Software Design for PDEs on GPUs

Matthew Knepley

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

Advanced Algorithms on GPUs
SIAM Conference on Computational Science and Engineering
Reno, Feb. 28, 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
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
The **PyLith Team:**

- **Dr. Brad Aagaard** *(PyLith)*
  - United States Geological Survey, Menlo Park, CA

- **Dr. Charles Williams** *(PyLith)*
  - GNS Science, Wellington, NZ
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.
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

1. PETSc-GPU

2. FEM-GPU
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 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`)
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
PETSc-Objects now hold a coherence flag

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PETSC_CUDA_UNALLOCATED</td>
<td>No allocation on the GPU</td>
</tr>
<tr>
<td>PETSC_CUDA_GPU</td>
<td>Values on GPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_CPU</td>
<td>Values on CPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_BOTH</td>
<td>Values on both are current</td>
</tr>
</tbody>
</table>

**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 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
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 aijcus -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_syncrhize       # Setup run
    -log_summary
```
Outline

1. PETSc-GPU

2. FEM-GPU
   - Analytic Flexibility
   - Computational Flexibility
   - Efficiency
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
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

2 FEM-GPU
- Analytic Flexibility
- Computational Flexibility
- Efficiency
Analytic Flexibility

Laplacian

\[ \int_T \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dx \]  \tag{1}

\begin{verbatim}
  element = FiniteElement('Lagrange', tetrahedron, 1)
  v = TestFunction(element)
  u = TrialFunction(element)
  a = inner(grad(v), grad(u))*dx
\end{verbatim}
\[ \int_T \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dx \]  

```python
element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u)) * dx
```
\[ \frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : \left( \nabla \phi_j(x) + \nabla^T \phi_j(x) \right) \, dx \]  

\[
\text{element} = \text{VectorElement('Lagrange', \text{tetrahedron}, 1)} \\
v = \text{TestFunction(element)} \\
u = \text{TrialFunction(element)} \\
a = \text{inner(sym(grad(v)), sym(grad(u)))} * dx
\]
\[ \frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : \left( \nabla \phi_j(x) + \nabla^T \phi_j(x) \right) \, dx \]  

\begin{verbatim}
element = VectorElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(sym(grad(v)), sym(grad(u))) * dx
\end{verbatim}
\[
\frac{1}{4} \int_{\mathcal{T}} \left( \nabla \mathbf{\phi}_i(\mathbf{x}) + \nabla^T \mathbf{\phi}_i(\mathbf{x}) \right) : C : \left( \nabla \mathbf{\phi}_j(\mathbf{x}) + \nabla^T \mathbf{\phi}_j(\mathbf{x}) \right) \, d\mathbf{x}
\]  

(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
\[
\frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : C : \left( \nabla \phi_j(x) + \nabla \phi_j(x) \right) \, dx \tag{3}
\]

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(\text{grad}(v))[i,j] \cdot C[i,j,k,l] \cdot sym(\text{grad}(u))[k,l] \cdot dx

Currently broken in FEniCS release
\[
\frac{1}{4} \int_{\mathcal{T}} \left( \nabla \vec{\phi}_i(x) + \nabla^T \vec{\phi}_i(x) \right) : \mathbf{C} : \left( \nabla \vec{\phi}_j(x) + \nabla \vec{\phi}_j(x) \right) \, dx
\] (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

2. FEM-GPU
   - Analytic Flexibility
   - Computational Flexibility
   - Efficiency
Element integrals are decomposed into **analytic** and **geometric** parts:

\[
\int_{\mathcal{T}} \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dx \\
= \int_{\mathcal{T}} \frac{\partial \phi_i(x)}{\partial x_\alpha} \frac{\partial \phi_j(x)}{\partial x_\alpha} \, dx \\
= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |J| \, dx \\
= \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} \, dx \\
= G^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ij}
\]

Coefficients are also put into the geometric part.
Form Decomposition

Additional fields give rise to multilinear forms.

\[
\int_{\mathcal{T}} \phi_i(x) \cdot (\phi_k(x) \nabla \phi_j(x)) \, dA
\]

\[
= \int_{\mathcal{T}} \phi_i^\beta(x) \left( \phi_k^\alpha(x) \frac{\partial \phi_j^\beta(x)}{\partial x_\alpha} \right) \, dA
\]

\[
= \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| \, dA
\]

\[
= \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} \, dA
\]

\[
= G^{\alpha\gamma}(T) K_{\alpha\gamma}^{ijk}
\]

The index calculus is fully developed by Kirby and Logg in

A Compiler for Variational Forms.
Isoparametric Jacobians also give rise to multilinear forms

\[
\int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) \, dA = \int_{\mathcal{T}} \frac{\partial \phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial \phi_j(\mathbf{x})}{\partial x_\alpha} \, dA = \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \phi_i(\xi)}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial x_\alpha} \left| J \right| \, dA = |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k J^\beta_\alpha \phi_i \phi_j \, dA = J^\beta_\alpha J^\gamma_\alpha |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k \phi_l \, dA = G^{\beta\gamma}_{kl} (\mathcal{T}) K^{ijkl}_{\beta\gamma}
\]
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
We **generate** different computations on the fly, and can change

- Element Batch Size
- Number of Concurrent Elements
- Loop unrolling
- Interleaving stores with computation
Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{ij}^{\beta\gamma}$
Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K_{ij}^{\beta\gamma}$
Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{ij}^{\beta\gamma}$
Computational Flexibility

Basic Contraction

$$G^\beta\gamma(T)K_{\beta\gamma}^{ij}$$

**Figure:** Tensor Contraction $G^\beta\gamma(T)K_{\beta\gamma}^{ij}$
Computational Flexibility

Element Batch Size

**Figure:** Tensor Contraction $G^{\beta\gamma}(T)K_{i\beta}^{ij}$
Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{\beta\gamma}^{ij}$
Figure: Tensor Contraction $G^\beta_\gamma(T)K^{ii}_{\beta_\gamma}$
Computational Flexibility

Element Batch Size

Figure: Tensor Contraction $G^{\beta\gamma}(T)K^{ii}_{\beta\gamma}$
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

\[ K_{\beta\gamma}(T)_{ij}^{\alpha} \]

M. Knepley (UC)
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

\[ K \beta\gamma (T) K^{ij} \beta\gamma \]

M. Knepley (UC)
PDE on GPU
CSE '11 30 / 42
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

\[ K^{\beta\gamma}(T)_{ij} \]
Computational Flexibility

Concurrent Elements

Figure: Tensor Contraction

\[ K_{\beta\gamma} (T_{\beta\gamma}^\text{T}) K_{ij} \]

thread 0

thread 5

thread 15

thread 16

thread 21

thread 31

\( G^0 \)

\( G^1 \)

\( G^0 \)

\( G^1 \)

\( G^0 \)

\( G^1 \)

\( G^0 \)

\( G^1 \)
/* G K contraction: unroll = full */
E[0] += G[0] * K[0];
E[0] += G[1] * K[1];
E[0] += G[7] * K[7];
E[0] += G[8] * K[8];
Loop Unrolling

```c
/* 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];
        }
    }
}
```
```c
/* 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];
```
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;
FEM-GPU

Outline

1. Analytic Flexibility
2. Computational Flexibility
3. Efficiency
Performance
Peak Performance

GPU Flop Rate for
3D $P_1$ Lagrange Laplacian and 2D $P_1$ Lagrange Elasticity

- Blue line: Laplacian bs128 ce2 is
- Green line: Elasticity bs256 ce2 is

MFlops/s vs. Number of Elements
# Price-Performance Comparison of CPU and GPU

3D $P_1$ Laplacian Integration

<table>
<thead>
<tr>
<th>Model</th>
<th>Price ($)</th>
<th>GF/s</th>
<th>MF/s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX285</td>
<td>390</td>
<td>90</td>
<td>231</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>300</td>
<td>2</td>
<td>6.6</td>
</tr>
</tbody>
</table>
Price-Performance Comparison of CPU and GPU
3D $P_1$ Laplacian Integration

<table>
<thead>
<tr>
<th>Model</th>
<th>Price ($)</th>
<th>GF/s</th>
<th>MF/s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX285</td>
<td>390</td>
<td>90</td>
<td>231</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>300</td>
<td>12*</td>
<td>40</td>
</tr>
</tbody>
</table>

* Jed Brown Optimization Engine
CPU vs. GPU Flop Rate for 2D $P_1$ Lagrange ['Elasticity']

Interleave Stores = 1
Loop Unrolling = full
Performance
Influence of Code Structure

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange Laplacian

Element Blocksize = 128
Concurrent Elem = 2

Number of Elements

M. Knepley (UC)
PDE on GPU
CSE ’11
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
Structured code generation, can allow easy integration of novel hardware and reconcile user physics with system traversals.
Structured code generation, can allow easy integration of novel hardware and reconcile user physics with system traversals.
Structured code generation, can allow easy integration of novel hardware and reconcile user physics with system traversals.