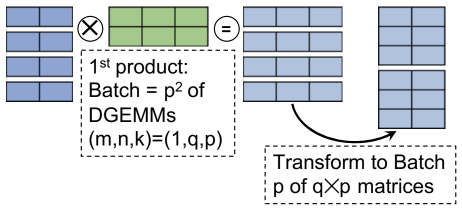

Fig. 3. First product in a 3D interpolation basis action. For (p, q) = (2, 3), the first product is a batch DGEMM with 4 operations of size (m, n, k) = (1, 3, 2).

The output matrices of the first product are transformed in shared memory into a batch p of $q\times p$ matrices. The transformation also reorganizes the threads so that they are properly indexed in their respective GEMM operations. The output is a batch p of $q\times q$ matrices.

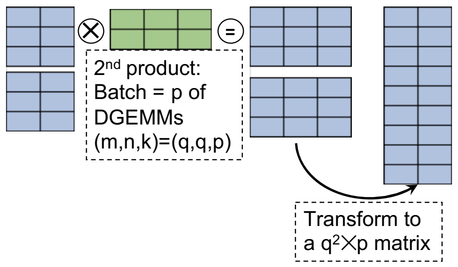

Fig. 4. Second product in a 3D interpolation basis action. For (p, q) = (2, 3), the second product is a batch DGEMM with 2 operations of size (m, n, k) = (3, 3, 2).

The final product is one GEMM operation, so the batch size is one. The output of the second product is transformed into a single matrix of size ($q^2\times p$). The final output is a $q^2\times q$ matrix that is stored in the rV register array.

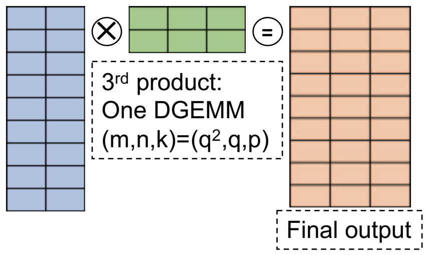

Fig. 5. Third product in a 3D interpolation basis action. For (p, q) = (2, 3), the third product is single DGEMM operation of size (m, n, k) = (9, 3, 2).

A 3D grad basis action can be viewed as performing the interp action three times for each dimension idim ∈{1, 2, 3}. There are some differences, though. **First**, the `grad` basis action takes two basis matrices instead of one: `dinterp1d`, which is the one-dimensional interpolation operator $\hat{\mathcal{J}}$, and `dgrad1d`, the one-dimensional gradient matrix $\hat{\mathcal{D}}$. **Second**, three batch `GEMMs` take place for each value of `idim`. The `dgrad1d` matrix is used in the first batch `GEMM` for `idim`$=0$, in the second batch `GEMM` for `idim`$=1$, and in the third batch `GEMM` for `idim`$=2$. Otherwise, the `dinterp1d` matrix is used; this corresponds to the combination of $\hat{\mathcal{J}}$ and $\hat{\mathcal{D}}$ in Eq. 3. **Third**, for the non-transposed mode, we read one u-vector for all values of `idim`, and produce three different v-vectors. **Fourth**, for the transposed mode, we read a different u-vector for each value of `idim`, and accumulate the result across `idim` into one v-vector.

V. Nontensor Basis Computations

In the case of non-tensor bases, the operators for the `interp`, `grad`, and `weight` actions cannot be represented as tensor products of one-dimensional matrices. Instead, in libCEED, the user provides matrices corresponding to the interpolation (J), gradient (D), and quadrature weights (W) at every quadrature node in the one-, two-, or three-dimensional element. This means the `weight` action for non-tensor is no longer a computation; we will focus solely on `interp` and `grad` . The `interp` action for an element is now a standard matrix-matrix multiplication,

$$v_l = J_{li} u_i, \tag{5}$$

with input u and output v represented as vectors, since they are no longer ordered on a tensor grid. The matrix J_{li} will be of size $\bar{Q} \times \bar{P}$. The `grad` action is similar, except the gradient matrix D is of size $(dim \times \bar{Q}) \times \bar{P}$, with dim blocks of $\bar{Q}$ rows for each component of the gradient.

Because we need to compute these matrix multiplications for every element, we can think of an input vector for multiple elements as a matrix U_{ie}, with each column representing an element. Then the interpolation action for all elements can be written as one large matrix-matrix multiplication,

$$V_{le} = J_{li} U_{ie}, \tag{6}$$

and similarly for the `grad` action. In the case of a vector-valued field, we can replace the index e with $E = e + c * N_e$, where $c \in [1, \text{ncomp}]$ is the component and N_e is the total number of elements being processed by the basis action.

A. *Standard vs. Batch GEMM*

As we have formulated the non-tensor basis action in terms of standard matrix multiplication, the actions can be performed with standard `GEMM` calls. Figure 6 shows the typical shape of the `GEMM` call in libCEED. The dimensions of the matrices (m, n, k) usually involve small values of m and k, but a large value of n. While the size range may vary, we consider the typical dimensions in the libCEED bake-off problems.

Ideally, a single `GEMM` operation would be enough to reach the GPU peak performance (e.g. using `cublasDgemm`). However, the large value of n compared to the small m and k values can hinder performance. Therefore, we also consider performing the `DGEMM` operation in Figure 6 as a batch `DGEMM` operation, splitting the problem across the n dimension to potentially create a more balanced workload for the GPU. The batch has the same $\bar{A}$, with different $\hat{B}$ and $\hat{C}$ matrices within a fixed stride from each other. Factoring n into `batchCount`$\times\eta$, describing the number of batch `GEMM` calls and the number of columns in each $\hat{B}$ and $\hat{C}$ (η), facilitates performance tuning for the batch `GEMM` call. Transforming the `DGEMM` in Figure 6 into a batch `DGEMM` does not require setting up pointer arrays that may impact the performance. Both cuBLAS and MAGMA provide stride-based batch `DGEMM` kernels.

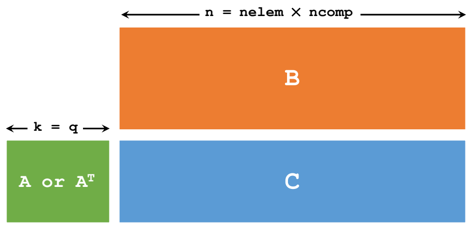

Fig. 6. Shape of the `DGEMM` operation for the non-tensor basis action in libCEED.

Figures 7, 8, and 9 show three different behaviors for the best performing `DGEMM` configuration on three different problem sizes. The $(\bar{P}, \bar{Q})$ sizes are typical in the standard MFEM benchmarks for libCEED . We also assume a relatively large n =`ncomp`$\times$`nelements`=10,000 in order to test the asymptotic performance of the GPU. Each figure marks the achieved performance of the "non-batch" `DGEMM` kernel in the cuBLAS and MAGMA libraries with horizontal dashed lines. Each figure also shows various performance numbers for the batch `DGEMM` kernels in both libraries according to different (`batchCount`, η) pairs. By trying out different combinations of (`batchCount`, η), we can find some cases where the batch kernels achieve better performance than a single `DGEMM`.

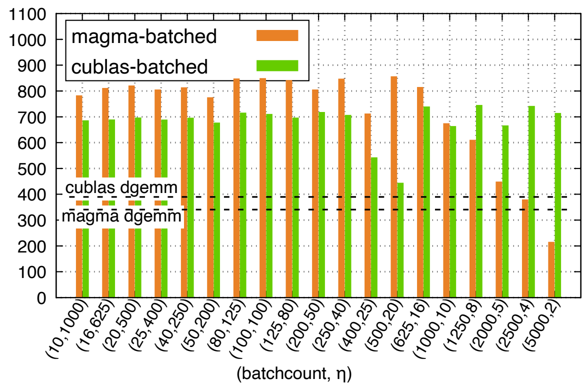

Fig. 7. Performance of different `DGEMM` configurations using cuBLAS and MAGMA. Results are shown for $(\bar{P}, \bar{Q}) = (27, 64)$ on a Tesla V100 GPU using CUDA 10.1 Toolkit.

For relatively small sizes (Figure 7), the batch DGEMM routine in MAGMA is the best performing kernel. Medium sizes such as the ones in Figure 8 show a winning scenario for the batch cuBLAS kernel. As we increase the sizes of $(\bar{P}, \bar{Q})$, we reach an asymptotic behavior, where the non-batch cuBLAS DGEMM is on par with its batch variant. Both of the cuBLAS kernels are within 85% of the GPU peak performance. In this case, it is usually better to call the non-batch kernel, since the subdivision size η that achieves the best performance might not fully divide the original n.

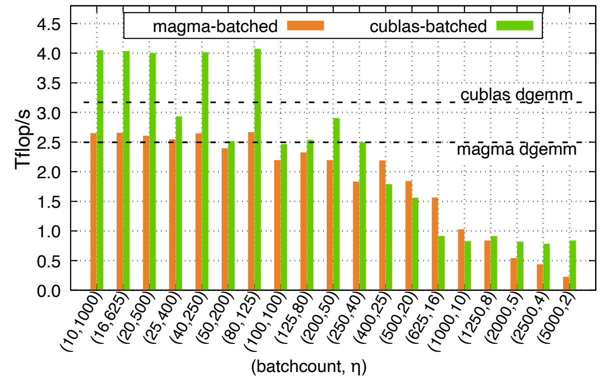

Fig. 8. Performance of different DGEMM configurations using cuBLAS and MAGMA. Results are shown for $(\bar{P}, \bar{Q}) = (125, 216)$ on a Tesla V100 GPU using CUDA 10.1 Toolkit.

All the autotuning sweeps for the best performing (batchCount, η) pair against the regular DGEMM kernels were conducted offline. The collected results led to the development of a very lightweight layer that selects the best performing kernel out of the four variants that have been tested during the offline sweep. Although the non-tensor basis actions usually trail the tensor basis mode in performance (because of the extra computation, scaling with $\bar{P}\bar{Q}$ rather than $\sim \hat{p}^4$ for the tensor case), the former is more portable due to the reliance on standard kernels that are usually highly optimized by vendors and widely-used open source libraries.

VI. LIBCEED BENCHMARKS: BAKE-OFF PROBLEMS

To compare and improve performance across a number of high-level finite element libraries involved in the project, CEED defined a series of bake-off problems (BPs) [19]. The BPs centered on variations of solving the positive definite Helmholtz equation,

$$-\nabla \cdot \mu \nabla u + \beta u = f \quad \text{in}\, \Omega, \tag{7}$$

with μ and f nonnegative functions in the domain Ω. For BP1, μ is taken to be zero and β to be one, which results in solving an interpolation problem with a standard mass matrix. For BP3, μ is one and β is zero, creating a diffusion problem with the standard stiffness matrix. In terms of the basis actions, BP1 will test interp, while BP3 tests grad.

The results in Fischer et al. [19] focused on optimized implementations for the matrix-free calculations of each operator

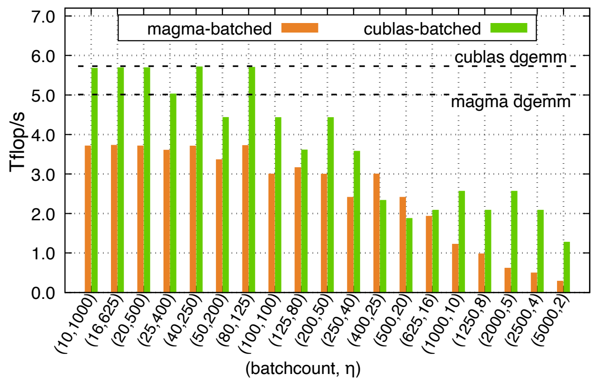

Fig. 9. Performance of different DGEMM configurations using cuBLAS and MAGMA. Results are shown for $(\bar{P}, \bar{Q}) = (729, 1000)$ on a Tesla V100 GPU using CUDA 10.1 Toolkit.

as implemented in Nek5000, MFEM, libParanumal, and deal.ii [20], without reliance on libCEED. Our results here compare the performance of standard "non-fused" libCEED backends as described in III-B on two problems modeled after BP1 and BP3. We use an implementation of the mass and diffusion problems provided through MFEM's integration with libCEED backends [21]. One slight variation is that the BPs in [19] used homogeneous Neumann conditions for BP1 and homogeneous Dirichlet conditions for BP3, while we use Dirichlet conditions for both. Furthermore, the implementation in [19] used diagonally preconditioned CG, while the implementation we used does not have any preconditioning.

VII. PERFORMANCE RESULTS

We now discuss the performance results of the MAGMA basis actions, comparing to the previously best-performing libCEED CUDA backends for each case, tensor and non-tensor. Because libCEED's computations are at the local level, we demonstrate backend performance improvements on a single GPU. The main implication of large-scale MPI parallelism for our work is a decrease in the local problem size per GPU (the x-axis on e.g. Fig. 10). For in-depth discussion of strong scaling for the CEED BPs, see [19].

A. *Tensor Results*

For the tensor benchmark tests, we consider a three-dimensional block mesh and standard Q-type hexahedral finite elements using the Gauss-Lobatto-Legendre (GLL) nodes. We use basis function orders of $\hat{p} \in [1, 8]$. The experiments were conducted with a Tesla V100 GPU with CUDA 10.2.89.

The results of the mass problem (BP1) and diffusion problem (BP3) for the cuda-shared and MAGMA backends are shown in Figures 10 and 11, respectively. A representative subset of tested basis function orders is shown for clarity. The y-axis, which shows the number of degrees of freedom (DOFs) in the mesh times the number of conjugate gradient (CG) iterations divided by time in the solver, represents the rate at which MFEM using the specified libCEED backend was able to process the necessary data to apply the matrix-free operator; alternatively, we can consider this the rate at which the implementation is able to do the necessary work for solving the problem. In Figures 12 and 13 we show the ratio of

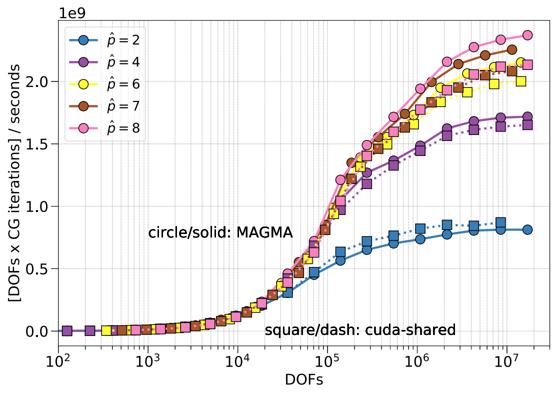

Fig. 10. MAGMA and cuda-shared backend performance for tensor-basis mass problem (BP1).

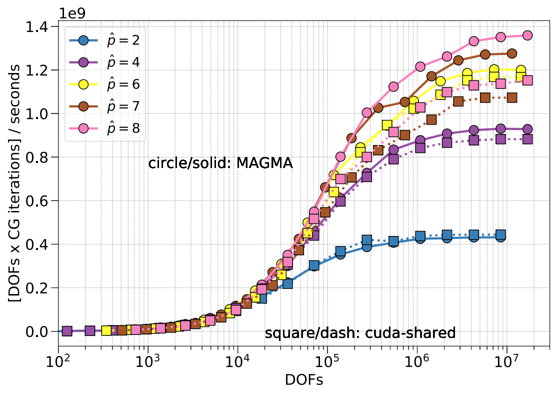

Fig. 11. MAGMA and cuda-shared backend performance for tensor-basis diffusion problem (BP3).

this "DOFs processing rate" metric for the MAGMA backend divided by that of cuda-shared. The benefit of MAGMA's fused batch-BLAS approach is greatest for higher orders of basis functions, particularly for the diffusion problem using the grad action, where several cases show approximately 1.2 times speedup; many more are within the range of 1.1 times. This is important because orders 7 and higher are routinely used, e.g., for incompressible flow simulations [3], [5].

B. Non-tensor Results

To compare the non-tensor performance, the meshes of the tensor benchmarks were modified to use P-type tetrahedron elements, with each element of the hexahedral mesh divided into six tetrahedrons. In Figures 14 and 15 we consider the performance of the backends in terms of the rate of DOFs processed in the CG solver. (Again, a representative subset of basis function orders was chosen to simplify the

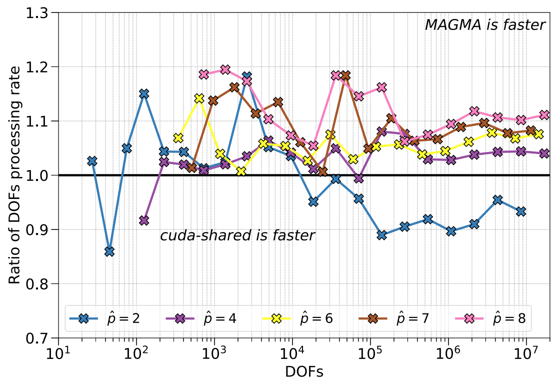

Fig. 12. Ratio of DOFs processed by MAGMA to the cuda-shared backend for the mass problem (BP1).

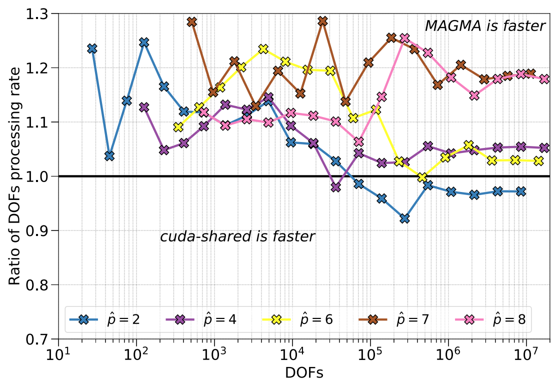

Fig. 13. Ratio of DOFs processed by MAGMA to the cuda-shared backend for the diffusion problem (BP3).

figures.) Now we are comparing the MAGMA backend's tuned batch GEMM approach to the cuda-ref backend, as cuda-shared does not implement non-tensor bases. In Figure 16, we show the ratio of the metric for MAGMA compared to cuda-ref, this time with BP1 and BP3 on the

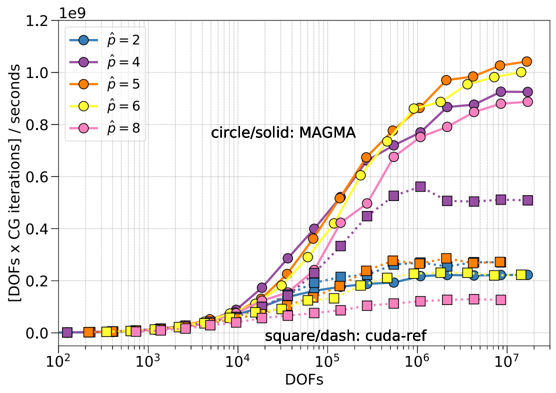

Fig. 14. MAGMA and cuda-ref backend performance for non-tensor mass problem (BP1).

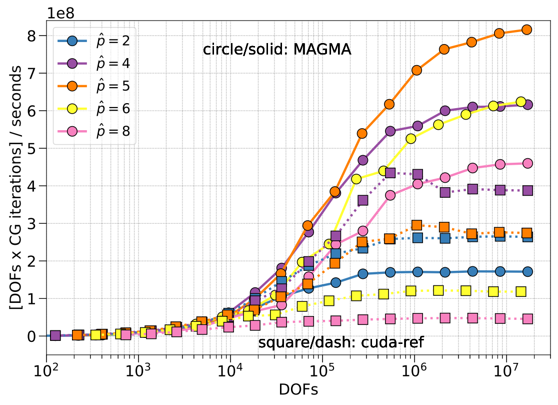

Fig. 15. MAGMA and `cuda-ref` backend performance for non-tensor diffusion problem (BP3).

same plot. We see similar trends as in the tensor case, in that there is a greater benefit of the MAGMA backend's approach for higher orders of basis functions, with a clear change in behavior for both BP1 and BP3 for $\hat{p} > 3$. A notable difference between the tensor case, however, is the continued increase of speedup for MAGMA as the number of elements in the mesh increases (larger number of DOFs), where MAGMA can be up to 10 times faster than `cuda-ref` for $\hat{p} = 8$.

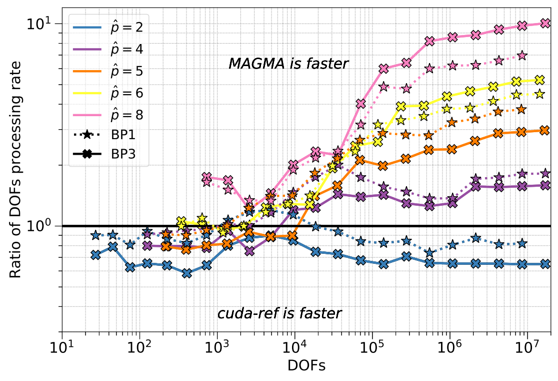

Fig. 16. Ratio of DOFs processed by MAGMA to the `cuda-ref` backend for the mass (dash/star) and diffusion (solid/×).

VIII. Conclusion and Future Work

We have presented improvements to a GPU backend for high-order matrix-free operator in libCEED. The backend is based on the MAGMA library. It uses both standard and customized batch matrix multiplication in order to perform different basis actions as defined in libCEED. The customized fused batch `GEMM` proves to outperform other GPU backends that provide a similar functionality for the tensor basis. Non-tensor basis actions are implemented using standard `GEMM` routines from both MAGMA and cuBLAS, which enable them outperform other backends as well. Future directions include adding support for AMD GPUs based on the HIP programming model, improving the GPU occupancy for relatively low-order problems, and designing a standard API for the device-level batch `GEMM`, which users can integrate into customized tensor contraction kernels.

Acknowledgments

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL-CONF-815596.

References

[1] "CEED," 2020. [Online]. Available: https://ceed.exascaleproject.org/
[2] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev *et al.*, "MFEM: a modular finite element methods library," *Computers & Mathematics with Applications*, 2020.
[3] M. O. Deville, P. F. Fischer, and E. H. Mund, *High-Order Methods for Incompressible Fluid Flow*, ser. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2002.
[4] "MFEM." [Online]. Available: https://github.com/mfem/mfem
[5] "Nek5000." [Online]. Available: https://github.com/Nek5000/Nek5000
[6] "libParanumal." [Online]. Available: https://github.com/paranumal/libparanumal
[7] "libCEED," 2020. [Online]. Available: https://github.com/ceed/libceed
[8] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *J. Phys.: Conf. Ser.*, vol. 180, no. 1, 2009.
[9] "MAGMA." [Online]. Available: http://icl.cs.utk.edu/magma/
[10] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.
[11] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, 2018.
[12] J. Huang, D. A. Matthews, and R. A. van de Geijn, "Strassen's algorithm for tensor contraction," *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. C305–C326, 2018.
[13] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on GPUs," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 85–95.
[14] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal, "Optimizing tensor contraction expressions for hybrid cpu-gpu execution," *Cluster computing*, vol. 16, no. 1, pp. 131–155, 2013.
[15] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, "Tensor contractions with extended BLAS kernels on CPU and GPU," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 193–202.
[16] A. Abdelfattah, M. Baboulin, V. Dobrev, J. J. Dongarra, C. W. Earl, J. Falcou, A. Haidar, I. Karlin, T. V. Kolev, I. Masliah, and S. Tomov, "High-Performance Tensor Contractions for GPUs," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, 2016, pp. 108–118. [Online]. Available: https://doi.org/10.1016/j.procs.2016.05.302
[17] K. Świrydowicz, N. Chalmers, A. Karakus, and T. Warburton, "Acceleration of tensor-product operations for high-order finite element methods," *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 735–757, 2019.
[18] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 3, pp. 1–29, 2018.
[19] P. Fischer, M. Min, T. Rathnayake, S. Dutta, T. Kolev, V. Dobrev, J.-S. Camier, M. Kronbichler, T. Warburton, K. Świrydowicz, and J. Brown, "Scalability of high-performance PDE solvers," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 562–586, 2020. [Online]. Available: https://doi.org/10.1177/1094342020915762
[20] "deal.ii." [Online]. Available: https://github.com/dealii/dealii
[21] "CEED benchmarks," 2020. [Online]. Available: https://github.com/CEED/benchmarks