Finite Element Integration using CUDA and OpenCL

Matthew Knepley, Karl Rupp, Andy Terrel

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

GPU-SMP 2013
Changchun, China  July 28–Aug 2, 2013
Collaborators

ViennaCL Creator
ANL

SciPy 2013 Chair
TACC

Karl Rupp

Andy Terrel
Research Products

- Efficiently vectorized FEM algorithm
  - Traversals are handled by the PETSc library
  - Separates physics from discretization

- Open implementation in PETSc
  - Runs in normal package examples
  - Needed OpenCL, too unstructured for OpenMP
Efficiently vectorized FEM algorithm
Traversals are handled by the PETSc library
Separates physics from discretization

Open implementation in PETSc
Runs in normal package examples
Needed OpenCL, too unstructured for OpenMP
- Efficiently vectorized FEM algorithm
  Traversals are handled by the PETSc library
  Separates physics from discretization

- Open implementation in PETSc
  Runs in normal package examples
  Needed OpenCL, too unstructured for OpenMP
Introduction

Research Products

- Efficiently vectorized FEM algorithm
  Traversals are handled by the PETSc library
  Separates physics from discretization

- Open implementation in PETSc
  Runs in normal package examples
  Needed OpenCL, too unstructured for OpenMP
Research Products

- Efficiently vectorized FEM algorithm
  Traversals are handled by the PETSc library
  Separates physics from discretization

- Open implementation in PETSc
  Runs in normal package examples
  Needed OpenCL, too unstructured for OpenMP
Outline

1. Vectorizing FEM

2. Performance
Why is Vectorization Important?

For vector length $k$, without vectorization we can attain only $\frac{1}{k}$ of peak performance.

For GTX580, $k = 32$

so that unvectorized code runs at 3% of peak.
Vectorizing FEM

Why is Vectorization Important?

For vector length $k$, without vectorization we can attain only $\frac{1}{k}$ of peak performance.

For GTX580, $k = 32$

so that unvectorized code runs at 3% of peak.
Why is Vectorization Important?

For streaming computations, other factors are less important:

- except coalesced (vectorized) loads
- little cache reuse
- tiling not as important
- latency covered by computation
Why is Vectorization Important?

Concurrent loads are necessary to saturate the memory bandwidth

<table>
<thead>
<tr>
<th>Architecture</th>
<th>STREAMS¹ (GB/s)</th>
<th>Peak (GB/s)</th>
<th>Eff (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GTX 285</td>
<td>134</td>
<td>159</td>
<td>84</td>
</tr>
<tr>
<td>NVIDIA GTX 580</td>
<td>166</td>
<td>192</td>
<td>86</td>
</tr>
<tr>
<td>AMD HD7970</td>
<td>199</td>
<td>264</td>
<td>75</td>
</tr>
<tr>
<td>Dual Intel E5-2670²</td>
<td>80</td>
<td>101</td>
<td>79</td>
</tr>
<tr>
<td>Intel Xeon Phi</td>
<td>95</td>
<td>220³</td>
<td>43</td>
</tr>
</tbody>
</table>

¹ Results benefit from autotuning

² See also https://panthema.net/2013/pmbw/Intel-Xeon-E5-2670-64GB

³ This is the ring bus limit, not the processor limit of 320 GB/s
Compilers cannot vectorize arbitrary code, and users typically do not vectorize

```
for (q = 0; q < N_q; ++q) {
  for (b = 0; b < N_b; ++b) {
    /* Calculate residual for test function res_0 and derivative res_1 */
    b_q = basis[q*N_b+b];
    db_q = basisDer[q*N_b+b];
    r_b += b_q * res_0;
    r_b += db_q * res_1;
  }
}
```

OpenCL results show large variations, depending on the compiler
Vectorization is complicated by hardcoding physics routines

```cpp
for (q = 0; q < N_q; ++q) {
  /* Calculate field and derivative at quadrature point */
  for (b = 0; b < N_b; ++b) {
    b_q = basis[q*N_b+b];
    db_q = basisDer[q*N_b+b];
    r_b += b_q * F(u_q, du_q);
    r_b += db_q * G(u_q, du_q);
  }
}
```

We avoid hardcoding by adopting a separated model for integration.
We consider weak forms dependent only on fields and gradients,

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

(1)

Discretizing we have

\[ \sum_e \mathcal{E}_e^T \begin{bmatrix} B^T W^q f_0(u^q, \nabla u^q) + \sum_k D_k^T W^q \vec{f}_{1k}^k(u^q, \nabla u^q) \end{bmatrix} = 0 \]  

(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
Many levels of blocking are necessary:

- **Chunk**: Basic tile
- **Batch**: Executed in serial
- **Block**: Executed concurrently and are more easily dealt with generically by the library.

We illustrate these sizes in the next section.
Vectorization over basis functions increases required bandwidth by a factor $N_b$
Vectorization over basis functions increases required bandwidth by a factor $N_b$
Vectorization over basis functions increases required bandwidth by a factor $N_b$
Vectorization over quadrature points increases required bandwidth by a factor $N_q$. 
If we vectorize first over quadrature points,

\[ q_1 \quad q_0 \quad q_1 \quad q_0 \quad q_1 \quad q_0 \quad q_1 \quad q_0 \quad q_1 \quad q_0 \]

and then over basis functions

\[ b_0 \quad b_1 \quad b_2 \quad b_0 \quad b_1 \quad b_2 \quad b_0 \quad b_1 \quad b_2 \quad b_0 \]

for a batch of cells, there must be a reduction over quadrature points.
If we vectorize first over quadrature points,

\[ q_1 \quad q_0 \quad q_1 \quad q_0 \quad q_1 \quad q_0 \quad q_0 \]

and then over basis functions

\[ b_2 \quad b_1 \quad b_2 \quad b_1 \quad b_2 \quad b_1 \quad b_2 \]

for a batch of cells, there must be a reduction over quadrature points.
If we vectorize first over quadrature points,

\[ q_0 \quad q_0 \quad q_0 \quad q_0 \quad q_0 \quad q_0 \]

and then over basis functions

\[ b_0 \quad b_0 \quad b_0 \quad b_0 \quad b_0 \quad b_0 \]

\[ b_1 \quad b_1 \quad b_1 \quad b_1 \quad b_1 \quad b_1 \]

\[ b_2 \quad b_2 \quad b_2 \quad b_2 \quad b_2 \quad b_2 \]

for a batch of cells, there must be a reduction over quadrature points.
Thread Transposition

Map values at quadrature points to coefficients

Evaluate basis and process values at quadrature points

Continue with kernel
Basis Phase

\[ N_{bc} = 12 \]
\[ N_t = 24 \]
\[ N_{bl} = 2 \]
\[ N_{sbc} = 3 \]

Quadrature Phase

\[ N_{sqc} = 2 \]
\[ N_t = 24 \]
\[ N_{bc} = 12 \]
\[ N_{bl} = 2 \]
\[ N_{bs} = 6 \]
Thread Transposition

- Removes reduction
- Single pass through memory
  - Operate in unassembled space
  - Could do scattered load (better with cache)
  - Our cell tiling would aid this
- Needs local memory
  - Bounded by $N_b N_q$, good for low order
All our runs may be reproduced from the PETSc development branch:

```
git clone https://bitbucket.org/petsc/petsc petsc-dev
cd petsc-dev
git fetch
git checkout next
```

To run the benchmarks, you configure using

```
./configure --with-shared-libraries --with-dynamic-loading
--download-mpich
--download-scientificpython --download-fiat
--download-generator
--download-triangle --download-chaco
```
Vectorizing FEM

Open Implementation

Building

All our runs may be reproduced from the PETSc development branch:

```
  git clone https://bitbucket.org/petsc/petsc petsc-dev
  cd petsc-dev
  git fetch
  git checkout next
```

To run the benchmarks, you configure using

```
./configure --with-shared-libraries --with-dynamic-loading
  --download-mpich
  --download-scientificpython --download-fiat
  --download-generator
  --download-triangle --download-chaco
```

and for CUDA you also need

```
  --with-cudac='nvcc -m64' --with-cuda-only
```
Open Implementation
Building

All our runs may be reproduced from the PETSc development branch:

```
    git clone https://bitbucket.org/petsc/petsc petsc-dev
    cd petsc-dev
    git fetch
    git checkout next
```

To run the benchmarks, you configure using

```
./configure --with-shared-libraries --with-dynamic-loading
    --download-mpich
    --download-scientificpython --download-fiats
    --download-generator
    --download-triangle --download-chaco
```

and for OpenCL you also need

```
    --with-opencl
```
Open Implementation

Building

All our runs may be reproduced from the PETSc development branch:

```bash
git clone https://bitbucket.org/petsc/petsc petsc-dev
cd petsc-dev
git fetch
git checkout next
```

To run the benchmarks, you configure using

```bash
./configure --with-shared-libraries --with-dynamic-loading
 --download-mpich
 --download-scientificpython --download-fiat
 --download-generator
 --download-triangle --download-chaco
```

and for OpenCL (on Mac) you also need

```bash
--with-opencl-include=/System/Library/Frameworks/
OpenCL.framework/Headers/
--with-opencl-lib=/System/Library/Frameworks/
OpenCL.framework/OpenCL
```
All our runs may be reproduced from the PETSc development branch:

```bash
git clone https://bitbucket.org/petsc/petsc petsc-dev
cd petsc-dev
git fetch
git checkout next
```

To run the benchmarks, you configure using

```bash
./configure --with-shared-libraries --with-dynamic-loading --download-mpich --download-scientificpython --download-fiat --download-generator --download-triangle --download-chaco
```

To build, use

```bash
make
```
All our runs may be reproduced from the PETSc development branch:

```
git clone https://bitbucket.org/petsc/petsc petsc-dev
cd petsc-dev
git fetch
git checkout next
```

To run the benchmarks, you configure using

```
./configure --with-shared-libraries --with-dynamic-loading
--download-mpich
--download-scientificpython --download-fiat
--download-generator
--download-triangle --download-chaco
```

To build with Python, use

```
./config/builder2.py build
```
A representative run for the $P_1$ Laplacian:

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

All run parameters are listed in the forthcoming paper.
A representative run for the $P_1$ Laplacian:
which is translated to

```
./\${PETSC_ARCH}/lib/ex52-obj/ex52
  -refinement_limit 0.0625 -compute_function -batch
  -gpu -gpu_batches 8 -gpu_blocks 16
  -log_summary summary.dat -log_summary_python
  -dm_view -show_residual 0 -preload off
```

All run parameters are listed in the forthcoming paper.
Outline

1. Vectorizing FEM

2. Performance
Performance on SNES Example 52

- GPU-OpenCL-32 IntegGPUOnly
- GPU-CUDA-32 IntegGPUOnly
Performance on SNES Example 52

- GPU-32 IntegBatchCPU
- CPU-32 IntegBatchCPU
- GPU-32 IntegBatchGPU
- GPU-32 IntegGPUOnly
Performance on SNES Example 52

Computation Rate (GF/s) vs Number of Dof

- GPU-OpenCL-32 IntegGPUOnly
- GPU-CUDA-32 IntegGPUOnly
Performance

Block size variation

Nvidia GTX580

Performance on SNES Example 52 - NVIDIA GTX 580

![Graph showing performance comparison of different block sizes on Nvidia GTX 580. The x-axis represents the number of dofs, and the y-axis represents the computation rate (GF/s). Different lines represent different block sizes, with green for blockExp 3, dashed green for blockExp 4, solid brown for blockExp 5, dotted red for blockExp 6, and solid red for blockExp 7. The graph shows varying performance across different block sizes.]
Performance on SNES Example 52

- GPU-32 IntegBatchCPU
- CPU-32 IntegBatchCPU
- GPU-32 IntegBatchGPU
- GPU-32 IntegGPUOnly

Computation Rate (MF/s) vs. Number of Dof
Performance Block size variation ATI HD7970

Performance on SNES Example 52 - AMD Radeon HD 7970

- blockExp 3
- blockExp 4
- blockExp 5
- blockExp 6
Performance on SNES Example 52

Graph showing the performance of different configurations on an Intel Xeon Phi.

- GPU-32 IntegBatchCPU
- CPU-32 IntegBatchCPU
- GPU-32 IntegBatchGPU
- GPU-32 IntegGPUOnly

The graph plots the computation rate (MF/s) against the number of degrees of freedom (Dof).
Scaling on the TACC Longhorn cluster

Performance on SNES Example 52

- GPU-32 IntegGPUOnly

Computation Rate (GF/s)

Number of Processors
Conclusions

- Traversals should be handled by the library
  - Allows efficient vectorization
  - Separates physics from discretization

- Performance portability requires better compilers
  - Vectorization is somewhat behind
  - MIC programming model is broken
Conclusions

- Traversals should be handled by the library
  Allows efficient vectorization
  Separates physics from discretization

- Performance portability requires better compilers
  Vectorization is somewhat behind
  MIC programming model is broken
Conclusions

- Traversals should be handled by the library
  Allows efficient vectorization
  Separates physics from discretization
- Performance portability requires better compilers
  Vectorization is somewhat behind
  MIC programming model is broken
Conclusions

- Traversals should be handled by the library
  Allows efficient vectorization
  Separates physics from discretization
- Performance portability requires better compilers
  Vectorization is somewhat behind
  MIC programming model is broken
Conclusions

- Traversals should be handled by the library
  Allows efficient vectorization
  Separates physics from discretization
- Performance portability requires better compilers
  Vectorization is somewhat behind
  MIC programming model is broken
Competing Models

How should kernels be integrated into libraries?

**CUDA/OpenCL**
- 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/OpenCL
- 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
How should kernels be integrated into libraries?

CUDA/OpenCL
- 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
How should kernels be integrated into libraries?

CUDA/OpenCL
- 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
Conclusions

Competing Models

How should kernels be integrated into libraries?

CUDA/OpenCL
- 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