Software Design for PDEs on GPUs

Matthew Knepley

Computation Institute
University of Chicago

Department of Molecular Biology and Physiology
Rush University Medical Center

Conference on Simulation and Optimization
In Honor of Gisbert Stoyan
Győr, Hungary June 29–July 1, 2011

Collaborators

Chicago Automated Scientific Computing Group:

- Prof. Ridgway Scott
  - Dept. of Computer Science, University of Chicago
  - Dept. of Mathematics, University of Chicago
- Peter Brune, (biological DFT)
  - Dept. of Computer Science, University of Chicago
- Dr. Andy Terrel, (Rheagen)
  - Dept. of Computer Science and TACC, University of Texas at Austin

Collaborators

The PetscGPU team:

- Dr. Barry Smith
  - Mathematics and Computer Science Division, ANL
- Satish Balay
  - Mathematics and Computer Science Division, ANL
- Victor Minden
  - Dept. of Mathematics, Tufts University

Collaborators

The PyLith Team:

- Dr. Brad Aagaard (PyLith)
  - United States Geological Survey, Menlo Park, CA
- Dr. Charles Williams (PyLith)
  - GNS Science, Wellington, NZ

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.

To be widely accepted,

GPU computing must be transparent to the user,

and reuse existing infrastructure.

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

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

Outline

Thrust

Thrust is a CUDA library of parallel algorithms

- Interface similar to C++ Standard Template Library
- Containers (`vector`) on both host and device
- Algorithms: `sort`, `reduce`, `scan`
- Freely available, part of PETSc configure (`-with-thrust-dir`)
- Included as part of CUDA 4.0 installation

Cusp

Cusp is a CUDA library for sparse linear algebra and graph computations

- Builds on data structures in Thrust
- Provides sparse matrices in several formats (CSR, Hybrid)
- Includes some preliminary preconditioners (Jacobi, SA-AMG)
- Freely available, part of PETSc configure (`–with-cusp-dir`)

VECCUDA

Strategy: Define a new **Vec** implementation

- Uses Thrust for data storage and operations on GPU
- Supports full PETSc **Vec** interface
- Inherits PETSc scalar type
- Can be activated at runtime, `-vec_type cuda`
- PETSc provides memory coherence mechanism

Memory Coherence

PETSc Objects now hold a coherence flag

PETSC_CUDA_UNALLOCATED	No allocation on the GPU
PETSC_CUDA_GPU	Values on GPU are current
PETSC_CUDA_CPU	Values on CPU are current
PETSC_CUDA_BOTH	Values on both are current

Table: Flags used to indicate the memory state of a PETSc CUDA **Vec** object.

MATAIJCUDA

Also define new **Mat** implementations

- Uses Cusp for data storage and operations on GPU
- Supports full PETSc **Mat** interface, some ops on CPU
- Can be activated at runtime, `-mat_type aijcuda`
- Notice that parallel matvec necessitates off-GPU data transfer

Solvers

Solvers come for Free

Preliminary Implementation of PETSc Using GPU,
Minden, Smith, Knepley, 2010

- All linear algebra types work with solvers
- Entire solve can take place on the GPU
  - Only communicate scalars back to CPU
- GPU communication cost could be amortized over several solves
- Preconditioners are a problem
  - Cusp has a promising AMG

Installation

PETSc only needs

```
# Turn on CUDA
--with-cuda
# Specify the CUDA compiler
--with-cudac='nvcc -m64'
# Indicate the location of packages
#  --download-* will also work soon
--with-thrust-dir=/PETSc3/multicore/thrust
--with-cusp-dir=/PETSc3/multicore/cusp
# Can also use double precision
--with-precision=single
```

Example

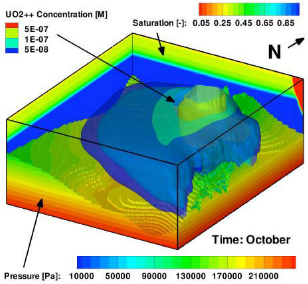
Driven Cavity Velocity-Vorticity with Multigrid

```
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
```

Example
PFLOTRAN

Flow Solver
$32 \times 32 \times 32$ grid

Routine	Time (s)	MFlops	MFlops/s
CPU			
KSPSolve	8.3167	4370	526
MatMult	1.5031	769	512
GPU			
KSPSolve	1.6382	4500	2745
MatMult	0.3554	830	2337



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

Outline

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

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

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

Analytic Flexibility
Linear Elasticity

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

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

Analytic Flexibility
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

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

```
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

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

```
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

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

```
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

Outline

Form Decomposition

Element integrals are decomposed into analytic and geometric parts:

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{4}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} \tag{5}$$

$$= \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| d\mathbf{x} \tag{6}$$

$$= \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} d\mathbf{x} \tag{7}$$

$$= G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma} \tag{8}$$

Coefficients are also put into the geometric part.

Form Decomposition

Additional fields give rise to multilinear forms.

$$\int_{\mathcal{T}} \phi_i(\mathbf{x}) \cdot \left(\phi_k(\mathbf{x}) \nabla \phi_j(\mathbf{x})\right) dA \tag{9}$$

$$= \int_{\mathcal{T}} \phi_i^\beta(\mathbf{x}) \left(\phi_k^\alpha(\mathbf{x}) \frac{\partial \phi_j^\beta(\mathbf{x})}{\partial x_\alpha} \right) dA \tag{10}$$

$$= \int_{\mathcal{T}_{\mathrm{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| dA \tag{11}$$

$$= \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} dA \tag{12}$$

$$= G^{\alpha\gamma}(\mathcal{T}) K_{\alpha\gamma}^{ijk} \tag{13}$$

The index calculus is fully developed by Kirby and Logg in A Compiler for Variational Forms.

Form Decomposition

Isoparametric Jacobians also give rise to multilinear forms

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) dA \tag{14}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} dA \tag{15}$$

$$= \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| dA \tag{16}$$

$$= |J| \int_{\mathcal{T}_{\mathrm{ref}}} \phi_k J_k^{\beta\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l J_l^{\gamma\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{17}$$

$$= J_k^{\beta\alpha} J_l^{\gamma\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \phi_k \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{18}$$

$$= G_{kl}^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ijkl} \tag{19}$$

A different space could also be used for Jacobians

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
```

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

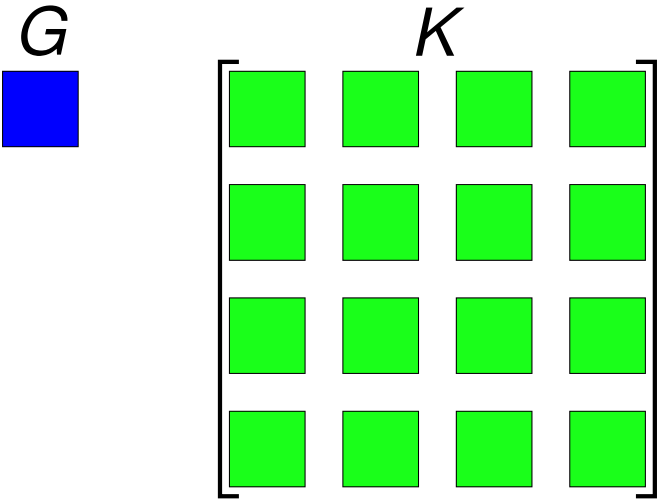


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

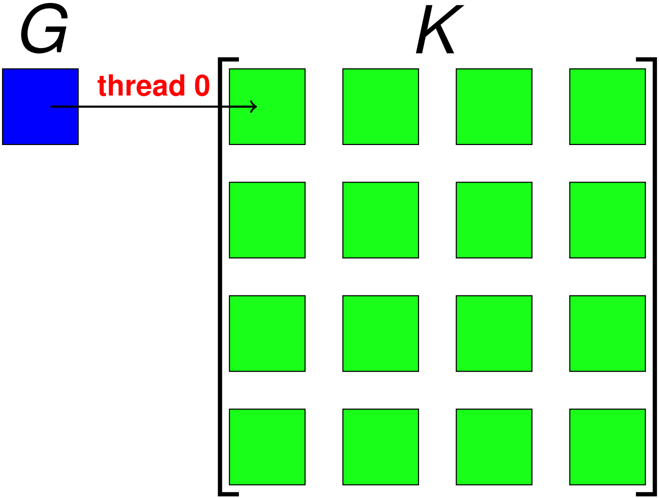


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

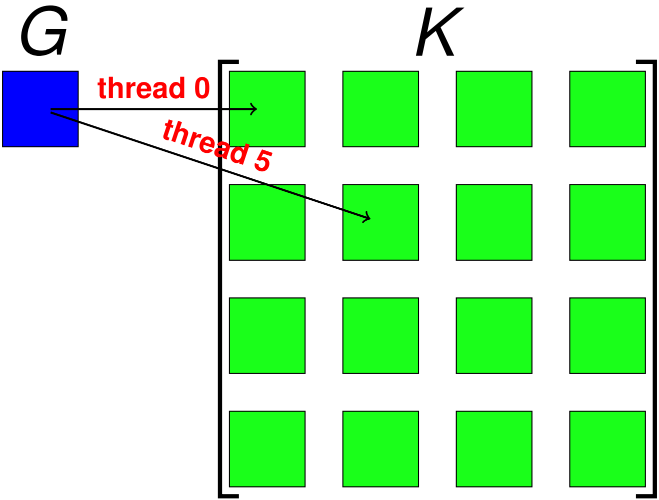


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

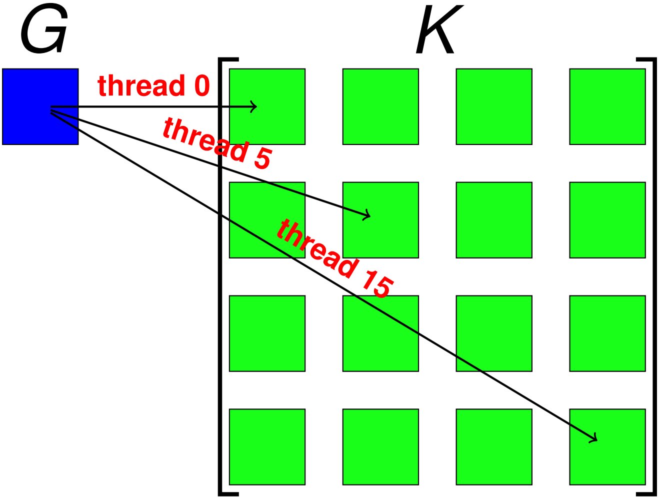


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Element Batch Size

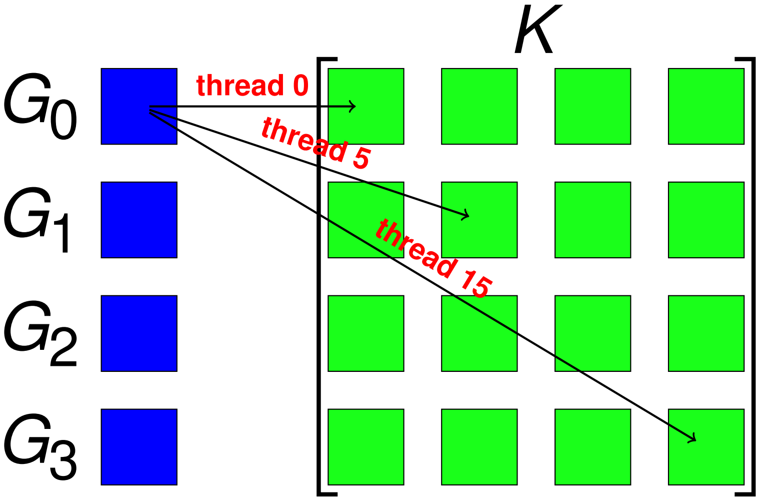


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Element Batch Size

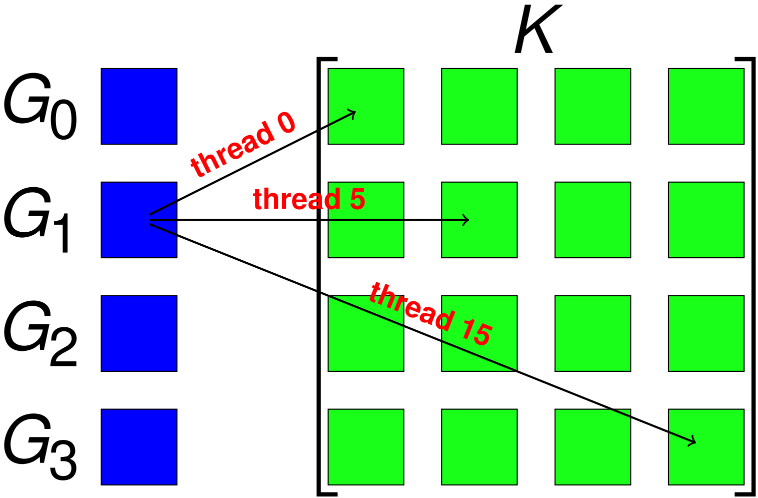


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Element Batch Size

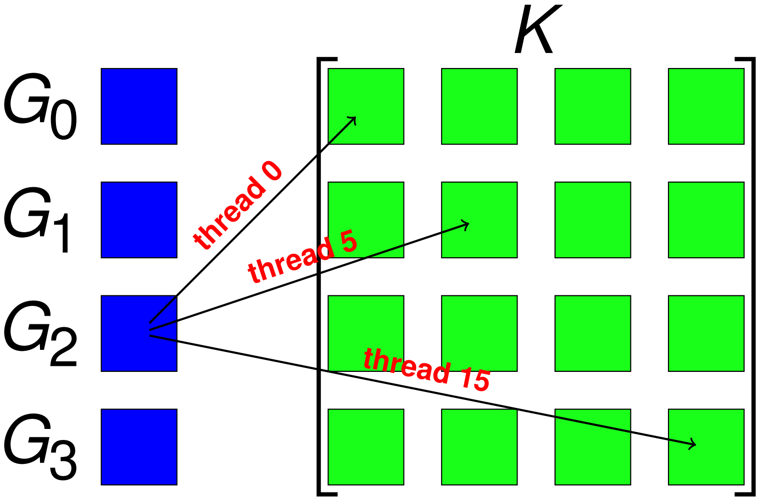


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Element Batch Size

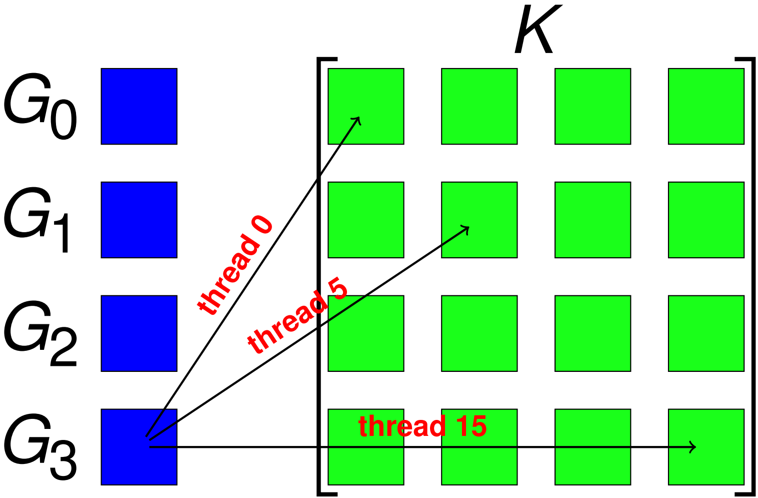
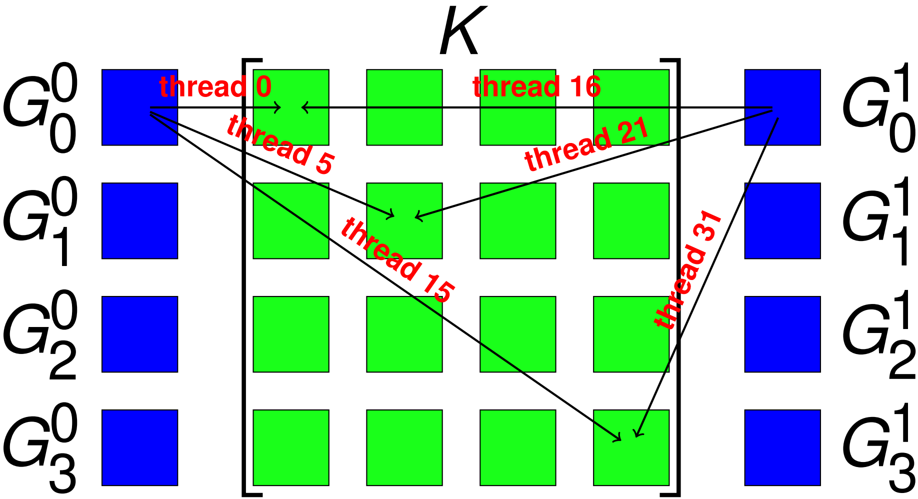
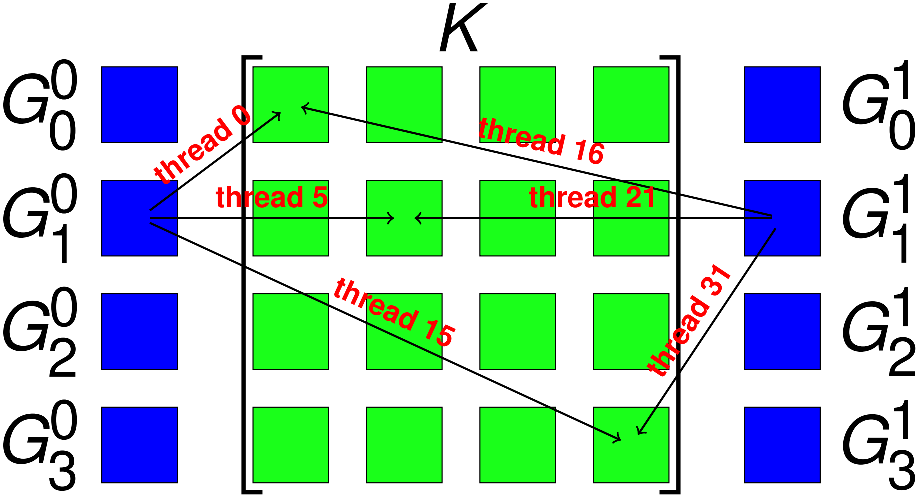


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

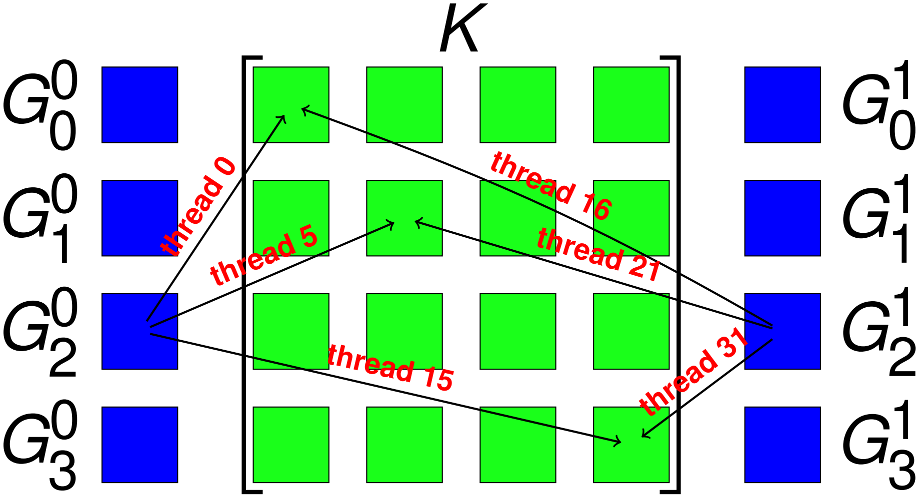
Computational Flexibility
Concurrent Elements



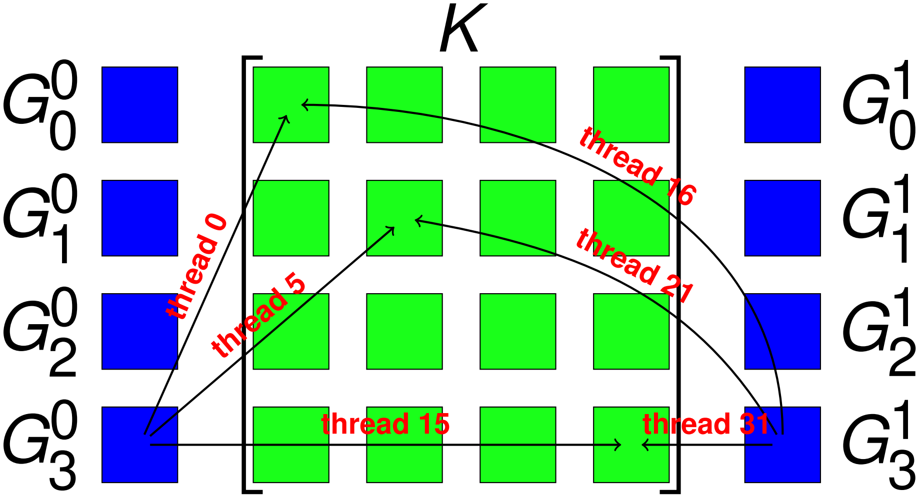
Computational Flexibility
Concurrent Elements



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

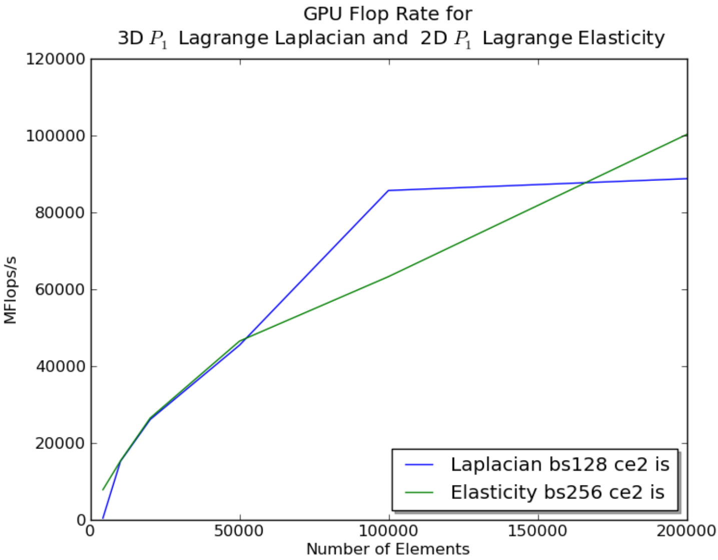
```
/* G K contraction: unroll = none */
for(int b = 0; b < 4; ++b) {
  const int n = b*1;
  for(int alpha = 0; alpha < 3; ++alpha) {
    for(int beta = 0; beta < 3; ++beta) {
      E[b] += G[n*9+alpha*3+beta] * K[alpha*3+beta];
    }
  }
}
/* Store contraction results */
elemMat[Eoffset+idx+0]  = E[0];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



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

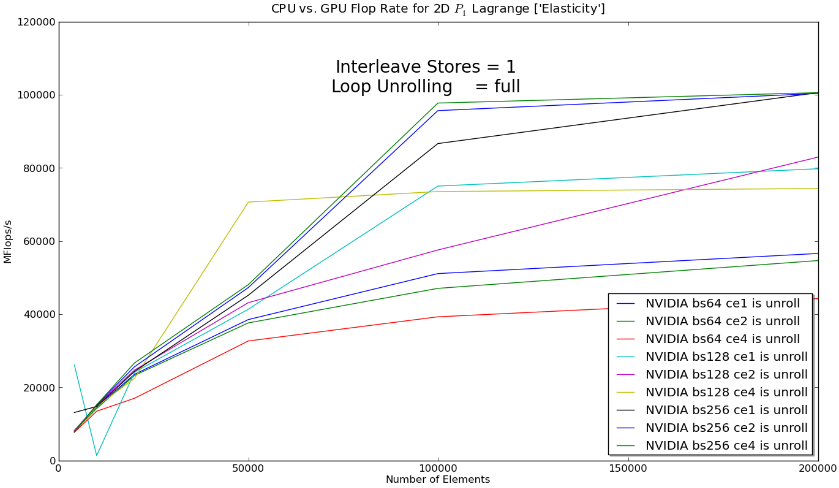
Performance

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

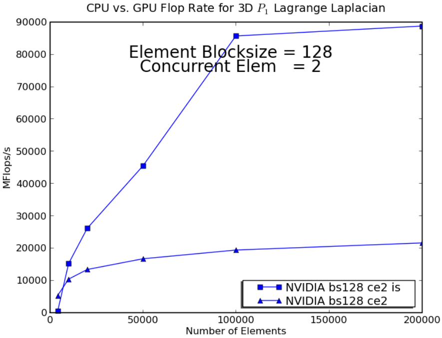
Model	Price ($)	GF/s	MF/s$
GTX285	390	90	231
Core 2 Duo	300	12*	40

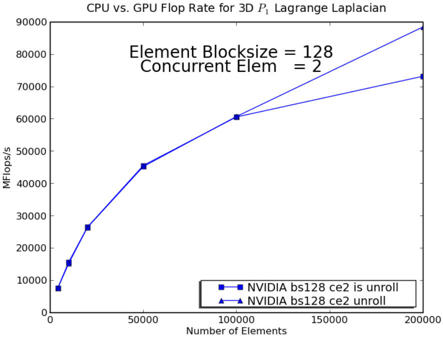
* Jed Brown Optimization Engine

Performance
Influence of Element Batch Sizes



Performance
Influence of Code Structure





Explaining performance

- Increase shared memory and work/thread until you top out
  - Occupancies go down or level out as performance goes up
- Does not work without interleaved stores
  - Scheduler can switch to kernels who are computing
  - Larger number of smaller computations makes better fit
- Should I worry about detailed explanations for performance?
  - Sensible decompositions, coupled with exploration
  - FLAME methodology

Automated Tuning System

Components of our performance evaluation system:

- Generate set of kernels using:
    - Loop slicing, store reordering, etc.
    - Loop invariants ala FLAME
    - High level constructs ala Rheagen and FEniCS
- Store results and metadata in HDF5 using PyTables
    - Thousands of tests for this talk
- Interrogate and plot with Matplotlib
- Eventually couple to build system
    - FFTW, Spiral, FLAME

Why Should You Try This?

Structured code generation,

can allow easy integration
of novel hardware

and reconcile user physics
with system traversals.

Why Should You Try This?

Structured code generation,

can allow easy integration of novel hardware

and reconcile user physics with system traversals.

Why Should You Try This?

Structured code generation,

can allow easy integration
of novel hardware

and reconcile user physics
with system traversals.