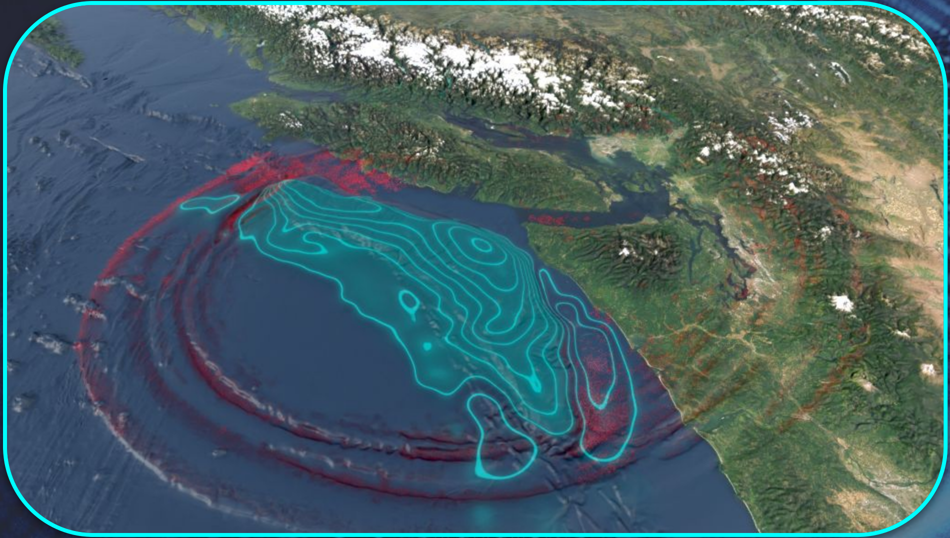
MFEM Community Workshop

A Guided Tour of MFEM GPU Kernel Optimization Techniques

John Camier - LLNL Collaborator,
Veselin Dobrev, Tzanio Kolev - LLNL
Stefan Henneking - Oden Institute, The University of Texas at Austin
Jiqun Tu - NVIDIA



MFEM Workshop - GPU Kernel Optimizations Guided Tour

- **Welcome students, new users & developers**
- **Exploring GPU kernel optimization strategies in MFEM**

- Arbitrary order curvilinear **mesh** elements
- Arbitrary order **H1**, H(curl), H(div) and **L2** elements
- **Bilinear**/linear forms for: Galerkin, DG, etc.
- MPI-scalable assembly and linear solvers
- **GPU** acceleration on **AMD**, **NVIDIA** hardware
- Non-linear operators and non-linear solvers
- Explicit and implicit high-order time integration
- Integration with: hypre, SUNDIALS, SuperLU, PETSc, etc.

⇒ **Real-time Bayesian inference at extreme scale:**
A digital twin for tsunami early warning applied to the Cascadia subduction zone [1]

mfem.org v4.8 - Apr 2025

ECP

CEED
EXASCALE DISCRETIZATIONS

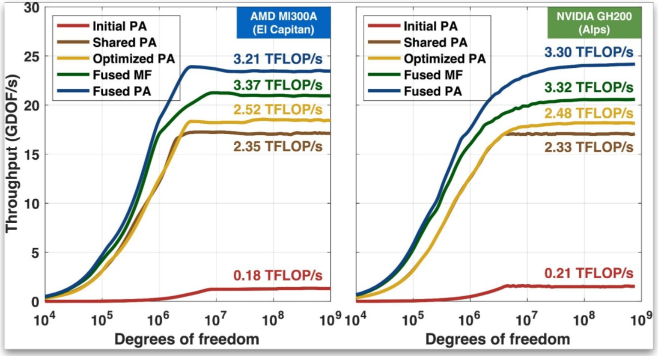
2020 R&D 100 FINALIST

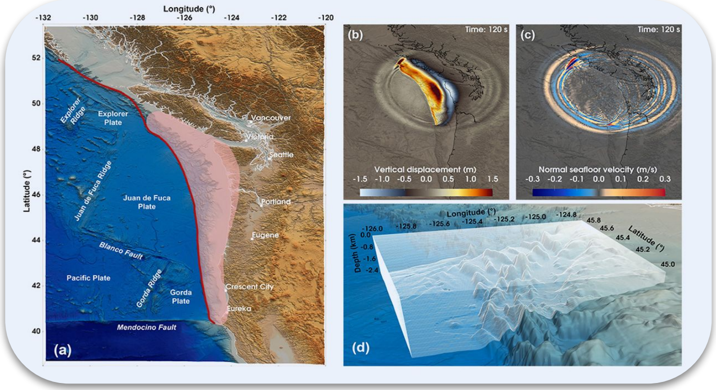
Application - A Digital Twin For Tsunami Early Warning

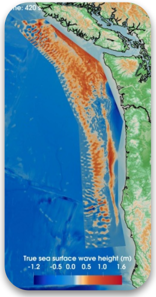
- Important and challenging problem
- Forecast wave heights or onshore inundation
- Produce better early warning systems for tsunamis

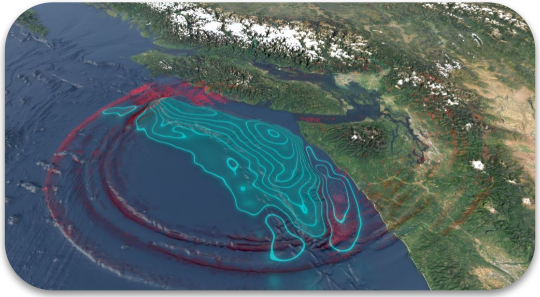
$$\left(A\begin{bmatrix}\vec{u}\\p\end{bmatrix},\begin{bmatrix}\vec{\tau}\\v\end{bmatrix}\right) := \begin{bmatrix}0 & (\nabla p,\vec{\tau})\\ -(\vec{u},\nabla v) & \langle Z^{-1}p, v\rangle_{\partial\Omega_a}\end{bmatrix}$$

- Problem size ⇒ memory optimizations
- Key kernels ⇒ performance optimizations



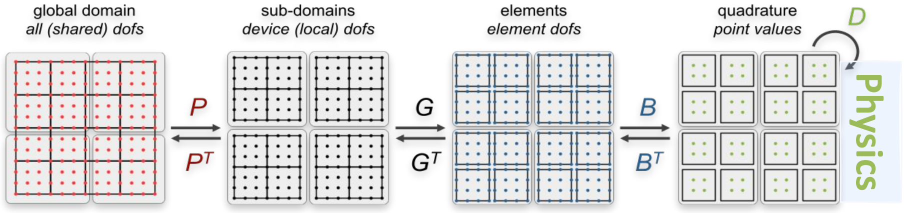






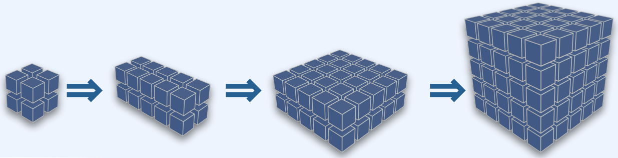
MFEM Operator Decomposition for GPU Kernels

- **Partial Assembled HO Finite Element Operators**
  - **$A = P^T G^T B^T D B G P$**
- **Optimal memory, near-optimal FLOPs**
- **Matrix free: no assembly of the full matrix A**

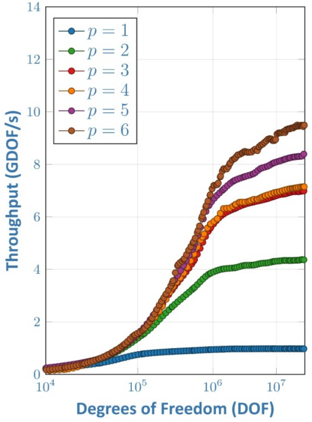


Parallel **Geometry** **Basis**

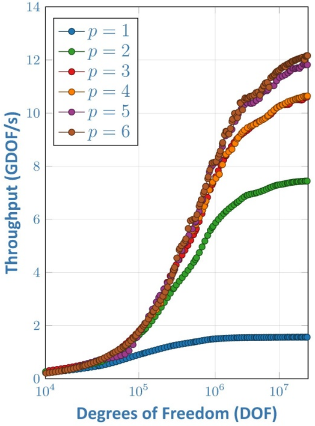
- **GPU Kernel optimization: focus on $G^T B^T D B G$**
- **B:**



MFEM BP1 (standard) @ GH200



MFEM BP1 (atomics) @ GH200



MFEM GPU Kernel Overview

MFEM Features Examples ▾ Documentation ▾ Community ▾ Gallery Download

Integration

MFEM's spatial integrations are performed in the usual finite element manner by first splitting the spatial domain into a collection of non-overlapping "elements" which cover the domain. This is usually referred to as the "mesh". An integral can then be computed separately in each element and the results added together:

$$\int_{\Omega} f(x)\, d\Omega = \sum_i \int_{\Omega_i} f(x)\, d\Omega \quad (1)$$

Where Ω is the full domain and Ω_i is the domain of the i-th element. In MFEM this sum over elements is performed in classes such as the `BilinearForm` or `LinearForm` and their parallel counterparts.

MFEM Features Examples ▾ Documentation ▾ Community ▾ Gallery Download

Bilinear Form Integrators

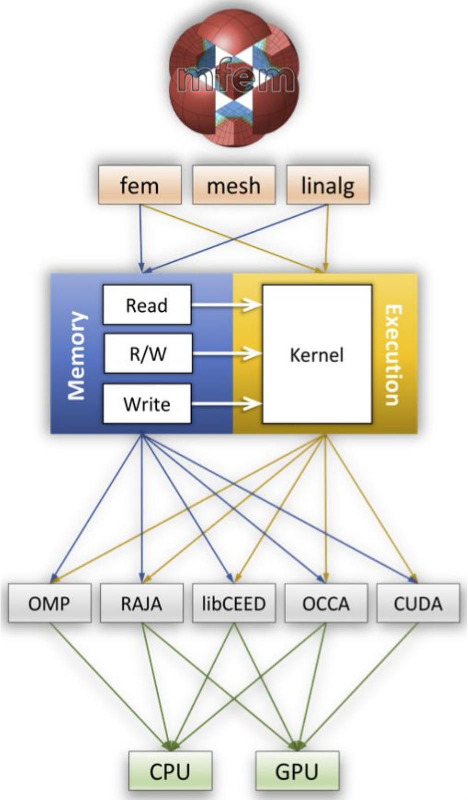
Bilinear form integrators are at the heart of any finite element method, they are used to compute the integrals of products of basis functions over individual mesh elements (or sometimes over edges or faces). Typically each element is contained in the support of several basis functions of both the domain and range spaces, therefore bilinear integrators simultaneously compute the integrals of all combinations of the relevant basis functions from the domain and range spaces. This produces a two dimensional array of results that are arranged into a small dense matrix of integral values called a *local element (stiffness) matrix.*

To put this another way, the `BilinearForm` class builds a global, sparse, finite element matrix, `glb_mat`, by performing the outer loop in the following pseudocode snippet whereas the `BilinearFormIntegrator` class performs the nested inner loops to compute the dense local element matrix, `loc_mat`.

```
for each elem in elements
   loc_mat = 0.0
   for each pt in quadrature_points
      for each u_j in elem
         for each v_i in elem
            loc_mat(i,j) += w(pt) * u_j(pt) v_i(pt)
         end
      end
   end
   glb_mat += loc_mat
end
```

Mixed Operators

Class Name	Domain	Range	Coef.	Dimension	Operator
VectorDivergenceIntegrator	$H1^d$, $L2^d$	H1, L2	S	1D, 2D, 3D	$(\lambda\nabla\cdot\vec{u}, v)$
GradientIntegrator	H1	$H1^d$, $L2^d$	S	1D, 2D, 3D	$(\lambda\nabla u, \vec{v})$



MFEM Features Examples ▾ Documentation ▾ Community ▾ Gallery Download

GPU support in MFEM

MFEM relies mainly on two features for running algorithms on devices such as GPUs:

- The memory manager handles transparently the moving of data between the host (CPU) and the device (e.g. GPU),
- The `mfem::forall` function to abstract `for` loops to parallelize the execution on an arbitrary device.

```
Vector u;
Vector v;
// ...
const auto u_data = u.Read(); // Express the intent to read u
auto v_data = v.ReadWrite(); // Express the intent to read and write v

// Abstract the loop: for(int i=0; i<u.Size(); i++)
mfem::forall(u.Size(), [=] MFEM_HOST_DEVICE (int i)
{
      v_data[i] *= u_data[i]; // This block of code is executed on the chosen device
});
```

Supported Integrators	native MFEM
Mass Integrator	✓
Vector Mass Integrator	✓
Vector FE Mass Integrator	✓
Convection Integrator	✓
Non-linear Convection Integrator	✓
Diffusion Integrator	✓
Vector Diffusion Integrator	✓
DGTrace Integrator	✓
Mixed Vector Gradient Integrator	✓
Mixed Vector Curl Integrator	✓
Mixed Vector Weak Curl Integrator	✓
Gradient Integrator	✓
Vector Divergence Integrator	✓
Vector FE Divergence Integrator	✓
Curl Curl Integrator	✓
Div Div Integrator	✓

- **Memory**: input/outputs data management
- **Execution**: outer/inner `forall` loops
- **Kernel**: Integrator, $G^T B^T D B G$

Initial PA - Integrators Implementation

- Mixed integrator: B_{test}, B_{trial}
- Fixed order: $\{4_{test}, 5_{trial}\}$ dofs, 5 at quadrature points

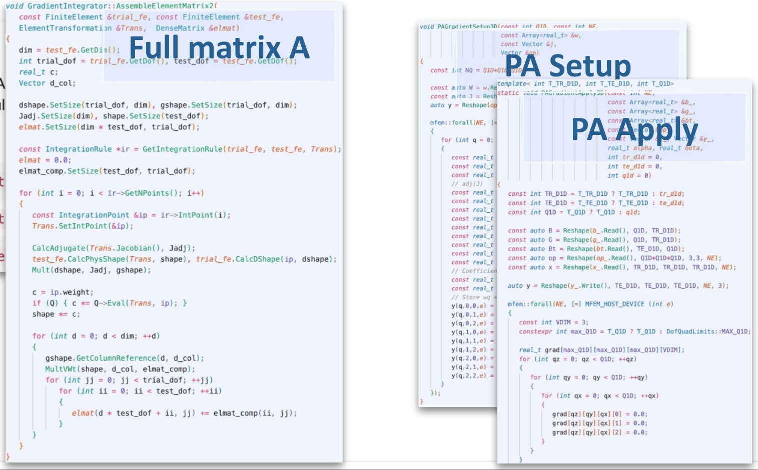
- CPU development, GPU portable
- Extract: PA setup, Q function

Writing Custom Integrators

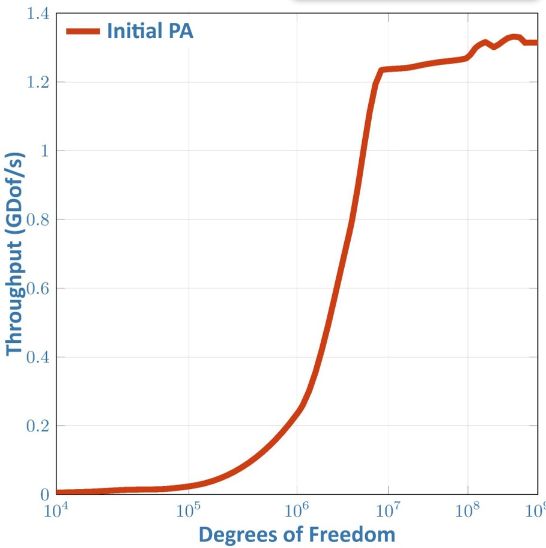
Element-wise integration arises in various places in the finite element method. A square and rectangular bilinear form operators, linear functionals, and the calcul

Type	Primary Function Needing Implementation
Square Operators	`BilinearFormIntegrator::AssembleElementMat`
Rectangular Operators	`BilinearFormIntegrator::AssembleElementMat`
Linear Functionals	`LinearFormIntegrator::AssembleRHSElementVe`

Name	C++ Expression	Formula
Jacobian Matrix	`const DenseMatrix &J = Trans.Jacobian()`	$J_{ij} = \frac{\partial x_i}{\partial u_j}$
Jacobian Determinant	`double detJ = Trans.Weight()`	$\det(J)$
Inverse Jacobian	`const DenseMatrix &InvJ = Trans.InverseJacobian()`	J^{-1}
Adjugate Jacobian	`const DenseMatrix &AdjJ = Trans.AdjugateJacobian()`	$\det(J)\,J^{-1}$



AMD MI300A



GPU Kernel Optimizations - Profiling and Benchmarking

- Using Google benchmark: agile development, CPU & GPU timings
- `tests/benchmarks` examples

Benchmark	Time	CPU	Iterations	Dofs	MDof/s	p	version
WaveOp/1/4/160	333 ms	333 ms	2	16.4613M	49.4931/s	4	1
WaveOp/8/4/160	305 ms	305 ms	2	16.4613M	53.9399/s	4	8
WaveOp/3/4/160	187 ms	187 ms	4	16.4613M	88.1353/s	4	3
WaveOp/4/4/160	144 ms	144 ms	5	16.4613M	114.076/s	4	4
WaveOp/6/4/160	174 ms	174 ms	4	16.4613M	94.6562/s	4	6
WaveOp/9/4/160	473 ms	473 ms	2	16.4613M	34.799/s	4	9
WaveOp/13/4/160	242 ms	242 ms	3	16.4613M	67.9619/s	4	13

Apple M2 Pro

benchmark

Benchmark	Time	CPU	Iterations	Dofs	MDof/s	p	version
WaveOp/1/4/160	17.4 ms	17.1 ms	41	16.4613M	960.653/s	4	1
WaveOp/8/4/160	1.83 ms	1.82 ms	385	16.4613M	9.06081k/s	4	8
WaveOp/3/4/160	0.897 ms	0.897 ms	773	16.4613M	18.3594k/s	4	3
WaveOp/4/4/160	0.703 ms	0.700 ms	983	16.4613M	23.5143k/s	4	4
WaveOp/6/4/160	1.20 ms	1.19 ms	586	16.4613M	13.8101k/s	4	6
WaveOp/9/4/160	1.07 ms	1.06 ms	656	16.4613M	15.531k/s	4	9
WaveOp/13/4/160	0.818 ms	0.810 ms	860	16.4613M	20.316k/s	4	13

AMD MI300A

- `-Rpass-analysis=kernel-resource-usage`
  [S,V,A]GPRs, ScratchSize, Occupancy, LDS Size
- `--ptxas-options=-v`

5M elements, 1.3B dofs

- rocprofv3 & https://ui.perfetto.dev
- NVIDIA Nsight Systems/Compute

Perfetto

Shared Memory PA Kernel Optimizations

void mfem::HipKernel1D<mfem::RK4... | void mfem::hip::HipKernel3D<128, 1, mfem::internal::SmemPAGra... | void mfem::HipKernel... | void mf... | void mfem::hip::HipKernel3D<128, 1, mfem::internal::SmemPA... | v | void mfem::HipKerne...

$B^T_{trial} D^T B_{test}$ G^T_{trial} G_{trial} $B^T_{test} D B_{trial}$

AMD MI300A

```
template <int D1DR, int D1DE, int Q1D>
void SmemPAGradientApplyTranspose3D(const int NE,
                                    const Array<real_t> &trial_bt,
                                    const Array<real_t> &trial_gt,
                                    const Array<real_t> &test_b,
                                    const Vector &d,
                                    const Vector &x, Vector &y,
                                    real_t a, real_t b)
{
   constexpr int MD1 = D1DE > D1DR ? D1DE : D1DR;

   const auto br_t = Reshape(trial_bt.Read(), D1DR, Q1D);
   const auto gr_t = Reshape(trial_gt.Read(), D1DR, Q1D);
   const auto be = Reshape(test_b.Read(), Q1D, D1DE);

   const auto D = Reshape(d.Read(), Q1D * Q1D * Q1D, 3, 3, NE);
   const auto X = Reshape(x.Read(), D1DE, D1DE, D1DE, NE, 3);
   auto Y = Reshape(y.Write(), D1DR, D1DR, D1DR, NE);

   mfem::forall_3D(NE, Q1D, Q1D, Q1D, [=] MFEM_HOST_DEVICE(int e)
   {
      MFEM_SHARED real_t BG[2][Q1D * MD1];
      MFEM_SHARED real_t sm0[3][Q1D * Q1D * Q1D], sm1[3][Q1D * Q1D * Q1D];

      kernels::internal::LoadX<MD1, Q1D>(e, D1DE, X, sm0);
      kernels::internal::LoadB<MD1, Q1D>(D1DE, Q1D, be, BG[0]);

      kernels::internal::EvalX<MD1, Q1D>(D1DE, Q1D, BG[0], sm0, sm1);
      kernels::internal::EvalY<MD1, Q1D>(D1DE, Q1D, BG[0], sm1, sm0);
      kernels::internal::EvalZ<MD1, Q1D>(D1DE, Q1D, BG[0], sm0, sm1);

      MFEM_FOREACH_THREAD(qz, z, Q1D)
      {
         MFEM_FOREACH_THREAD(qy, y, Q1D)
         {
            MFEM_FOREACH_THREAD(qx, x, Q1D)
            {
               real_t G[3], A[3];
               const int q = qx + (qy + qz * Q1D) * Q1D;
               kernels::internal::PullEval<Q1D>(Q1D, qx, qy, qz, sm1, G);
               A[0] = (D(q,0,0,e) * G[0]) + (D(q,0,1,e) * G[1]) + (D(q,0,2,e) * G[2]);
               A[1] = (D(q,1,0,e) * G[0]) + (D(q,1,1,e) * G[1]) + (D(q,1,2,e) * G[2]);
               A[2] = (D(q,2,0,e) * G[0]) + (D(q,2,1,e) * G[1]) + (D(q,2,2,e) * G[2]);
               kernels::internal::PushGrad<Q1D>(Q1D, qx, qy, qz, A, sm0);
            }
         }
      }
      MFEM_SYNC_THREAD;

      kernels::internal::LoadBtGt<MD1,Q1D>(D1DR,Q1D,br_t, gr_t, BG);
      kernels::internal::GradZt<MD1, Q1D>(D1DR, Q1D, BG, sm0, sm1);
      kernels::internal::GradYt<MD1, Q1D>(D1DR, Q1D, BG, sm1, sm0);
      kernels::internal::GradScaleXt<MD1, Q1D>(D1DR, Q1D, BG, sm0, Y, e, a, b);
   });
}
```

SHARED

B_{test}

D^T

B^T_{trial}

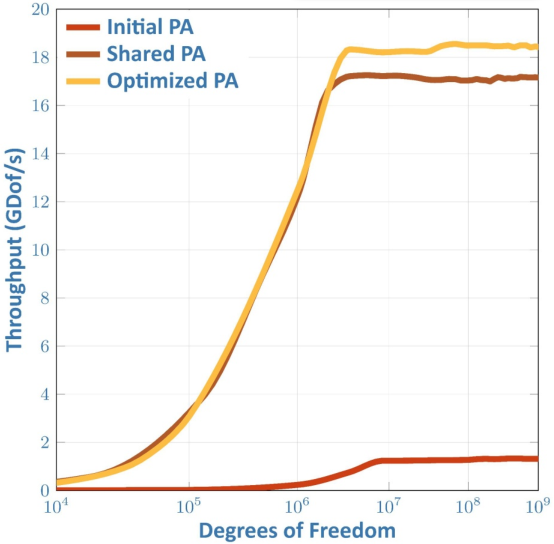
```
template <int D1DR, int D1DE, int Q1D>
void SmemPAGradientApply3D(const int NE,
                           const Array<real_t> &trial_b,
                           const Array<real_t> &trial_g,
                           const Array<real_t> &test_bt,
                           const Vector &d,
                           const Vector &x, Vector &y,
                           real_t a, real_t b)
{
   constexpr int MD1 = D1DE > D1DR ? D1DE : D1DR;

   const auto br = Reshape(trial_b.Read(), Q1D, D1DR);
   const auto gr = Reshape(trial_g.Read(), Q1D, D1DR);
   const auto be_t = Reshape(test_bt.Read(), D1DE, Q1D);

   const auto D = Reshape(d.Read(), Q1D * Q1D * Q1D, 3, 3, NE);
   const auto X = Reshape(x.Read(), D1DR, D1DR, D1DR, NE);
   auto Y = Reshape(y.Write(), D1DE, D1DE, D1DE, NE, 3);

   mfem::forall_3D(NE, Q1D, Q1D, Q1D, [=] MFEM_HOST_DEVICE(int e)
   {
      MFEM_SHARED real_t BG[2][Q1D * MD1];
      MFEM_SHARED real_t sm0[3][Q1D * Q1D * Q1D], sm1[3][Q1D * Q1D * Q1D];

      kernels::internal::LoadX<Q1D>(e, D1DR, X, sm0[0]);
      kernels::internal::LoadBG<MD1, Q1D>(D1DR, Q1D, br, gr, BG);

      kernels::internal::GradX<MD1, Q1D>(D1DR, Q1D, BG, sm0, sm1);
      kernels::internal::GradY<MD1, Q1D>(D1DR, Q1D, BG, sm1, sm0);
      kernels::internal::GradZ<MD1, Q1D>(D1DR, Q1D, BG, sm0, sm1);

      MFEM_FOREACH_THREAD(qz, z, Q1D)
      {
         MFEM_FOREACH_THREAD(qy, y, Q1D)
         {
            MFEM_FOREACH_THREAD(qx, x, Q1D)
            {
               real_t G[3], A[3];
               const int q = qx + (qy + qz * Q1D) * Q1D;
               kernels::internal::PullGrad<Q1D>(Q1D, qx, qy, qz, sm1, G);
               A[0] = (D(q, 0, 0, e) * G[0]) + (D(q, 1, 0, e) * G[1]) + (D(q, 2, 0, e) * G[2]);
               A[1] = (D(q, 0, 1, e) * G[0]) + (D(q, 1, 1, e) * G[1]) + (D(q, 2, 1, e) * G[2]);
               A[2] = (D(q, 0, 2, e) * G[0]) + (D(q, 1, 2, e) * G[1]) + (D(q, 2, 2, e) * G[2]);
               kernels::internal::PushEval<Q1D>(Q1D, qx, qy, qz, A, sm0);
            }
         }
      }
      MFEM_SYNC_THREAD;

      kernels::internal::LoadBt<MD1, Q1D>(D1DE, Q1D, be_t, BG[0], true);
      kernels::internal::EvalXt<MD1, Q1D>(D1DE, Q1D, BG[0], sm0, sm1);
      kernels::internal::EvalYt<MD1, Q1D>(D1DE, Q1D, BG[0], sm1, sm0);
      kernels::internal::EvalScaleZt<MD1, Q1D>(D1DE, Q1D, BG[0], sm0, Y, e, a, b);
   });
}
```

SHARED

B_{trial}

D

B^T_{test}

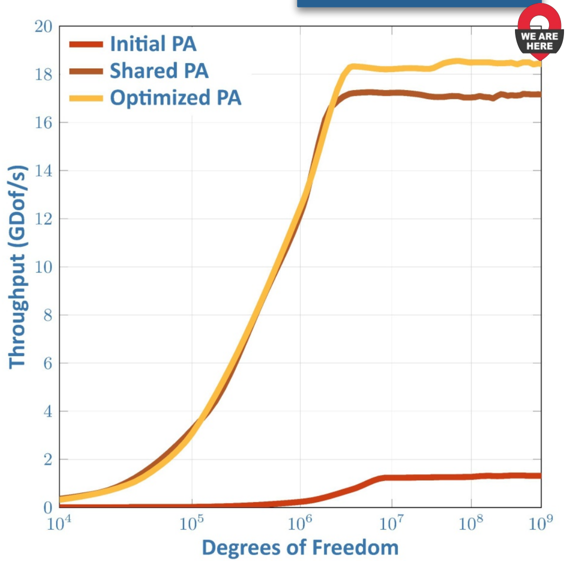


- G^T,G: Local to Element (L to E) vectors handled by MFEM
- "Optimized PA" reached by fixing the launch bounds

Application - HPC Context & Algorithms

$B^T_{trial} D^T B_{test}$ G^T_{trial} G_{trial} $B^T_{test} D B_{trial}$

AMD MI300A



- Vector sizes: $\mathcal{O}$(GB)
- Multiple read of the same data should be avoided
- Overall optimization fusion pass

- G^T, G: replaced by indirection + atomics
- $B^T_{test} D B_{trial} G_{trial} \; G^T_{trial} B^T_{trial} D^T B_{test} \; X_{u,p} \Rightarrow B^T_{test} B^T_{trial} D D^T B_{trial} B_{test} \; X_{u,p}$

$B^T_{trial} B^T_{test} D D^T B_{trial} B_{test}$

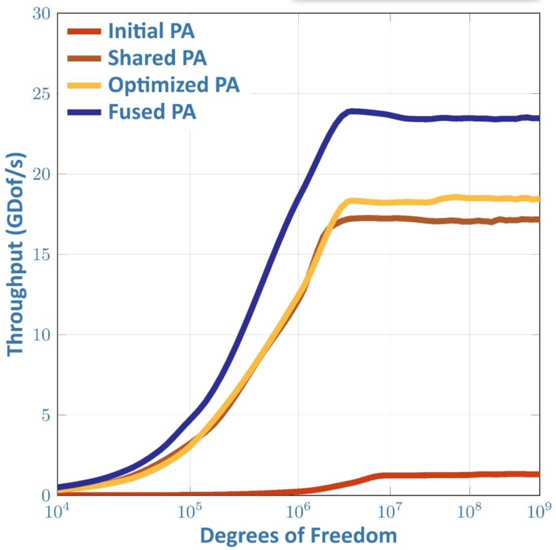
Fused PA Kernel Optimizations

- **Reduced memory access: PA data read once**

- **Challenges:**
  - **register pressure**
  - **increased complexity**
  - **shared memory usage**

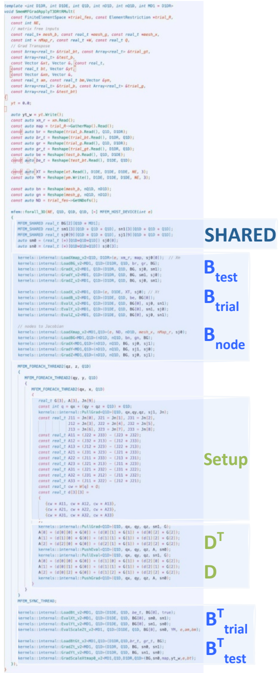
- **PA data uses 1/3 of the memory, w/:**
  - avoiding caching large vectors
  - recomputing on-the-fly some values
  - reusing temporary vectors from RK4

AMD MI300A

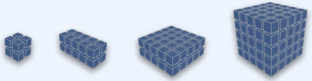


$B^T_{trial} B^T_{test} D D^T B_{trial} B_{test}$

Fused MF Kernel Optimizations - 1/2

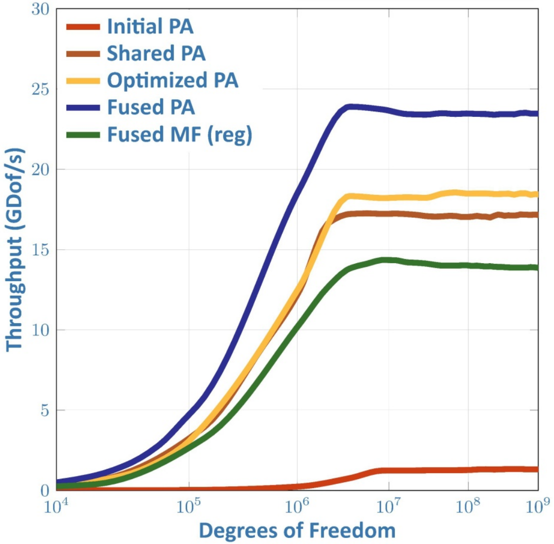


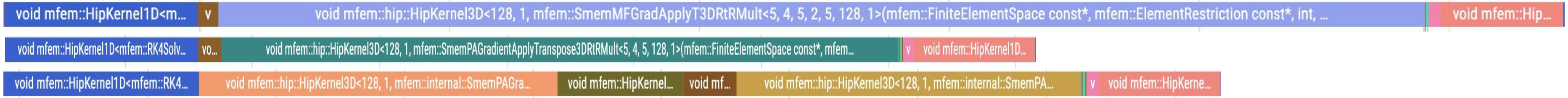
- No PA data stored at Quadrature points
- Extra input vectors & computations
  - Indirections, basis arrays
  - Mesh coordinates: used for 'setup'
  - Sum factorisation 3D vector grad basis



- Multiple implementations
  - `smem:` default, with 3D block of smem & threads
  - `regs:` less shared mem, 2D thread blocks

AMD MI300A

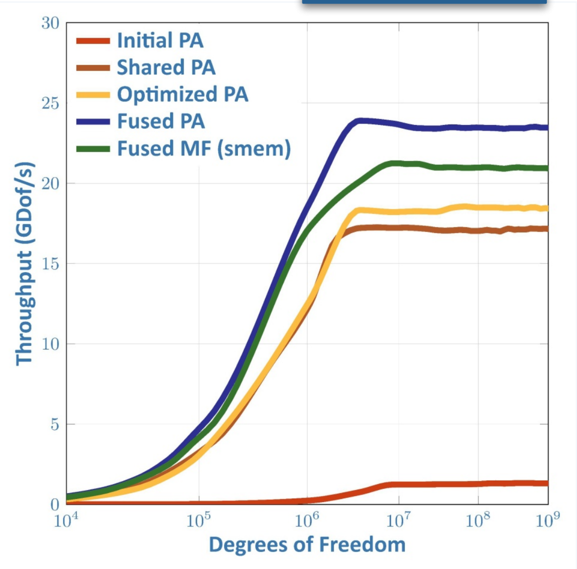




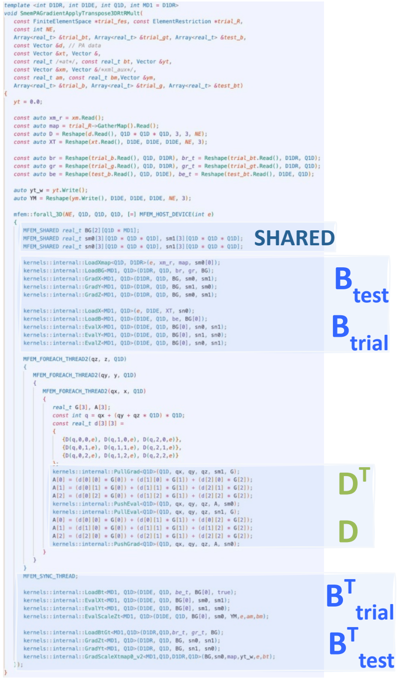
Fused MF Kernel Optimizations - 2/2

- Increasing the occupancy: number of wavefronts
- Use compiler output:
  - 170 max VREG
  - 3 waves ⇒ 1638 maximum `fp64`
- Reducing register usage:
  - `FORALL_DIRECT`
- Reducing the amount of shared memory
  - move B_{trial}, B_{test} data to `constant` memory
  - shuffle/re-use vector grad computation

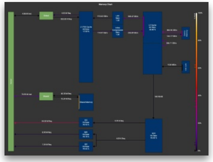
AMD MI300A



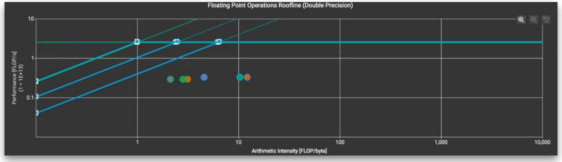
Shared Memory, Fused PA & Fused MF NVIDIA Kernel Optimizations

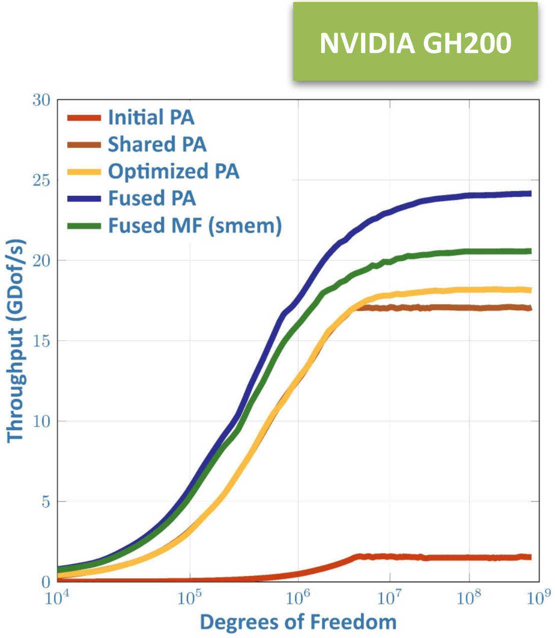


- Switch seamlessly to NVIDIA hardware
- Resilient to the different *versions*
- ✔ Shared memory bound kernel

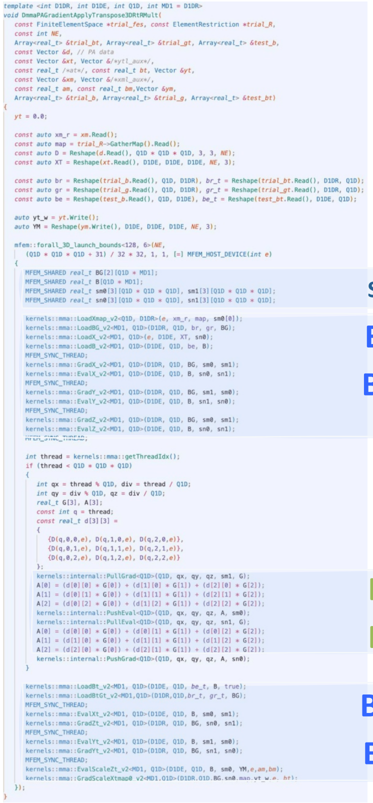


Fused PA





Unlocking Next-Level Performance Opportunities



SHARED

B_{test}

B_{trial}

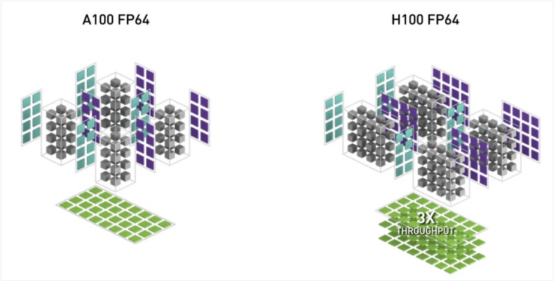
D^T

D

B^T_{trial}

B^T_{test}

- Jiqun Tu added Tensor Core based contractions
- M-by-N-by-K warp-synchronous collectives

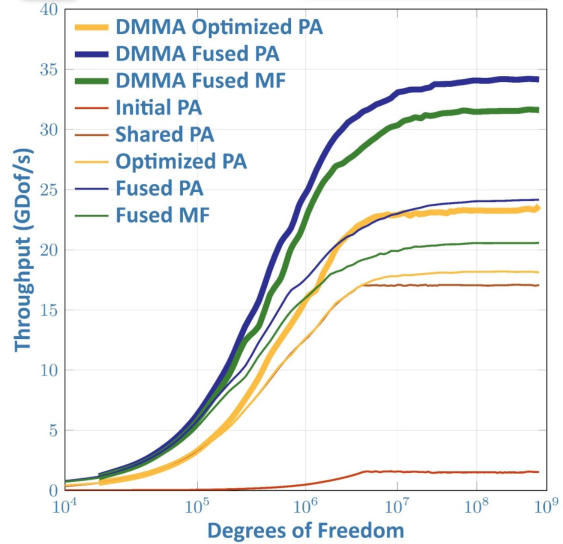


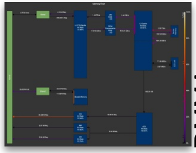
- 4th generation Matrix Multiply-Add (MMA)
  - D = op(A, B) + C
  - ⇒ Higher throughput
  - ⇒ More efficient way to share data

- For shared memory bound kernels ⇒ speedup

+10 GDof/s

NVIDIA GH200





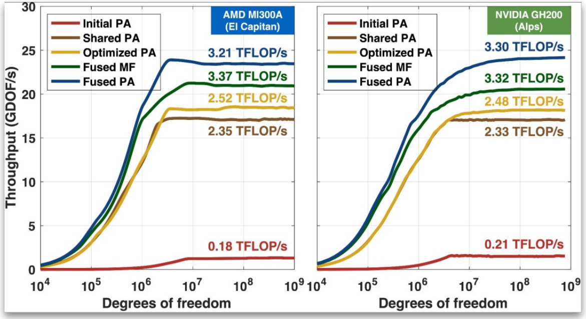
DMMA Fused PA

Conclusion

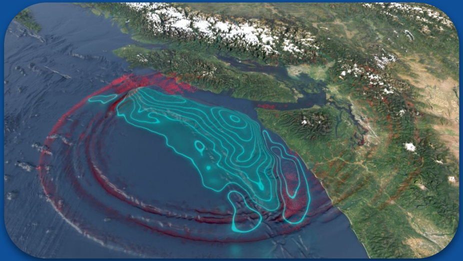
- Practical insights for enhancing FE HPC computations
- Contributions are welcome!

- Holistic Kernel Fusion Approach
  - Not only limited to kernel launch overhead
  - Re-use data, avoid in-&-out data transfers

- WIP tensor contraction API to support:
  - Low vs. high order algorithms
  - Arbitrary number of arguments for ∂FEM



mfem.org



[1] Henneking, Stefan, et al., Real-time Bayesian inference at extreme scale: A digital twin for tsunami early warning applied to the Cascadia subduction zone

Lawrence Livermore National Laboratory