libCEED GPU Strategy

Jeremy L Thompson

University of Colorado Boulder

jeremy@jeremylt.org

11 Feb 2025

libCEED, Ratel, and HONEE Team

libCEED Repo: https://github.com/CEED/libCEED
HONEE Repo: https://gitlab.com/phypid/HONEE
Ratel Repo: https://gitlab.com/micromorph/ratel

Developers: **Zach Atkins**, Jed Brown, Fabio Di Gioacchino, Leila Ghaffari, Kenneth Jansen, **Rezgar Shakeri**, James Wright, **Jeremy L Thompson**

The authors acknowledge support by the Department of Energy, National Nuclear Security Administration, Predictive Science Academic Alliance Program (PSAAP) under Award Number DE-NA0003962.

Overview

1. Background
2. General GPU Strategy
   - Ref Operators
   - Shared Memory Bases
   - Gen Operators
3. MPM Support
   - Shared Memory Bases
4. Operator Assembly
   - Diagonal Assembly
   - Full Assembly
5. Questions

ECP Roots

- libCEED + PETSc projects follow from ECP CEED work
- libCEED provides high-performance operator evaluation
- libCEED provides CUDA, ROCm, and SYCL support
- PETSc provides linear/non-linear solvers and time steppers

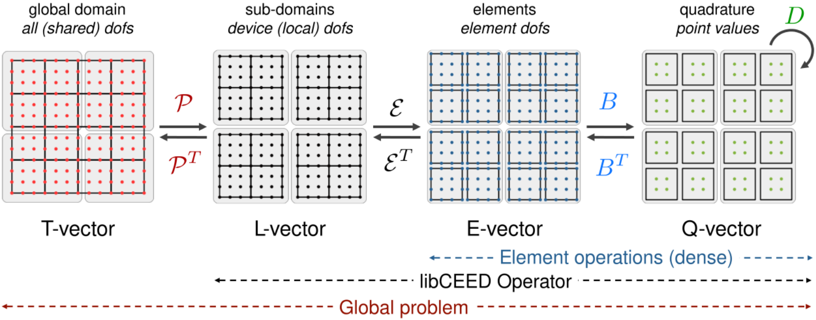
libCEED Projects

Several projects built using libCEED

- Ratel - solid mechanics FEM and MPM (PSAAP)
- HONEE - fluid dynamics FEM & differential filtering (PHASTA)
- MFEM - various applications, libCEED integrators (LLNL)
- Palace - Electromagnetics FEM with MFEM + libCEED (Amazon)
- RDycore - River dynamical core with PETSc + libCEED (SciDAC)

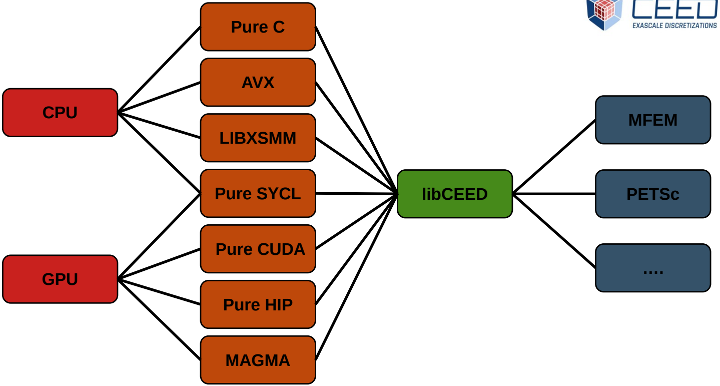
Matrix-Free Operators from libCEED

$$A = \mathcal{P}^T \mathcal{E}^T B^T D B \mathcal{E} \mathcal{P}$$



libCEED provides arbitrary order matrix-free operator evaluation

Performance Portability from libCEED



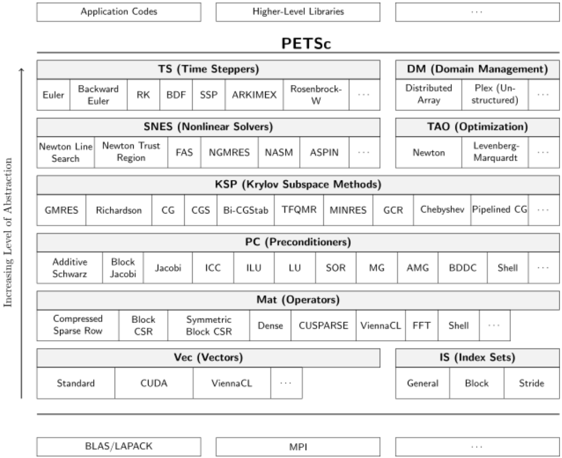
libCEED backends target different hardware at runtime

Extensible Solvers from PETSc

CeedEvaluator

MatCeed

CeedVector



libCEED provides the local operator action for PETSc objects

Two Families of Approaches

Three libCEED backends with two approaches to operator application

- Separate kernels
  - `/gpu/*/ref` and `/gpu/*/shared`
  - $\mathcal{E}$, B, and D all separate kernels
  - Higher overall memory usage, multiple kernel launches
- Fused kernel
  - `/gpu/*/gen`
  - Single kernel JiTed with data from $\mathcal{E}$, B, and D
  - Lower overall memory usage, single kernel launch

Ref Operator Application

`/gpu/*/ref` and `/gpu/*/shared` use largely the same code

$A_L = \mathcal{E}^T B^T D B \mathcal{E}$ use separate kernels

$\mathcal{E}$ source comes from the `/gpu/*/ref`

`/gpu/*/ref` uses basic kernels for B

`/gpu/*/shared` uses shared memory for B

D source is given by the user

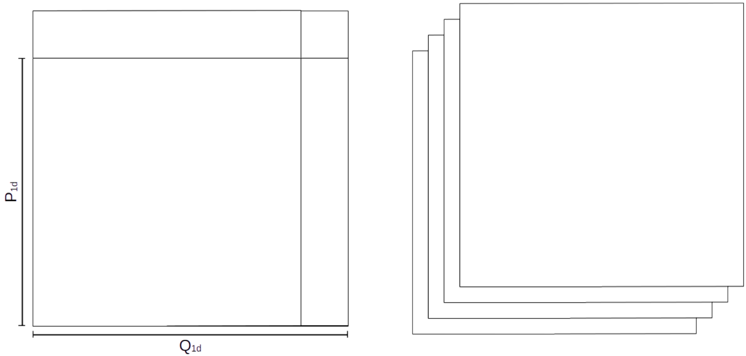
Shared Basis Code

```
extern "C" __launch_bounds__(BASIS_GRAD_BLOCK_SIZE) __global__
    void Grad(const CeedInt num_elem, const CeedScalar *c_B,
    const CeedScalar *c_G, const CeedScalar *__restrict__ d_U,
    CeedScalar *__restrict__ d_V) {
  // Setup (omitted)
  // Apply basis element by element
  for (CeedInt elem=blockIdx.x*blockDim.z+threadIdx.z; elem<num_elem; elem+=gridDim.x*blockDim.z) {
    ReadElementStrided2d<NUM_COMP,P_1D>(data,elem,1,P_1D*P_1D*num_elem,P_1D*P_1D,d_U,r_U);
    GradTensor2d<NUM_COMP,P_1D,Q_1D>(data,r_U,s_B,s_G,r_V);
    WriteElementStrided2d<NUM_COMP*DIM,Q_1D>(data,elem,1,Q_1D*Q_1D*num_elem,Q_1D*Q_1D,r_V,d_V);
  }
}
```

x and y thread index gives point (2D) or column of points (3D)

z thread index gives the element

Thread Usage



x and y thread index gives point (2D) or column of points (3D)

3D strategy works on 2D slabs of points

Shared Basis Code

```
template <int NUM_COMP, int P_1D, int Q_1D>
inline __device__ void GradTensor2d(SharedData_Hip &data,
    const CeedScalar *__restrict__ r_U, const CeedScalar *c_B,
    const CeedScalar *c_G, CeedScalar *__restrict__ r_V) {
  CeedScalar r_t[1];
  for (CeedInt comp = 0; comp < NUM_COMP; comp++) {
    ContractX2d<NUM_COMP,P_1D,Q_1D>(data, &r_U[comp], c_G, r_t);
    ContractY2d<NUM_COMP,P_1D,Q_1D>(data, r_t, c_B, &r_V[comp+0*NUM_COMP]);
    ContractX2d<NUM_COMP,P_1D,Q_1D>(data, &r_U[comp], c_B, r_t);
    ContractY2d<NUM_COMP,P_1D,Q_1D>(data, r_t, c_G, &r_V[comp+1*NUM_COMP]);
  }
}

template <int NUM_COMP, int P_1D, int Q_1D>
inline __device__ void GradTransposeTensor2d(SharedData_Hip &data,
    const CeedScalar *__restrict__ r_U, const CeedScalar *c_B,
    const CeedScalar *c_G, CeedScalar *__restrict__ r_V) {
  CeedScalar r_t[1];
  for (CeedInt comp = 0; comp < NUM_COMP; comp++) {
    ContractTransposeY2d<NUM_COMP,P_1D,Q_1D>(data, &r_U[comp+0*NUM_COMP], c_B, r_t);
    ContractTransposeX2d<NUM_COMP,P_1D,Q_1D>(data, r_t, c_G, &r_V[comp]);
    ContractTransposeY2d<NUM_COMP,P_1D,Q_1D>(data, &r_U[comp+1*NUM_COMP], c_G, r_t);
    ContractTransposeAddX2d<NUM_COMP,P_1D,Q_1D>(data, r_t, c_B, &r_V[comp]);
  }
}
```

Loop over components to reduce total shared memory needed

Shared Basis Code

```
template <int NUM_COMP, int P_1D, int Q_1D>
inline __device__ void ContractX3d(SharedData_Hip &data, const CeedScalar *U,
    const CeedScalar *B, CeedScalar *V) {
  CeedScalar r_B[P_1D];
  for (CeedInt i = 0; i < P_1D; i++) r_B[i] = B[i + data.t_id_x * P_1D];

  for (CeedInt k = 0; k < P_1D; k++) {
    data.slice[data.t_id_x + data.t_id_y * T_1D] = U[k];
    __syncthreads();
    V[k] = 0.0;
    if (data.t_id_x < Q_1D && data.t_id_y < P_1D) {
      for (CeedInt i = 0; i < P_1D; i++) {
        V[k] += r_B[i] * data.slice[i + data.t_id_y * T_1D];
      }
    }
    __syncthreads();
  }
}
```

Each thread computes all node's contributions to one quadrature point

3D loops over 2D slabs for tensor contraction

Gen Operator Application

`/gpu/*/gen` generates a single kernel for the operator

$A_L = \mathcal{E}^T B^T D B \mathcal{E}$ uses a single kernel

$\mathcal{E}$ source comes from the `/gpu/*/ref`

B source comes from `/gpu/*/shared`

D source is given by the user

Generated Operator Kernel

```
extern "C" __global__ void CeedKernelCudaGenOperator_mass(CeedInt num_elem,
    void* ctx, FieldsInt_Cuda indices, Fields_Cuda fields, Fields_Cuda B,
    Fields_Cuda G, CeedScalar *W, Points_Cuda points) {
  // Setup kernel data

  // Input and Output field constants and basis data

  // Element loop
  __syncthreads();
  for (CeedInt elem = blockIdx.x*blockDim.z + threadIdx.z; elem < num_elem; elem +=
       gridDim.x*blockDim.z) {
    // -- Input field restrictions (E) and basis actions (B)

    // -- Output field setup
    {
      // -- Apply QFunction (D)
      mass(ctx, 1, inputs, outputs);
    }

    // -- Output field basis actions (B^T) and restrictions (E^T)
  }
}
// -----------------------------------------------------------------------------
```

$A_L = \mathcal{E}^T B^T D B \mathcal{E}$ in a single kernel

Generated Operator Kernel

```
// Setup kernel data
const CeedScalar *d_in_0 = fields.inputs[0];
const CeedScalar *d_in_1 = fields.inputs[1];
CeedScalar *d_out_0 = fields.outputs[0];
const CeedInt dim = 1;
const CeedInt Q_1d = 8;
extern __shared__ CeedScalar slice[];
SharedData_Cuda data;
data.t_id_x = threadIdx.x;
data.t_id_y = threadIdx.y;
data.t_id_z = threadIdx.z;
data.t_id  = threadIdx.x+threadIdx.y*blockDim.x+threadIdx.z*blockDim.y*blockDim.x;
data.slice = slice + data.t_id_z*T_1D;
```

Set up pointers to basis data and shared memory

Generated Operator Kernel

```
// Input field constants and basis data
// -- Input field 0
const CeedInt P_1d_in_0 = 8;
const CeedInt num_comp_in_0 = 1;
// EvalMode: none
// -- Input field 1
const CeedInt P_1d_in_1 = 5;
const CeedInt num_comp_in_1 = 1;
// EvalMode: interpolation
__shared__ CeedScalar s_B_in_1[40];
LoadMatrix<P_1d_in_1, Q_1d>(data, B.inputs[1], s_B_in_1);

// Output field constants and basis data
// -- Output field 0
const CeedInt P_1d_out_0 = 5;
const CeedInt num_comp_out_0 = 1;
// EvalMode: interpolation
__shared__ CeedScalar s_B_out_0[40];
LoadMatrix<P_1d_out_0, Q_1d>(data, B.outputs[0], s_B_out_0);
```

Basis data and constants loaded

Generated Operator Kernel

```
      // Scratch restriction buffer space
      CeedScalar r_e_scratch[8];

      // -- Input field restrictions and basis actions
      // ---- Input field 0
      CeedScalar r_e_in_0[num_comp_in_0*P_1d_in_0];
      // Strides: {1, 8, 8}
      ReadLVecStrided1d<num_comp_in_0, P_1d_in_0,1,8,8>(data, elem, d_in_0, r_e_in_0);
      // EvalMode: none
      CeedScalar *r_q_in_0 = r_e_in_0;
      // ---- Input field 1
      CeedScalar *r_e_in_1 = r_e_scratch;
      const CeedInt l_size_in_1 = 61;
      // CompStride: 1
      ReadLVecStandard1d<num_comp_in_1, 1, P_1d_in_1>(data, l_size_in_1, elem, indices.
           inputs[1], d_in_1, r_e_in_1);
      // EvalMode: interpolation
      CeedScalar r_q_in_1[num_comp_in_1*Q_1d];
      Interp1d<num_comp_in_1, P_1d_in_1, Q_1d>(data, r_e_in_1, s_B_in_1, r_q_in_1);

      // -- Output field setup
      // ---- Output field 0
      CeedScalar r_q_out_0[num_comp_out_0*Q_1d];
```

Restrict and apply basis for each input

Setup output data buffers

Generated Operator Kernel

```
      // Note: Using full elements
      {
        // -- Input fields
        // ---- Input field 0
        CeedScalar *r_s_in_0 = r_q_in_0;
        // ---- Input field 1
        CeedScalar *r_s_in_1 = r_q_in_1;
        // -- Output fields
        // ---- Output field 0
        CeedScalar *r_s_out_0 = r_q_out_0;

        // -- QFunction inputs and outputs
        // ---- Inputs
        CeedScalar *inputs[2];
        // ------ Input field 0
        inputs[0] = r_s_in_0;
        // ------ Input field 1
        inputs[1] = r_s_in_1;
        // ---- Outputs
        CeedScalar *outputs[1];
        // ------ Output field 0
        outputs[0] = r_s_out_0;

        // -- Apply QFunction
        mass(ctx, 1, inputs, outputs);
      }
```

Apply QFunction at each quadrature point

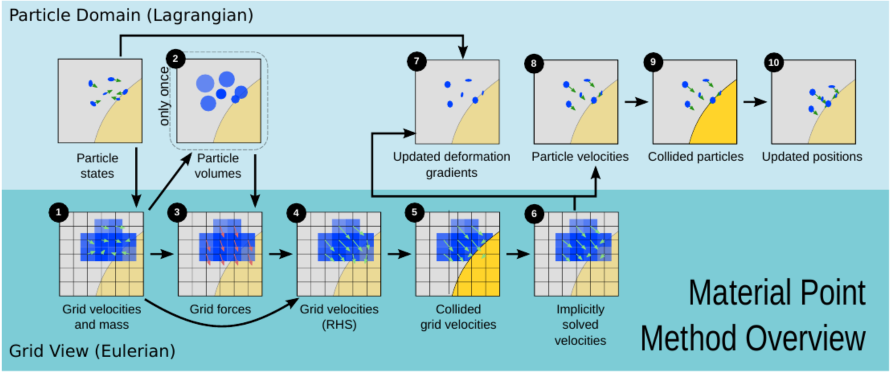
May apply to 2D slabs or full elements in 3D

Generated Operator Kernel

```
      // -- Output field basis action and restrictions
      // ---- Output field 0
      // EvalMode: interpolation
      CeedScalar *r_e_out_0 = r_e_scratch;
      InterpTranspose1d<num_comp_out_0, P_1d_out_0, Q_1d>(data, r_q_out_0, s_B_out_0,
           r_e_out_0);
      const CeedInt l_size_out_0 = 61;
      // CompStride: 1
      WriteLVecStandard1d<num_comp_out_0, 1, P_1d_out_0>(data, l_size_out_0, elem,
           indices.outputs[0], r_e_out_0, d_out_0);
```

Output basis action and restriction to assemble result

What is MPM?

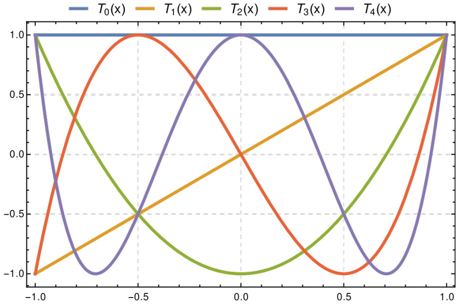


- Continuum based particle method with background mesh for gradients
- Extension of FLIP (which is an extension of PIC)
- Used in rendering for the movie *Frozen*

What does MPM have to do with FEM?

- Problem on background mesh changes when material points move
- Natural fit for matrix-free representation
- Similar reasoning to use matrix-free for adaptive methods
- Ratel/libCEED FEM infrastructure provides fast background mesh solves

libCEED Basis Evaluation AtPoints



- Interpolate from primal to dual (quadrature) space
- Fit Chebyshev polynomials to values at quadrature points
- Evaluate Chebyshev polynomials at reference coords of material points
- Transpose the order for projection to mesh from material points

Shared Basis Code

```
extern "C" __launch_bounds__(BASIS_INTERP_BLOCK_SIZE) __global__
    void InterpAtPoints(const CeedInt num_elem, const CeedScalar *c_B,
    const CeedInt *points_per_elem, const CeedScalar *__restrict__ d_X,
    const CeedScalar *__restrict__ d_U, CeedScalar *__restrict__ d_V) {
  // Setup (omitted)
  // Apply basis element by element
  for (CeedInt elem=blockIdx.x*blockDim.z+threadIdx.z; elem<num_elem; elem+=gridDim.x
       *blockDim.z) {
    // Map from nodes to Chebyshev coefficients
    ReadElementStrided2d<NUM_COMP,P_1D>(data,elem,1,P_1D*P_1D*num_elem,P_1D*P_1D,d_U,
         r_U);
    InterpTensor2d<NUM_COMP,P_1D,Q_1D>(data,r_U,s_B,r_C);
    // Map from Chebyshev coefficients to points
    for (CeedInt i=threadIdx.x+threadIdx.y*blockDim.x; i<point_loop_bound; i+=
         blockDim.x*blockDim.y) {
      const CeedInt p = i % NUM_PTS;
      ReadPoint<DIM,NUM_PTS>(data,elem,p,NUM_PTS,1,num_elem*NUM_PTS,NUM_PTS,d_X,r_X);
      InterpAtPoints2d<NUM_COMP,NUM_PTS,Q_1D>(data,i,r_C,r_X,r_V);
      WritePoint<NUM_COMP,NUM_PTS>(data,elem,p,NUM_PTS,1,num_elem*NUM_PTS,NUM_PTS,r_V
           ,d_V);
    }
  }
}
```

Threadblock maps to Chebeshev coeffs on element (standard interpolation)

Each thread maps from Chebyshev coeffs to single point

Shared Basis Code

```
template <int NUM_COMP, int NUM_POINTS, int Q_1D>
inline __device__ void InterpAtPoints2d(SharedData_Hip &data, const CeedInt p, const
    CeedScalar *__restrict__ r_C, const CeedScalar *__restrict__ r_X,
                                        CeedScalar *__restrict__ r_V) {
  for (CeedInt i = 0; i < NUM_COMP; i++) r_V[i] = 0.0;
  for (CeedInt comp = 0; comp < NUM_COMP; comp++) {
    CeedScalar buffer[Q_1D];
    CeedScalar chebyshev_x[Q_1D];
    // Load coefficients
    if (data.t_id_x<Q_1D && data.t_id_y<Q_1D) {
      data.slice[data.t_id_x+data.t_id_y*Q_1D] = r_C[comp];
    }
    __syncthreads();
    // Contract x direction
    ChebyshevPolynomialsAtPoint<Q_1D>(r_X[0], chebyshev_x);
    for (CeedInt i = 0; i < Q_1D; i++) {
      buffer[i] = 0.0;
      for (CeedInt j = 0; j < Q_1D; j++) {
        buffer[i] += chebyshev_x[j] * data.slice[j + i*Q_1D];
      }
    }
    // Contract y direction
    ChebyshevPolynomialsAtPoint<Q_1D>(r_X[1], chebyshev_x);
    for (CeedInt i = 0; i < Q_1D; i++) {
      r_V[comp] += chebyshev_x[i] * buffer[i];
    }
  }
}
```

Each thread (point) needs separate contraction buffers

Preconditioning Support

Some operator assembly needed for preconditioning

Diagonal assembly for Jacobi, Chebyshev

Full assembly for AMG or LU

FEM Diagonal Assembly

FEM diagonal assembly consists of two phases

- QFunction assembly
  - Assemble small matrix at each quadrature point
  - Each active input individually set to 1
  - QFunction kernel D called to populate assembled row
- Diagonal assembly
  - Compute diagonal entries of $B^T D B$ on element
  - Element diagonals assembled via $\mathcal{E}^T$ for local diagonal

MPM Diagonal Assembly

MPM diagonal assembly consists of a single phase

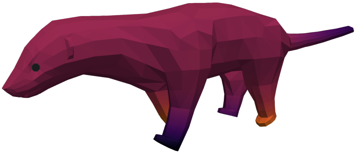
- Each input node on elements individually set to 1
- Basis B applied to get values at material points
- QFunction kernel D called to populate result
- Basis B^T applied to get values at nodes
- Corresponding diagonal entry copied into element diagonal
- Element diagonals assembled via $\mathcal{E}^T$ for local diagonal

FEM Full Assembly

FEM full assembly consists of three phases

- Sparsity pattern
  - Compute local (on process) matrix COO indices for entries
- QFunction assembly
  - Same as for diagonal assembly
- Full assembly
  - Compute entries of $B^T D B$ for each element
  - Populate assembled array per the sparsity pattern

Questions?



libCEED Repo: https://github.com/CEED/libCEED
HONEE Repr: https://gitlab.com/phypid/HONEE
Ratel Repo: https://gitlab.com/micromorph/ratel

Grant: Predictive Science Academic Alliance Program (DE-NA0003962)