The University of Texas at Austin
Oden Institute for Computational Engineering and Sciences

Development of a DG compressible Navier-Stokes solver with MFEM

M. Bolinches - PECOS development team

10/20/2021 · MFEM Workshop

ODEN INSTITUTE

PECOS Predictive Engineering & Computational Science

Index

Index

Introduction

- Oden Institute (UT Austin) interested in **high-fidelity simulations** of Inductively Coupled Plasma (ICP) Torch
  - as part of PSAAP3 project
  - initially different physics simulated independently (here flow only)
  - fully coupled simulations to come
- **MFEM library** chosen as framework for development of simulation infrastructure
- **High-order (HO) compact schemes** particularly efficient for **GPU** architectures
  - Large number of operations per DOF and independent from neighbors
- **Discontinuous Galerkin** (DG) scheme initially chosen
  - no GPU supported by MFEMv4.2

Introduction

CPU based code

- Baseline CPU code implemented
  - Based on MFEM example 18
  - Verified using MASA library (MMS)
- Characteristics provided by MFEM
  - Discontinuous Galerkin (DG) method, i.e. FE method
  - arbitrary order of accuracy
  - MPI parallel
  - unstructured
- Main implemented features
  - compressible
  - upwind flux (Roe/LF) at interfaces, i.e. dissipative
  - HDF5 output and restart
  - adiabatic & isothermal wall BCs
  - reflecting & non-reflecting in/out BCs
  - communication/computation overlap
  - restart with arbitrary #MPI tasks

Introduction

GPU code

- GPU code based on CPU version
- Some functions **duplicated for GPU** support
  - Makes use of MFEM functions where possible
  - Takes over some loops for higher degree of parallelism
  - **Uses MFEM GPU directives** for kernel coding
- GPU implementation efforts in two areas
  - increased level of parallelism
  - kernel optimization
- Source code `https://github.com/pecos/tps`
- Documentation `https://pecos.github.io/tps-docs/`

Index

Index

DG discretization

- Weak DG formulation of Navier-Stokes (NS) equations

$$\int_{\Omega_e} \frac{\partial U^h}{\partial t} \phi_j \mathrm{d}\Omega = \int_{\Omega_e} \mathbf{F}^h \cdot \nabla \phi_j \mathrm{d}\Omega - \int_{\partial\Omega_e} \mathbf{F}^* \cdot \mathbf{n} \phi_j \mathrm{d}\left(\partial\Omega\right)$$

- Superscript h denotes numerical solution; $\mathbf{F}^*$ numerical flux at interface
- Volume integrals result in **element-wise** matrix-vector multiplication
- Last term involves **data from neighboring elements**

Implementation approach

- MFEM "for-loops" executing kernels **substituted by single kernel**
  - **Increases the level of parallelism** of computation
  - more complex kernels

	MFEM	**Implemented**
Element-wise functions	for each element execute element GPU kernel	single kernel where each thread group computes contribution to one element
Face integrals	for each face execute face GPU kernel	single kernel where each thread group computes all face contributions for one element

- **Example 18** has been implemented using these two approaches
  - Single kernel performed better
  - mfem::NonLinearForm kept transferring data GPU-CPU for both v4.2 and v4.3

MFEM GPU macros

- MFEM GPU macros allow for **hardware independent** coding
- GPU code generated at compile time
  - CUDA macros

```
#define MFEM_SHARED __shared__
#define MFEM_SYNC_THREAD __syncthreads()
#define MFEM_THREAD_ID(k) threadIdx.k
#define MFEM_THREAD_SIZE(k) blockDim.k
#define MFEM_FOREACH_THREAD(i,k,N) for(int i=threadIdx.k; i<N; i+=blockDim.k)
#define MFEM_FORALL_2D(i,N,X,Y,BZ,...)   ForallWrap<2>(true,N,...
```

  - HIP macros

```
#define MFEM_SHARED __shared__
#define MFEM_SYNC_THREAD __syncthreads()
#define MFEM_THREAD_ID(k) hipThreadIdx_ ##k
#define MFEM_THREAD_SIZE(k) hipBlockDim_ ##k
#define MFEM_FOREACH_THREAD(i,k,N)
#define MFEM_FORALL_2D(i,N,X,Y,BZ,...)   ForallWrap<2>(true,N,...
```

Example element-wise function

Inverse mass matrix multiplication

- For-loop controlling kernel execution

```
for(int el=0; el<NumElems;el++){
  // Get data
  // Get element inverse mass matrix
  // GPU matrix-vector multiplication kernel
  // Add to global array
}
```

- Single kernel implementation

```
MFEM_FORALL_2D(el,NumElems,dof,1,1,{
  MFEM_FOREACH_THREAD(i,x,dof){
     // Load data
     // Matrix-vector multiplication
     // Save to global vector
  }
});
```

Face integration

- Loop over element faces

```
for(int i=0; i<mesh->GetNumFaces(); i++){
  // Get data elems 1 & 2
  // Perform GPU face integration
  // Add face contribution to element
}
```

- Single kernel by faces not possible
  - faces belonging to same element override each other
  - face contributions implemented **by element**

```
MFEM_FORALL_2D(el,NumElemType,elDof,1,1,{
    MFEM_FOREACH_THREAD(i,x,elDof){
      // loop through faces
      // add total contribution to elemenent
    }
}
```

Index

Computations on GPU

- Most (simple) functions are **memory bound**
  - Accessing data more expensive than operations
- Different memory types have very different **access rates**

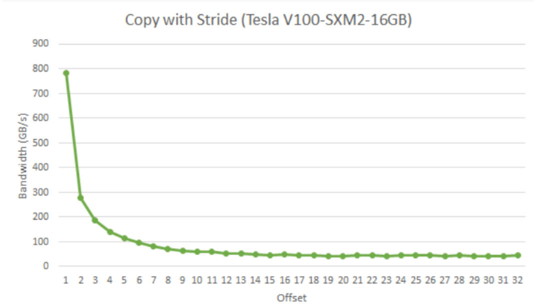
Access type	CPU$\rightleftharpoons$GPU	Global GPU	Shared
Bandwidth (peak)	$\sim$32GB/s	900GB/s	"Much faster"

[Shared data access rate for the particular GPU not found but reported as "much faster" in the NVIDIA developer guide]

- Low GPU$\rightleftharpoons$CPU rates imply all operations must happen on GPU
- **Memory management is critical** in GPU computation

Memory Access Bandwidth

- Global memory accesses rates can vary dramatically with access patterns
  - strided accesses to be avoided
- Shared memory used throughout
  1. can reduce memory accesses
  2. can improve memory access patterns a.k.a. **coalesced memory accesses**



- Efficient kernels can be achieved by
  - minimizing global memory access
  - maximizing operations for loaded data (great for compact HO FE)

[In line with MFEM webside https://mfem.org/gpu-support/]

Shared memory optimizations

- **Coalesced accesses** can be achieved by loading data in the array order
  - data ordering $[\ \rho_1 \cdots \rho_N \quad u_1 \cdots u_N \quad v_1 \cdots v_N \quad w_1 \cdots w_N \quad p_1 \cdots p_N\ ]$
  - e.g. fluxes computation kernel will load first density for each node, then velocities etc.
- **Reducing global memory accesses**
  - can be done by storing data in shared arrays
- Shared memory is **scarce** (needs to be used wisely)
  - 64KB including read register memory for a NVIDIA V100

Example

Multiplication by inverse of mass matrix

- If shared memory not used
  - data in array *d_z* is accessed multiple times
  - kernel looks simpler

```
MFEM_FORALL_2D(el,NE,dof,1,1,{
  int eli = el + elemOffset;
  int offsetInv = d_posDofInvM[2*eli];
  int offsetIds = d_posDofIds[2*eli];
  MFEM_FOREACH_THREAD(eq,y,num_equation){
    MFEM_FOREACH_THREAD(i,x,dof){
      int index = d_nodesIDs[offsetIds+i];
      double temp = 0;
      for(int k=0;k<dof;k++){
        int indexk = d_nodesIDs[offsetIds +k];
        temp += d_invM[offsetInv +i*dof +k]*d_z[indexk + eq*totNumDof];
      }
      d_y[index+eq*totNumDof] = temp;
    }
  }
});
```

Example using shared memory

Multiplication by inverse of mass matrix

- Using shared data avoids accessing data in d_z repeatedly
  - this kernel takes 55% of the time needed to compute the previous

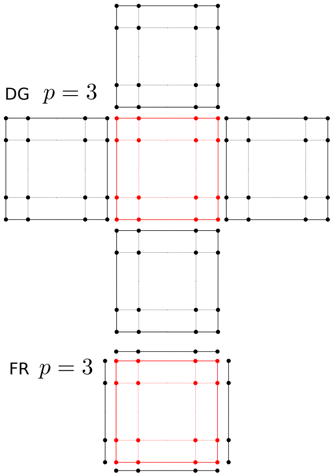
```
MFEM_FORALL_2D(el,NE,dof,1,1,{
  MFEM_FOREACH_THREAD(i,x,dof){
    MFEM_SHARED double data[216*5];
    int eli = el + elemOffset;
    int offsetInv = d_posDofInvM[2*eli];
    int offsetIds = d_posDofIds[2*eli];
    int index = d_nodesIDs[offsetIds+i];

    for(int eq=0;eq<num_equation;eq++)
        data[i+eq*dof] = d_z[index + eq*totNumDof];
    MFEM_SYNC_THREAD;
    for(int eq=0;eq<num_equation;eq++){
      double tmp = 0.;
      for(int k=0;k<dof;k++) tmp += d_invM[offsetInv +i*dof +k]*data[k+eq*dof];
      d_y[index+eq*totNumDof] = tmp;
    }
  }
});
```

Index

Drawback of DG face integration

- Most complex and expensive kernel
  - Contains lots of non-consecutive global memory accesses
  - 47% of total execution time
- Face contribution kernels always more expensive than volume contributions
  - involves loading data from neighboring elements
  - memory accesses **always non-ordered**
- Particularly damaging in DG
  - interpolation to integration points requires **loading all element solution points**
- In contrast, other methods use only nodes at common faces, e.g. FR



Index

Final comments

- DG code for the solution of the NS equations has been developed
  - CPU version coded following example 18
- GPU code approach
  - **increased level of parallelism**
  - optimized/minimized global memory accesses via **shared memory**
- Face integration most expensive kernel
  - large number of data accessed
  - data access cannot be coalesced
  - it is the drawback of DG
  - **improvement is underway**

Code and acknowledgment

- Source code `https://github.com/pecos/tps`
- Documentation `https://pecos.github.io/tps-docs/`
- This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.