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

Karl Rupp

SciPy 2013 Chair
TACC

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

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

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

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 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

Architecture	STREAMS[1] (GB/s)	Peak (GB/s)	Eff (%)
NVIDIA GTX 285	134	159	84
NVIDIA GTX 580	166	192	86
AMD HD7970	199	264	75
Dual Intel E5-2670[2]	80	101	79
Intel Xeon Phi	95	220[3]	43

[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

Impediments to Vectorization
Compiler Complexity

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

Impediments to Vectorization
User-specified physics routines

Vectorization is complicated by hardcoding physics routines

```
for (q = 0; q < N_q; ++q) {
  /* Calculate field and derviative 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.

FEM Integration Model

Proposed by Jed Brown

We consider weak forms dependent only on fields and gradients,

$$\int_\Omega \phi \cdot f_0(u, \nabla u) + \nabla\phi : \vec{f}_1(u, \nabla u) = 0. \qquad (1)$$

Discretizing we have

$$\sum_e \mathcal{E}_e^T \left[B^T W^q f_0(u^q, \nabla u^q) + \sum_k D_k^T W^q \vec{f}_1^k(u^q, \nabla u^q) \right] = 0 \qquad (2)$$

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

Impediments to Vectorization
Code Complexity

Many levels of blocking are necessary:

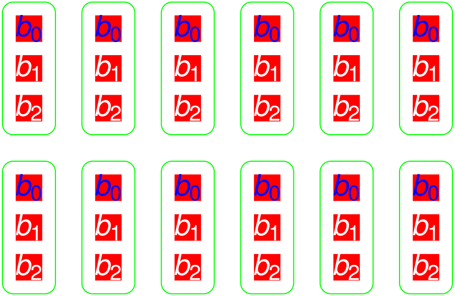
- **Chunk**: Basic tile
- **Batch**: Executed in serial
- **Block**: Executed concurrently

and are more easily dealt with generically by the library.

We illustrate these sizes in the next section.

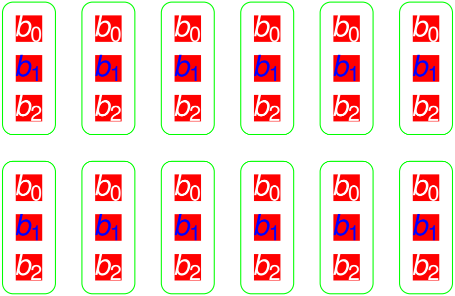
Impediments to Vectorization
Memory bandwidth

Vectorization over basis functions increases required bandwidth by a factor N_b



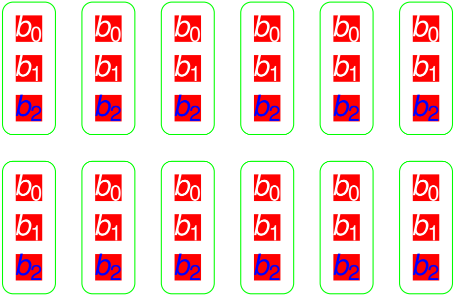
Impediments to Vectorization
Memory bandwidth

Vectorization over basis functions increases required bandwidth by a factor N_b



Impediments to Vectorization
Memory bandwidth

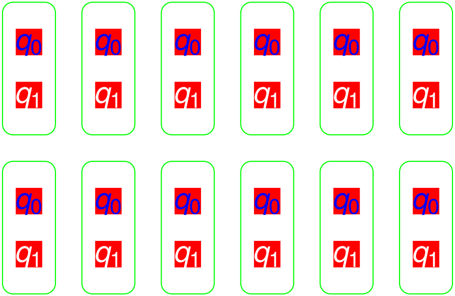
Vectorization over basis functions increases required bandwidth by a factor N_b



Impediments to Vectorization

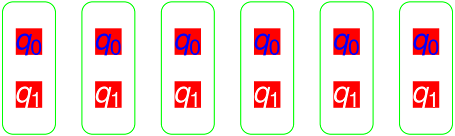
Memory bandwidth

Vectorization over quadrature points increases required bandwidth by a factor N_q



Impediments to Vectorization
Reductions

If we vectorize first over quadrature points,



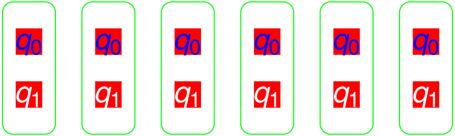
and then over basis functions

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

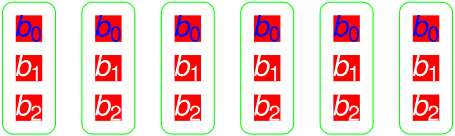
Impediments to Vectorization

Reductions

If we vectorize first over quadrature points,



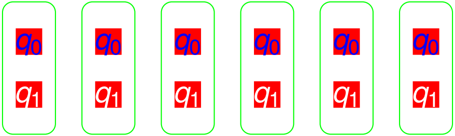
and then over basis functions



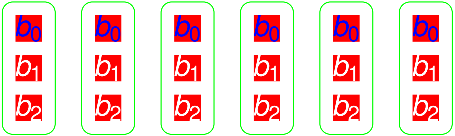
for a batch of cells, there must be a reduction over quadrature points.

Impediments to Vectorization
Reductions

If we vectorize first over quadrature points,

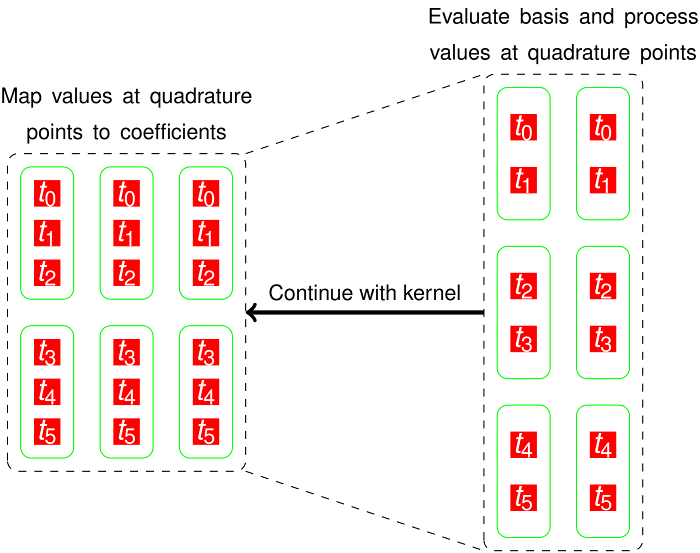


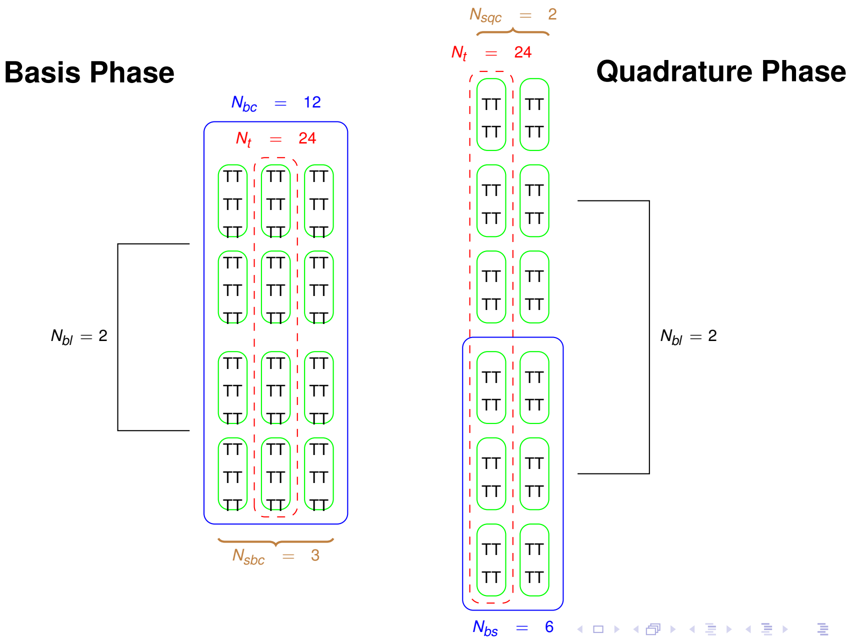
and then over basis functions



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

Thread Transposition





Basis Phase
$N_{bc} = 12$
$N_t = 24$
$N_{bl} = 2$
$N_{sbc} = 3$
Quadrature Phase
$N_{sqc} = 2$
$N_t = 24$
$N_{bl} = 2$
$N_{bs} = 6$

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

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
```

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-fiat
  --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:

```
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 OpenCL (on Mac) you also need

```
  --with-opencl-include=/System/Library/Frameworks/
OpenCL.framework/Headers/
  --with-opencl-lib=/System/Library/Frameworks/
OpenCL.framework/OpenCL
```

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
```

To build, use

```
make
```

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
```

To build with Python, use

```
./config/builder2.py build
```

Open Implementation
Running

A representative run for the P_1 Laplacian:

```
./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.

Open Implementation
Running

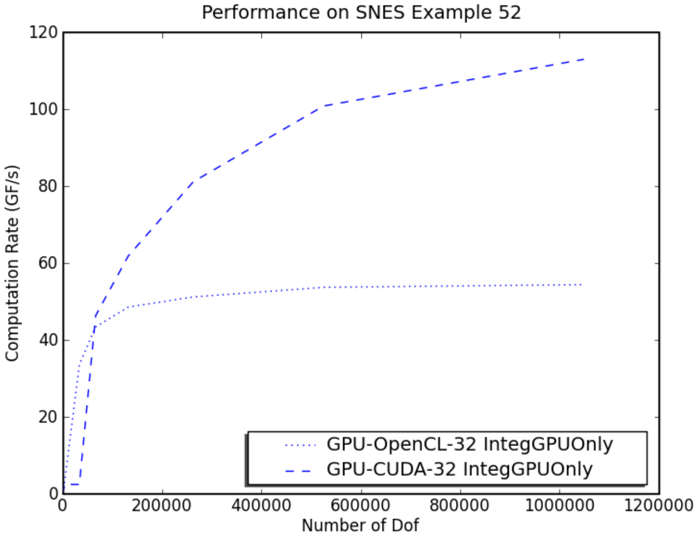
A representative run for the P_1 Laplacian:
which is translated to

```
./\${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
```

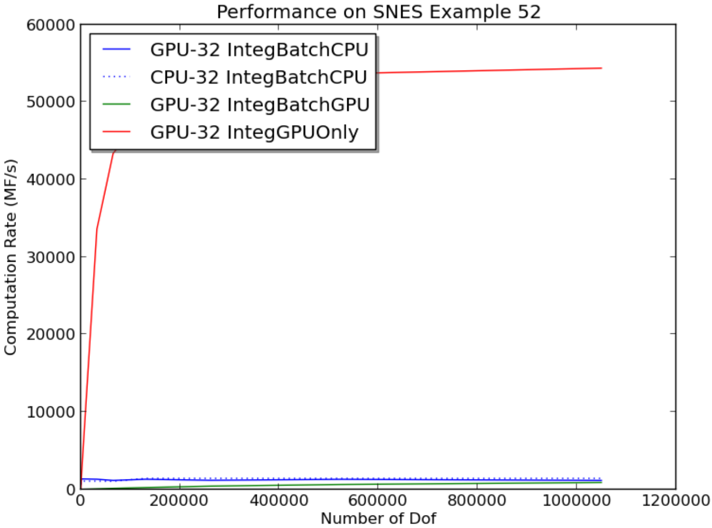
All run parameters are listed in the forthcoming paper.

Outline

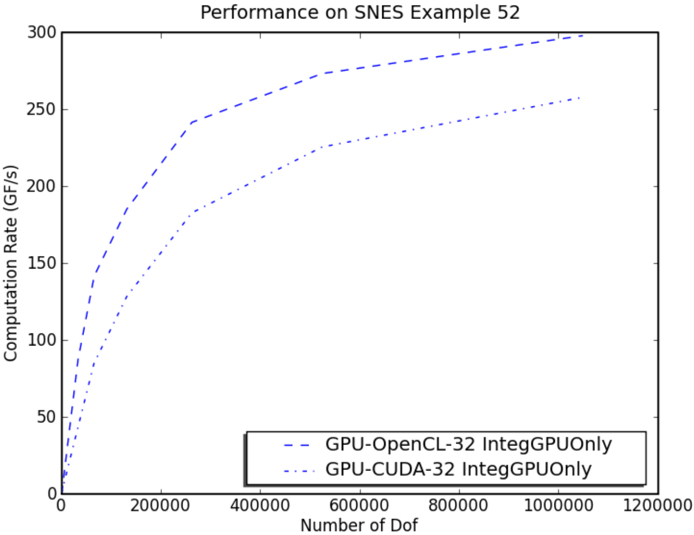
Nvidia GTX285 CUDA



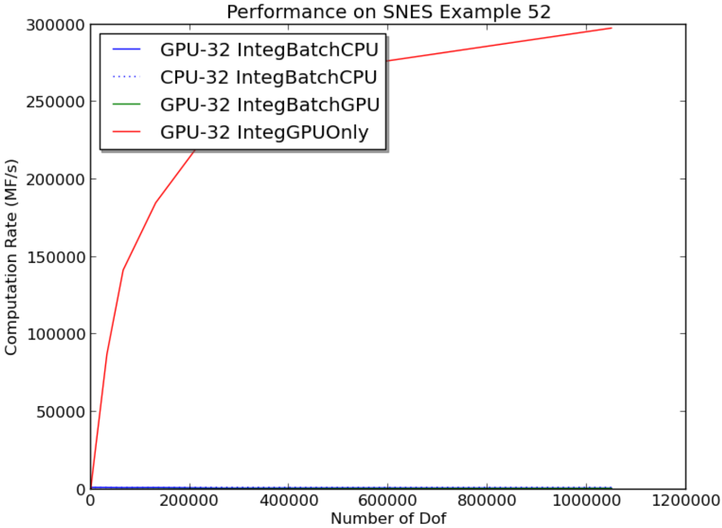
Nvidia GTX285 OpenCL



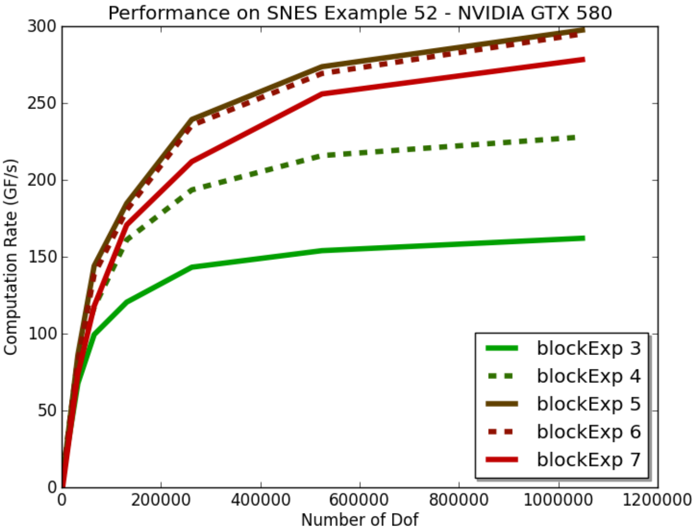
Nvidia GTX580 CUDA



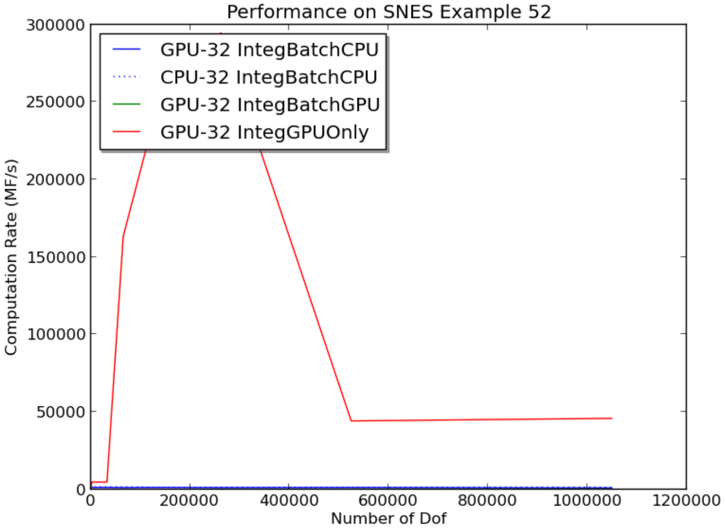
Nvidia GTX580 OpenCL



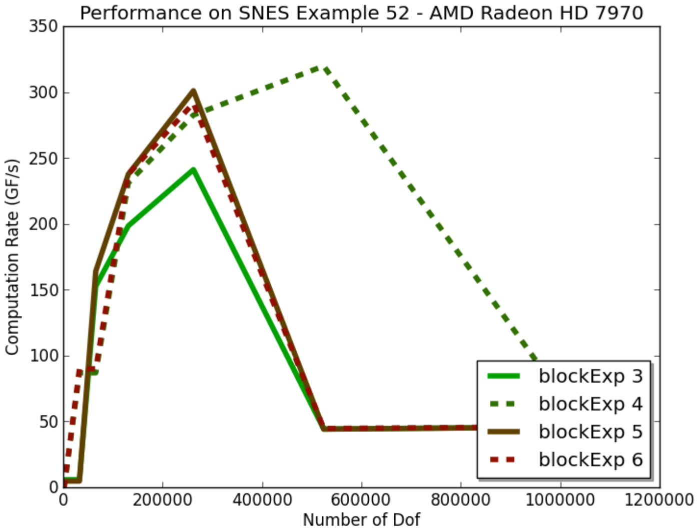
Block size variation
Nvidia GTX580



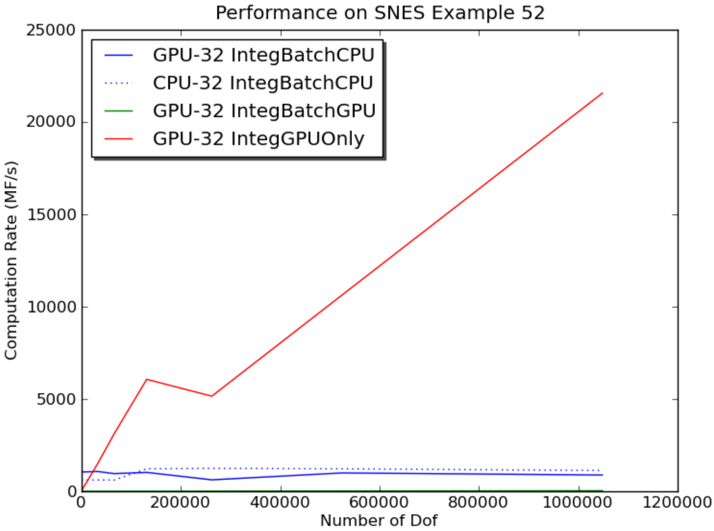
ATI HD7970


Performance on SNES Example 52
GPU-32 IntegBatchCPU
CPU-32 IntegBatchCPU
GPU-32 IntegBatchGPU
GPU-32 IntegGPUOnly
Computation Rate (MF/s)
Number of Dof
300000
250000
200000
150000
100000
50000
0
0
200000
400000
600000
800000
1000000
1200000

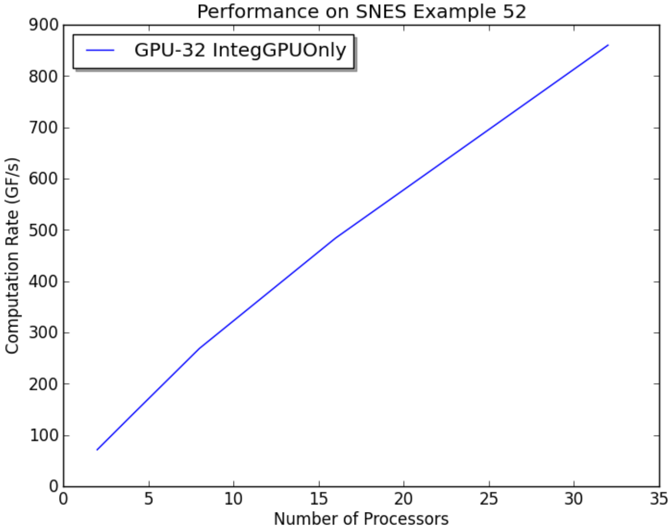
Block size variation
ATI HD7970



Intel Xeon Phi



Scaling on the TACC Longhorn cluster



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

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

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