Incorporation of Multicore FEM Integration Routines into Scientific Libraries

Matthew Knepley

Computation Institute
University of Chicago

Department of Molecular Biology and Physiology
Rush University Medical Center

SIAM Annual Meeting 2012
Minneapolis, MN July 9–13, 2012

Collaborators

Andy R. Terrel

- Andreas Klöckner
- Jed Brown
- Robert Kirby

Main Point

To be widely accepted,

GPU computing must be transparent to the user,

and reuse existing infrastructure.

Main Point

To be widely accepted,

GPU computing must be
transparent to the user,

and reuse existing
infrastructure.

Main Point

To be widely accepted,

GPU computing must be transparent to the user,

and reuse existing infrastructure.

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

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

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

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 Integration

PETSc FEM Organization

GPU evaluation is transparent to the user:

User Input		**Automation**		**Solver Input**
domain	==	Triangle/TetGen	==>	Mesh
element	==	FIAT	==>	Tabulation
f_n	==	Generic Evaluation	==>	Residual

- User provides point-wise physics functions
- Loops are done in batches, remainder cells handled by CPU
- One batch integration method with compile-time sizes
    - CPU, multicore CPU, MIC, GPU, etc.
- PETSc ex52 is a single-field example

PETSc FEM Organization

GPU evaluation is transparent to the user:

User Input		Automation		Solver Input
domain	==	Triangle/TetGen	==>	Mesh
element	==	FIAT	==>	Tabulation
f_n	==	Generic Evaluation	==>	Residual

- User provides point-wise physics functions
- Loops are done in batches, remainder cells handled by CPU
- One batch integration method with compile-time sizes
    - CPU, multicore CPU, MIC, GPU, etc.
- PETSc ex52 is a single-field example

PETSc FEM Organization

GPU evaluation is transparent to the user:

User Input		Automation		Solver Input
domain	==	Triangle/TetGen	==>	Mesh
element	==	FIAT	==>	Tabulation
f_n	==	Generic Evaluation	==>	Residual

- User provides point-wise physics functions
- Loops are done in batches, remainder cells handled by CPU
- One batch integration method with compile-time sizes
  - CPU, multicore CPU, MIC, GPU, etc.
- PETSc ex52 is a single-field example

PETSc FEM Organization

GPU evaluation is transparent to the user:

User Input		Automation		Solver Input
domain	==	Triangle/TetGen	==>	Mesh
element	==	FIAT	==>	Tabulation
f_n	==	Generic Evaluation	==>	Residual

- User provides point-wise physics functions
- Loops are done in batches, remainder cells handled by CPU
- One batch integration method with compile-time sizes
  - CPU, multicore CPU, MIC, GPU, etc.
- PETSc ex52 is a single-field example

PETSc FEM Organization

GPU evaluation is transparent to the user:

User Input		Automation		Solver Input
domain	==	Triangle/TetGen	==>	Mesh
element	==	FIAT	==>	Tabulation
f_n	==	Generic Evaluation	==>	Residual

- User provides point-wise physics functions
- Loops are done in batches, remainder cells handled by CPU
- One batch integration method with compile-time sizes
    - CPU, multicore CPU, MIC, GPU, etc.
- PETSc ex52 is a single-field example

FEM Integration Model

Proposed by Jed Brown

We consider weak forms dependent only on fields and gradients,

$$\int_\Omega \phi \cdot f_0(u, \nabla u) + \nabla\phi : \vec{f}_1(u, \nabla u) = 0. \qquad (1)$$

Discretizing we have

$$\sum_e \mathcal{E}_e^T \left[B^T W^q f_0(u^q, \nabla u^q) + \sum_k D_k^T W^q \vec{f}_1^k(u^q, \nabla u^q) \right] = 0 \qquad (2)$$

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

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

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

Physics code

$$\nabla \phi_i \cdot \nabla u$$

Physics code

$$\nabla \phi_i \cdot \nabla u$$

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

Physics code

$$\nabla\phi_i \cdot (\nabla u + \nabla u^T)$$

Physics code

$$\nabla \phi_i \cdot (\nabla u + \nabla u^T)$$

```
__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;
}
```

Physics code

$$\nabla \phi_i \cdot \nabla u + \phi_i k^2 u$$

Physics code

$$\nabla\phi_i \cdot \nabla u + \phi_i k^2 u$$

```
__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];
}
```

Physics code

$$\nabla \phi_i \cdot \nabla \vec{u} - (\nabla \cdot \phi) p$$

Physics code

$$\nabla\phi_i \cdot \nabla\vec{u} - (\nabla \cdot \phi)p$$

```
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];
  }
}
```

Physics code

$$\nabla \phi_i \cdot \nu_0 e^{-\beta T} \nabla \vec{u} - (\nabla \cdot \phi) p$$

Physics code

$$\nabla\phi_i \cdot \nu_0 e^{-\beta T} \nabla\vec{u} - (\nabla \cdot \phi) p$$

```
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];
  }
}
```

Why Not Quadrature?

Vectorization is a Problem

Strategy	**Problem**

Why Not Quadrature?

Vectorization is a Problem

Strategy	**Problem**
Vectorize over Quad Points	Reduction needed to compute Basis Coefficients

Why Not Quadrature?

Vectorization is a Problem

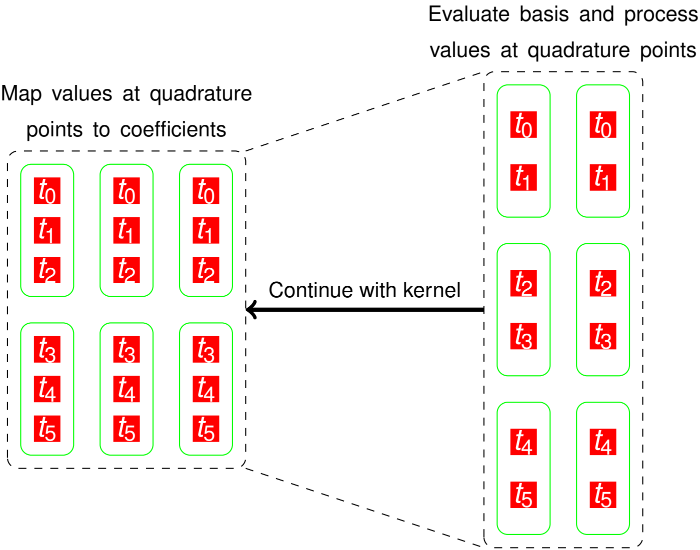
Strategy	**Problem**
Vectorize over Quad Points	Reduction needed to compute Basis Coefficients
Vectorize over Basis Coef for each Quad Point	Too many passes through global memory

Why Not Quadrature?

Vectorization is a Problem

Strategy	**Problem**
Vectorize over Quad Points	Reduction needed to compute Basis Coefficients
Vectorize over Basis Coef for each Quad Point	Too many passes through global memory
Vectorize over Basis Coef and Quad Points	Some threads idle when sizes are different

Thread Transposition



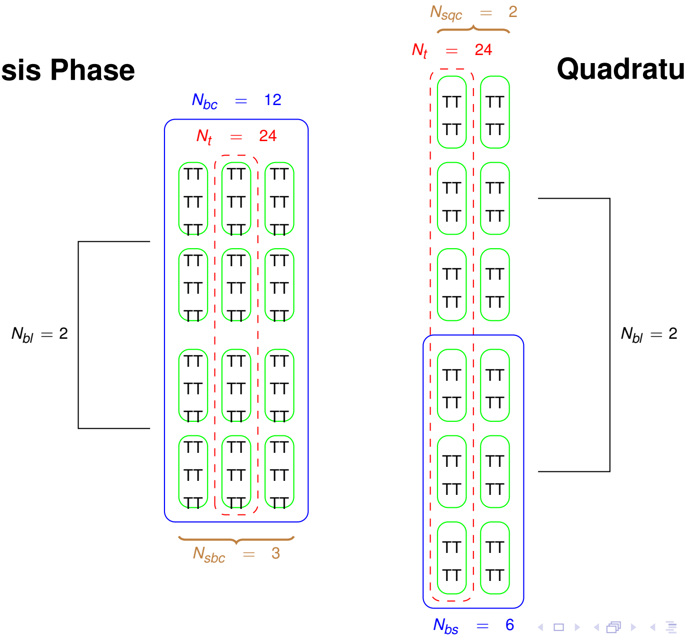
Evaluate basis and process values at quadrature points

Map values at quadrature points to coefficients

Continue with kernel

Basis Phase

Quadrature Phase



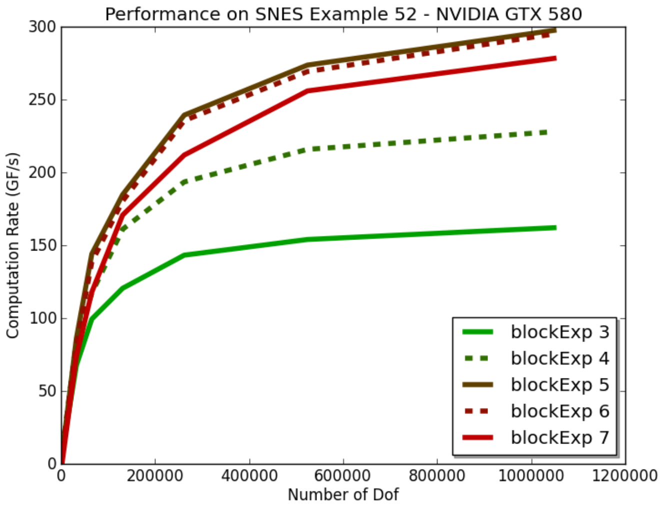
Performance Expectations

Element Integration

FEM Integration, at the element level, is also limited by memory bandwidth, rather than by peak flop rate.

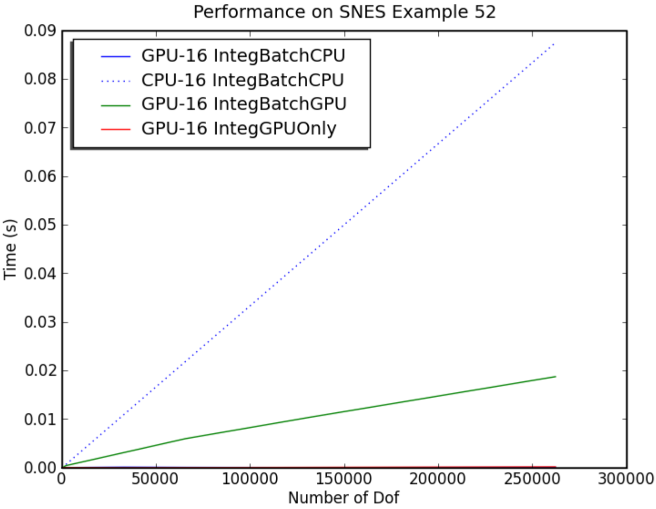
- We expect bandwidth ratio speedup (3x–6x for most systems)
- Input for FEM is a vector of coefficients (auxiliary fields)
- Output is a vector of coefficients for the residual

2D P_1 Laplacian Performance



Reaches 100 GF/s by 100K elements

2D P_1 Laplacian Performance



Linear scaling for both GPU and CPU integration

2D P_1 Laplacian Performance

Configuring PETSc

```
$PETSC_DIR/configure
    –download-triangle –download-chaco
    –download-scientificpython –download-fiat –download-generator
    –with-cuda
    –with-cudac=’nvcc -m64’ –with-cuda-arch=sm_10
    –with-cusp-dir=/PETSc3/multicore/cusp
    –with-thrust-dir=/PETSc3/multicore/thrust
    –with-cuda-only
    –with-precision=single
```

2D P_1 Laplacian Performance

Configuring PETSc

```
$PETSC_DIR/configure
    –download-triangle –download-chaco
    –download-scientificpython –download-fiat –download-generator
    –with-cuda
    –with-cudac='nvcc -m64' –with-cuda-arch=sm_10
    –with-cusp-dir=/PETSc3/multicore/cusp
    –with-thrust-dir=/PETSc3/multicore/thrust
    –with-cuda-only
    –with-precision=single
```

2D P_1 Laplacian Performance

Configuring PETSc

```
$PETSC_DIR/configure
    –download-triangle –download-chaco
    –download-scientificpython –download-fiat –download-generator
    –with-cuda
    –with-cudac='nvcc -m64' –with-cuda-arch=sm_10
    –with-cusp-dir=/PETSc3/multicore/cusp
    –with-thrust-dir=/PETSc3/multicore/thrust
    –with-cuda-only
    –with-precision=single
```

2D P_1 Laplacian Performance

Configuring PETSc

```
$PETSC_DIR/configure
    –download-triangle –download-chaco
    –download-scientificpython –download-fiat –download-generator
    –with-cuda
    –with-cudac=’nvcc -m64’ –with-cuda-arch=sm_10
    –with-cusp-dir=/PETSc3/multicore/cusp
    –with-thrust-dir=/PETSc3/multicore/thrust
    –with-cuda-only
    –with-precision=single
```

2D P_1 Laplacian Performance

Configuring PETSc

```
$PETSC_DIR/configure
    –download-triangle –download-chaco
    –download-scientificpython –download-fiat –download-generator
    –with-cuda
    –with-cudac='nvcc -m64' –with-cuda-arch=sm_10
    –with-cusp-dir=/PETSc3/multicore/cusp
    –with-thrust-dir=/PETSc3/multicore/thrust
    –with-cuda-only
    –with-precision=single
```

2D P_1 Laplacian Performance

Configuring PETSc

```
$PETSC_DIR/configure
    –download-triangle –download-chaco
    –download-scientificpython –download-fiat –download-generator
    –with-cuda
    –with-cudac='nvcc -m64' –with-cuda-arch=sm_10
    –with-cusp-dir=/PETSc3/multicore/cusp
    –with-thrust-dir=/PETSc3/multicore/thrust
    –with-cuda-only
    –with-precision=single
```

2D P_1 Laplacian Performance

Configuring PETSc

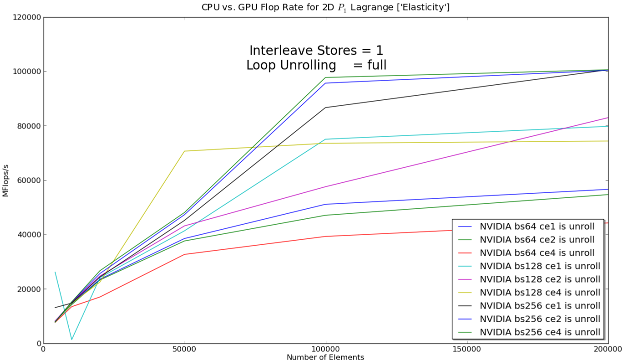
```
$PETSC_DIR/configure
    –download-triangle –download-chaco
    –download-scientificpython –download-fiat –download-generator
    –with-cuda
    –with-cudac=’nvcc -m64’ –with-cuda-arch=sm_10
    –with-cusp-dir=/PETSc3/multicore/cusp
    –with-thrust-dir=/PETSc3/multicore/thrust
    –with-cuda-only
    –with-precision=single
```

2D P_1 Laplacian Performance

Running the example

```
$PETSC_DIR/src/benchmarks/benchmarkExample.py
    --daemon --num 52 DMComplex
    --events IntegBatchCPU IntegBatchGPU IntegGPUOnly
    --refine 0.0625 0.00625 0.000625 0.0000625 0.00003125
    0.000015625 0.0000078125 0.00000390625
    --order=1 --blockExp 4
    CPU=’dm_view show_residual=0 compute_function batch’
    GPU=’dm_view show_residual=0 compute_function batch gpu
    gpu_batches=8’
```

2D P_1 Rate-of-Strain Performance



Reaches 100 GF/s by 100K elements

2D P_1 Rate-of-Strain Performance

Running the example

```
$PETSC_DIR/src/benchmarks/benchmarkExample.py
    --daemon --num 52 DMComplex
    --events IntegBatchCPU IntegBatchGPU IntegGPUOnly
    --refine 0.0625 0.00625 0.000625 0.0000625 0.00003125
    0.000015625 0.0000078125 0.00000390625
    --operator=elasticity --order=1 --blockExp 4
    CPU=’dm_view op_type=elasticity show_residual=0
    compute_function batch’
    GPU=’dm_view op_type=elasticity show_residual=0
    compute_function batch gpu gpu_batches=8’
```

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

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

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

Competing Models

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

Competing Models

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

Competing Models

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

Competing Models

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

Competing Models

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