GPUs in Computational Science

Matthew Knepley\textsuperscript{1}

\textsuperscript{1}Computation Institute
University of Chicago

Colloquium
GFD Group, ETH Zurich
Switzerland, September 8, 2010
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
New Model for Scientific Software

Simplifying Parallelization of Scientific Codes by a Function-Centric Approach in Python

Jon K. Nilsen, Xing Cai, Bjorn Hoyland, and Hans Petter Langtangen

- **Python** at the application level
- **numpy** for data structures
- **petsc4py** for linear algebra and solvers
- **PyCUDA** for integration (physics) and assembly
Introduction

New Model for Scientific Software

Application

sympy

symbolics

FFC/SyFi
eqn. definition

data structures

solvers

integration/assembly

petsc4py

numpy

PyCUDA

PETSc

CUDA

OpenCL
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
  - Boundary conditions
    - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
Introduction

What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy

- Depth First Search
  - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  -Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy

- Depth First Search
  - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
Outline

1. Introduction

2. Tools
   - numpy
   - petsc4py
   - PyCUDA
   - FEniCS

3. FEM on the GPU

4. PETSc-GPU

5. Conclusions
Tools

- numpy
- petsc4py
- PyCUDA
- FEniCS
**numpy** is ideal for building Python data structures

- Supports multidimensional arrays
- Easily interfaces with C/C++ and Fortran
- High performance BLAS/LAPACK and functional operations
- Python 2 and 3 compatible
- Used by petsc4py to talk to PETSc
Outline

2 Tools

- numpy
- petsc4py
- PyCUDA
- FEniCS
**petsc4py** provides Python bindings for PETSc

- Provides **ALL** PETSc functionality in a Pythonic way
  - Logging using the Python `with` statement

- Can use Python callback functions
  - `SNESSetFunction()`, `SNESSetJacobian()`

- Manages all memory (creation/destruction)

- Visualization with `matplotlib`
**petsc4py Installation**

- **Automatic**
  - `pip install -install-options=-user petsc4py`
  - **Uses** `$PETSC_DIR` and `$PETSC_ARCH`
  - **Installed into** `$HOME/.local`
  - No additions to `PYTHONPATH`

- **From Source**
  - `virtualenv python-env`
  - `source ./python-env/bin/activate`
  - **Now everything installs into your proxy Python environment**
  - `hg clone https://petsc4py.googlecode.com/hg petsc4py-dev`
  - `ARCHFLAGS="-arch x86_64" python setup.py sdist`
  - `ARCHFLAGS="-arch x86_64" pip install dist/petsc4py-1.1.2.tar.gz`
  - **ARCHFLAGS** only necessary on Mac OSX
petsc4py Examples

- externalpackages/petsc4py-1.1/demo/bratu2d/bratu2d.py
  - Solves Bratu equation (SNES ex5) in 2D
  - Visualizes solution with matplotlib

- src/ts/examples/tutorials/ex8.py
  - Solves a 1D ODE for a diffusive process
  - Visualize solution using -vec_view_draw
  - Control timesteps with -ts_max_steps
Tools

- numpy
- petsc4py
- PyCUDA
- FEniCS
PyCUDA and PyOpenCL

Python packages by Andreas Klöckner for embedded GPU programming

- Handles unimportant details automatically
  - CUDA compile and caching of objects
  - Device initialization
  - Loading modules onto card

- Excellent Documentation & Tutorial

- Excellent platform for Metaprogramming
  - Only way to get portable performance
  - Road to FLAME-type reasoning about algorithms
<%namespace name="pb" module="performanceBenchmarks" />
${pb.globalMod(isGPU)} void kernel(${pb.gridSize(isGPU)}) float *output) {
${pb.gridLoopStart(isGPU, load, store)}
${pb.threadLoopStart(isGPU, blockDimX)}
float G[${dim*dim}] = {${', '.join(['3.0']*(dim*dim))}};
float K[${dim*dim}] = {${', '.join(['3.0']*(dim*dim))}};
float product = 0.0;
const int Ooffset = blockIdx*${numThreads};

// Contract G and K
% for n in range(numLocalElements):
%   for alpha in range(dim):
%     for beta in range(dim):
<% gIdx = (n*dim + alpha)*dim + beta %>
<% kIdx = alpha*dim + beta %>
    product += G[${gIdx}] * K[${kIdx}];
% endfor
% endfor
% endfor
output[Ooffset+idx] = product;
${pb.threadLoopEnd(isGPU)}
${pb.gridLoopEnd(isGPU)}
return;
We render code template into strings using a dictionary of inputs.

```python
args = {'dim': self.dim,
        'numLocalElements': 1,
        'numThreads': self.threadBlockSize}
kernelTemplate = self.getKernelTemplate()
gpuCode = kernelTemplate.render(isGPU = True, **args)
cpuCode = kernelTemplate.render(isGPU = False, **args)
```
__global__ void kernel( float *output) {
    const int gridIdx = blockIdx.x + blockIdx.y * blockDim.x;  // This is \((i, j)\)
    const int idx = threadIdx.x + threadIdx.y * 1;  // This is \((i, j)\)
    float G[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
    float K[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
    float product = 0.0;
    const int Ooffset = gridIdx * 1;

    // Contract G and K
    product += G[0] * K[0];
    product += G[1] * K[1];
    product += G[4] * K[4];
    product += G[5] * K[5];
    product += G[7] * K[7];
    product += G[8] * K[8];
    output[Ooffset+idx] = product;
    return;
}
```c
void kernel(int numInvocations, float *output) {
    for(int gridIdx = 0; gridIdx < numInvocations; ++gridIdx) {
        for(int i = 0; i < 1; ++i) {
            for(int j = 0; j < 1; ++j) {
                const int idx = i + j*1; // This is (i, j)
                float G[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float K[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float product = 0.0;
                const int Ooffset = gridIdx*1;

                // Contract G and K
                product += G[0] * K[0];
                product += G[1] * K[1];
                product += G[4] * K[4];
                product += G[5] * K[5];
                product += G[7] * K[7];
                product += G[8] * K[8];
                output[Ooffset+idx] = product;
            }
        }
    }
}
```
Creating a Module

CPU:

```python
# Output kernel and C support code
self.outputKernelC(cpuCode)
self.writeMakefile()
out, err, status = self.executeShellCommand('make')
end{minted}

\bigskip

GPU:
\begin{minted}{python}
from pycuda.compiler import SourceModule

mod = SourceModule(gpuCode)
self.kernel = mod.get_function('kernel')
self.kernelReport(self.kernel, 'kernel')
\end{minted}
import pycuda.driver as cuda
import pycuda.autoinit

blockDim = (self.dim, self.dim, 1)
start = cuda.Event()
end = cuda.Event()
grid = self.calculateGrid(N, numLocalElements)
start.record()
for i in range(iter):
    self.kernel(cuda.Out(output),
                block = blockDim, grid = grid)
end.record()
end.synchronize()
gpuTimes.append(start.time_till(end)*1e−3/iter)
2 Tools

- numpy
- petsc4py
- PyCUDA
- FEniCS
# $P^k$ element

```python
element = FiniteElement("Lagrange", domains[ self.dim ], k)
v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(element)

a = inner(grad(v), grad(u))*dx
L = v*f*dx
```
Element integrals are decomposed into \textit{analytic} and \textit{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.
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

1. Introduction
2. Tools
3. FEM on the GPU
4. PETSc-GPU
5. Conclusions
Element integrals are decomposed into analytic and geometric parts:

\[ \int_{\mathcal{T}} \nabla \phi_i(x) \cdot \nabla \phi_j(x) dx \]

\[ = \int_{\mathcal{T}} \frac{\partial \phi_i(x)}{\partial x_\alpha} \frac{\partial \phi_j(x)}{\partial x_\alpha} dx \]

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

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

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

Coefficients are also put into the geometric part.
Additional fields give rise to multilinear forms.

\[
\int_T \phi_i(\mathbf{x}) \cdot (\phi_k(\mathbf{x}) \nabla \phi_j(\mathbf{x})) \, dA \\
= \int_T \phi_i^\beta(\mathbf{x}) \left( \phi_k^\alpha(\mathbf{x}) \frac{\partial \phi_j^\beta(\mathbf{x})}{\partial x_\alpha} \right) \, dA \\
= \int_{T_{ref}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial x_\alpha}{\partial \xi_\gamma} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| \, dA \\
= \frac{\partial x_\gamma}{\partial \xi_\alpha} |J| \int_{T_{ref}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} dA \\
= G^{\alpha\gamma}(T) K_{ijk}^{\alpha\gamma} 
\]

The index calculus is fully developed by Kirby and Logg in

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

\[ \int_{\mathcal{T}} \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dA = \int_{\mathcal{T}} \frac{\partial \phi_i(x)}{\partial x_\alpha} \frac{\partial \phi_j(x)}{\partial x_\alpha} \, dA \]  
\[ = \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\gamma} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\beta} |J| \, dA \]  
\[ = |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k J^\alpha_k \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \phi_l J^\gamma_l \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} \, dA \]  
\[ = J^\beta_k J^\gamma_l |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \phi_l \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} \, dA \]  
\[ = G^{\beta \gamma}_{kl} (\mathcal{T}) K^{ijkl}_{\beta \gamma} \]  

A different space could also be used for Jacobians
Element matrix $K$ is now made up of small tensors

Contract all tensor elements with each the geometry tensor $G(\mathcal{T})$

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-4</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>-4</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>-8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>-8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-8</td>
<td>-4</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-8</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-8</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>-8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>-8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>-8</td>
<td>-4</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>-8</td>
<td>-4</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| M. Knepley | GPU | 9/1/10 | 30 / 63 |
For $N$ elements, map blocks of $N_L$ elements to each Thread Block (TB)

- Launch grid must be $g_x \times g_y = \frac{N}{N_L}$
- TB grid will depend on the specific algorithm
- Output is size $N_{\text{basis}} \times N_{\text{basis}} \times N_L$

We can split a TB to work on multiple, $N_B$, elements at a time

- Note that each TB always gets $N_L$ elements, so $N_B$ must divide $N_L$
Mapping $G^{\alpha\beta}K^{ij}_{\alpha\beta}$ to the GPU

Kernel Arguments

```c
__global__
void integrateJacobian(float *elemMat,
                        float *geometry,
                        float *analytic)
```

- **geometry**: Array of $G$ tensors for each element
- **analytic**: $K$ tensor
- **elemMat**: Array of $E = G : K$ tensors for each element
We can interleave stores with computation, or wait until the end

- Waiting could improve coalescing of writes
- Interleaving could allow overlap of writes with computation

Also need to

- Coalesce accesses between global and local/shared memory
  (use moveArray())
- Limit use of shared and local memory
Superior GPU memory bandwidth is due to both 

*bus width and clock speed.*

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Width (bits)</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>Bus Clock Speed (MHz)</td>
<td>400</td>
<td>1600</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/s)</td>
<td>3</td>
<td>102</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>240</td>
<td>600</td>
</tr>
</tbody>
</table>

Tesla always accesses blocks of 64 or 128 bytes
Choose strategies to minimize reductions

- Only reductions occur in summation for contractions
  - Similar to the reduction in a quadrature loop

- **Strategy #1**: Each thread uses all of $K$

- **Strategy #2**: Do each contraction in a separate thread
Each thread computes an entire element matrix, so that

$$\text{blockDim} = \left( \frac{N_L}{N_B}, 1, 1 \right)$$

We will see that there is little opportunity to overlap computation and memory access.
FEM on the GPU

Strategy #1
Analytic Part

Read $K$ into shared memory (need to synchronize before access)

```c
__shared__ float K[$\{\text{dim} \times \text{dim} \times \text{numBasisFuncs} \times \text{numBasisFuncs}\}$];

${\text{fm.moveArray('K', 'analytic', }
\text{dim} \times \text{dim} \times \text{numBasisFuncs} \times \text{numBasisFuncs}, '\', \text{numThreads})}

__syncthreads();
```
Strategy #1
Geometry

- Each thread handles $N_B$ elements
- Read $G$ into local memory (not coalesced)
- Interleaving means writing after each thread does a single element matrix calculation

```c
float G[{dim*dim*numBlockElements}];

if (interleaved) {
    const int Goffset = (gridIdx*{numLocalElements} + idx)*{dim*dim};
    for n in range(numBlockElements):
        ${fm.moveArray('G', 'geometry', dim*dim, 'Goffset',
            blockNumber = n*numLocalElements/numBlockElements,
            localBlockNumber = n, isCoalesced = False )}
} else {
    const int Goffset = (gridIdx*{numLocalElements/numBlockElements} + idx)*{dim*dim*numBlockElements};
    ${fm.moveArray('G', 'geometry', dim*dim*numBlockElements, 'Goffset',
        isCoalesced = False )}
}
```
We write element matrices out contiguously by TB

```cpp
const int matSize = numBasisFuncs * numBasisFuncs;
const int Eoffset = gridIdx * matSize * numLocalElements;

if (interleaved) {
    const int elemOff = idx * matSize;
    __shared__ float E[matSize * numLocalElements / numBlockElements];
} else {
    const int elemOff = idx * matSize * numBlockElements;
    __shared__ float E[matSize * numLocalElements];
}```
matSize = numBasisFuncs * numBasisFuncs
if interleaveStores:
    for b in range(numBlockElements):
        # Do 1 contraction for each thread
        __syncthreads();
        fm.moveArray('E', 'elemMat',
                      matSize * numLocalElements / numBlockElements,
                      'Eoffset', numThreads, blockNumber = n, isLoad = 0)
else:
    # Do numBlockElements contractions for each thread
    __syncthreads();
    fm.moveArray('E', 'elemMat',
                 matSize * numLocalElements,
                 'Eoffset', numThreads, isLoad = 0)
Each thread computes a single element of an element matrix, so that

$$\text{blockDim} = (N_{\text{basis}}, N_{\text{basis}}, N_B)$$

This allows us to overlap computation of another element in the TB with writes for the first.
Strategy #2

Analytic Part

- Assign an \((i,j)\) block of \(K\) to local memory
- \(N_B\) threads will simultaneously calculate a contraction

```cpp
const int Kidx = threadIdx.x + threadIdx.y * numBasisFuncs; // This is \((i,j)\)
const int idx = Kidx + threadIdx.z * (numBasisFuncs * numBasisFuncs);
const int Koffset = Kidx * (dim * dim);
float K[dim * dim];

for alpha in range(dim):
    for beta in range(dim):
        kIdx = alpha * dim + beta
        K[kIdx] = analytic[Koffset + kIdx];
```

M. Knepley
GPU
9/1/10
Strategy #2
Geometry

- Store $N_L$ G tensors into shared memory
- Interleaving means writing after each thread does a single element calculation

```cpp
const int Goffset = gridIdx * {dim*dim*numLocalElements};
__shared__ float G[ {dim*dim*numLocalElements} ];
${fm.moveArray('G', 'geometry', dim*dim*numLocalElements, 'Goffset', numThreads)}
__syncthreads();
```
We write element matrices out contiguously by TB
If interleaving stores, only need a single product
Otherwise, need $N_L/N_B$, one per element processed by a thread

```cpp
const int matSize = numBasisFuncs * numBasisFuncs;
const int Eoffset = gridIdx * matSize * numLocalElements;

if (interleaved) {
    float product = 0.0;
    const int elemOff = idx * matSize;
} else {
    float product [ numLocalElements / numBlockElements ];
    const int elemOff = idx * matSize * numBlockElements;
}
```
if interleaveStores:
    for n in range(numLocalElements / numBlockElements):
        # Do 1 contraction for each thread
        __syncthreads()
        # Do coalesced write of element matrix
        elemMat[Eoffset+idx + n*numThreads] = product
    else:
        # Do numLocalElements/numBlockElements contractions
        # save results in product[]
        for n in range(numLocalElements / numBlockElements):
            elemMat[Eoffset+idx + n*numThreads] = product[n]
CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 32
FEM on the GPU

Results

GTX 285

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

Element Blocksize = 32
Interleaving stores
Results
GTX 285, 2 Simultaneous Elements

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

Element Blocksize = 32
Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D \( P_1 \) Lagrange

Element Blocksize = 32
Interleaving stores
CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 64
Results
GTX 285

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

Element Blocksize = 64
Interleaving stores
Results

GTX 285, 2 Simultaneous Elements

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

Element Blocksize = 64
Results
GTX 285, 2 Simultaneous Elements

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

Element Blocksize = 64
Interleaving stores
Results

GTX 285

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

Element Blocksize = 128
Results

GTX 285

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

Element Blocksize = 128
Interleaving stores
Results
GTX 285, 2 Simultaneous Elements

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

Element Blocksize = 128
FEM on the GPU

Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D \( P_1 \) Lagrange

Element Blocksize = 128
Interleaving stores
Results
GTX 285, 2 Simultaneous Elements

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

Element Blocksize = 256
Results
GTX 285, 2 Simultaneous Elements

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

Element Blocksize = 256
Interleaving stores
Results
GTX 285, 4 Simultaneous Elements

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

Element Blocksize = 256
Results
GTX 285, 4 Simultaneous Elements

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

Element Blocksize = 256
Interleaving stores
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>PETSC_CUDA_UNALLOCATED</th>
<th>No allocation on the GPU</th>
</tr>
</thead>
<tbody>
<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.
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 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

M. Knepley  GPU  9/1/10  58 / 63
Outline

1. Introduction
2. Tools
3. FEM on the GPU
4. PETSc-GPU
5. Conclusions
How Will Algorithms Change?

- **Massive concurrency** is necessary
  - Mix of vector and thread paradigms
  - Demands new analysis

- **More attention to memory management**
  - Blocks will only get larger
  - Determinant of performance
Results

9400M

CPU vs. GPU Flop Rate for 3D P_1 Lagrange

Element Blocksize = 1
Results

9400M

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

Element Blocksize = 2
Results

9400M

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

Element Blocksize = 4
Results

9400M

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

Element Blocksize = 8

Number of Elements

MFlops/s
Results

9400M

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

Element Blocksize = 16
Results

9400M

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

Element Blocksize = 16
Interleaving stores
Conclusions

Results

9400M

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

Element Blocksize = 32
Results

9400M

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

Element Blocksize = 32
Interleaving stores