Developing GPU-Enabled Scientific Libraries

Matthew Knepley

Computation Institute
University of Chicago

Department of Molecular Biology and Physiology
Rush University Medical Center

CBC Workshop on CBC Key Topics
Simula Research Laboratory
Oslo, Norway   August 25–26, 2011

RUSH UNIVERSITY MEDICAL CENTER

Outline

1. Scientific Libraries
   - What is PETSc?
2. Linear Systems
3. Assembly
4. Integration
5. Yet To be Done

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.

Main Point

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

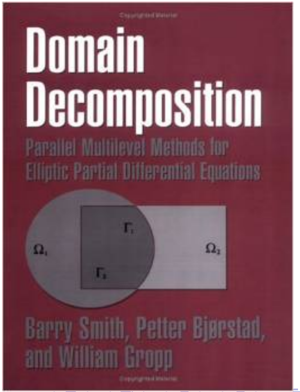
1 Scientific Libraries
- What is PETSc?

How did PETSc Originate?

PETSc was developed as a Platform for Experimentation

We want to experiment with different

- Models
- Discretizations
- Solvers
- Algorithms
    - which blur these boundaries



The Role of PETSc

Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort.

PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver, nor a silver bullet.

— Barry Smith

Advice from Bill Gropp

You want to think about how you decompose your data structures, how you think about them globally. [...] If you were building a house, you'd start with a set of blueprints that give you a picture of what the whole house looks like. You wouldn't start with a bunch of tiles and say. "Well I'll put this tile down on the ground, and then I'll find a tile to go next to it." But all too many people try to build their parallel programs by creating the smallest possible tiles and then trying to have the structure of their code emerge from the chaos of all these little pieces. You have to have an organizing principle if you're going to survive making your code parallel.

(http://www.rce-cast.com/Podcast/rce-28-mpich2.html)

What is PETSc?

A freely available and supported research code for the parallel solution of nonlinear algebraic equations

Free

- Download from http://www.mcs.anl.gov/petsc
- Free for everyone, including industrial users

Supported

- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov

Usable from C, C++, Fortran 77/90, Matlab, Julia, and Python

What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems
    - Cray XT6, BG/Q, NVIDIA Fermi, K Computer
  - Loosely coupled systems, such as networks of workstations
    - IBM, Mac, iPad/iPhone, PCs running Linux or Windows
- PETSc History
  - Begun September 1991
  - Over 60,000 downloads since 1995 (version 2)
  - Currently 400 per month
- PETSc Funding and Support
  - Department of Energy
    - SciDAC, MICS Program, AMR Program, INL Reactor Program
  - National Science Foundation
    - CIG, CISE, Multidisciplinary Challenge Program

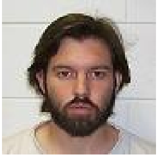
The PETSc Team


Bill Gropp


Barry Smith


Satish Balay


Jed Brown


Matt Knepley


Lisandro Dalcin


Hong Zhang


Mark Adams


Toby Issac

Who Uses PETSc?

Computational Scientists

- Earth Science
  - PyLith (CIG)
  - Underworld (Monash)
  - Magma Dynamics (LDEO, Columbia, Oxford)
- Subsurface Flow and Porous Media
  - STOMP (DOE)
  - PFLOTRAN (DOE)

Who Uses PETSc?

Computational Scientists

- CFD
  - Firedrake
  - Fluidity
  - OpenFOAM
  - freeCFD
  - OpenFVM
- MicroMagnetics
  - MagPar
- Fusion
  - XGC
  - BOUT++
  - NIMROD

Who Uses PETSc?

Algorithm Developers

- Iterative methods
  - Deflated GMRES
  - LGMRES
  - QCG
  - SpecEst
- Preconditioning researchers
  - Prometheus (Adams)
  - ParPre (Eijkhout)
  - FETI-DP (Klawonn and Rheinbach)

Who Uses PETSc?

Algorithm Developers

- Finite Elements
  - libMesh
  - MOOSE
  - PETSc-FEM
  - Deal II
  - OOFEM
- Other Solvers
  - Fast Multipole Method (PetFMM)
  - Radial Basis Function Interpolation (PetRBF)
  - Eigensolvers (SLEPc)
  - Optimization (TAO)

What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media
- PETSc has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL
- PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code

What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media
- PETSc has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL
- PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code

What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
    - UNIC on BG/P and XT5
    - PFLOTRAN for flow in porous media
- PETSc has run on over 290,000 cores efficiently
    - UNIC on the IBM BG/P Jugene at Jülich
    - PFLOTRAN on the Cray XT5 Jaguar at ORNL
- PETSc applications have run at 23% of peak (600 Teraflops)
    - Jed Brown on NERSC Edison
    - HPGMG code

Interface Questions

How should the user interact with manycore systems?

Through computational libraries

How should the user interact with the library?

Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?

- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions

Interface Questions

How should the user interact with manycore systems?

Through computational libraries

How should the user interact with the library?

Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?

- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions

Interface Questions

How should the user interact with manycore systems?

Through computational libraries

How should the user interact with the library?

Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?

- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions

Interface Questions

How should the user interact with manycore systems?

Through computational libraries

How should the user interact with the library?

Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?

- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions

Interface Questions

How should the user interact with manycore systems?

Through computational libraries

How should the user interact with the library?

Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?

- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions

Interface Questions

How should the user interact with manycore systems?

Through computational libraries

How should the user interact with the library?

Strong, data structure-neutral API (Smith and Gropp, 1996)

How should the library interact with manycore systems?

- Existing library APIs
- Code generation (CUDA, OpenCL, PyCUDA)
- Custom multi-language extensions

Performance Analysis

In order to understand and predict the performance of GPU code, we need:

good models for the computation, which make it possible to evaluate the efficiency of an implementation;

a flop rate, which tells us how well we are utilizing the hardware;

timing, which is what users care about;

Outline

1. Scientific Libraries
2. Linear Systems
3. Assembly
4. Integration
5. Yet To be Done

Performance Expectations

Linear Systems

The Sparse Matrix-Vector product (SpMV) is limited by system memory bandwidth, rather than by peak flop rate.

- We expect bandwidth ratio speedup (3x–6x for most systems)
- Memory movement is more important than minimizing flops
- Kernel is a vectorized, segmented sum (Blelloch, Heroux, and Zagha: CMU-CS-93-173)

Memory Bandwidth

All computations in this presentation are memory bandwidth limited. We have a *bandwidth peak*, the maximum flop rate achievable given a bandwidth. This depends on β, the ratio of bytes transferred to flops done by the algorithm.

Processor	BW (GB/s)	Peak (GF/s)	BW Peak* (GF/s)
Core 2 Duo	4	34	1
GeForce 9400M	21	54	5
GTX 285	159	1062	40
Tesla M2050	144	1030	36

*Bandwidth peak is shown for $\beta = 4$

STREAM Benchmark

Simple benchmark program measuring sustainable memory bandwidth

- Protoypical operation is Triad (WAXPY): $\mathbf{w} = \mathbf{y} + \alpha\mathbf{x}$
- Measures the memory bandwidth bottleneck (much below peak)
- Datasets outstrip cache

Machine	Peak (MF/s)	Triad (MB/s)	MF/MW	Eq. MF/s
Matt's Laptop	1700	1122.4	12.1	93.5 (5.5%)
Intel Core2 Quad	38400	5312.0	57.8	442.7 (1.2%)
Tesla 1060C	984000	102000.0*	77.2	8500.0 (0.8%)

Table: Bandwidth limited machine performance

http://www.cs.virginia.edu/stream/

Analysis of Sparse Matvec (SpMV)

Assumptions

- No cache misses
- No waits on memory references

Notation

- m Number of matrix rows
- nz Number of nonzero matrix elements
- V Number of vectors to multiply

We can look at bandwidth needed for peak performance

$$\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V} \text{ byte/flop} \tag{1}$$

or achieveable performance given a bandwith BW

$$\frac{Vnz}{(8V+2)m + 6nz} BW \text{ Mflop/s} \tag{2}$$

Towards Realistic Performance Bounds for Implicit CFD Codes, Gropp, Kaushik, Keyes, and Smith.

Linear Algebra Interfaces

Strong interfaces mean:

- Easy code interoperability (LAPACK, Trilinos)
- Easy portability (GPU)
- Seamless optimization

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

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

Example

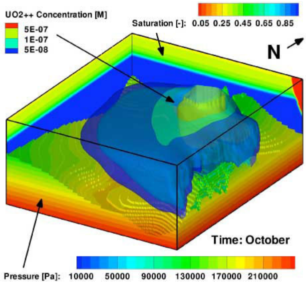
Driven Cavity Velocity-Vorticity with Multigrid

```
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

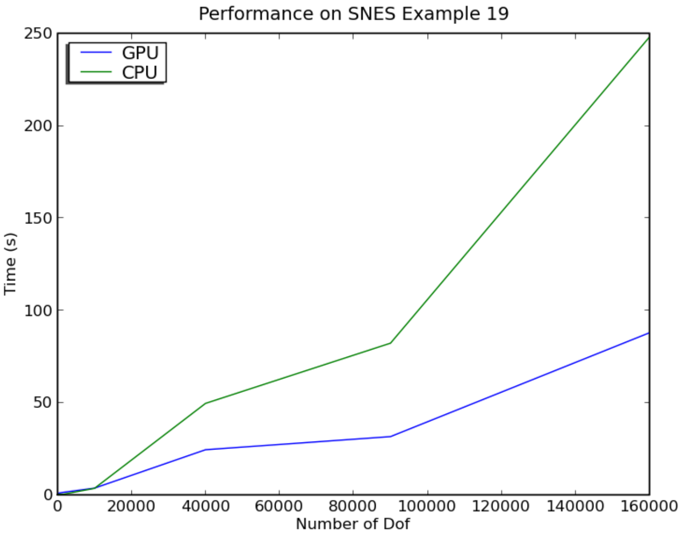
Flow Solver
$32 \times 32 \times 32$ grid

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

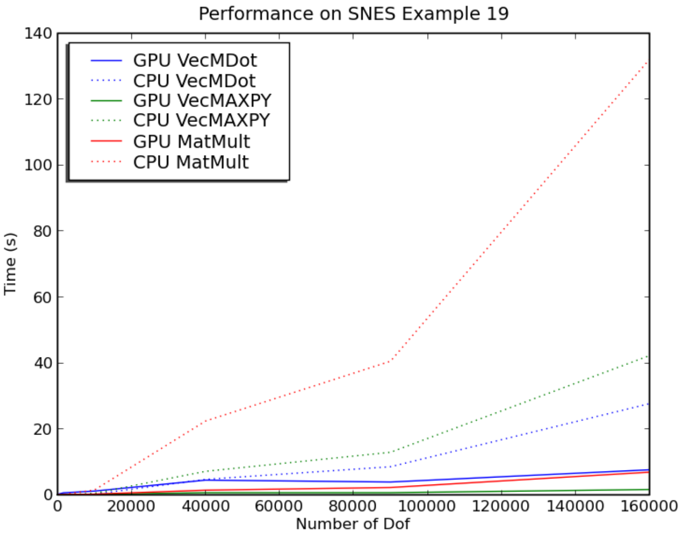


P. Lichtner, G. Hammond, R. Mills, B. Phillip

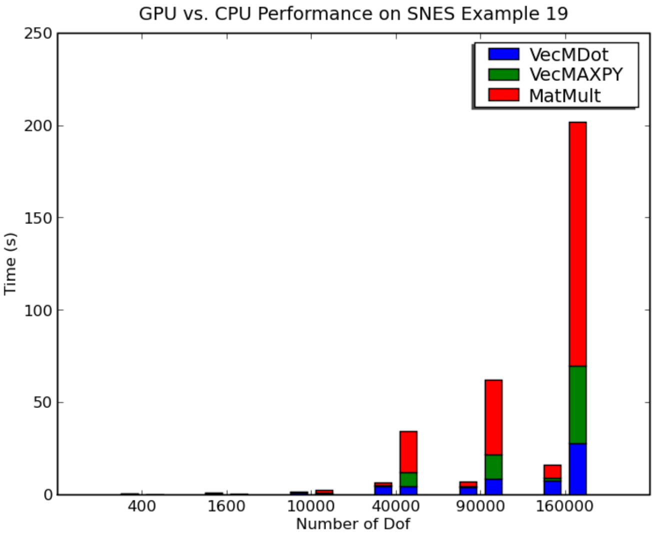
Serial Performance
NVIDIA GeForce 9400M



Serial Performance
NVIDIA Tesla M2050



Serial Performance
NVIDIA Tesla M2050



Outline

1. Scientific Libraries
2. Linear Systems
3. Assembly
4. Integration
5. Yet To be Done

Performance Expectations
Matrix Assembly

Matrix Assembly, aggregation of inputs, is also limited by memory bandwidth, rather than by peak flop rate.

- We expect bandwidth ratio speedup (3x–6x for most systems)
- Input for FEM is a set of element matrices
- Kernel is dominated by sort (submission to TOMS)

Assembly Interface

A single new method is added:

```
MatSetValuesBatch(Mat J, PetscInt Ne, PetscInt Nl,
                  PetscInt *elemRows,
                  PetscScalar *elemMats)
```

Thus, a user just batches his input to achieve massive concurrency.

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input

Convenience Iterators

```
repeated_range<IndexArrayIterator>
  rowInd(elemRows.begin(), elemRows.end(), NI);
tiled_range<IndexArrayIterator>
  colInd(elemRows.begin(), elemRows.end(), NI, NI);
```

$$N_I = 3$$

elemRows	0 1 3
rowInd	0 0 0 | 1 1 1 | 3 3 3
colInd	0 1 3 | 0 1 3 | 0 1 3

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
    1. Get permutation from (stably) sorting columns
    2. Gather rows with this permutation
    3. Get permutation from (stably) sorting rows
    4. Gather columns with this permutation
    5. Gather values with this permutation

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
    1. Get permutation from (stably) sorting columns
    2. Gather rows with this permutation
    3. Get permutation from (stably) sorting rows
    4. Gather columns with this permutation
    5. Gather values with this permutation

Multikey Sort

Initial input

(1	0)
(3	1)
(0	0)
(1	1)
(3	3)
(0	1)
(0	3)
(3	0)
(1	3)

Multikey Sort

Number pairs

		Index
(1	0)	0
(3	1)	1
(0	0)	2
(1	1)	3
(3	3)	4
(0	1)	5
(0	3)	6
(3	0)	7
(1	3)	8

Multikey Sort

After stable sort of columns

		Index
(1	0)	0
(0	0)	2
(3	0)	7
(3	1)	1
(1	1)	3
(0	1)	5
(3	3)	4
(0	3)	6
(1	3)	8

Multikey Sort

After gather of rows
using column permutation,
and implicit renumbering

		Index
(1	0)	0
(0	0)	1
(3	0)	2
(3	1)	3
(1	1)	4
(0	1)	5
(3	3)	6
(0	3)	7
(1	3)	8

Multikey Sort

After stable sort of rows, and gather of columns using row permutation

		Index
(0	0)	1
(0	1)	5
(0	3)	7
(1	0)	0
(1	1)	4
(1	3)	8
(3	0)	2
(3	1)	3
(3	3)	6

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
5. Compute number of unique (i,j) entries using inner_product()

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
5. Compute number of unique (i,j) entries using inner_product()

Counting Unique Entries

Initial input

(0	0)
(0	1)
(0	1)
(0	3)
(1	0)
(1	1)
(3	0)
(3	0)
(3	0)

Counting Unique Entries

Duplicate input

(0 0)	(0 0)
(0 1)	(0 1)
(0 1)	(0 1)
(0 3)	(0 3)
(1 0)	(1 0)
(1 1)	(1 1)
(3 0)	(3 0)
(3 0)	(3 0)
(3 0)	(3 0)

Counting Unique Entries

Shift new sequence
and truncate initial input

(0	0)	(0	1)
(0	1)	(0	1)
(0	1)	(0	3)
(0	3)	(1	0)
(1	0)	(1	1)
(1	1)	(3	0)
(3	0)	(3	0)
(3	0)	(3	0)

Counting Unique Entries

“Multiply entries” using not-equals binary operator

$$
\begin{array}{cc|cccc}
(0 & 0) & (0 & 1) & \Longrightarrow & 1 \\
(0 & 1) & (0 & 1) & \Longrightarrow & 0 \\
(0 & 1) & (0 & 3) & \Longrightarrow & 1 \\
(0 & 3) & (1 & 0) & \Longrightarrow & 1 \\
(1 & 0) & (1 & 1) & \Longrightarrow & 1 \\
(1 & 1) & (3 & 0) & \Longrightarrow & 1 \\
(3 & 0) & (3 & 0) & \Longrightarrow & 0 \\
(3 & 0) & (3 & 0) & \Longrightarrow & 0
\end{array}
$$

Counting Unique Entries

Reduction of entries plus 1 gives number of unique entries

			1
(0 0)	(0 1)	$\Longrightarrow$	1
(0 1)	(0 1)	$\Longrightarrow$	0
(0 1)	(0 3)	$\Longrightarrow$	1
(0 3)	(1 0)	$\Longrightarrow$	1
(1 0)	(1 1)	$\Longrightarrow$	1
(1 1)	(3 0)	$\Longrightarrow$	1
(3 0)	(3 0)	$\Longrightarrow$	0
(3 0)	(3 0)	$\Longrightarrow$	0
			6

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
5. Compute number of unique (i,j) entries using inner_product()
6. Allocate COO storage for final matrix
7. Sum values with the same (i,j) index using reduce_by_key()
8. Convert to AIJ matrix
9. Copy from GPU (if necessary)

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
5. Compute number of unique (i,j) entries using inner_product()
6. Allocate COO storage for final matrix
7. Sum values with the same (i,j) index using reduce_by_key()
8. Convert to AIJ matrix
9. Copy from GPU (if necessary)

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
5. Compute number of unique (i,j) entries using inner_product()
6. Allocate COO storage for final matrix
7. Sum values with the same (i,j) index using reduce_by_key()
8. Convert to AIJ matrix
9. Copy from GPU (if necessary)

Serial Assembly Steps

1. Copy elemRows and elemMat to device
2. Allocate storage for intermediate COO matrix
3. Use repeat&tile iterators to expand row input
4. Sort COO matrix by row and column
5. Compute number of unique (i,j) entries using inner_product()
6. Allocate COO storage for final matrix
7. Sum values with the same (i,j) index using reduce_by_key()
8. Convert to AIJ matrix
9. Copy from GPU (if necessary)

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
   1. Find number of off-process rows (serial)
   2. Map rows to processes (serial)
   3. Send number of rows to each process (collective)

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
   1. Find number of off-process rows (serial)
   2. Map rows to processes (serial)
   3. Send number of rows to each process (collective)

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
    1. Find number of off-process rows (serial)
    2. Map rows to processes (serial)
    3. Send number of rows to each process (collective)

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
   1. Partition into diagonal and off-diagonal&off-process using partition_copy ()
   2. Partition again into off-diagonal and off-process using stable_partition ()

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
   1. Partition into diagonal and off-diagonal&off-process using partition_copy ()
   2. Partition again into off-diagonal and off-process using stable_partition ()

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
    1. Partition into diagonal and off-diagonal&off-process using partition_copy ()
    2. Partition again into off-diagonal and off-process using stable_partition ()

Partitioning Entries

Process owns rows [0, 3)

Initial input

$$\left(\begin{array}{ccc|c} (0,0) & \cdots & (0,2) & (0,3) \\ \vdots & \ddots & \vdots & (0,3) \\ (2,0) & \cdots & (2,2) & (0,3) \\ \hline (3,0) & (3,1) & (3,2) & (3,3) \end{array}\right)$$

$$\begin{array}{cc} (3 & 0) \\ (0 & 1) \\ (3 & 3) \\ (0 & 3) \\ (0 & 0) \\ (3 & 1) \\ (1 & 3) \\ (1 & 1) \\ (1 & 0) \end{array}$$

Partitioning Entries

Process owns rows $[0, 3)$

Partition into diagonal, and off-diagonal & off-process entries

Diagonal	(0	0)
	(1	1)
	(0	1)
	(1	0)
Off-diagonal and Off-process	(3	1)
	(3	0)
	(1	3)
	(3	3)
	(0	3)

Partitioning Entries

Process owns rows $[0, 3)$

Partition again into off-diagonal and off-process entries

Diagonal	$(0 \quad 0)$ $(1 \quad 1)$ $(0 \quad 1)$ $(1 \quad 0)$
Off-diagonal	$(1 \quad 3)$ $(0 \quad 3)$
Off-process	$(3 \quad 1)$ $(3 \quad 0)$ $(3 \quad 3)$

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
6. Send off-process entries
7. Allocate storage for intermediate off-diagonal COO matrix
8. Repartition entries into diagonal and off-diagonal using partition_copy()
9. Repeat serial assembly on both matrices

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
6. Send off-process entries
7. Allocate storage for intermediate off-diagonal COO matrix
8. Repartition entries into diagonal and off-diagonal using partition_copy ()
9. Repeat serial assembly on both matrices

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
6. Send off-process entries
7. Allocate storage for intermediate off-diagonal COO matrix
8. Repartition entries into diagonal and off-diagonal using partition_copy()
9. Repeat serial assembly on both matrices

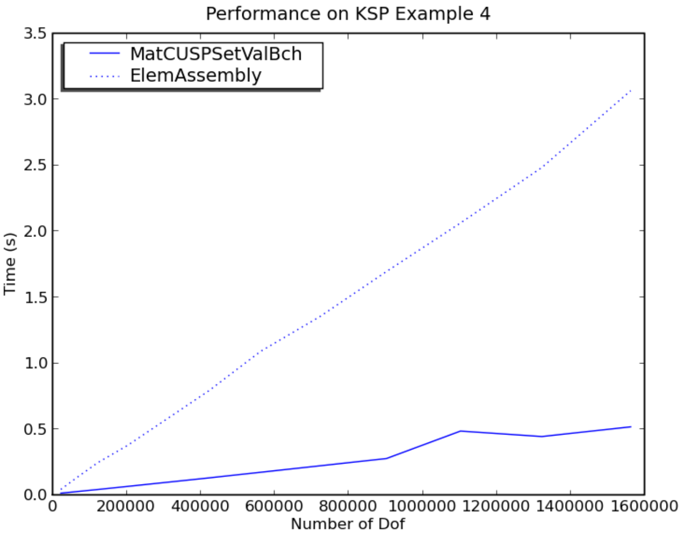
Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
6. Send off-process entries
7. Allocate storage for intermediate off-diagonal COO matrix
8. Repartition entries into diagonal and off-diagonal using `partition_copy()`
9. Repeat serial assembly on both matrices

Parallel Assembly Steps

1. Copy elemRows and elemMat to device
2. Use repeat&tile iterators to expand row input
3. Communicate off-process entry sizes
4. Allocate storage for intermediate diagonal COO matrix
5. Partition entries
6. Send off-process entries
7. Allocate storage for intermediate off-diagonal COO matrix
8. Repartition entries into diagonal and off-diagonal using `partition_copy ()`
9. Repeat serial assembly on both matrices

Serial Performance
NVIDIA GTX 285



Outline

1. Scientific Libraries
2. Linear Systems
3. Assembly
4. Integration
   - Analytic Flexibility
   - Computational Flexibility
   - Efficiency
5. Yet To be Done

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

4 Integration
- Analytic Flexibility
- Computational Flexibility
- Efficiency

Analytic Flexibility
Laplacian

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{3}$$

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

Analytic Flexibility

Laplacian

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{3}$$

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

Analytic Flexibility

Linear Elasticity

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

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

Analytic Flexibility
Linear Elasticity

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

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

Analytic Flexibility

Full Elasticity

$$\frac{1}{4}\int_{\mathcal{T}}\left(\nabla\vec{\phi}_i(\mathbf{x})+\nabla^T\vec{\phi}_i(\mathbf{x})\right):C:\left(\nabla\vec{\phi}_j(\mathbf{x})+\nabla\vec{\phi}_j(\mathbf{x})\right)d\mathbf{x} \tag{5}$$

```
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

$$\frac{1}{4}\int_{\mathcal{T}} \left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : C : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right) d\mathbf{x} \qquad (5)$$

```
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

$$\frac{1}{4}\int_{\mathcal{T}}\left(\nabla\vec{\phi}_i(\mathbf{x})+\nabla^T\vec{\phi}_i(\mathbf{x})\right):C:\left(\nabla\vec{\phi}_j(\mathbf{x})+\nabla\vec{\phi}_j(\mathbf{x})\right)dx \tag{5}$$

```
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

Form Decomposition

Element integrals are decomposed into analytic and geometric parts:

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \quad (6)$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} \quad (7)$$

$$= \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| d\mathbf{x} \quad (8)$$

$$= \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} d\mathbf{x} \quad (9)$$

$$= G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma} \quad (10)$$

Coefficients are also put into the geometric part.

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
```

Outline

4 Integration
- Analytic Flexibility
- Computational Flexibility
- Efficiency

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

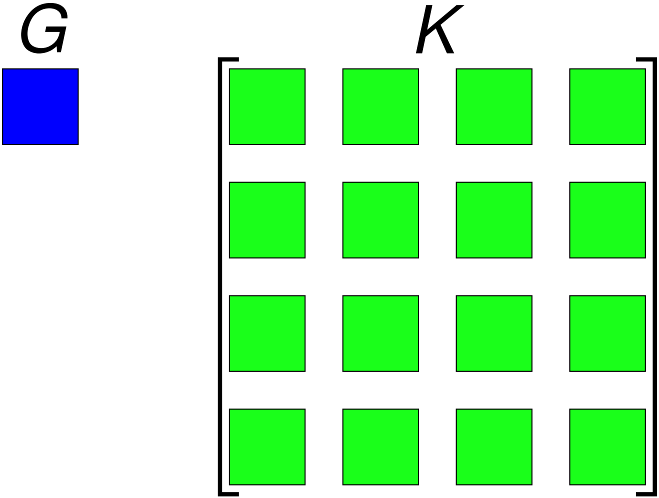


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

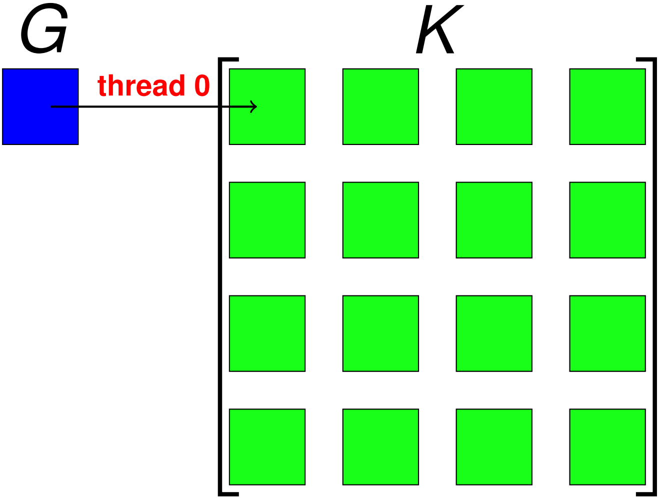


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

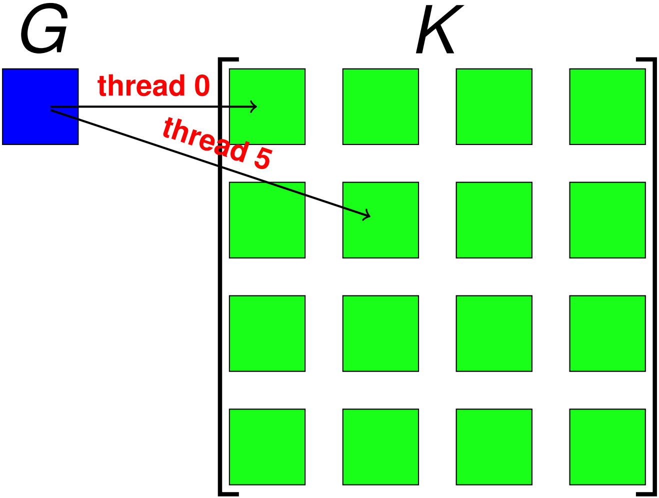


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

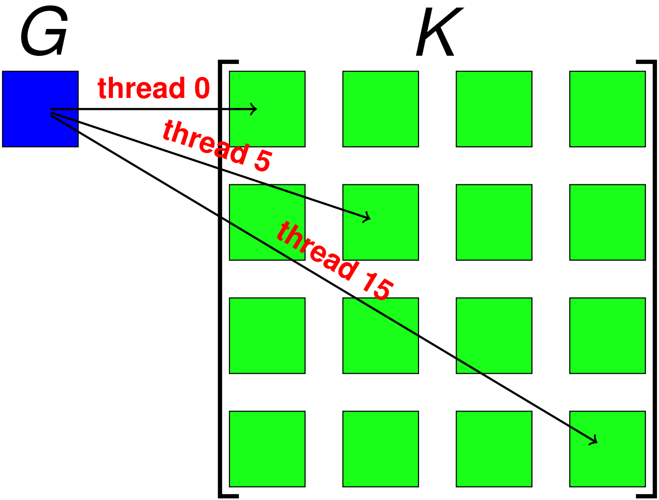


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Element Batch Size

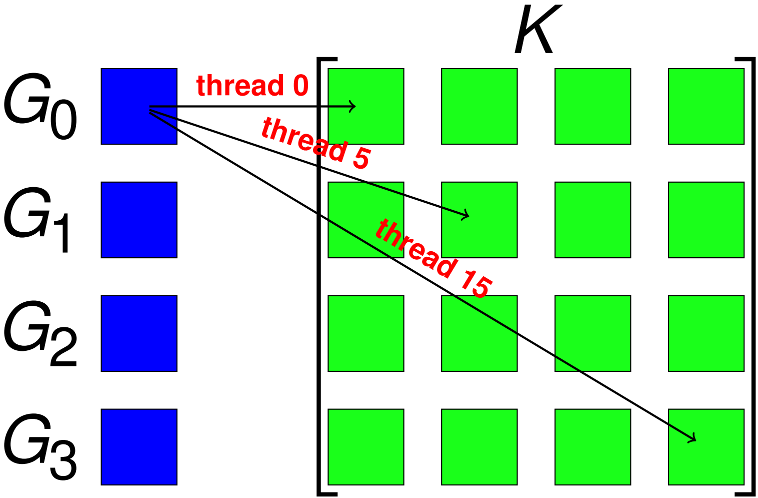


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Element Batch Size

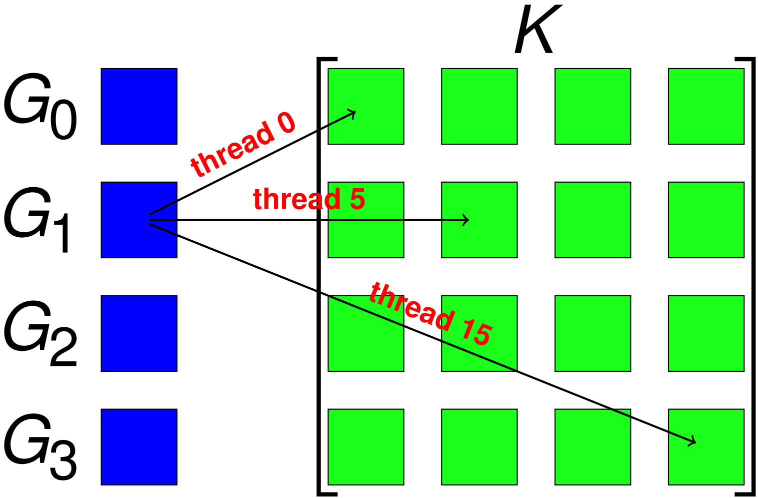


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

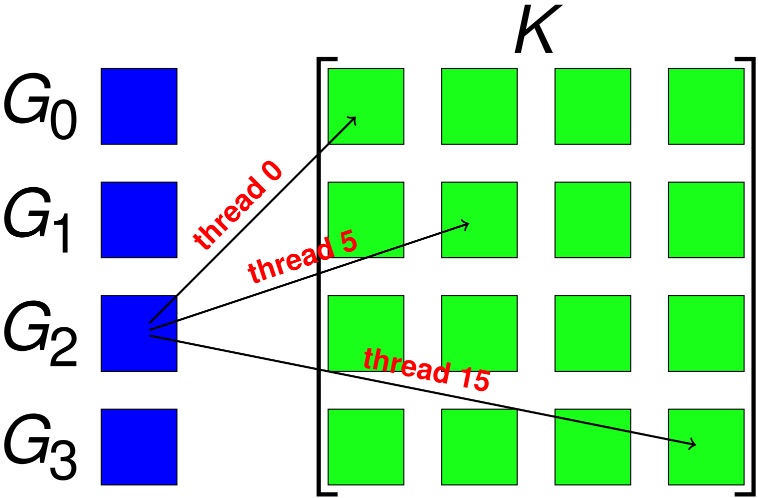


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Element Batch Size

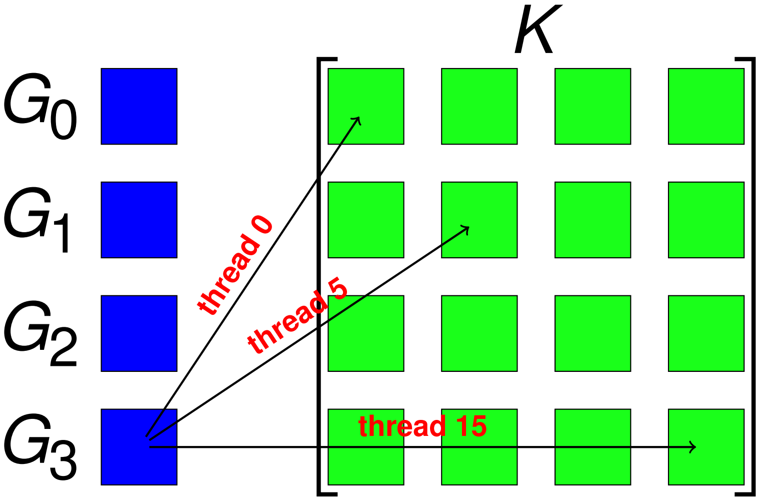
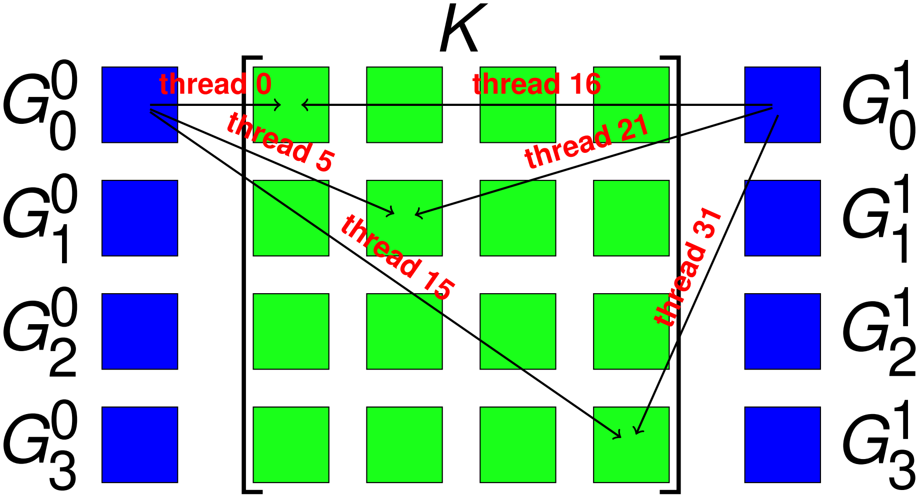
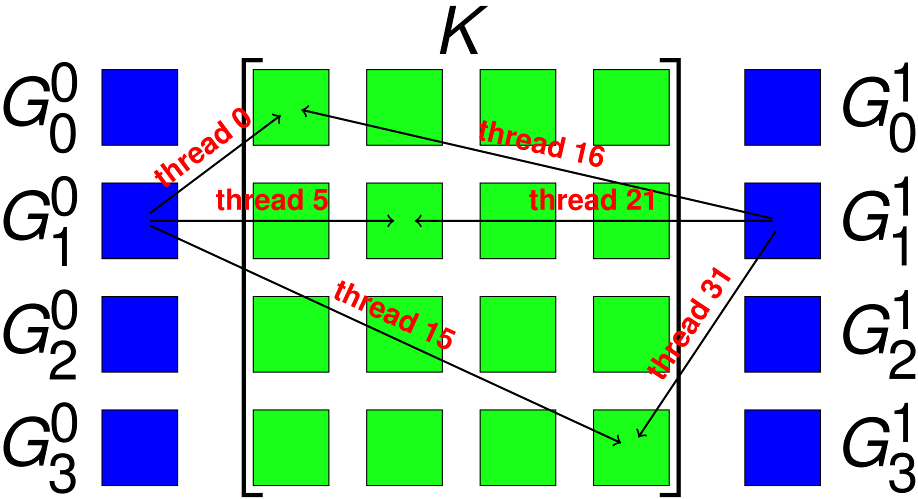


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Concurrent Elements

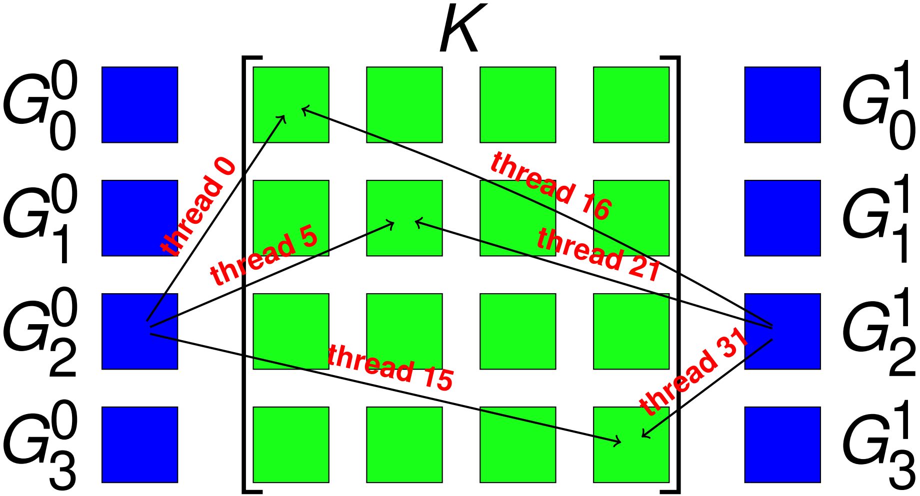


Computational Flexibility
Concurrent Elements

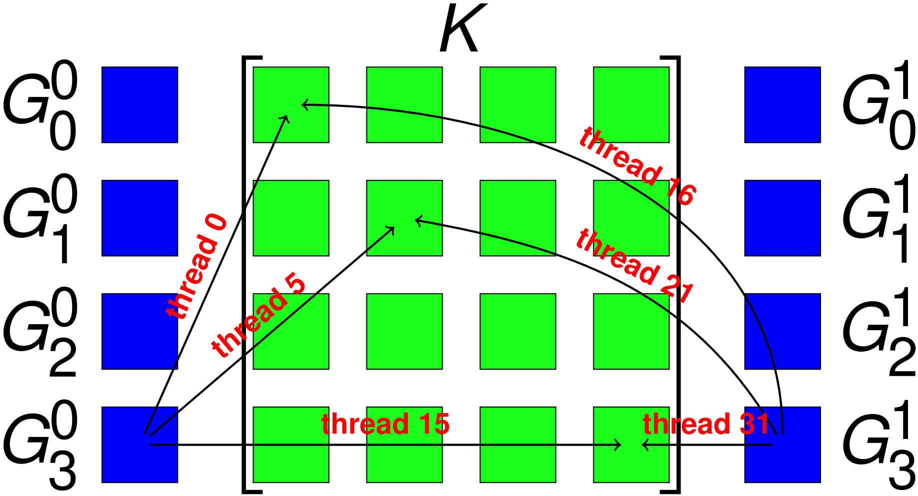


Computational Flexibility

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

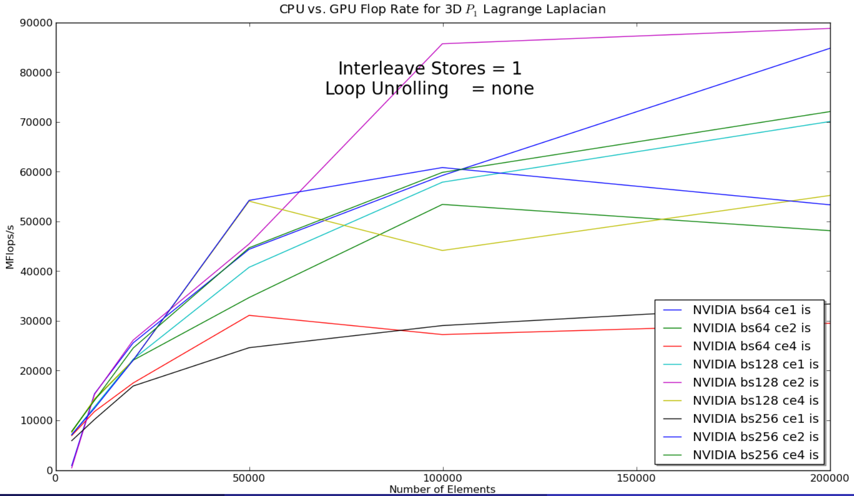
Interleaving stores

```
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

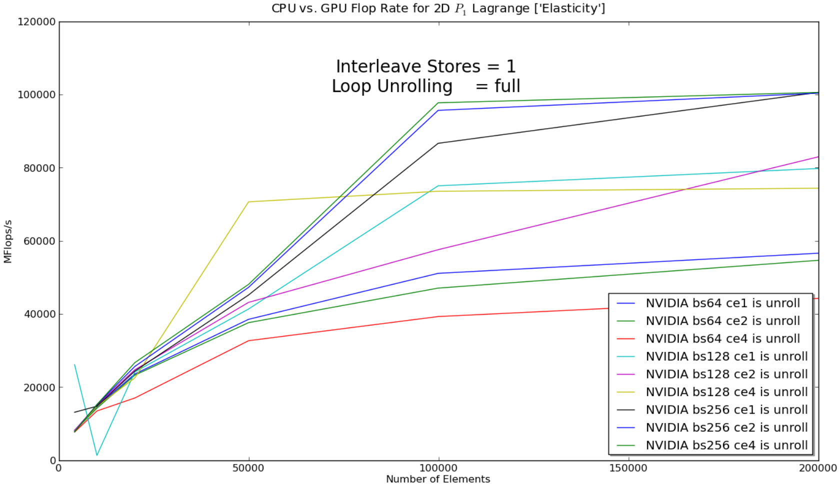
4 Integration
- Analytic Flexibility
- Computational Flexibility
- Efficiency

Performance
Influence of Element Batch Sizes



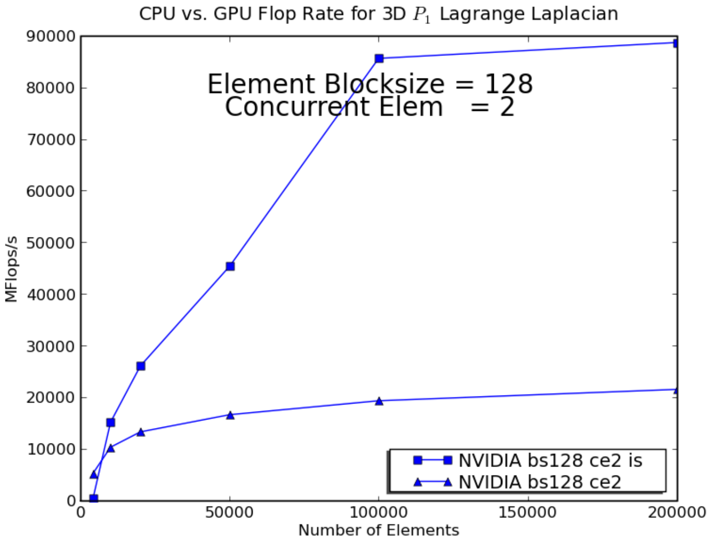
Performance

Influence of Element Batch Sizes

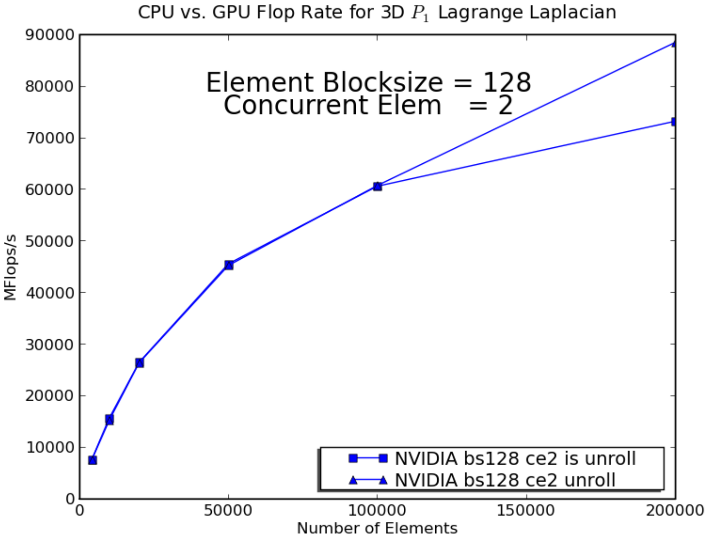


Performance

Influence of Code Structure



Performance
Influence of Code Structure



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

Outline

1 Scientific Libraries

2 Linear Systems

3 Assembly

4 Integration

5 Yet To be Done

Competing Models

How should modern scientific computing be structured?

Current Model: PETSC

- Single language
- Hand optimized
- 3rd party libraries
- new hardware

Competing Models

How should modern scientific computing be structured?

Current Model: PETSC

- Single language
- Hand optimized
- 3rd party libraries
- new hardware

Competing Models

How should modern scientific computing be structured?

Current Model: PETSC

- Single language
- Hand optimized
- 3rd party libraries
- new hardware

Competing Models

How should modern scientific computing be structured?

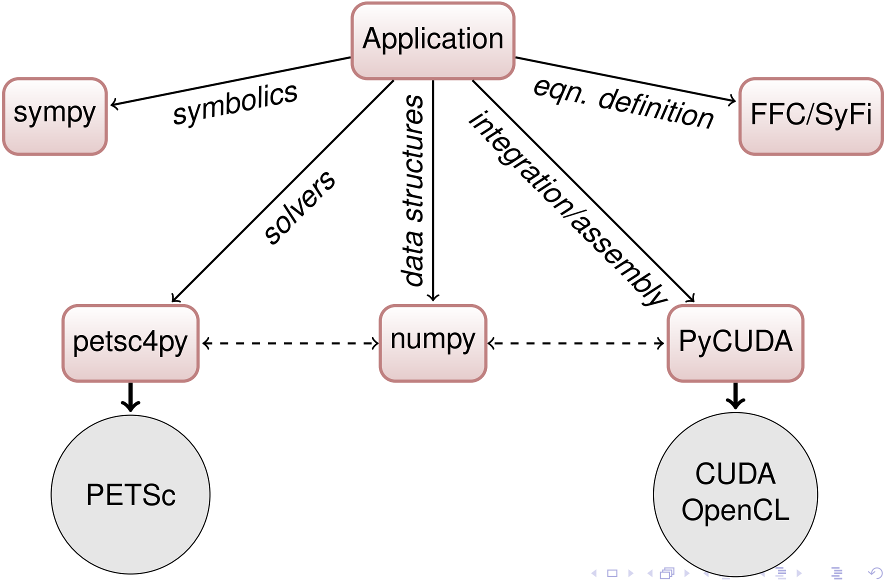
Current Model: PETSC

- Single language
- Hand optimized
- 3rd party libraries
- new hardware

Alternative Model: PetCLAW

- Multiple language through Python
- Optimization through code generation
- 3rd party libaries through wrappers
- New hardware through code generation

New Model for Scientific Software



What Do We Still Need?

- Better integration of code generation
  - Match CUDA driver interface to CUDA runtime interface
  - Extend code generation to quadrature schemes
  - Kernel fusion in assembly
- Better hierarchical parallelism
  - Larger scale parallel GPU tests
  - Synchronization reduction in current algorithms