Low Order Finite Elements on the GPU

Matthew Knepley

Computation Institute
University of Chicago

Department of Molecular Biology and Physiology
Rush University Medical Center

Large-Scale Geosciences Applications
Using GPU and Multicore Architectures
San Francisco, December 16, 2010

Collaborators

- Dr. Andy Terrel (FEniCS)
    - Dept. of Computer Science, University of Texas
    - Texas Advanced Computing Center, University of Texas
- Prof. Andreas Klöckner (PyCUDA)
    - Courant Institute of Mathematical Sciences, New York University
- Dr. Brad Aagaard (PyLith)
    - United States Geological Survey, Menlo Park, CA
- Dr. Charles Williams (PyLith)
    - GNS Science, Wellington, NZ

Other Software

- High Order, Discontinuous Galerkin FEM
  - Hedge, Andreas Klöckner
- Cartesian, Finite Difference Multigrid
  - OpenCurrent, Jon Cohen
- Fast Multipole Method
  - PetFMM, Lorena Barba, Felipe Cruz, Matthew Knepley
- Parallel Linear Algebra and Solvers
  - PETSc, Barry Smith, et.al.
  - Cusp, Nathan Bell, et.al.
  - CUSPARSE, NVIDIA

What are the Benefits for current PDE Code?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem

What are the Benefits for current PDE Code?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem

What are the Benefits for current PDE Code?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem

What are the Benefits for current PDE Code?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem

Outline

Analytic Flexibility

Laplacian

$$\int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) d\mathbf{x} \tag{1}$$

```
element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u))*dx
```

Analytic Flexibility

Laplacian

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{1}$$

```
element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u))*dx
```

Analytic Flexibility

Linear Elasticity

$$\frac{1}{4}\int_{\mathcal{T}} \left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right) dx \qquad (2)$$

```
element = VectorElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(sym(grad(v)), sym(grad(u)))*dx
```

Analytic Flexibility

Linear Elasticity

$$\frac{1}{4}\int_{\mathcal{T}} \left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right) d\mathbf{x} \tag{2}$$

```
element = VectorElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(sym(grad(v)), sym(grad(u)))*dx
```

Analytic Flexibility

Full Elasticity

$$\frac{1}{4}\int_{\mathcal{T}} \left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : C : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right) d\mathbf{x} \tag{3}$$

```
element  = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                         (dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release

Analytic Flexibility

Full Elasticity

$$\frac{1}{4}\int_{\mathcal{T}} \left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : C : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right) d\mathbf{x} \qquad (3)$$

```
element  = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                         (dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release

Analytic Flexibility

Full Elasticity

$$\frac{1}{4}\int_{\mathcal{T}}\left(\nabla\vec{\phi}_i(\mathbf{x})+\nabla^T\vec{\phi}_i(\mathbf{x})\right):C:\left(\nabla\vec{\phi}_j(\mathbf{x})+\nabla\vec{\phi}_j(\mathbf{x})\right)dx \qquad (3)$$

```
element  = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                         (dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release

Form Decomposition

Element integrals are decomposed into analytic and geometric parts:

$$
\begin{aligned}
& \int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} && (4)\\
= & \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} && (5)\\
= & \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| d\mathbf{x} && (6)\\
= & \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} d\mathbf{x} && (7)\\
= & G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma} && (8)
\end{aligned}
$$

Coefficients are also put into the geometric part.

Weak Form Processing

```
from ffc.analysis import analyze_forms
from ffc.compiler import compute_ir

parameters = ffc.default_parameters()
parameters['representation'] = 'tensor'
analysis = analyze_forms([a,L], {}, parameters)
ir = compute_ir(analysis, parameters)

a_K = ir[2][0]['AK'][0][0]
a_G = ir[2][0]['AK'][0][1]

K = a_K.A0.astype(numpy.float32)
G = a_G
```

Outline

Computational Flexibility

We generate different computations on the fly,

and can change

- Element Batch Size
- Number of Concurrent Elements
- Loop unrolling
- Interleaving stores with computation

Computational Flexibility

Basic Contraction

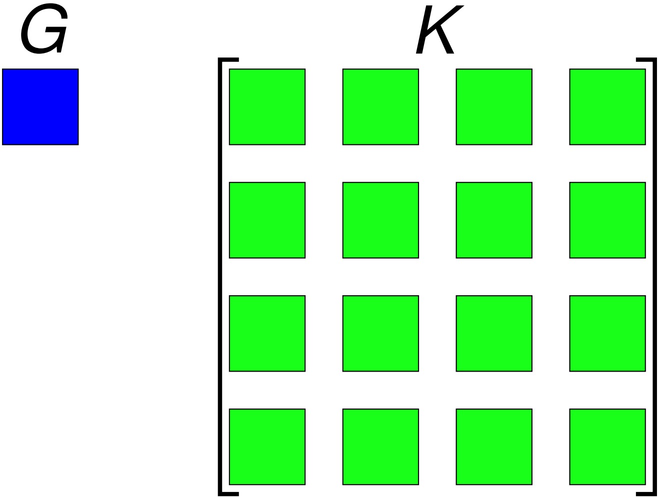


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Basic Contraction

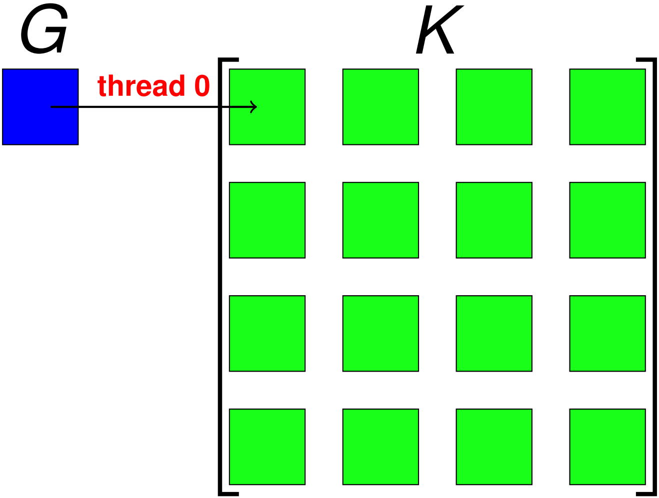


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Basic Contraction

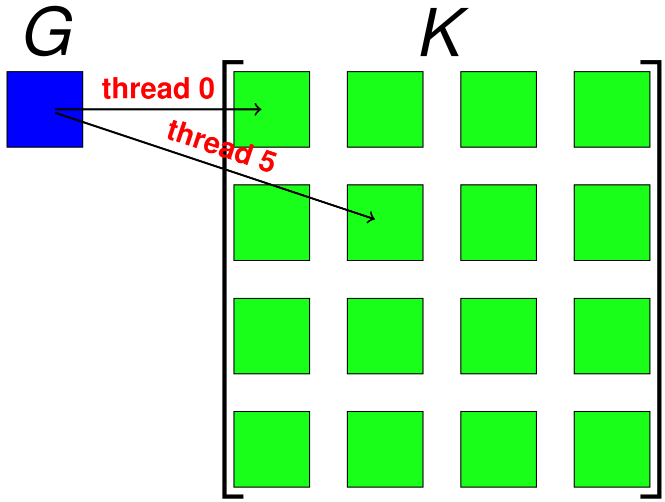


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Basic Contraction

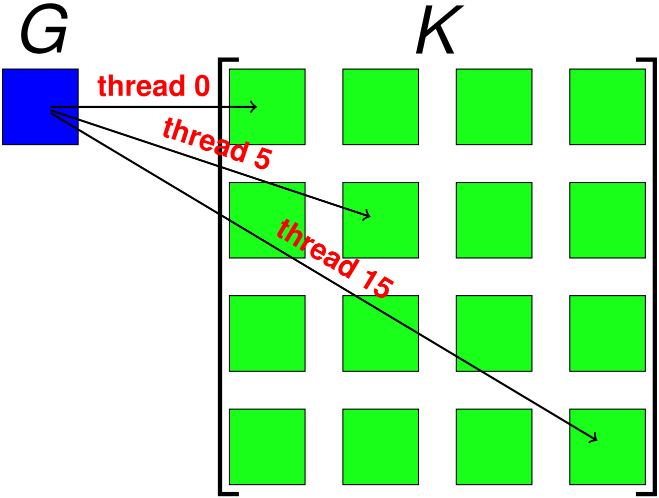


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

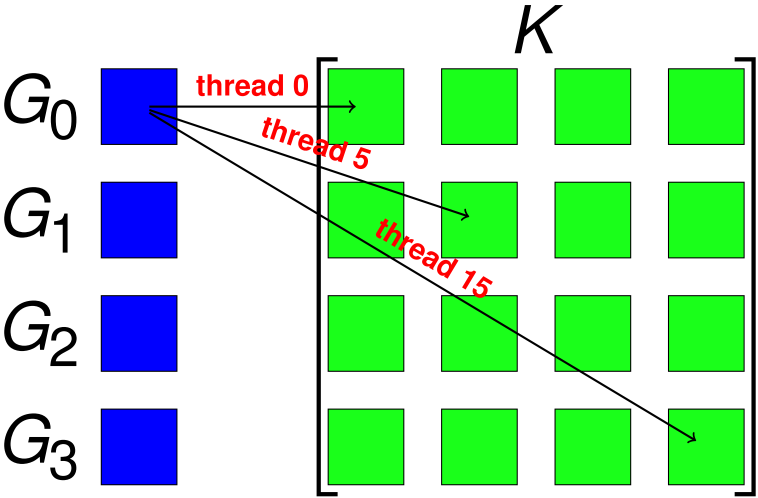


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

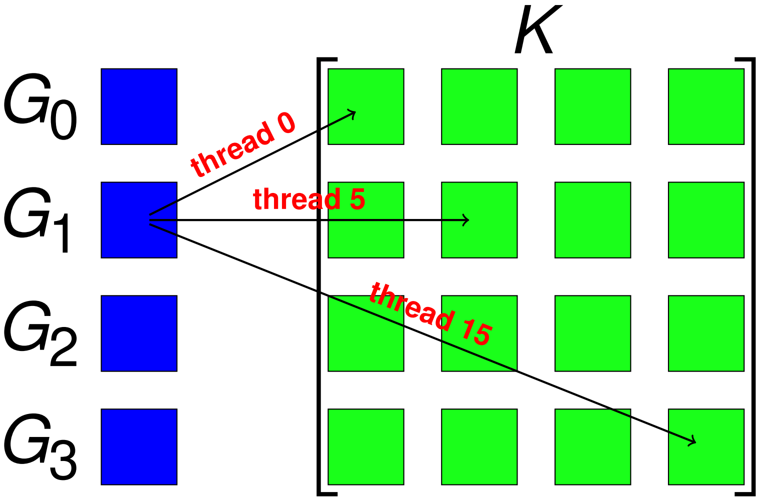


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

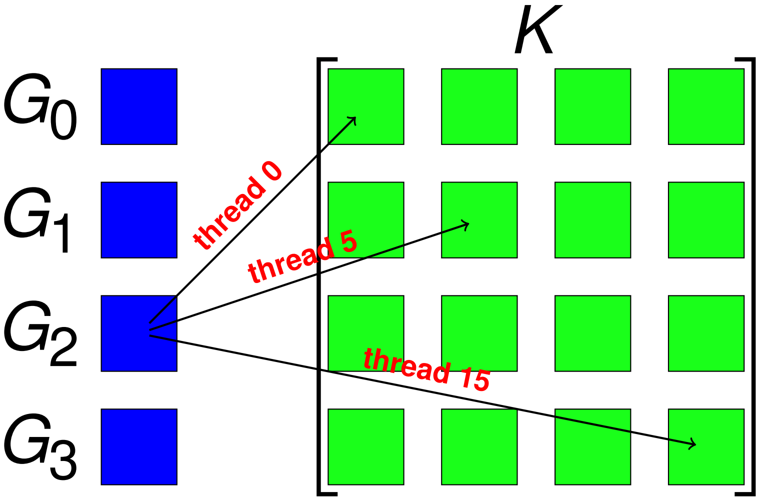


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

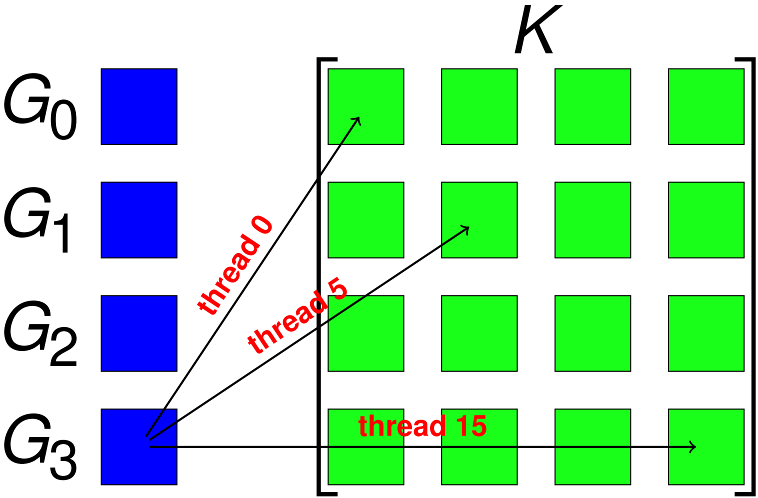
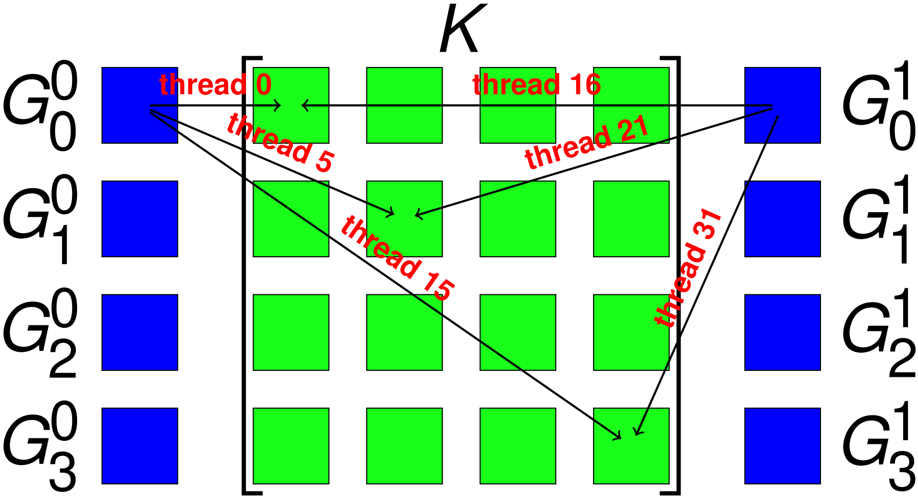


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

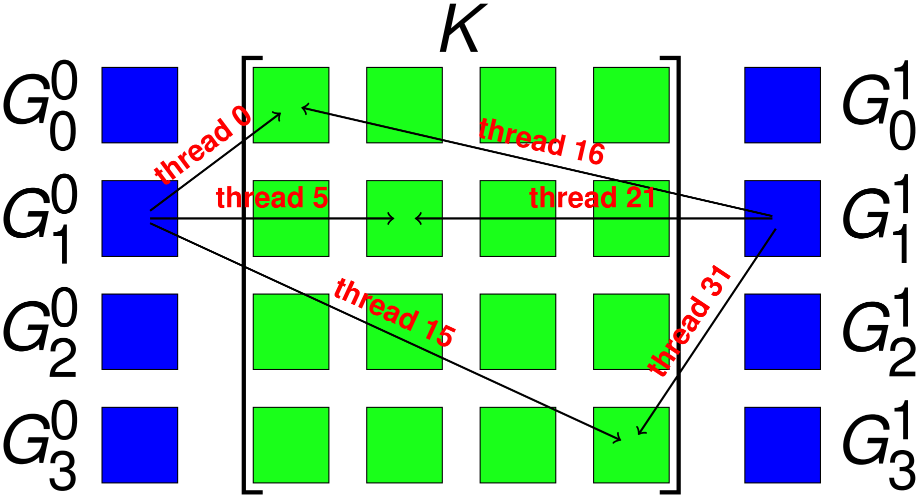
Computational Flexibility

Concurrent Elements



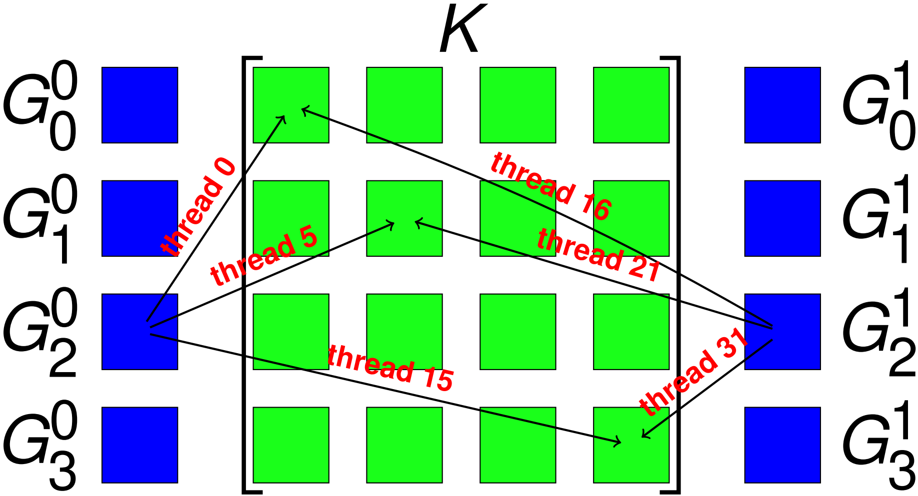
Computational Flexibility

Concurrent Elements



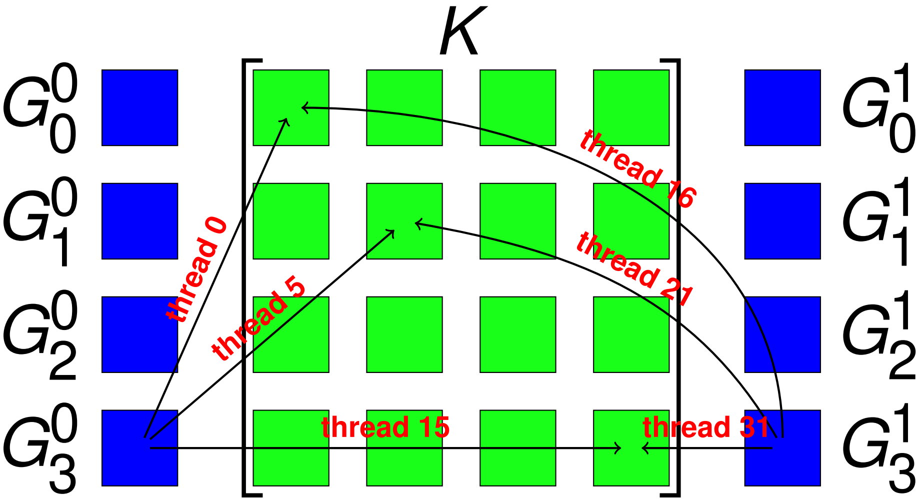
Computational Flexibility

Concurrent Elements



Computational Flexibility

Concurrent Elements



Computational Flexibility

Loop Unrolling

```
/* G K contraction: unroll = full */
E[0] += G[0] * K[0];
E[0] += G[1] * K[1];
E[0] += G[2] * K[2];
E[0] += G[3] * K[3];
E[0] += G[4] * K[4];
E[0] += G[5] * K[5];
E[0] += G[6] * K[6];
E[0] += G[7] * K[7];
E[0] += G[8] * K[8];
```

Computational Flexibility

Loop Unrolling

```
/* G K contraction: unroll = none */
for(int b = 0; b < 1; ++b) {
  const int n = b*1;
  for(int alpha = 0; alpha < 3; ++alpha) {
    for(int beta = 0; beta < 3; ++beta) {
      E[b] += G[n*9+alpha*3+beta] * K[alpha*3+beta];
    }
  }
}
```

Computational Flexibility

Interleaving stores

```
/* G K contraction: unroll = none */
for(int b = 0; b < 4; ++b) {
  const int n = b*1;
  for(int alpha = 0; alpha < 3; ++alpha) {
    for(int beta = 0; beta < 3; ++beta) {
      E[b] += G[n*9+alpha*3+beta] * K[alpha*3+beta];
    }
  }
}
/* Store contraction results */
elemMat[Eoffset+idx+0]  = E[0];
elemMat[Eoffset+idx+16] = E[1];
elemMat[Eoffset+idx+32] = E[2];
elemMat[Eoffset+idx+48] = E[3];
```

Computational Flexibility

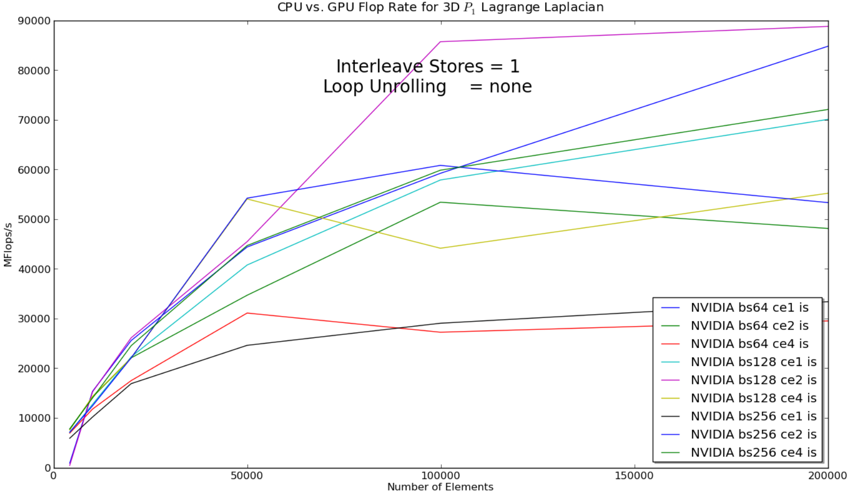
Interleaving stores

```
n = 0;
for(int alpha = 0; alpha < 3; ++alpha) {
  for(int beta = 0; beta < 3; ++beta) {
    E += G[n*9+alpha*3+beta] * K[alpha*3+beta];
  }
}
/* Store contraction result */
elemMat[Eoffset+idx+0] = E;
n = 1; E = 0.0; /* contract */
elemMat[Eoffset+idx+16] = E;
n = 2; E = 0.0; /* contract */
elemMat[Eoffset+idx+32] = E;
n = 3; E = 0.0; /* contract */
elemMat[Eoffset+idx+48] = E;
```

Outline

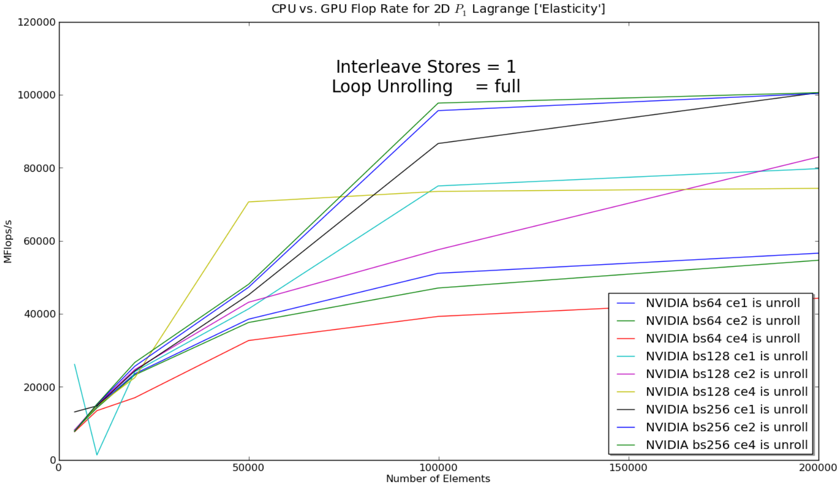
Performance

Influence of Element Batch Sizes



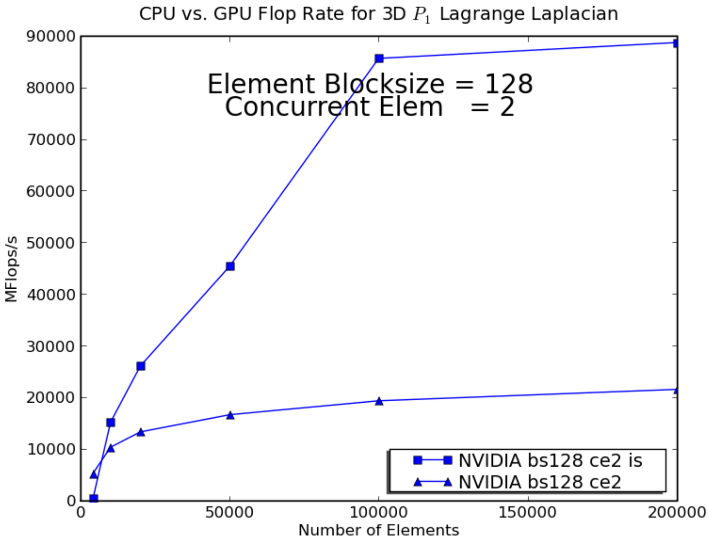
Performance

Influence of Element Batch Sizes



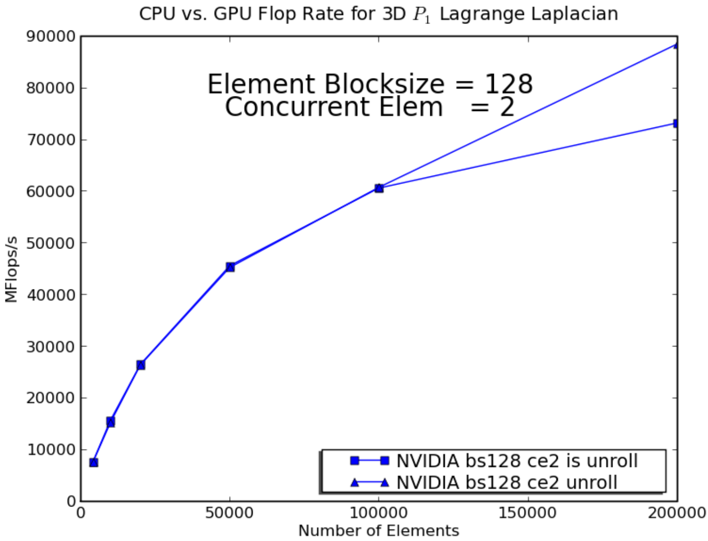
Performance

Influence of Code Structure



Performance

Influence of Code Structure



Performance

Price-Performance Comparison of CPU and GPU
3D P_1 Laplacian Integration

Model	Price ($)	GF/s	MF/s$
GTX285	390	90	231
Core 2 Duo	300	2	6.6

Performance

Price-Performance Comparison of CPU and GPU
3D P_1 Laplacian Integration

Model	Price ($)	GF/s	MF/s$
GTX285	390	90	231
Core 2 Duo	300	12*	40

* Jed Brown Optimization Engine

Why Should You Try This?

Many Codes Today use Low Order FEM,
GPUs can Help

- Analytic Flexibility
- Computational Flexibility
- Efficiency

Extension to Quadrature

Formulation due to Jed Brown

Add additional contraction over quadrature points:

$$\int_\Omega \phi \cdot f_0(u, \nabla u) + \nabla\phi : f_1(u, \nabla u) = 0 \tag{9}$$

$$\sum_e \mathcal{E}_e^T \left[B^T W^q f_0(u^q, \nabla u^q) + \sum_k D_k^T W^q f_1^k(u^q, \nabla u^q) \right] = 0 \tag{10}$$

Single thread computes quadrature loops to avoid reductions,
just like contractions