FEM Integration with Quadrature on the GPU

Matthew Knepley

Computation Institute
University of Chicago
Department of Molecular Biology and Physiology
Rush University Medical Center

GPU-SMP 2012
Shenzhen, China  June 1–4, 2012
Collaborators

- Andy R. Terrel
- Andreas Klöckner
- Jed Brown
- Robert Kirby
Why Scientific Libraries?

1. Why Scientific Libraries?
   - What is PETSc?

2. Linear Systems are Easy

3. Finite Element Integration

4. Future Direction
Main Point

To be widely accepted, GPU computing must be transparent to the user, and reuse existing infrastructure.
To be widely accepted, GPU computing must be transparent to the user, and reuse existing infrastructure.
To be widely accepted, GPU computing must be transparent to the user, and reuse existing infrastructure.
Failure

- Parallelizing Compilers
- Automatic program decomposition

Success

- MPI (Library Approach)
- PETSc (Parallel Linear Algebra)
- User provides only the mathematical description
Why Scientific Libraries?

Lessons from Clusters and MPPs

Failure

- Parallelizing Compilers
- Automatic program decomposition

Success

- MPI (Library Approach)
- PETSc (Parallel Linear Algebra)
- User provides only the mathematical description
Why Scientific Libraries?

- What is PETSc?
What is PETSc?

A freely available and supported research code for the parallel solution of nonlinear algebraic equations

Free
- Download from http://www.mcs.anl.gov/petsc
- Free for everyone, including industrial users

Supported
- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov

Usable from C, C++, Fortran 77/90, Matlab, Julia, and Python
What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems
    - Cray XT6, BG/Q, NVIDIA Fermi, K Computer
  - Loosely coupled systems, such as networks of workstations
    - IBM, Mac, iPad/iPhone, PCs running Linux or Windows

PETSc History

- Begun September 1991
- Over 60,000 downloads since 1995 (version 2)
- Currently 400 per month

PETSc Funding and Support

- Department of Energy
  - SciDAC, MICS Program, AMR Program, INL Reactor Program
- National Science Foundation
  - CIG, CISE, Multidisciplinary Challenge Program
The PETSc Team

Bill Gropp  Barry Smith  Satish Balay

Jed Brown  Matt Knepley  Lisandro Dalcin

Hong Zhang  Mark Adams  Toby Issac
Who Uses PETSc?

Computational Scientists

- Earth Science
  - PyLith (CIG)
  - Underworld (Monash)
  - Magma Dynamics (LDEO, Columbia, Oxford)

- Subsurface Flow and Porous Media
  - STOMP (DOE)
  - PFLOTRAN (DOE)
Who Uses PETSc?

Computational Scientists

- CFD
  - Firedrake
  - Fluidity
  - OpenFOAM
  - freeCFD
  - OpenFVM

- MicroMagnetics
  - MagPar

- Fusion
  - XGC
  - BOUT++
  - NIMROD
Algorithm Developers

- **Iterative methods**
  - Deflated GMRES
  - LGMRES
  - QCG
  - SpecEst

- **Preconditioning researchers**
  - Prometheus (Adams)
  - ParPre (Eijkhout)
  - FETI-DP (Klawonn and Rheinbach)
Who Uses PETSc?

Algorithm Developers

- **Finite Elements**
  - libMesh
  - MOOSE
  - PETSc-FEM
  - Deal II
  - OOFEM

- **Other Solvers**
  - Fast Multipole Method (**PetFMM**)
  - Radial Basis Function Interpolation (**PetRBF**)
  - Eigensolvers (**SLEPc**)
  - Optimization (**TAO**)

M. Knepley (UC)
What Can We Handle?

- PETSc has run implicit problems with over **500 billion** unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- PETSc has run on over **290,000** cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- PETSc applications have run at 23% of peak (**600 Teraflops**)
  - Jed Brown on NERSC Edison
  - HPGMG code
PETSc has run implicit problems with over 500 billion unknowns

- UNIC on BG/P and XT5
- PFLOTRAN for flow in porous media

PETSc has run on over 290,000 cores efficiently

- UNIC on the IBM BG/P Jugene at Jülich
- PFLOTRAN on the Cray XT5 Jaguar at ORNL

PETSc applications have run at 23% of peak (600 Teraflops)

- Jed Brown on NERSC Edison
- HPGMG code
PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

PETSc has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code
Interface Questions

How should the user interact with manycore systems?
Through computational libraries

How should the user interact with the library?
Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?
- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions
Interface Questions

How should the user interact with manycore systems?
Through computational libraries

How should the user interact with the library?
Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?
- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions
Interface Questions

How should the user interact with manycore systems?
Through computational libraries

How should the user interact with the library?
Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?
- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions
Interface Questions

How should the user interact with manycore systems?
Through computational libraries

How should the user interact with the library?
Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?
- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions
How should the user interact with manycore systems?
Through computational libraries

How should the user interact with the library?
Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?
- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions
Interface Questions

How should the user interact with manycore systems?
Through computational libraries

How should the user interact with the library?
Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?
- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions
Outline

1. Why Scientific Libraries?
2. Linear Systems are Easy
3. Finite Element Integration
4. Future Direction
Some parts of PDE computation are less mature

Linear Algebra
- One universal interface
  - BLAS, PETSc, Trilinos, FLAME, Elemental
- Entire problem can be phrased in the interface
  - $Ax = b$
- Standalone component

Finite Elements
- Many Interfaces
  - FEniCS, FreeFEM++, DUNE, dealII, Fluent
- Problem definition requires general code
- Physics, boundary conditions
- Crucial interaction with other simulation components
- Discretization, mesh/geometry
Some parts of PDE computation are less mature

Linear Algebra
- One universal interface
  - BLAS, PETSc, Trilinos, FLAME, Elemental
- Entire problem can be phrased in the interface
  - $Ax = b$
- Standalone component

Finite Elements
- Many Interfaces
  - FEniCS, FreeFEM++, DUNE, dealII, Fluent
- Problem definition requires general code
- Physics, boundary conditions
- Crucial interaction with other simulation components
- Discretization, mesh/geometry
Some parts of PDE computation are less mature

### Linear Algebra
- One universal interface
  - BLAS, PETSc, Trilinos, FLAME, Elemental
- Entire problem can be phrased in the interface
  - $Ax = b$
- Standalone component

### Finite Elements
- Many Interfaces
  - FEniCS, FreeFEM++, DUNE, dealII, Fluent
- Problem definition requires general code
  - Physics, boundary conditions
- Crucial interaction with other simulation components
  - Discretization, mesh/geometry
Linear Systems are Easy

Interface Maturity

Some parts of PDE computation are less mature

Linear Algebra
- One universal interface
  - BLAS, PETSc, Trilinos, FLAME, Elemental
- Entire problem can be phrased in the interface
  - \( Ax = b \)
- Standalone component

Finite Elements
- Many Interfaces
  - FEniCS, FreeFEM++, DUNE, dealII, Fluent
- Problem definition requires general code
  - Physics, boundary conditions
- Crucial interaction with other simulation components
  - Discretization, mesh/geometry
PETSc-GPU

**PETSc now has support for Krylov solves on the GPU**

- `-with-cuda=1` `-with-cusp=1` `-with-thrust=1`
  - Also possibly `-with-precision=single`
- **New classes** `VECCUDA` and `MATAIJCUDA`
  - Just change type on command line, `-vec_type veccuda`
- Uses **Thrust** and **Cusp** libraries from Nvidia guys
- Does not communicate vectors during solve
Example
Driven Cavity Velocity-Vorticity with Multigrid

```
ex50 -da_vec_type seqcusp
     -da_mat_type aijcusp -mat_no_inode  # Setup types
     -da_grid_x 100 -da_grid_y 100  # Set grid size
     -pc_type none -pc_mg_levels 1  # Setup solver
     -preload off -cuda_synchronize  # Setup run
     -log_summary
```
**Flow Solver**

32 × 32 × 32 grid

<table>
<thead>
<tr>
<th>Routine</th>
<th>Time (s)</th>
<th>MFlops</th>
<th>MFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KSPSolve</td>
<td>8.3167</td>
<td>4370</td>
<td>526</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.5031</td>
<td>769</td>
<td>512</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KSPSolve</td>
<td>1.6382</td>
<td>4500</td>
<td>2745</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.3554</td>
<td>830</td>
<td>2337</td>
</tr>
</tbody>
</table>

P. Lichtner, G. Hammond, R. Mills, B. Phillip
Outline

1. Why Scientific Libraries?
2. Linear Systems are Easy
3. Finite Element Integration
4. Future Direction
Element integrals are decomposed into **analytic** and **geometric** parts:

\[ \int_{\mathcal{T}} \nabla \phi_i(x) \cdot \nabla \phi_j(x) d\mathbf{x} = \int_{\mathcal{T}} \frac{\partial \phi_i(x)}{\partial x_\alpha} \frac{\partial \phi_j(x)}{\partial x_\alpha} d\mathbf{x} \]

\[ = \int_{\mathcal{T}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |J| d\mathbf{x} \]

\[ = \left( \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \right) \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} d\mathbf{x} \]

\[ = G^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ij} \]

Coefficients are also put into the geometric part.
FEniCS based code achieves

90 GF/s on 3D $P_1$ Laplacian
100 GF/s on 2D $P_1$ Elasticity

- Relies on analytic integration
- Dot products are workhorse
- Crossover point with quadrature with multiple fields

Finite Element Integration on GPUs, ACM TOMS, Andy R. Terrel and Matthew G. Knepley
Why Quadrature?

Quadrature can handle

- many fields (linearization)
- non-affine elements (Argyris)
- non-affine mappings (isoparametric)
- functions not in the FEM space

Optimizations for Quadrature Representations of Finite Element Tensors through Automated Code Generation, ACM TOMS, Kristian B. Ølgaard and Garth N. Wells
Jed Brown’s Model

We consider weak forms dependent only on fields and gradients,

\[ \int_{\Omega} \phi \cdot f_0(u, \nabla u) + \nabla \phi : \vec{f}_1(u, \nabla u) = 0. \]  \hspace{1cm} (6)

Discretizing we have

\[ \sum_e \mathcal{E}_e^T \left[ B^T W^q f_0(u^q, \nabla u^q) + \sum_k D_k^T W^q \vec{f}_1^k(u^q, \nabla u^q) \right] = 0 \]  \hspace{1cm} (7)

- \( f_n \): pointwise physics functions
- \( u_q \): field at a quad point
- \( W^q \): diagonal matrix of quad weights
- \( B, D \): basis function matrices which reduce over quad points
- \( \mathcal{E} \): assembly operator
\nabla \phi_i \cdot \nabla u
\[ \nabla \phi_j \cdot \nabla u \]

```c
__device__ vecType f1(realType u[], vecType gradU[], int comp) {
    return gradU[comp];
}
```
\nabla \phi_i \cdot (\nabla u + \nabla u^T)
\[ \nabla \phi_i \cdot (\nabla u + \nabla u^T) \]

```c
__device__ vecType f1 (realType u[], vecType gradU[], int comp) {
    vecType f1;

    switch (comp) {
    case 0:
        f1.x = 0.5*(gradU[0].x + gradU[0].x);
        f1.y = 0.5*(gradU[0].y + gradU[1].x);
        break;
    case 1:
        f1.x = 0.5*(gradU[1].x + gradU[0].y);
        f1.y = 0.5*(gradU[1].y + gradU[1].y);
    }
    return f1;
}
```
\nabla \phi_i \cdot \nabla u + \phi_i k^2 u
\[ \nabla \phi_i \cdot \nabla u + \phi_i k^2 u \]

```c
__device__ vecType f1(realType u[], vecType gradU[], int comp) {
    return gradU[comp];
}

__device__ realType f0(realType u[], vecType gradU[], int comp) {
    return k * k * u[0];
}
```
\[ \nabla \phi_i \cdot \nabla \vec{u} - (\nabla \cdot \phi)p \]
\[ \nabla \phi_i \cdot \nabla \bar{u} - (\nabla \cdot \phi) p \]

```c
void f1(PetscScalar u[], const PetscScalar gradU[], PetscScalar f1[]) {
  const PetscInt dim = SPATIAL_DIM_0;
  const PetscInt Ncomp = NUM_BASIS_COMPONENTS_0;
  PetscInt comp, d;

  for (comp = 0; comp < Ncomp; ++comp) {
    for (d = 0; d < dim; ++d) {
      f1[comp*dim+d] = gradU[comp*dim+d];
    }
    f1[comp*dim+comp] -= u[Ncomp];
  }
}
```
\[ \nabla \phi_i \cdot \nu_0 e^{-\beta T} \nabla \bar{u} - (\nabla \cdot \phi) p \]
Finite Element Integration

Physics code

\[ \nabla \phi_i \cdot \nu_0 e^{-\beta T} \nabla \vec{u} - (\nabla \cdot \phi) \rho \]

```c
void f1(PetscScalar u[], const PetscScalar gradU[], PetscScalar f1[]) {
    const PetscInt dim = SPATIAL_DIM_0;
    const PetscInt Ncomp = NUM_BASIS_COMPONENTS_0;
    PetscInt comp, d;

    for (comp = 0; comp < Ncomp; ++comp) {
        for (d = 0; d < dim; ++d) {
            f1[comp*dim+d] = nu_0*exp(-beta*u[Ncomp+1])*gradU[comp*dim+d];
        }
        f1[comp*dim+comp] -= u[Ncomp];
    }
}
```
Finite Element Integration

Why Not Quadrature?

Vectorization is a Problem

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Vectorization is a Problem

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorize over Quad Points</td>
<td>Reduction needed to compute Basis Coefficients</td>
</tr>
</tbody>
</table>
## Why Not Quadrature?

Vectorization is a Problem

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorize over Quad Points</td>
<td>Reduction needed to compute Basis Coefficients</td>
</tr>
<tr>
<td>Vectorize over Basis Coef for each Quad Point</td>
<td>Too many passes through global memory</td>
</tr>
</tbody>
</table>
## Vectorization is a Problem

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorize over Quad Points</td>
<td>Reduction needed to compute Basis Coefficients</td>
</tr>
<tr>
<td>Vectorize over Basis Coef for each Quad Point</td>
<td>Too many passes through global memory</td>
</tr>
<tr>
<td>Vectorize over Basis Coef and Quad Points</td>
<td>Some threads idle when sizes are different</td>
</tr>
</tbody>
</table>
Thread Transposition

Map values at quadrature points to coefficients

Evaluate basis and process values at quadrature points

Continue with kernel
Basis Phase

\[ N_{bc} = 12 \]

\[ N_t = 24 \]

\[ N_{bl} = 2 \]

\[ N_{sbc} = 3 \]

Quadrature Phase

\[ N_{sqc} = 2 \]

\[ N_t = 24 \]

\[ N_{bl} = 2 \]

\[ N_{bs} = 6 \]
PETSc FEM Organization

GPU evaluation is **transparent** to the user:

<table>
<thead>
<tr>
<th>User Input</th>
<th>Automation</th>
<th>Solver Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>Triangle/TetGen</td>
<td>Mesh</td>
</tr>
<tr>
<td>element</td>
<td>FIAT</td>
<td>Tabulation</td>
</tr>
<tr>
<td>$f_n$</td>
<td>Generic Evaluation</td>
<td>Residual</td>
</tr>
</tbody>
</table>

- Loops are done in batches
- Remainder cells are integrated on the CPU
- PETSc **ex52** is a single-field example
PETSc FEM Organization

GPU evaluation is **transparent** to the user:

<table>
<thead>
<tr>
<th>User Input</th>
<th>Automation</th>
<th>Solver Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>Triangle/TetGen</td>
<td>Mesh</td>
</tr>
<tr>
<td>element</td>
<td>FIAT</td>
<td>Tabulation</td>
</tr>
<tr>
<td>$f_n$</td>
<td>Generic Evaluation</td>
<td>Residual</td>
</tr>
</tbody>
</table>

- Loops are done in **batches**
- Remainder cells are integrated on the CPU
- PETSc **ex52** is a single-field example
PETSc Multiphysics

Each block of the Jacobian is evaluated separately:

- Reuse single-field code
- Vectorize over cells, rather than fields
- Retain sparsity of the Jacobian

Solver integration is seamless:

- Nested Block preconditioners from the command line
- Segregated KKT MG smoothers from the command line
- Fully composable with AMG, LU, Schur complement, etc.

PETSc ex62 solves the Stokes problem, and ex31 adds temperature
Each block of the Jacobian is evaluated separately:
- Reuse single-field code
- Vectorize over cells, rather than fields
- Retain sparsity of the Jacobian

Solver integration is seamless:
- Nested Block preconditioners from the command line
- Segregated KKT MG smoothers from the command line
- Fully composable with AMG, LU, Schur complement, etc.

PETSc ex62 solves the Stokes problem, and ex31 adds temperature
PETSc Multiphysics

Each block of the Jacobian is evaluated separately:

- Reuse single-field code
- Vectorize over cells, rather than fields
- Retain sparsity of the Jacobian

Solver integration is seamless:

- Nested Block preconditioners from the command line
- Segregated KKT MG smoothers from the command line
- Fully composable with AMG, LU, Schur complement, etc.

PETSc ex62 solves the Stokes problem, and ex31 adds temperature
FEM Integration, at the element level, is also limited by memory bandwidth, rather than by peak flop rate.

- We expect bandwidth ratio speedup (3x–6x for most systems)
- Input for FEM is a vector of coefficients (auxiliary fields)
- Output is a vector of coefficients for the residual
2D $P_1$ Laplacian Performance

Reaches **100 GF/s by 100K elements**
Finite Element Integration

2D $P_1$ Laplacian Performance

Performance on SNES Example 52

- Linear scaling for both GPU and CPU integration

M. Knepley (UC)
2D $P_1$ Rate-of-Strain Performance

Reaches 100 GF/s by 100K elements
Outline

1. Why Scientific Libraries?
2. Linear Systems are Easy
3. Finite Element Integration
4. Future Direction
How should kernels be integrated into libraries?

**CUDA+Code Generation**
- Explicit vectorization
- Can inspect/optimize code
- Errors easily localized
- Can use high-level reasoning for optimization (FERari)
- Kernel fusion is easy

**TBB+C++ Templates**
- Implicit vectorization
- Generated code is hidden
- Notoriously difficult debugging
- Low-level compiler-type optimization
- Kernel fusion is really hard
How should kernels be integrated into libraries?

CUDA+Code Generation
- Explicit vectorization
- Can inspect/optimize code
- Errors easily localized
- Can use high-level reasoning for optimization (FERari)
- Kernel fusion is easy

TBB+C++ Templates
- Implicit vectorization
- Generated code is hidden
- Notoriously difficult debugging
- Low-level compiler-type optimization
- Kernel fusion is really hard
How should kernels be integrated into libraries?

CUDA+Code Generation
- Explicit vectorization
- Can inspect/optimize code
- Errors easily localized
- Can use high-level reasoning for optimization (FERari)
- Kernel fusion is easy

TBB+C++ Templates
- Implicit vectorization
- Generated code is hidden
- Notoriously difficult debugging
- Low-level compiler-type optimization
- Kernel fusion is really hard
How should kernels be integrated into libraries?

**CUDA + Code Generation**
- Explicit vectorization
- Can inspect/optimize code
- Errors easily localized
- Can use high-level reasoning for optimization (FERari)
- Kernel fusion is easy

**TBB + C++ Templates**
- Implicit vectorization
- Generated code is hidden
- Notoriously difficult debugging
- Low-level compiler-type optimization
- Kernel fusion is really hard
How should kernels be integrated into libraries?

CUDA+Code Generation
- Explicit vectorization
- Can inspect/optimize code
- Errors easily localized
- Can use high-level reasoning for optimization (FERari)
- Kernel fusion is easy

TBB+C++ Templates
- Implicit vectorization
- Generated code is hidden
- Notoriously difficult debugging
- Low-level compiler-type optimization
- Kernel fusion is really hard