Utilizing Emerging Hardware for Multiphysics Simulation Through Implicit High-Order Finite Element Methods With Tensor Product Structure

Jed Brown[1], Aron Ahmadia[2], Matt Knepley[3], Barry Smith[1]

[1]Mathematics and Computer Science Division, Argonne National Laboratory
[2]King Abdullah University of Science and Technology
[3]Computation Institute, University of Chicago

2011-12-05

The Roadmap

Hardware trends

- More cores (keep hearing $\mathcal{O}(1000)$ per node)
- Long vector registers (already 32 bytes for AVX and BG/Q)
- Must use SMT to hide memory latency
- Must use SMT for floating point performance (GPU, BG/Q)
- Large penalty for non-contiguous memory access

“Free flops”, but how can we use them?

- High order methods good: better accuracy per storage
- High order methods bad: work unit gets larger
- GPU threads have very little memory, must keep work unit small
- Need library composability, keep user contribution embarrassingly parallel

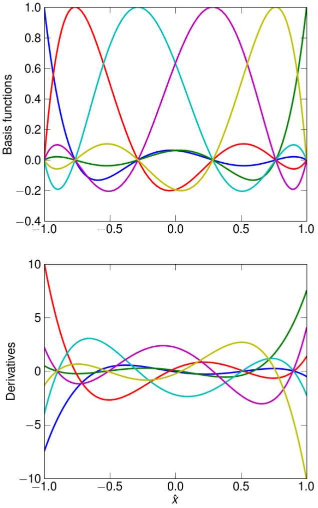
How to program this beast?

- Decouple physics from discretization
  - Expose small, embarrassingly parallel operations to user
  - Library schedules user threads for reuse between kernels
  - User provides physics in kernels run at each quadrature point
  - Continuous weak form: find $u \in \mathcal{V}_D$

$$v^T F(u) \sim \int_\Omega v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) = 0, \qquad \forall v \in \mathcal{V}_0$$

  - Similar form at faces, but may involve Riemann solve
- Library manages reductions
  - Interpolation and differentiation on elements
  - Exploit tensor product structure to keep working set small
  - Assembly into solution/residual vector (sum over elements)

Nodal hp-version finite element methods



1D reference element

- Lagrange interpolants on Legendre-Gauss-Lobatto points
- Quadrature $\hat{R}$, weights $\hat{W}$
- Evaluation: $\hat{B}, \hat{D}$

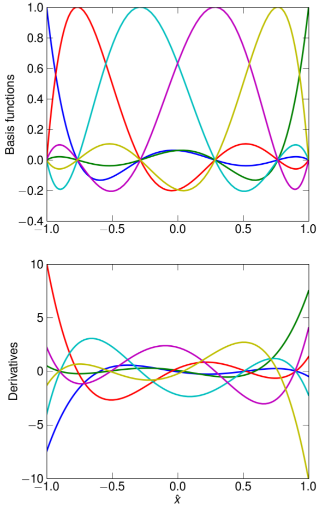
3D reference element

$$\hat{W} = \hat{W} \otimes \hat{W} \otimes \hat{W}$$
$$\hat{B} = \hat{B} \otimes \hat{B} \otimes \hat{B}$$

$$\hat{D}_0 = \hat{D} \otimes \hat{B} \otimes \hat{B}$$
$$\hat{D}_1 = \hat{B} \otimes \hat{D} \otimes \hat{B}$$
$$\hat{D}_2 = \hat{B} \otimes \hat{B} \otimes \hat{D}$$

These tensor product operations are very efficient, 70% of peak flop/s

Nodal hp-version finite element methods



1D reference element

- Lagrange interpolants on Legendre-Gauss-Lobatto points
- Quadrature $\hat{R}$, weights $\hat{W}$
- Evaluation: $\hat{B}, \hat{D}$

3D reference element

$$\hat{W} = \hat{W} \otimes \hat{W} \otimes \hat{W}$$
$$\hat{B} = \hat{B} \otimes \hat{B} \otimes \hat{B}$$

$$\hat{D}_0 = \hat{D} \otimes \hat{B} \otimes \hat{B}$$
$$\hat{D}_1 = \hat{B} \otimes \hat{D} \otimes \hat{B}$$
$$\hat{D}_2 = \hat{B} \otimes \hat{B} \otimes \hat{D}$$

These tensor product operations are very efficient, 70% of peak flop/s

Operations on physical elements

Mapping to physical space

$$x^e : \hat{K} \to K^e, \quad J^e_{ij} = \partial x^e_i / \partial \hat{x}_j, \quad (J^e)^{-1} = \partial \hat{x} / \partial x^e$$

Element operations in physical space

$$B^e = \hat{B} \qquad W^e = \hat{W} \Lambda(|J^e(r)|)$$

$$D^e_i = \Lambda\left(\frac{\partial \hat{x}_0}{\partial x_i}\right)\hat{D}_0 + \Lambda\left(\frac{\partial \hat{x}_1}{\partial x_i}\right)\hat{D}_1 + \Lambda\left(\frac{\partial \hat{x}_2}{\partial x_i}\right)\hat{D}_2$$

$$(D^e_i)^T = \hat{D}^T_0 \Lambda\left(\frac{\partial \hat{x}_0}{\partial x_i}\right) + \hat{D}^T_1 \Lambda\left(\frac{\partial \hat{x}_1}{\partial x_i}\right) + \hat{D}^T_2 \Lambda\left(\frac{\partial \hat{x}_2}{\partial x_i}\right)$$

Global problem is defined by assembly

$$F(u) = \sum_e \mathcal{E}^T_e \left[(B^e)^T W^e \Lambda(f_0(u^e, \nabla u^e)) + \sum_{i=0}^{d} (D^e_i)^T W^e \Lambda(f_{1,i}(u^e, \nabla u^e)) \right] = 0$$

where $u^e = B^e \mathcal{E}^e u$ and $\nabla u^e = \{D^e_i \mathcal{E}^e u\}_{i=0}^2$

Representation of Jacobians, Automation

- For unassembled representations, decomposition, and assembly
- Continuous weak form: find u

$$v^T F(u) \sim \int_\Omega v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) = 0, \qquad \forall v \in \mathcal{V}_0$$

- Weak form of the Jacobian $J(u)$: find w

$$v^T J(u) w \sim \int_\Omega \begin{bmatrix} v^T & \nabla v^T \end{bmatrix} \begin{bmatrix} f_{0,0} & f_{0,1} \\ f_{1,0} & f_{1,1} \end{bmatrix} \begin{bmatrix} w \\ \nabla w \end{bmatrix}$$

$$[f_{i,j}] = \begin{bmatrix} \frac{\partial f_0}{\partial u} & \frac{\partial f_0}{\partial \nabla u} \\ \frac{\partial f_1}{\partial u} & \frac{\partial f_1}{\partial \nabla u} \end{bmatrix} (u, \nabla u)$$

- Terms in $[f_{i,j}]$ easy to compute symbolically, AD more scalable.
- Nonlinear terms f_0, f_1 usually have the most expensive nonlinearities in the computation of scalar material parameters
  - Equations of state, effective viscosity, “star” region in Riemann solve
  - Compute gradient with reverse-mode, store at quadrature points.
  - Perturb scalars, then use forward-mode to complete the Jacobian.
  - Flip for action of the adjoint.

Conservative (non-Boussinesq) two-phase ice flow

Find momentum density ρu, pressure p, and total energy density E:

$$(\rho u)_t + \nabla\cdot(\rho u \otimes u - \eta D u_i + p 1) - \rho g = 0$$
$$\rho_t + \nabla\cdot\rho u = 0$$
$$E_t + \nabla\cdot\big((E+p)u - k_T \nabla T - k_\omega \nabla \omega\big) - \eta D u_i : D u_i - \rho u \cdot g = 0$$

- Solve for density ρ, ice velocity u_i, temperature T, and melt fraction ω using constitutive relations.
  - Simplified constitutive relations can be solved explicitly.
  - Temperature, moisture, and strain-rate dependent rheology η.
  - High order FEM, typically Q_3 momentum & energy, SUPG (yuck).
- DAEs solved implicitly after semidiscretizing in space.
- Preconditioning using nested fieldsplit

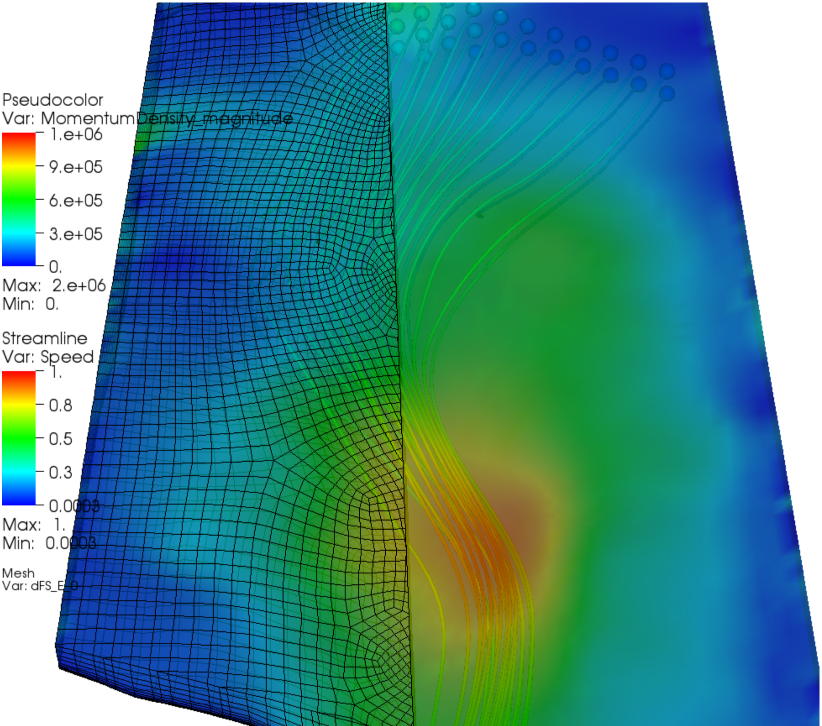
Traversal code

- CPU traversal computes coefficients of test functions,
  https://github.com/jedbrown/dohp/

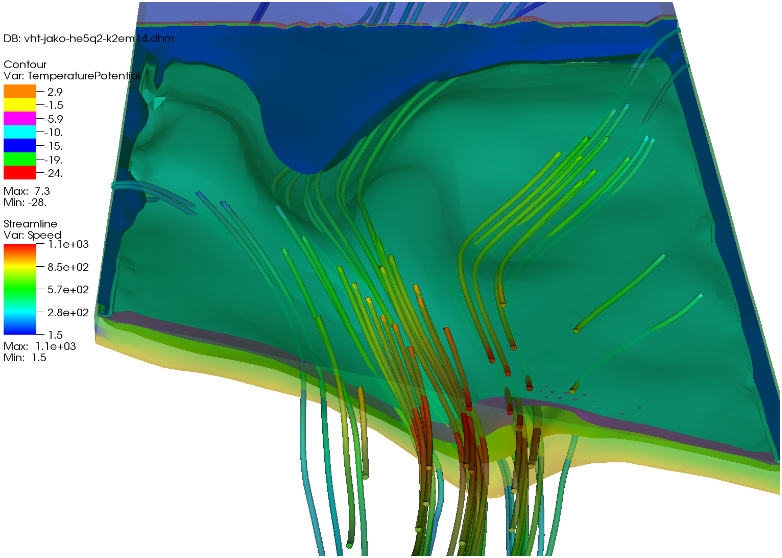
```
while (IteratorHasPatch(iter)) {
  IteratorGetPatchApplied(iter,&Q,&jw,
      &x,&dx,NULL,NULL,
      &u,&du,&u_,&du_, &p,&dp,&p_,NULL, &e,&de,&e_,&de_);
  IteratorGetStash(iter,NULL,&stash);
  for (dInt i=0; i<Q; i++) {
    PointwiseFunction(context,x[i],dx[i],jw[i],
        u[i],du[i],p[i],dp[i],e[i],de[i],
        &stash[i], u_[i],du_[i],p_[i],e_[i],de_[i]);
  }
  IteratorCommitPatchApplied(iter,INSERT_VALUES, NULL,NULL,
                             u_,du_, p_,NULL, e_,de_);
  IteratorNextPatch(iter);
}
```

- GPU version calls `PointwiseFunction()` directly.
- Unassembled Jacobian application reuses `stash`

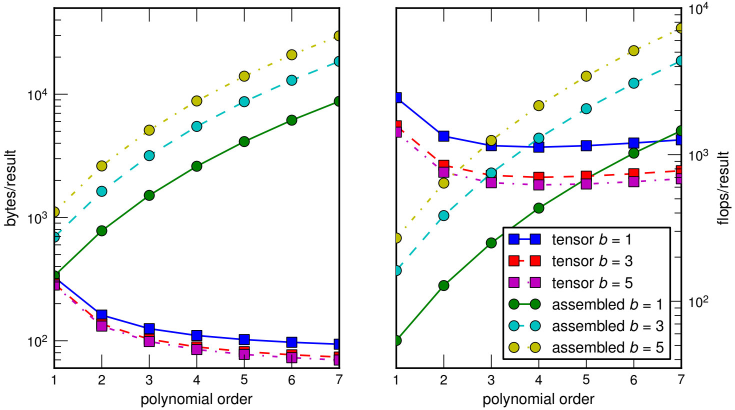
```
PointwiseJacobian(context,&stash[i],dx[i],jw[i],
                  u[i],du[i],p[i],dp[i],e[i],de[i],
                  u_[i],du_[i],p_[i],e_[i],de_[i]);
```



Pseudocolor
Var: MomentumDensity_magnitude
1.e+06
9.e+05
6.e+05
3.e+05
0.
Max: 2.e+06
Min: 0.
Streamline
Var: Speed
1.
0.8
0.5
0.3
0.0003
Max: 1.
Min: 0.0003
Mesh
Var: dFS_E_0


DB: vht-jako-he5q2-k2em14.dhm
Contour
Var: TemperaturePotential
2.9
-1.5
-5.9
-10.
-15.
-19.
-24.
Max: 7.3
Min: -28.
Streamline
Var: Speed
1.1e+03
8.5e+02
5.7e+02
2.8e+02
1.5
Max: 1.1e+03
Min: 1.5

Performance of assembled versus unassembled



- High order Jacobian stored unassembled using coefficients at quadrature points, can use local AD
- Choose approximation order at run-time, independent for each field
- Precondition high order using assembled lowest order method
- Implementation $> 70\%$ of FPU peak, SpMV bandwidth wall $< 4\%$

Memory Bandwidth

Operation	Arithmetic Intensity (flop/s per byte)
Sparse matrix-vector product	1/6
Dense matrix-vector product	1/4
Unassembled matrix-vector product	≈ 8
High-order residual evaluation	> 5

Processor	BW (GB/s)	Peak (GF/s)	Balanced AI (F/s/B)
Sandy Bridge 6-core	21*	150	7.2
Magny Cours 16-core	42*	281	6.7
Blue Gene/Q node	43	205	4.8
GeForce 9400M	21	54	2.6
GTX 285	159	1062	6.8
Tesla M2050	144	1030	7.1

Outlook

- Sparse matrix assembly (for preconditioning) not shown
  - > 100 GF/s for lowest order Stokes (Matt Knepley)
  - common physics code with CPU implementation
  - Dohp CPU version faster than libMesh and Deal.II for Q_1
  - Q_1 assembly embedded in higher order is 8% slower than hand-rolled
- Can't wait for OpenCL to implement indirect function calls
- Symbolic differentiation too slow, tired of hand-differentiation
- I want source-transformation AD with indirect function calls
- Find correct amount of reuse between face and cell integration
- Riemann solves harder to vectorize
- Finer grained parallelism in GPU tensor product kernels
- Hide dispatch to pointwise kernels inside library
  - Easy, but scary. Library/framework becomes **F**ramework.