Library Interface Design and Performance Portability

Jed Brown, Jeremy Thompson, Valeria Barra

SIAM CSE, Spokane, 2019-02-25

This talk: https://jedbrown.org/files/20190225-PerfPortability.pdf
What is performance portability?

- Performs well across range of architectures and problem configurations with modest development and maintenance effort.
- All architectures require massive parallelism
  - 28-core Xeon: dual-issue FMA with 16-lane registers, 6-cycle latency: 5376 in-flight flops
  - About 4x more for V100
- Architectural differences
  - Persistent cache on CPU, big register space on GPU
  - Programming model expression of coalesced loads (e.g., CUDA vs OpenMP/OpenACC)
  - Tolerance for lane divergence (SIMT versus masked SIMD); what does compiler need to know/use?
- Problem configurations: time to solution requirements (problem sizes)
  - constitutive models, etc.
Modest development and maintenance effort

- Some custom code may be needed; how to leverage?
- Granularity of extensibility
  - BLAS/LAPACK: Users compose large dense linear algebraic operations
  - BLIS: Users implement packing schemes; reuse “microkernel”
  - Users implement different PDE, discretization, and constitutive models
  - FEniCS: domain-specific language for finite element
- Maintenance
  - Interface stability for the extensible parts
  - Ability to rapidly affect “infrastructure” (turtles all the way down?)
## Approaches

<table>
<thead>
<tr>
<th>Type of software</th>
<th>Expressive</th>
<th>Interoperable/Environment</th>
<th>Contributors</th>
<th>New architecture</th>
<th>Packaging/Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library (LAPACK, FFTW, BLIS @MS129 ↓)</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td>Dynamic/extensible Library (libCEED)</td>
<td>0</td>
<td>+</td>
<td>+</td>
<td>plugin</td>
<td>if stable ABI</td>
</tr>
<tr>
<td>Template Library (Kokkos @MS129)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+</td>
<td>recompile</td>
</tr>
<tr>
<td>DSL/code generation/JIT (Firedrake)</td>
<td>+</td>
<td>−</td>
<td>0</td>
<td>+</td>
<td>?</td>
</tr>
<tr>
<td>New Language (Chapel)</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>?</td>
</tr>
</tbody>
</table>
Performance of assembled versus unassembled

- Arithmetic intensity for $Q_p$ elements
  - $< \frac{1}{4}$ (assembled), $\approx 10$ (unassembled), $\approx 5$ to 10 (hardware)

- store Jacobian information at Gauss quadrature points, can use AD
libCEED: Code for Efficient Extensible Discretization

- BSD-2 license, C library with Fortran interface
- Releases: v0.1 — v0.3 (2018); v0.4 to be released in March
- Purely algebraic interface
- Extensible backends
  - CPU: reference, blocked/vectorized, libXSMM
  - OCCA (just-in-time compilation): CPU, OpenMP, OpenCL, CUDA
  - MAGMA
  - CUDA using NVRTC (Steven Roberts & Yohann Dudouit, to be merged soon)
- Platform for collaboration with vendors
- Minimal assumptions about execution environment, parallel decomposition
- Primary target: high order finite element methods
  - $H^1$, $H(\text{div})$, $H(\text{curl})$
  - Also of interest to spectral difference, etc.
  - Exploit tensor product structure when possible
$$A = \mathbf{p}^T \mathbf{\epsilon}^T \mathbf{B}^T \mathbf{D} \mathbf{B} \mathbf{\epsilon} \mathbf{p}$$
Quadrature Function

\[ v^T F(u) \sim \int_{\Omega} v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) \quad v^T Jw \sim \int_{\Omega} \begin{bmatrix} v \\ \nabla v \end{bmatrix}^T \begin{bmatrix} f_{0,0} & f_{0,1} \\ f_{1,0} & f_{1,1} \end{bmatrix} \begin{bmatrix} w \\ \nabla w \end{bmatrix} \]

\[ u = B_I \mathcal{E}_e u_L \quad \nabla u = \frac{\partial X}{\partial x} B_\nabla \mathcal{E}_e u_L \]

\[ Jw = \sum_{e} \mathcal{E}_e^T \begin{bmatrix} B_I \\ B_\nabla \end{bmatrix}^T \begin{bmatrix} I \\ \left( \frac{\partial X}{\partial x} \right)^T \end{bmatrix} \begin{bmatrix} f_{0,0} & f_{0,1} \\ f_{1,0} & f_{1,1} \end{bmatrix} \begin{bmatrix} I \\ \left( \frac{\partial X}{\partial x} \right) \end{bmatrix} \begin{bmatrix} B_I \\ B_\nabla \end{bmatrix} \mathcal{E}_e w_L \]

- \( B \) and \( B_\nabla \) are tensor contractions – independent of element geometry
- Choice of how to order and represent gathers \( \mathcal{E} \) and scatters \( \mathcal{E}^T \)
- Who computes the metric terms and other coefficients?
- Similar for Neumann/Robin and nonlinear boundary conditions
Quadrature Functions

- Multiple inputs and outputs
- Independent operations at each of \( Q \) quadrature points
  - Ordering and number of elements not specified

```c
int L2residual(void *ctx, CeedInt Q,
    const CeedScalar *const in[],
    CeedScalar *const out[])
{
    const CeedScalar *u = in[0], *rho = in[1], *target = in[2];
    CeedScalar *v = out[0];
    #pragma omp simd
    for (CeedInt i=0; i<Q; i++)
        v[i] = rho[i] * (u[i] - target[i]);
    return 0;
}
CeedQFunctionAddInput(qf, "u", 1, CEED_EVAL_INTERP);
CeedQFunctionAddInput(qf, "rho", 1, CEED_EVAL_INTERP);
CeedQFunctionAddInput(qf, "target", 1, CEED_EVAL_INTERP);
CeedQFunctionAddOutput(qf, "v", 1, CEED_EVAL_INTERP);
```
Building Operators

\[
v^T F(u) \sim \int_\Omega v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u)
\]

\[
u = B_{lE} e u_L \quad \nabla u = \frac{\partial X}{\partial x} B_{\nabla E} e u_L
\]

- `CeedOperatorCreate(ceed, f, &op);`
- `CeedOperatorSetField(op, "velocity", E, B, u_L);`
- **Apply**: implementation handles batching, work buffers, and calling \( f(u, \nabla u) \).
- User links to interface library
- Backend implementation switchable at run-time
- Two-phase implementation enables connectivity analysis and JIT
libCEED Operator

\[
A = \mathcal{P}^T \mathcal{E}^T B^T \mathcal{DBE} \mathcal{P}
\]

- quadrature function \( D \)
- For each field: element restriction \( \mathcal{E} \), basis \( B \), where to find vector

  ```c
  CeedOperatorCreate(ceed, qf_L2residual, &op);
  CeedOperatorSetField(op, "u", E, Basis, CEED_VECTOR_ACTIVE);
  CeedOperatorSetField(op, "rho", E_id, CEED_BASIS_COLOCATED, rho);
  CeedOperatorSetField(op, "target", E_id, CEED_BASIS_COLOCATED, target);
  CeedOperatorSetField(op, "v", E, Basis, CEED_VECTOR_ACTIVE);

  CeedOperatorApply(op, u, v, &request);
  ```
Element restriction $\mathcal{E}_e$

- Conforming homogeneous mesh: boolean matrix with homogeneous block size
- Non-conforming mesh: anchored rows have linear combination
- Nek5000/DG-style E-vector: indexed identity
- libCEED backends are allowed to reorder, compress, etc.
- May be applied all at once or in batches
- CeedOperator implementation chooses if/how to fuse
Performance versatility: $n_{1/2}$ and $t_{1/2}$

- Suppose a linear scaling algorithm
- Let $r(n)$ be the performance rate (e.g., DOF/second or GF/s) for local problem size $n = N/P$
- Let $r_{\text{max}} = \max_n r(n)$ be the peak attainable performance
- $n_{1/2} = \min \{ n : r(n) \geq \frac{1}{2} r_{\text{max}} \}$
  - Local problem sizes $n < n_{1/2}$ will not yield acceptable efficiency
- $t_{1/2} = 2n_{1/2}/r_{\text{max}}$
  - Time to solution less than $t_{1/2}$ is not feasible with acceptable efficiency
Performance spectra

Points per compute node

[DOFs x CG iterations] / [compute nodes x seconds]

Time per iteration

[DOFs x CG iterations] / [compute nodes x seconds]
Vectorization techniques

- Vectorize within a single high-order element
  - Minimal working set (as small as one element)
  - Specialized implementation for different degree/# quadrature points
  - Hard to avoid cross-lane operations at modest degree
  - Nek5000

- Vectorize across elements in batches \([i,j,k,e]\)
  - Working set has at least vector length number of elements (e.g., 8)
  - Generic implementation is easy to optimize; no cross-lane operations
  - HPGMG-FE, Deal.II (Kronbichler and Kormann), MFEM (new)
libXSMM internal vs external vectorization
Outlook

- `spack install ceed` works; next release in March
- libCEED is interested in contributors and friendly users
- Need consistent strategy for JIT of Q-functions
  - How much fusion and inlining?
- Performance optimizations in progress
  - Backends should automatically choose internal versus external vectorization
  - Choice depends on architecture, element size, number of fields, problem size
  - Even/odd decomposition (symmetry)
- Mixed topology (works, but needs flattening)
- Hanging node interpolation (can be done in P)