Building Robust Scientific Codes

Matthew Knepley

Computation Institute
University of Chicago

Scientific Computing in the Americas:
The Challenge of Massive Parallelism
Valparaiso, Chile, January 2011

RUSH UNIVERSITY MEDICAL CENTER

Outline

1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS
2. GPU Computing
3. Linear Algebra and Solvers
4. First Lab: Assembling a Code

What I Need From You

- Tell me if you do not understand
- Tell me if an example does not work
- Suggest better wording or figures
- Followup problems at petsc-maint@mcs.anl.gov

Ask Questions!!!

- Helps **me** understand what you are missing
- Helps **you** clarify misunderstandings
- Helps **others** with the same question

How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov

How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov

How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov

How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov

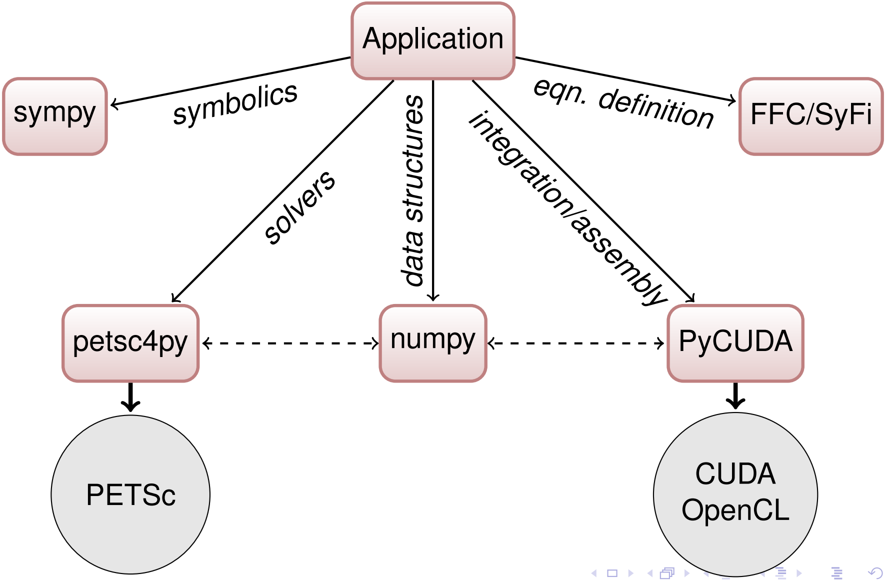
New Model for Scientific Software

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

New Model for Scientific Software



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???**

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. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS

Location and Retrieval

"Where's the Tarball"

- Version Control
    - Mercurial, Git, Subversion
- Hosting
    - BitBucket, GitHub, Launchpad
- Community involvement
    - arXiv, PubMed

Distributed Version Control

- CVS/SVN manage a single repository
    - Versioned data
    - Local copy for modification and checkin
- Mercurial manages many repositories
    - Identified by URLs
    - No one *Master*
- Repositories communicate by ChangeSets
    - Use `push` and `pull` to move changesets
    - Can move arbitrary changes with *patch queues*

Project Workflow

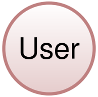


Figure: Single Repository

Project Workflow

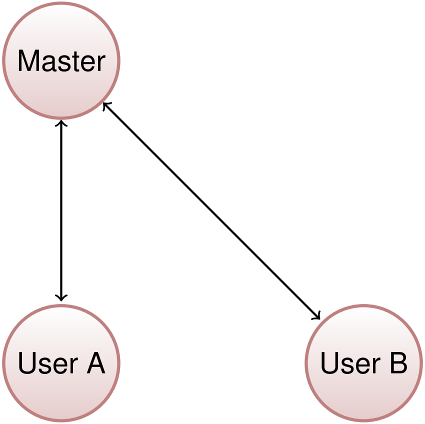


Figure: Master Repository with User Clones

Project Workflow

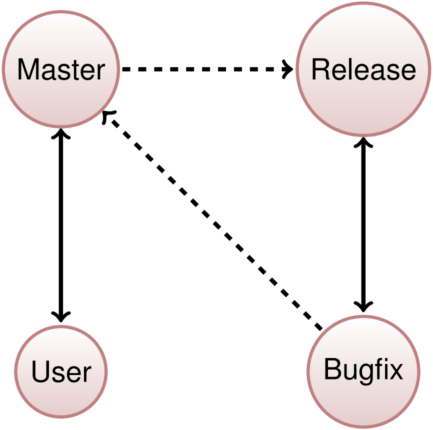


Figure: Project with Release and Bugfix Repositories

Outline

1 Tools and Infrastructure
- Version Control
- Configuration and Build
- PETSc
- numpy
- sympy
- petsc4py
- PyCUDA
- FEniCS

Configuration and Build

“It won’t run on my iPhone”

- Portability
  - PETSc BuildSystem, autoconf
- Dependencies
  - Does this work with UnsupportedGradStudentAMG?
- Configurable build
  - Build must integrate with the configuration system
  - CMake, SCons

BuildSystem

Provides tools for Configuration and Build

- Dependency tracking and analysis
- Package management and hierarchy
- Library of standard tests
- Standard build rules
- Automatic package build and integration

http://petsc.cs.iit.edu/petsc/BuildSystem
http://petsc.cs.iit.edu/petsc/SimpleConfigure

Configure
Modules

`BuildSystem.config.base` configures a specific functionality

- Entry points:
  - `setupHelp()`
  - `setupDependencies()`
  - `configure()`
- Builtin capabilities:
  - Preprocessing, compilation, linking, running
  - Manages languages
  - Checks for executables
- Output types:
  - Define, typedef, or prototype
  - Make macro or rule
  - Substitution (old-style)

Configure

Framework

`BuildSystem.config.framework` manages the configure run

- Manages configure modules
  - Dependencies with DAG, `require()`
  - Options table
  - Initialization, run, cleanup
- Outputs
  - Configure headers and log
  - Make variable and rules
  - Pickled configure tree

Configure
Third Party Packages

`BuildSystem.config.package` manages other packages

- `BuildSystem/config/packages/*` examples (MPI, FIAT, etc.)
- Standard location and install hooks
- Standard header and library tests
- Uniform interface for parameter retrieval
- Special support for GNU packages

Configure

Build Integration

A module can declare a dependency using:

```
fw       = self.framework
self.mpi = fw.require('config.packages.MPI', self)
```

so that MPI is configured before `self`. Information is retrieved during `configure()`:

```
if self.mpi.found:
  include.extend(self.mpi.include)
  libs.extend(self.mpi.lib)
```

Configure
Build Integration

A module can declare a dependency using:

```
fw       = self.framework
self.mpi = fw.require('config.packages.MPI', self)
```

so that MPI is configured before `self`. Information is retrieved during `configure()`:

```
if self.mpi.found:
  include.extend(self.mpi.include)
  libs.extend(self.mpi.lib)
```

Configure
Build Integration

A build system can acquire the information using:

```
class ConfigReader(script.Script):
  def __init__(self):
    import RDict
    argDB = RDict.RDict(None, None, 0, 0)
    argDB.saveFilename = os.path.join('path','RDict.db')
    argDB.load()
    script.Script.__init__(self, argDB = argDB)
    return

  def getMPIModule(self):
    self.setup()
    fw  = self.loadConfigure()
    mpi = fw.require('config.packages.MPI', None)
    return mpi
```

Make

GNU Make automates a package build

- Has a single predicate, older-than
- Executes shell code for actions
- PETSc has support for
  - configuration integration
  - automatic compilation
- Alternatives
  - SCons
  - CMake

builder

Simple replacement for GNU make

- Excellent configure integration
- User-defined predicates
- Dependency analysis and tracking
- Python actions
- Support for test execution

builder

Two Interfaces

The simple interface handles the entire build:

```
./config/builder.py
```

A more flexible front end allows finer control:

```
./config/builder2.py help [command]
./config/builder2.py clean
./config/builder2.py stubs fortran
./config/builder2.py build [src/snes/interface/snesj.c]
./config/builder2.py check [src/snes/examples/tutorials/ex10.c]
```

Testing

“They are identical in the eyeball norm”

- Unit tests
  - cppUnit
- Regression tests
  - buildbot
- Benchmarks
  - Cigma

Outline

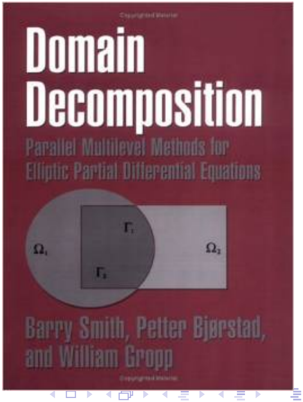
1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS

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


Domain Decomposition
Parallel Multilevel Methods for Elliptic Partial Differential Equations
Ω_1
Γ_1
Γ_2
Ω_2
Barry Smith, Petter Bjørstad, and William Gropp

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

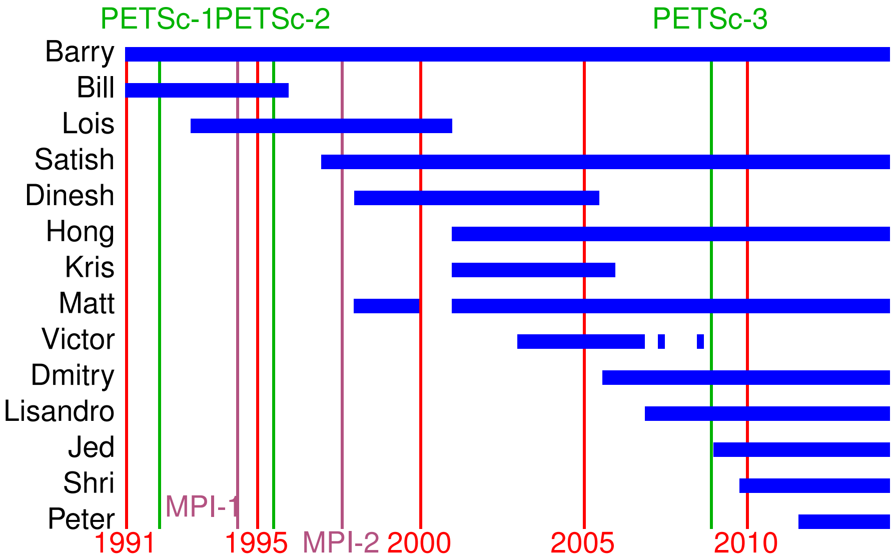
- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov

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

Timeline


PETSc-1
PETSc-2
PETSc-3
Barry
Bill
Lois
Satish
Dinesh
Hong
Kris
Matt
Victor
Dmitry
Lisandro
Jed
Shri
Peter
MPI-1
1991
1995
MPI-2
2000
2005
2010

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

Outline

1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS

numpy

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

1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS

sympy

sympy is useful for symbolic manipulation

- Interacts with numpy
- Derivatives and integrals
- Series expansions
- Equation simplification
- Small and open source

sympy
Example of Series Transform

Create the shifted polynomial

$$\sum_{i=0}^{order} \frac{c_i}{i!}(x-a)^i$$

```python
def constructShiftedPolynomial(order):
  from sympy import Symbol, collect, diff, limit
  from sympy import factorial as f
  c = [Symbol('c'+str(i)) for i in range(order)]
  g = sum([c[i]*(x-a)**i/f(i) for i in range(order)])
  # Convert to a monomial
  g = collect(g.expand(), x)
  return c, g
```

sympy

Example of Series Transform

Here is the shifted polynomial for order 5:

```
c0 - a*c1 + c2*a**2/2 - c3*a**3/6 + c4*a**4/24
+ x*(c1 - a*c2 + c3*a**2/2 - c4*a**3/6)
+ x**2*(c2/2 - a*c3/2 + c4*a**2/4)
+ x**3*(c3/6 - a*c4/6)
+ c4*x**4/24
```

sympy
Example of Series Transform

Construct matrix transform from

$$\sum_{i=0}^{order} \frac{c_i}{i!}(x-a)^i \quad \text{to} \quad \sum_{i=0}^{order} \frac{c_i}{i!}x^i$$

```
def constructTransformMatrix(order = 5):
  from sympy import diff, limit
  c, g = constructShiftedPolynomial(order, debug)
  M = []
  for o in range(order):
    exp = g.diff(x, o).limit(x, 0)
    M.append([exp.diff(c[p]) for p in range(order)])
  return M
```

sympy
Example of Series Transform

Here is the transform matrix M:

$$\begin{pmatrix} 1 & -a & \frac{a^2}{2} & -\frac{a^3}{6} & \frac{a^4}{24} \\ 0 & 1 & -a & \frac{a^2}{2} & -\frac{a^3}{6} \\ 0 & 0 & 1 & -a & \frac{a^2}{2} \\ 0 & 0 & 0 & 1 & -a \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Outline

1 Tools and Infrastructure
- Version Control
- Configuration and Build
- PETSc
- numpy
- sympy
- petsc4py
- PyCUDA
- FEniCS

petsc4py

petcs4py 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 petscp4y`
  - 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`

Outline

1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - 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

Code Template

```
<%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}] = {${','.join(['3.0']*(dim*dim))}};
  float product       = 0.0;
  const int Ooffset   = gridIdx*${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;
```

Rendering a Template

We render code template into strings using a dictionary of inputs.

```
args = {'dim':              self.dim,
        'numLocalElements': 1,
        'numThreads':       self.threadBlockSize}
kernelTemplate = self.getKernelTemplate()
gpuCode = kernelTemplate.render(isGPU = True, **args)
cpuCode = kernelTemplate.render(isGPU = False, **args)
```

GPU Source Code

```
__global__ void kernel(float *output) {
  const int      gridIdx = blockIdx.x + blockIdx.y*gridDim.x;
  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[2] * K[2];
  product += G[3] * K[3];
  product += G[4] * K[4];
  product += G[5] * K[5];
  product += G[6] * K[6];
  product += G[7] * K[7];
  product += G[8] * K[8];
  output[Ooffset+idx] = product;
  return;
}
```

CPU Source Code

```
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[2] * K[2];
  product += G[3] * K[3];
  product += G[4] * K[4];
  product += G[5] * K[5];
  product += G[6] * K[6];
  product += G[7] * K[7];
  product += G[8] * K[8];
  output[Ooffset+idx] = product;
      }
    }
```

Creating a Module

CPU:

```
# 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')
```

Executing a Module

```
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(iters):
  self.kernel(cuda.Out(output),
              block = blockDim, grid = grid)
end.record()
end.synchronize()
gpuTimes.append(start.time_till(end)*1e-3/iters)
```

Outline

1 Tools and Infrastructure
- Version Control
- Configuration and Build
- PETSc
- numpy
- sympy
- petsc4py
- PyCUDA
- FEniCS

FIAT

Finite Element Integrator And Tabulator by Rob Kirby

`http://www.fenics.org/fiat`

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

User can build arbitrary elements specifying the Ciarlet triple (K, P, P')

FIAT is part of the FEniCS project, as is the PETSc Sieve module

FFC

FFC is a compiler for variational forms by Anders Logg.

Here is a mixed-form Poisson equation:

$$a((\tau, w), (\sigma, u)) = L((\tau, w)) \qquad \forall (\tau, w) \in V$$

where

$$\begin{aligned} a((\tau, w), (\sigma, u)) &= \int_\Omega \tau\sigma - \nabla\cdot\tau u + w\nabla\cdot u \, dx \\ L((\tau, w)) &= \int_\Omega wf \, dx \end{aligned}$$

FFC
Mixed Poisson

```
shape = "triangle"

BDM1 = FiniteElement("Brezzi-Douglas-Marini",shape,1)
DG0  = FiniteElement("Discontinuous Lagrange",shape,0)

element      = BDM1 + DG0
(tau, w)     = TestFunctions(element)
(sigma, u)   = TrialFunctions(element)

a = (dot(tau, sigma) - div(tau)*u + w*div(sigma))*dx

f = Function(DG0)
L = w*f*dx
```

FFC

Here is a discontinuous Galerkin formulation of the Poisson equation:

$$a(v, u) = L(v) \qquad \forall v \in V$$

where

$$\begin{aligned}
a(v, u) &= \int_\Omega \nabla u \cdot \nabla v \, dx \\
&+ \sum_S \int_S - < \nabla v > \cdot [[u]]_n - [[v]]_n \cdot < \nabla u > -(\alpha/h) vu \, dS \\
&+ \int_{\partial\Omega} -\nabla v \cdot [[u]]_n - [[v]]_n \cdot \nabla u - (\gamma/h) vu \, ds \\
L(v) &= \int_\Omega vf \, dx
\end{aligned}$$

FFC
DG Poisson

```
DG1 = FiniteElement("Discontinuous Lagrange",shape,1)
v = TestFunctions(DG1)
u = TrialFunctions(DG1)
f = Function(DG1)
g = Function(DG1)
n = FacetNormal("triangle")
h = MeshSize("triangle")
a = dot(grad(v), grad(u))*dx
  - dot(avg(grad(v)), jump(u, n))*dS
  - dot(jump(v, n), avg(grad(u)))*dS
  + alpha/h*dot(jump(v, n) + jump(u, n))*dS
  - dot(grad(v), jump(u, n))*ds
  - dot(jump(v, n), grad(u))*ds
  + gamma/h*v*u*ds
L = v*f*dx + v*g*ds
```

Big Picture

- Usability is paramount
  - Need community by-in
  - Need complete workflow
- Leverage existing systems
  - Adoption is much easier with the familiar
  - arXiv, package managers

Outline

1. Tools and Infrastructure

2. GPU Computing
   - FEM-GPU
   - PETSc-GPU

3. Linear Algebra and Solvers

4. First Lab: Assembling a Code

5. Second Lab: Debugging and Performance Benchmarking

Collaborators

- Dr. Andy Terrel (FEniCS)
  - Dept. of Computer Science, University of Texas
  - Texas Advanced Computing Center, University of Texas
- Prof. Andreas Klöckner (PyCUDA)
  - Courant Institute of Mathematical Sciences, New York University
- Dr. Brad Aagaard (PyLith)
  - United States Geological Survey, Menlo Park, CA
- Dr. Charles Williams (PyLith)
  - GNS Science, Wellington, NZ

Outline

2 GPU Computing
- FEM-GPU
- PETSc-GPU

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

Analytic Flexibility

Laplacian

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{1}$$

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

Analytic Flexibility

Laplacian

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{1}$$

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

Analytic Flexibility

Linear Elasticity

$$\frac{1}{4}\int_{\mathcal{T}}\left(\nabla\vec{\phi}_i(\mathbf{x})+\nabla^T\vec{\phi}_i(\mathbf{x})\right):\left(\nabla\vec{\phi}_j(\mathbf{x})+\nabla\vec{\phi}_j(\mathbf{x})\right)d\mathbf{x} \tag{2}$$

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

Analytic Flexibility

Linear Elasticity

$$\frac{1}{4}\int_{\mathcal{T}}\left(\nabla\vec{\phi}_i(\mathbf{x})+\nabla^T\vec{\phi}_i(\mathbf{x})\right):\left(\nabla\vec{\phi}_j(\mathbf{x})+\nabla\vec{\phi}_j(\mathbf{x})\right)d\mathbf{x} \tag{2}$$

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

Analytic Flexibility

Full Elasticity

$$\frac{1}{4}\int_{\mathcal{T}} \left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : C : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right) d\mathbf{x} \tag{3}$$

```
element  = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                         (dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release

Analytic Flexibility

Full Elasticity

$$\frac{1}{4}\int_{\mathcal{T}} \left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : C : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right) d\mathbf{x} \tag{3}$$

```
element  = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                         (dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release

Analytic Flexibility

Full Elasticity

$$\frac{1}{4}\int_{\mathcal{T}}\left(\nabla\vec{\phi}_i(\mathbf{x}) + \nabla^T\vec{\phi}_i(\mathbf{x})\right) : C : \left(\nabla\vec{\phi}_j(\mathbf{x}) + \nabla\vec{\phi}_j(\mathbf{x})\right)\, d\mathbf{x} \tag{3}$$

```
element  = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                         (dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release

Form Decomposition

Element integrals are decomposed into <u>analytic</u> and <u>geometric</u> parts:

$$
\begin{aligned}
& \int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} & (4)\\
= \quad & \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} & (5)\\
= \quad & \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| d\mathbf{x} & (6)\\
= \quad & \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} d\mathbf{x} & (7)\\
= \quad & G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma} & (8)
\end{aligned}
$$

Coefficients are also put into the geometric part.

Form Decomposition

Additional fields give rise to multilinear forms.

$$
\begin{aligned}
& \int_{\mathcal{T}} \phi_i(\mathbf{x}) \cdot \left(\phi_k(\mathbf{x}) \nabla \phi_j(\mathbf{x})\right) dA && (9)\\
= & \int_{\mathcal{T}} \phi_i^\beta(\mathbf{x}) \left(\phi_k^\alpha(\mathbf{x}) \frac{\partial \phi_j^\beta(\mathbf{x})}{\partial x_\alpha}\right) dA && (10)\\
= & \int_{\mathcal{T}_{\mathrm{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| dA && (11)\\
= & \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} dA && (12)\\
= & G^{\alpha\gamma}(\mathcal{T}) K^{ijk}_{\alpha\gamma} && (13)
\end{aligned}
$$

The index calculus is fully developed by Kirby and Logg in A Compiler for Variational Forms.

Form Decomposition

Isoparametric Jacobians also give rise to multilinear forms

$$
\begin{aligned}
& \int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) dA && (14)\\
= & \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} dA && (15)\\
= & \int_{\mathcal{T}_{\mathrm{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| dA && (16)\\
= & |J| \int_{\mathcal{T}_{\mathrm{ref}}} \phi_k J_k^{\beta\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l J_l^{\gamma\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA && (17)\\
= & J_k^{\beta\alpha} J_l^{\gamma\alpha} |J| \int_{\mathcal{T}_{\mathrm{ref}}} \phi_k \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA && (18)\\
= & G_{kl}^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ijkl} && (19)
\end{aligned}
$$

A different space could also be used for Jacobians

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
```

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

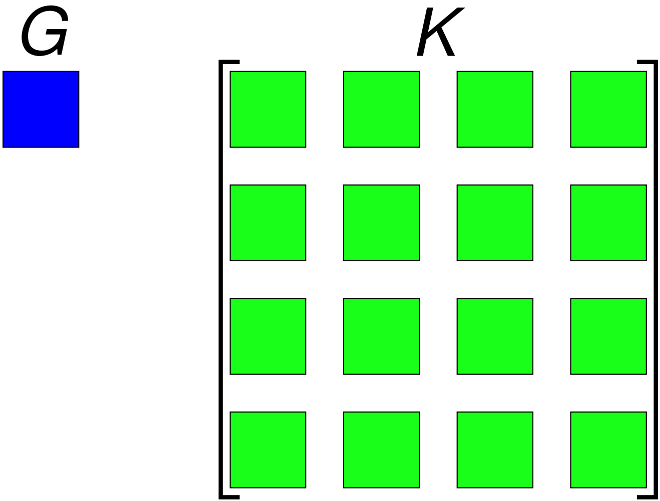


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Basic Contraction

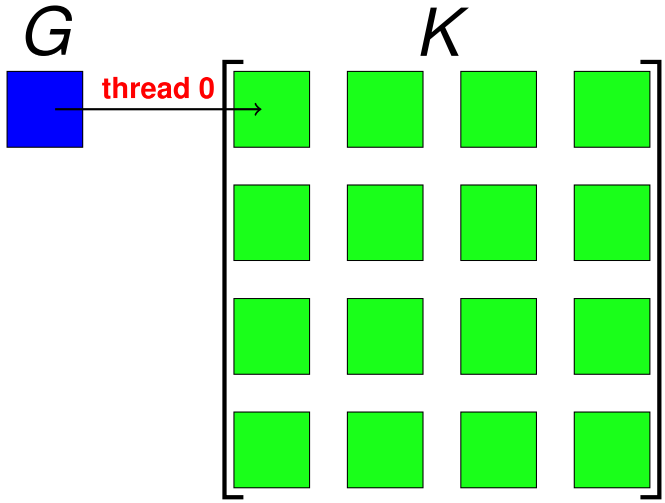


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

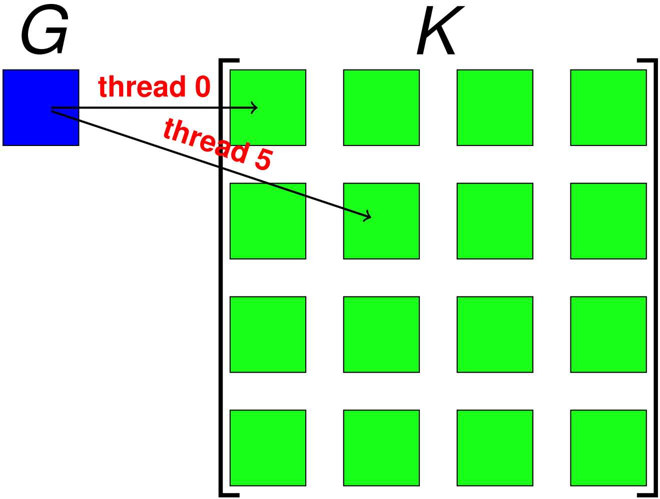


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility
Basic Contraction

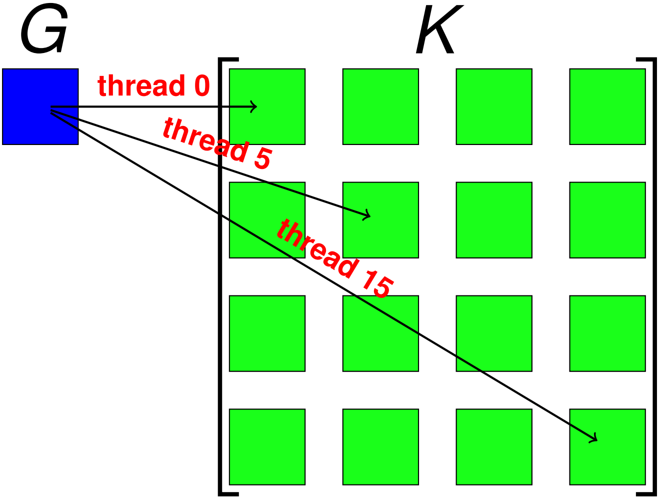


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

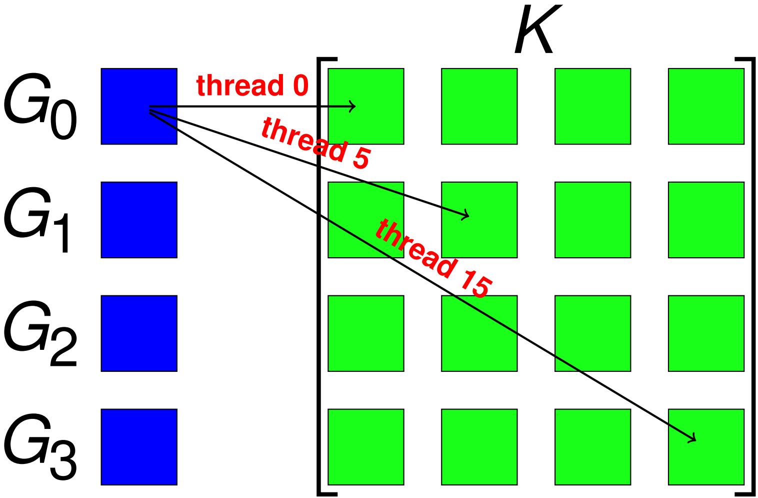


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

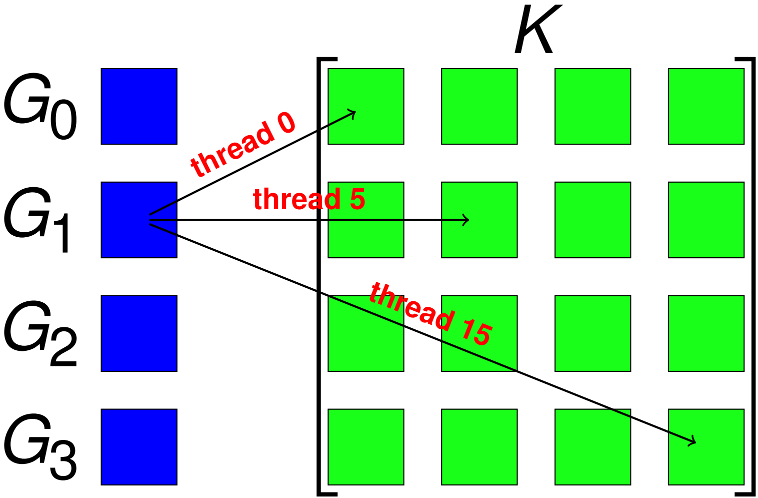


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

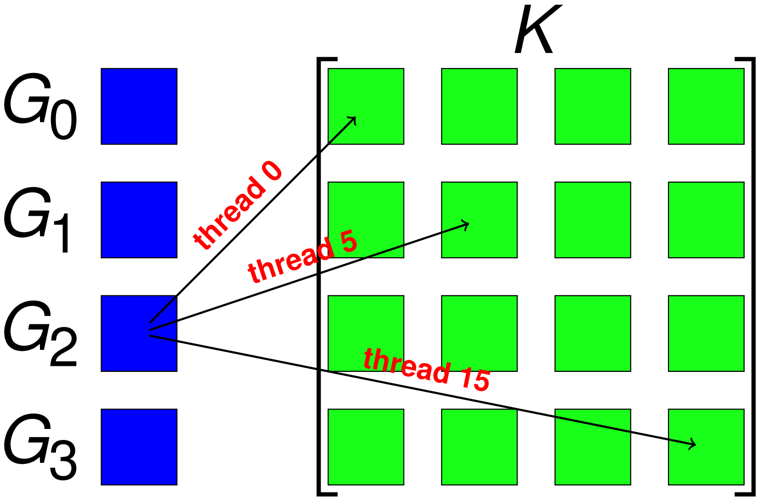


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

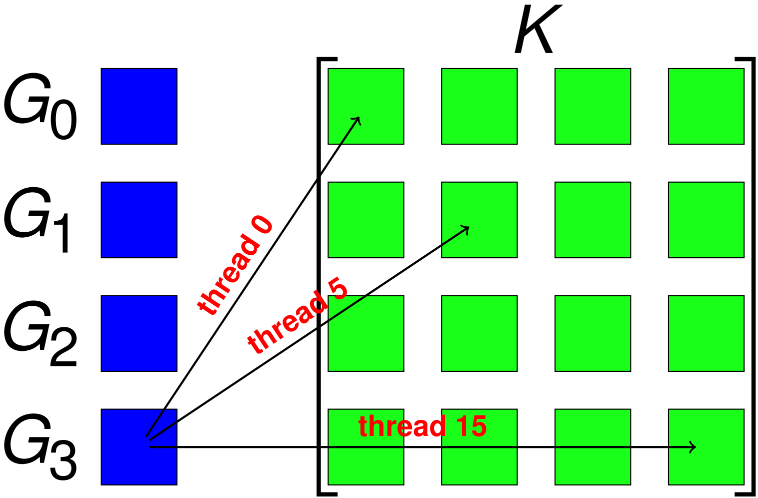
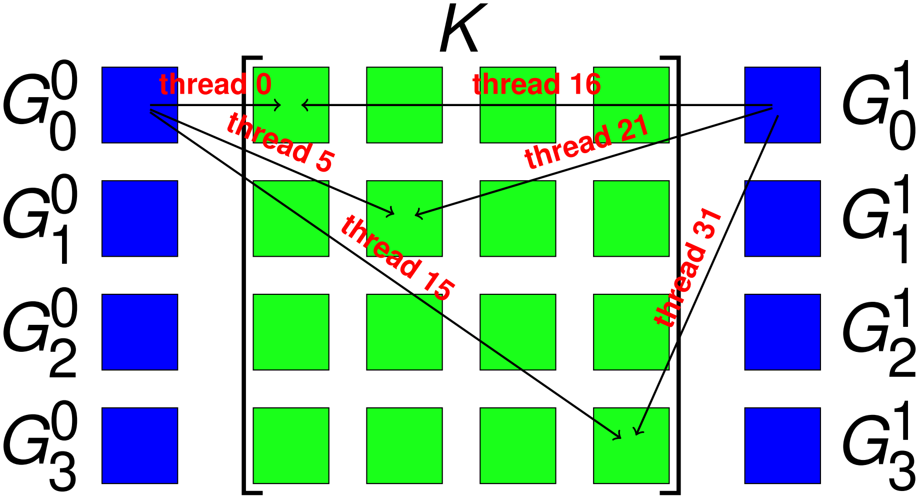


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

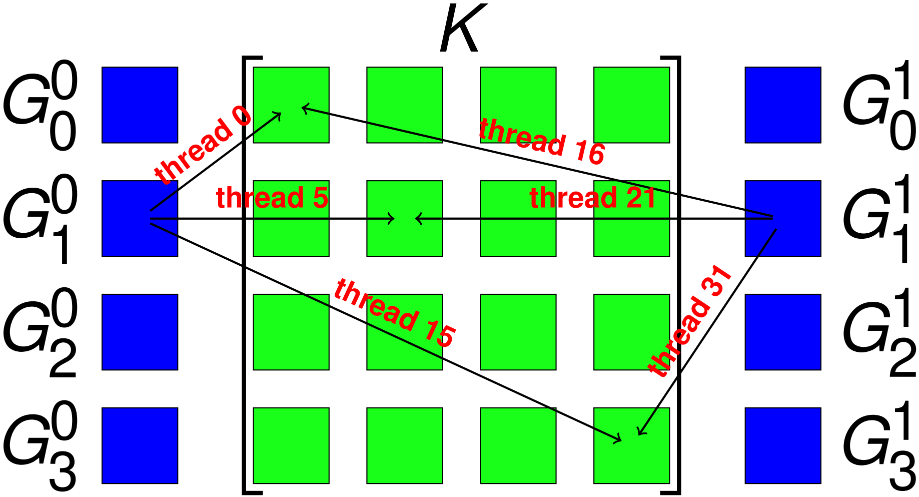
Computational Flexibility

Concurrent Elements

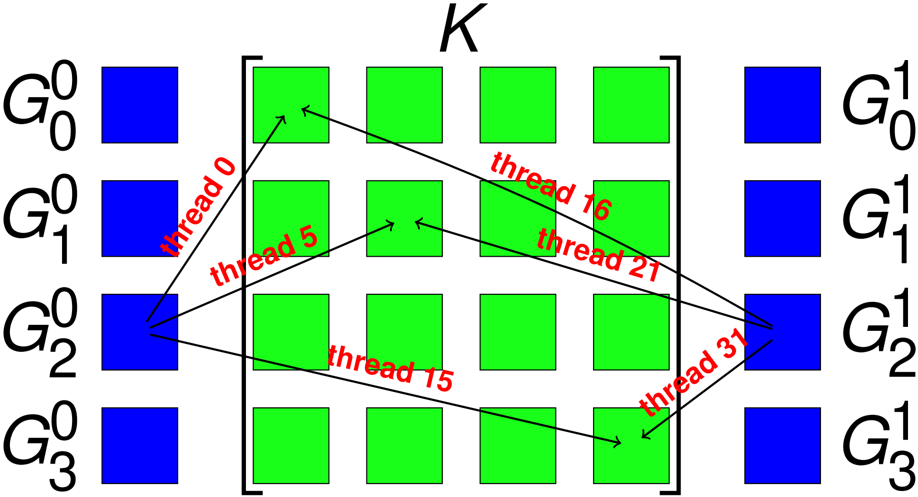


Computational Flexibility

Concurrent Elements

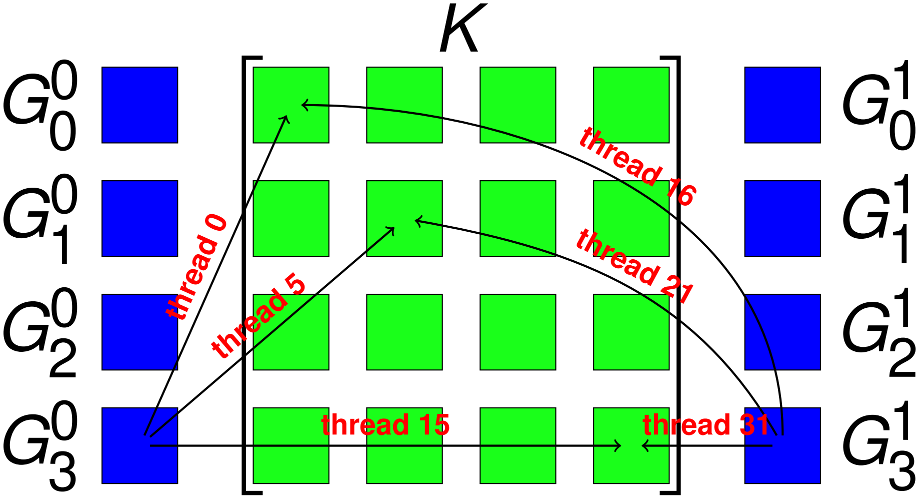


Computational Flexibility
Concurrent Elements



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;
```

Code Template

```
<%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}]  = {${','.join(['3.0']*(dim*dim))}};
  float product        = 0.0;
  const int Ooffset    = gridIdx*${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;
```

Rendering a Template

We render code template into strings using a dictionary of inputs.

```
args = {'dim':              self.dim,
        'numLocalElements': 1,
        'numThreads':       self.threadBlockSize}
kernelTemplate = self.getKernelTemplate()
gpuCode = kernelTemplate.render(isGPU = True, **args)
cpuCode = kernelTemplate.render(isGPU = False, **args)
```

GPU Source Code

```
__global__ void kernel(float *output) {
  const int       gridIdx = blockIdx.x + blockIdx.y*gridDim.x;
  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[2] * K[2];
  product += G[3] * K[3];
  product += G[4] * K[4];
  product += G[5] * K[5];
  product += G[6] * K[6];
  product += G[7] * K[7];
  product += G[8] * K[8];
  output[Ooffset+idx] = product;
  return;
}
```

CPU Source Code

```
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[2] * K[2];
  product += G[3] * K[3];
  product += G[4] * K[4];
  product += G[5] * K[5];
  product += G[6] * K[6];
  product += G[7] * K[7];
  product += G[8] * K[8];
  output[Ooffset+idx] = product;
      }
    }
```

Creating a Module

CPU:

```
# 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')
```

Executing a Module

```
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(iters):
  self.kernel(cuda.Out(output),
              block = blockDim, grid = grid)
end.record()
end.synchronize()
gpuTimes.append(start.time_till(end)*1e-3/iters)
```

Element Matrix Formation

- Element matrix K is now made up of small tensors
- Contract all tensor elements with each the geometry tensor $G(\mathcal{T})$

3 0 0 0	0 -1 0 0	1 1 0 0	-4 -4 0 0	0 4 0 0	0 0 0 0
0 0 -1 0	0 0 0 3	0 0 1 1	0 0 0 0	0 0 4 0	0 0 -4 -4
1 0 1 0	0 1 0 1	3 3 3 3	-4 0 -4 0	0 0 0 0	0 -4 0 -4
-4 0 -4 0	0 0 0 0	-4 -4 0 0	8 4 4 8	0 -4 -4 -8	0 4 4 0
0 0 4 0	0 4 0 0	0 0 0 0	0 -4 -4 -8	8 4 4 8	-8 -4 -4 0
0 0 0 0	0 -4 0 -4	0 0 -4 -4	0 4 4 0	-8 -4 -4 0	8 4 4 8

Mapping $G^{\alpha\beta}K^{ij}_{\alpha\beta}$ to the GPU

Problem Division

For N elements, map blocks of N_L elements to each Thread Block (TB)

- Launch `grid` must be $g_x \times g_y = {}^{N}/_{N_L}$
- TB grid will depend on the specific algorithm
- Output is size $N_{\mathrm{basis}} \times N_{\mathrm{basis}} \times N_L$

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

```
__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

Mapping $G^{\alpha\beta} K^{ij}_{\alpha\beta}$ to the GPU

Memory Movement

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

Memory Bandwidth

Superior GPU memory bandwidth is due to both

bus width and clock speed.

	CPU	GPU
Bus Width (bits)	64	512
Bus Clock Speed (MHz)	400	1600
Memory Bandwidth (GB/s)	3	102
Latency (cycles)	240	600

Tesla always accesses blocks of 64 or 128 bytes

Mapping $G^{\alpha\beta}K^{ij}_{\alpha\beta}$ to the GPU

Reduction

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

Strategy #1
TB Division

Each thread computes an entire element matrix, so that

$$\mathrm{blockDim} = \left({}^{N_L}\!/\!_{N_B}, 1, 1\right)$$

We will see that there is little opportunity to overlap computation and memory access

Strategy #1
Analytic Part

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

```
__shared__ float K[${dim*dim*numBasisFuncs*numBasisFuncs}];

${fm.moveArray('K', 'analytic',
               dim*dim*numBasisFuncs*numBasisFuncs, '', 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

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

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

Strategy #1
Output

We write element matrices out contiguously by TB

```
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];
}
```

Strategy #1

Contraction

```
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)
```

Strategy #2
TB Division

Each thread computes a single element of an element matrix, so that

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

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

```
const int Kidx    = threadIdx.x + threadIdx.y*${numBasisFuncs}; // This is
const int idx     = Kidx + threadIdx.z*${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}];
%   endfor
% endfor
```

Strategy #2
Geometry

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

```
const int        Goffset = gridIdx*${dim*dim*numLocalElements};
__shared__ float G[${dim*dim*numLocalElements}];

${fm.moveArray('G', 'geometry', dim*dim*numLocalElements,
               'Goffset', numThreads)}
__syncthreads();
```

Strategy #2

Output

- 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

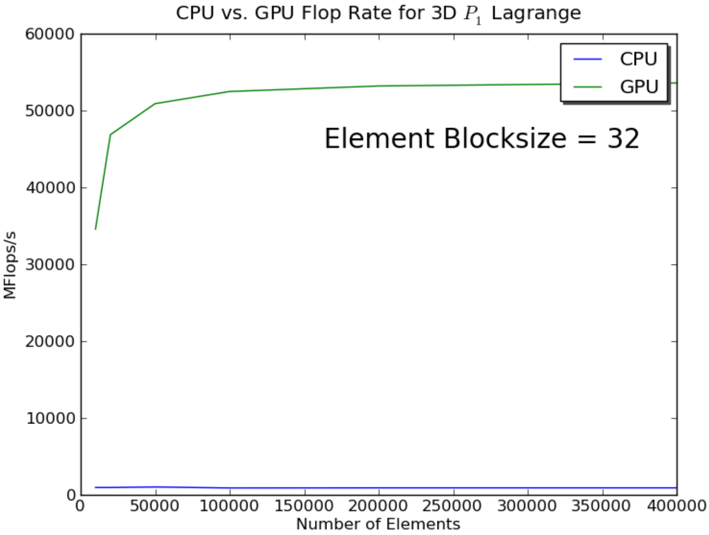
```
const int matSize = numBasisFuncs*numBasisFuncs;
const int Eoffset = gridIdx*matSize*numLocalElements;

if (interleaved) {
  float             product = 0.0;
  const int         elemOff = idx*matSize;
} else {
  float             product[numLocalElements/numBlockElements];
  const int         elemOff = idx*matSize*numBlockElements;
}
```

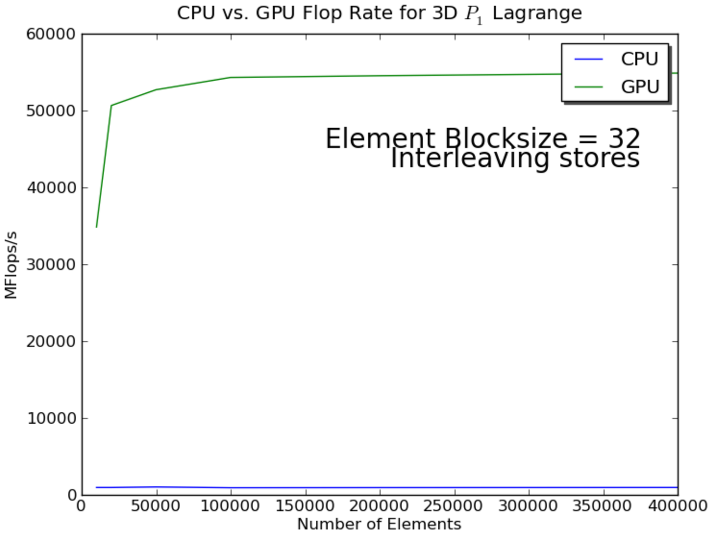
Strategy #2
Contraction

```
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]
```

Results
GTX 285

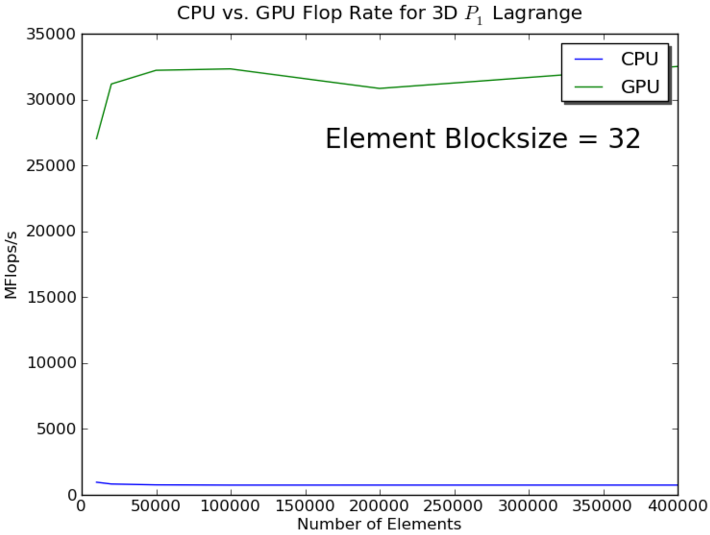


Results
GTX 285



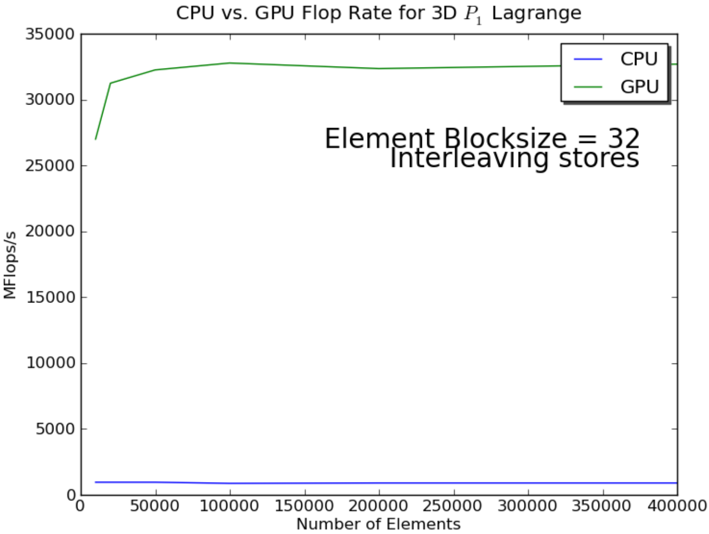
Results

GTX 285, 2 Simultaneous Elements

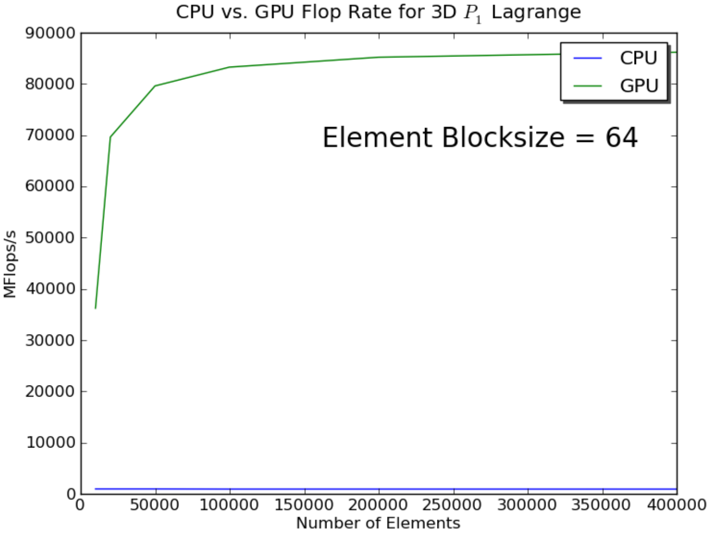


Results

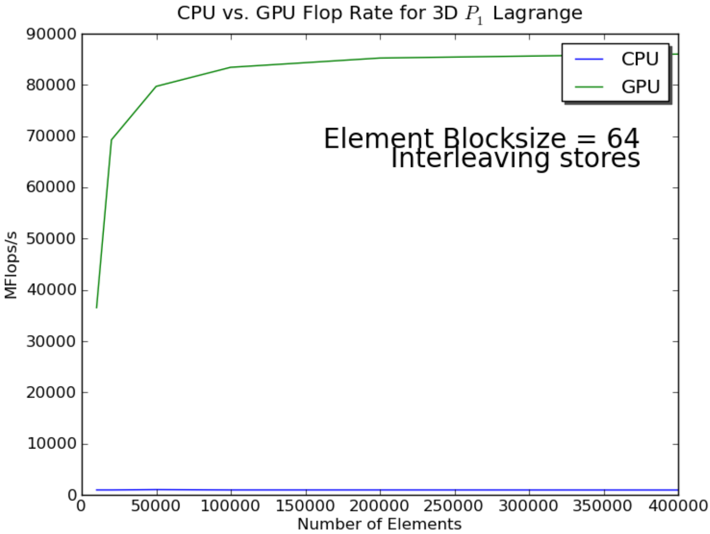
GTX 285, 2 Simultaneous Elements



Results
GTX 285

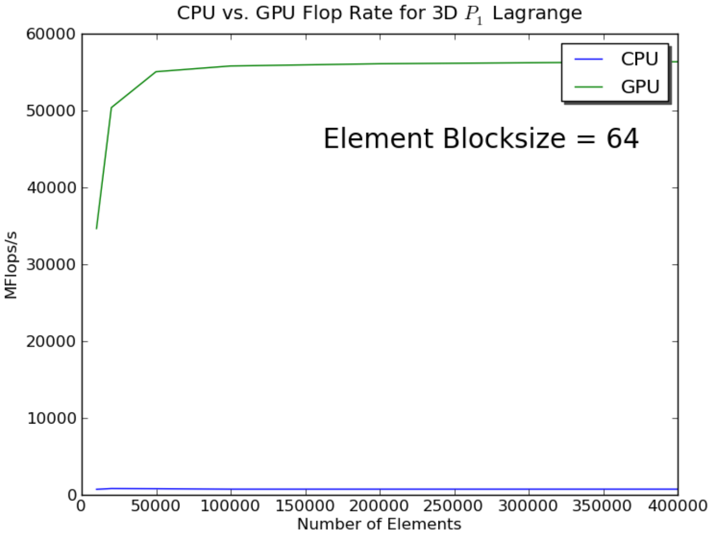


Results
GTX 285



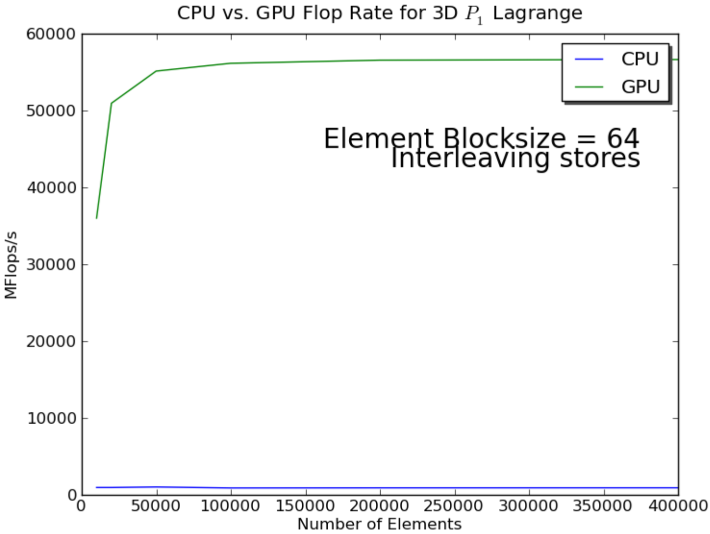
Results

GTX 285, 2 Simultaneous Elements



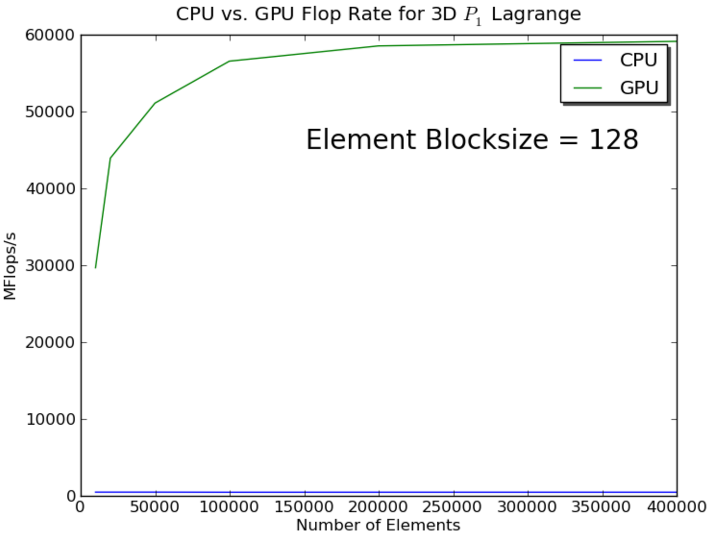
Results

GTX 285, 2 Simultaneous Elements

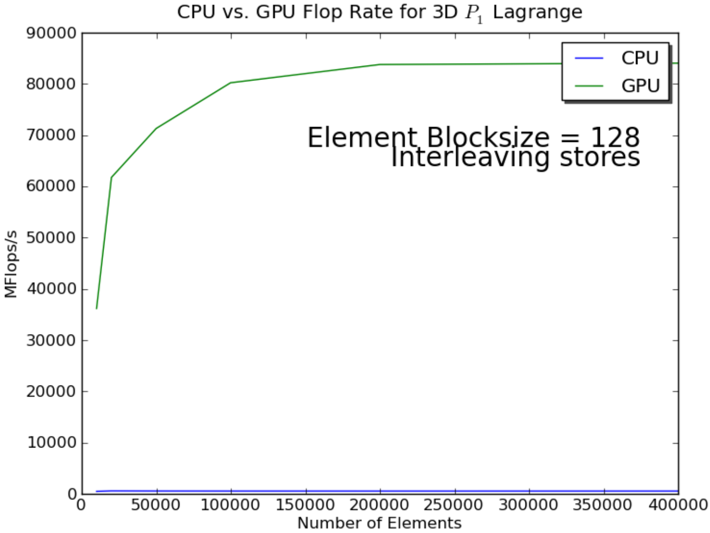


Results

GTX 285

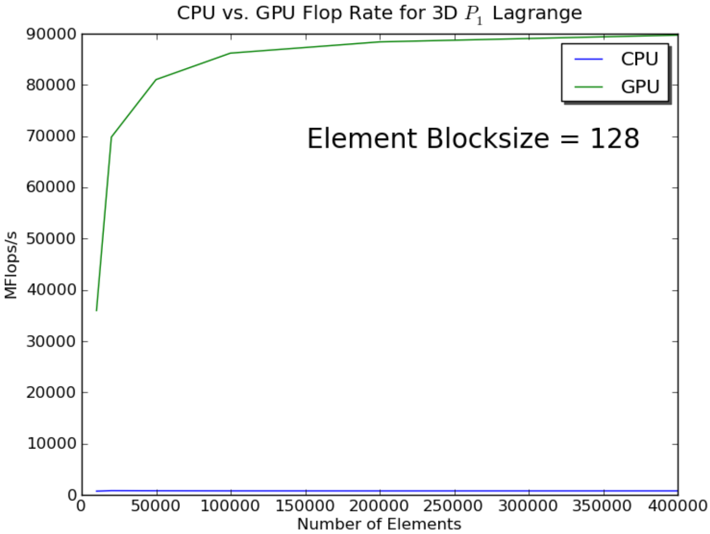


Results
GTX 285



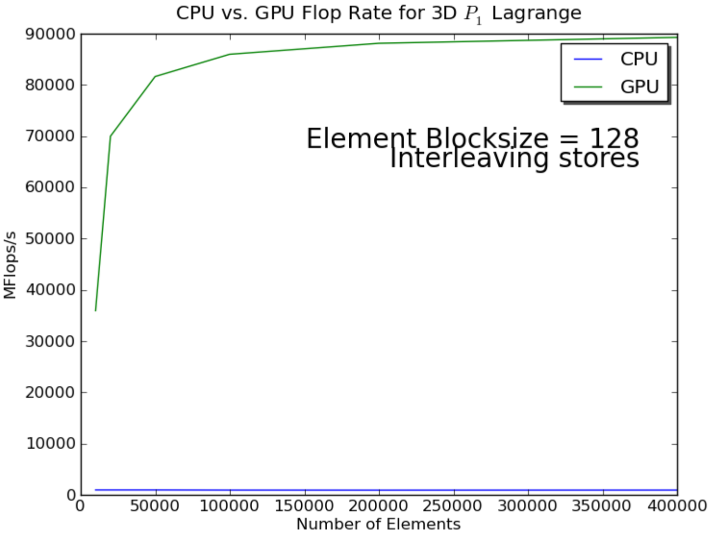
Results

GTX 285, 2 Simultaneous Elements



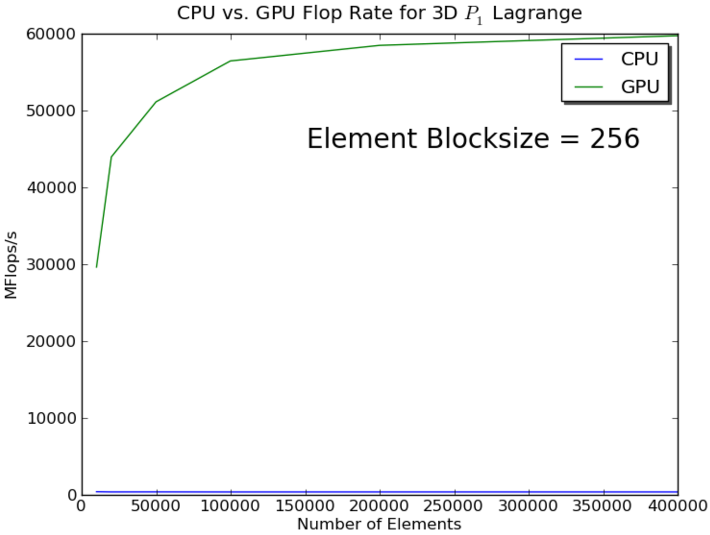
Results

GTX 285, 2 Simultaneous Elements



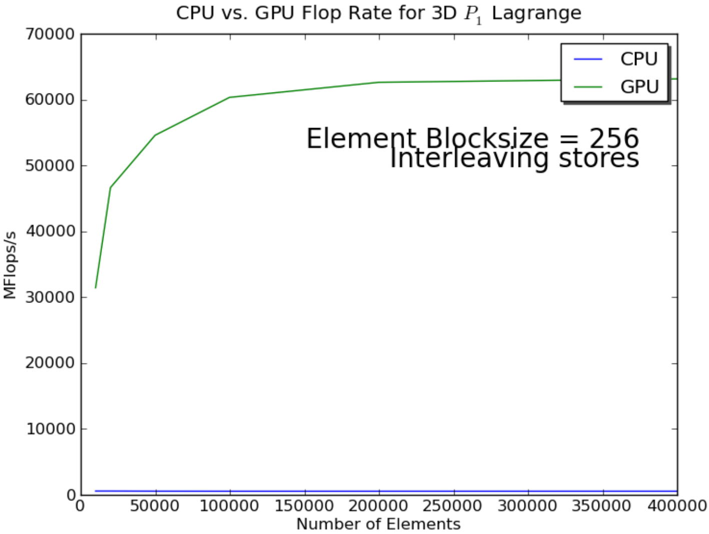
Results

GTX 285, 2 Simultaneous Elements



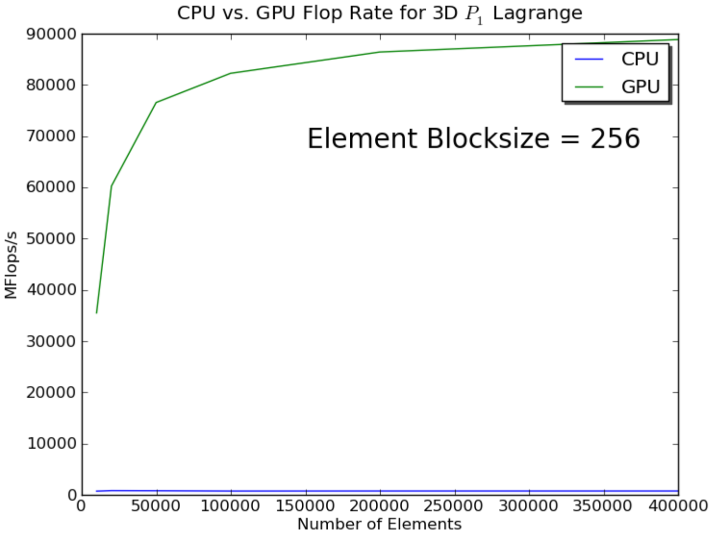
Results

GTX 285, 2 Simultaneous Elements



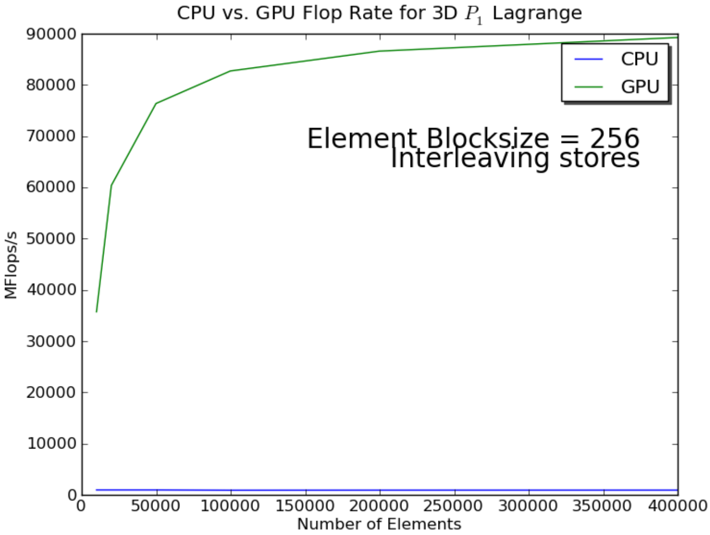
Results

GTX 285, 4 Simultaneous Elements



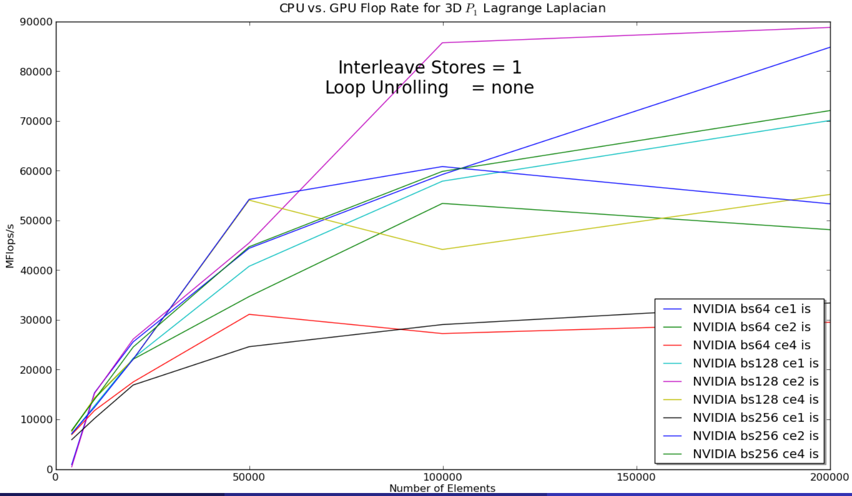
Results

GTX 285, 4 Simultaneous Elements



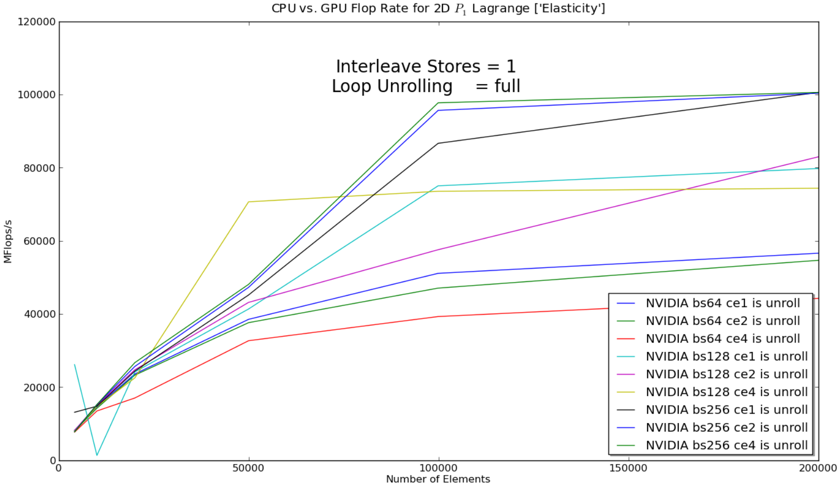
Performance

Influence of Element Batch Sizes



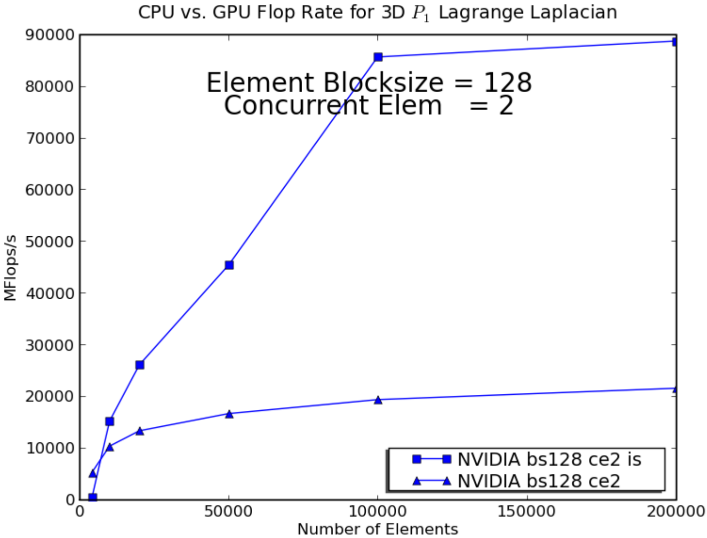
Performance

Influence of Element Batch Sizes



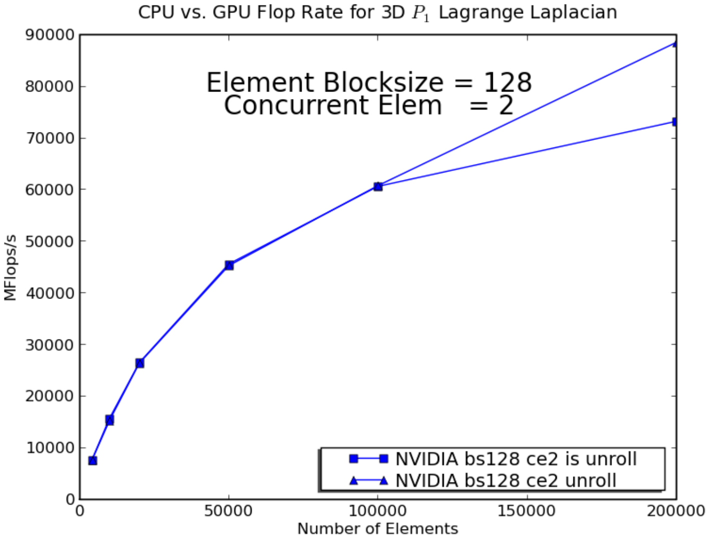
Performance

Influence of Code Structure



Performance

Influence of Code Structure



Outline

2 GPU Computing

- FEM-GPU
- PETSc-GPU

Thrust

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

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`)

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

Memory Coherence

PETSc Objects now hold a coherence flag

PETSC_CUDA_UNALLOCATED	No allocation on the GPU
PETSC_CUDA_GPU	Values on GPU are current
PETSC_CUDA_CPU	Values on CPU are current
PETSC_CUDA_BOTH	Values on both are current

Table: Flags used to indicate the memory state of a PETSc CUDA **Vec** object.

MATAIJCUDA

Also define new **Mat** implementations

- Uses Cusp for data storage and operations on GPU
- Supports full PETSc **Mat** interface, some ops on CPU
- Can be activated at runtime, `-mat_type aijcuda`
- Notice that parallel matvec necessitates off-GPU data transfer

Solvers

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

Installation

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
```

Outline

1 Tools and Infrastructure

2 GPU Computing

3 Linear Algebra and Solvers

- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit

4 First Lab: Assembling a Code

5 Second Lab: Debugging and Performance Benchmarking

Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit

Vector Algebra

What are PETSc vectors?

- Fundamental objects representing
  - solutions
  - right-hand sides
  - coefficients
- Each process locally owns a subvector of contiguous global data

Vector Algebra

How do I create vectors?

- VecCreate(MPI_Comm, Vec*)
- VecSetSizes(Vec, PetscIntn, PetscInt N)
- VecSetType(Vec, VecType typeName)
- VecSetFromOptions(Vec)
    - Can set the type at runtime

Vector Algebra

A PETSc Vec

- Supports all vector space operations
  - VecDot(), VecNorm(), VecScale()
- Has a direct interface to the values
  - VecGetArray(), VecGetArrayF90()
- Has unusual operations
  - VecSqrtAbs(), VecStrideGather()
- Communicates automatically during assembly
- Has customizable communication (**PetscSF**,**VecScatter**)

Parallel Assembly

Vectors and Matrices

- Processes may set an arbitrary entry
  - Must use proper interface
- Entries need not be generated locally
  - Local meaning the process on which they are stored
- PETSc automatically moves data if necessary
  - Happens during the assembly phase

Vector Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication

```
VecSetValues(Vec v, PetscInt n, PetscInt rows[],
             PetscScalar values[], InsertMode mode)
```

- Mode is either `INSERT_VALUES` or `ADD_VALUES`
- Two phases allow overlap of communication and computation
  - VecAssemblyBegin(Vecv)
  - VecAssemblyEnd(Vecv)

One Way to Set the Elements of a Vector

```
VecGetSize(x, &N);
MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
if (rank == 0) {
  val = 0.0;
  for(i = 0; i < N; ++i) {
    VecSetValues(x, 1, &i, &val, INSERT_VALUES);
    val += 10.0;
  }
}
/* These routines ensure that the data is
   distributed to the other processes */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

A Better Way to Set the Elements of a Vector

```
VecGetOwnershipRange(x, &low, &high);
val = low*10.0;
for(i = low; i < high; ++i) {
  VecSetValues(x, 1, &i, &val, INSERT_VALUES);
  val += 10.0;
}
/* No data will be communicated here */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

Selected Vector Operations

Function Name	Operation
VecAXPY(Vec y, PetscScalar a, Vec x)	$y = y + a * x$
VecAYPX(Vec y, PetscScalar a, Vec x)	$y = x + a * y$
VecWAYPX(Vec w, PetscScalar a, Vec x, Vec y)	$w = y + a * x$
VecScale(Vec x, PetscScalar a)	$x = a * x$
VecCopy(Vec y, Vec x)	$y = x$
VecPointwiseMult(Vec w, Vec x, Vec y)	$w_i = x_i * y_i$
VecMax(Vec x, PetscInt *idx, PetscScalar *r)	$r = \max r_i$
VecShift(Vec x, PetscScalar r)	$x_i = x_i + r$
VecAbs(Vec x)	$x_i = \|x_i\|$
VecNorm(Vec x, NormType type, PetscReal *r)	$r = \|\|x\|\|$

Working With Local Vectors

It is sometimes more efficient to directly access local storage of a `Vec`.

- PETSc allows you to access the local storage with
  - VecGetArray(Vec, double *[])
- You must return the array to PETSc when you finish
  - VecRestoreArray(Vec, double *[])
- Allows PETSc to handle data structure conversions
  - Commonly, these routines are fast and do not involve a copy

VecGetArray in C

```
Vec            v;
PetscScalar   *array;
PetscInt       n, i;

VecGetArray(v, &array);
VecGetLocalSize(v, &n);
PetscSynchronizedPrintf(PETSC_COMM_WORLD,
  "First element of local array is %f\n", array[0]);
PetscSynchronizedFlush(PETSC_COMM_WORLD);
for(i = 0; i < n; ++i) {
  array[i] += (PetscScalar) rank;
}
VecRestoreArray(v, &array);
```

VecGetArray in F77

```
#include "finclude/petsc.h"

      Vec            v;
      PetscScalar    array(1)
      PetscOffset    offset
      PetscInt       n, i
      PetscErrorCode ierr

      call VecGetArray(v, array, offset, ierr)
      call VecGetLocalSize(v, n, ierr)
      do i=1,n
        array(i+offset) = array(i+offset) + rank
      end do
      call VecRestoreArray(v, array, offset, ierr)
```

VecGetArray in F90

```
#include "finclude/petsc.h90"

      Vec              v;
      PetscScalar      pointer :: array(:)
      PetscInt         n, i
      PetscErrorCode   ierr


      call VecGetArrayF90(v, array, ierr)
      call VecGetLocalSize(v, n, ierr)
      do i=1,n
        array(i) = array(i) + rank
      end do
      call VecRestoreArrayF90(v, array, ierr)
```

VecGetArray in Python

```
with v as a:
  for i in range(len(a)):
    a[i] = 5.0*i
```

DMDAVecGetArray in C

```
DM              da;
Vec             v;
DMDALocalInfo  *info;
PetscScalar   **array;

DMDAVecGetArray(da, v, &array);
for(j = info->ys; j < info->ys+info->ym; ++j) {
  for(i = info->xs; i < info->xs+info->xm; ++i) {
    u       = x[j][i];
    uxx     = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
    uyy     = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
    f[j][i] = uxx + uyy;
  }
}
DMDAVecRestoreArray(da, v, &array);
```

Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit

Matrix Algebra

What are PETSc matrices?

- Fundamental objects for storing stiffness matrices and Jacobians
- Each process locally owns a contiguous set of rows
- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Matrix, etc.
- Supports structures for many packages
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack

How do I create matrices?

- MatCreate(MPI_Comm, Mat*)
- MatSetSizes(Mat, PetscIntm, PetscInt n, M, N)
- MatSetType(Mat, MatType typeName)
- MatSetFromOptions(Mat)
  - Can set the type at runtime
- MatSeqAIJPreallocation(Mat, PetscIntnz, const PetscInt nnz[])
- MatXAIJPreallocation(Mat, bs, dnz[], onz[], dnzu[], onzu[])
- MatSetValues(Mat, m, rows[], n, cols [], values [], InsertMode)
  - **MUST** be used, but does automatic communication

Matrix Polymorphism

The PETSc `Mat` has a single user interface,

- Matrix assembly
  - `MatSetValues()`
- Matrix-vector multiplication
  - `MatMult()`
- Matrix viewing
  - `MatView()`

but multiple underlying implementations.

- AIJ, Block AIJ, Symmetric Block AIJ,
- Dense
- Matrix-Free
- etc.

A matrix is defined by its interface, not by its data structure.

Matrix Assembly

- A three step process
    - Each process sets or adds values
    - Begin communication to send values to the correct process
    - Complete the communication
- MatSetValues(Matm, m, rows[], n, cols [], values [], mode)
    - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
    - Logically dense block of values
- Two phase assembly allows overlap of communication and computation
    - MatAssemblyBegin(Matm, MatAssemblyType type)
    - MatAssemblyEnd(Matm, MatAssemblyType type)
    - `type` is either `MAT_FLUSH_ASSEMBLY` or `MAT_FINAL_ASSEMBLY`

One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

```
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
if (rank == 0) {
  for(row = 0; row < N; row++) {
    cols[0] = row-1; cols[1] = row; cols[2] = row+1;
    if (row == 0) {
      MatSetValues(A,1,&row,2,&cols[1],&v[1],INSERT_VALUES);
    } else if (row == N-1) {
      MatSetValues(A,1,&row,2,cols,v,INSERT_VALUES);
    } else {
      MatSetValues(A,1,&row,3,cols,v,INSERT_VALUES);
    }
  }
}
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

A Better Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

```
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
MatGetOwnershipRange(A,&start,&end);
for(row = start; row < end; row++) {
  cols[0] = row-1; cols[1] = row; cols[2] = row+1;
  if (row == 0) {
    MatSetValues(A,1,&row,2,&cols[1],&v[1],INSERT_VALUES);
  } else if (row == N-1) {
    MatSetValues(A,1,&row,2,cols,v,INSERT_VALUES);
  } else {
    MatSetValues(A,1,&row,3,cols,v,INSERT_VALUES);
  }
}
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

Why Are PETSc Matrices That Way?

- No one data structure is appropriate for all problems
    - Blocked and diagonal formats provide performance benefits
    - PETSc has many formats
    - Makes it easy to add new data structures
- Assembly is difficult enough without worrying about partitioning
    - PETSc provides parallel assembly routines
    - High performance still requires making most operations local
    - However, programs can be incrementally developed.
    - `MatPartitioning` and `MatOrdering` can help
- Matrix decomposition in contiguous chunks is simple
    - Makes interoperation with other codes easier
    - For other ordering, PETSc provides “Application Orderings” (AO)

Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit

Solver Types

- **Explicit**:
  - Field variables are updated using local neighbor information
- **Semi-implicit**:
  - Some subsets of variables are updated with global solves
  - Others with direct local updates
- **Implicit**:
  - Most or all variables are updated in a single global solve

Linear Solvers

Krylov Methods

- Using PETSc linear algebra, just add:
  - KSPSetOperators(KSPksp, MatA, MatM, MatStructure flag)
  - KSPSolve(KSPksp, Vecb, Vecx)
- Can access subobjects
  - KSPGetPC(KSPksp, PC*pc)
- Preconditioners must obey PETSc interface
  - Basically just the KSP interface
- Can change solver dynamically from the command line
  - `-ksp_type` bicgstab

Nonlinear Solvers

- Using PETSc linear algebra, just add:
  - SNESSetFunction(SNESsnes, Vecr, residualFunc, void *ctx)
  - SNESSetJacobian(SNESsnes, MatA, MatM, jacFunc, void *ctx)
  - SNESSolve(SNESsnes, Vecb, Vecx)
- Can access subobjects
  - SNESGetKSP(SNESsnes, KSP*ksp)
- Can customize subobjects from the cmd line
  - Set the subdomain preconditioner to ILU with `-sub_pc_type ilu`

Basic Solver Usage

Use `SNESSetFromOptions()` so that everything is set dynamically

- Set the type
  - Use `-snes_type` (or take the default)
- Set the preconditioner
  - Use `-npc_snes_type` (or take the default)
- Override the tolerances
  - Use `-snes_rtol` and `-snes_atol`
- View the solver to make sure you have the one you expect
  - Use `-snes_view`
- For debugging, monitor the residual decrease
  - Use `-snes_monitor`
  - Use `-ksp_monitor` to see the underlying linear solver

3rd Party Solvers in PETSc

Complete table of solvers

1. Sequential LU
   - ILUDT (SPARSEKIT2, Yousef Saad, U of MN)
   - EUCLID & PILUT (Hypre, David Hysom, LLNL)
   - ESSL (IBM)
   - SuperLU (Jim Demmel and Sherry Li, LBNL)
   - Matlab
   - UMFPACK (Tim Davis, U. of Florida)
   - LUSOL (MINOS, Michael Saunders, Stanford)
2. Parallel LU
   - MUMPS (Patrick Amestoy, IRIT)
   - SPOOLES (Cleve Ashcroft, Boeing)
   - SuperLU_Dist (Jim Demmel and Sherry Li, LBNL)
3. Parallel Cholesky
   - DSCPACK (Padma Raghavan, Penn. State)
   - MUMPS (Patrick Amestoy, Toulouse)
   - CHOLMOD (Tim Davis, Florida)
4. XYTlib - parallel direct solver (Paul Fischer and Henry Tufo, ANL)

3rd Party Preconditioners in PETSc

Complete table of solvers

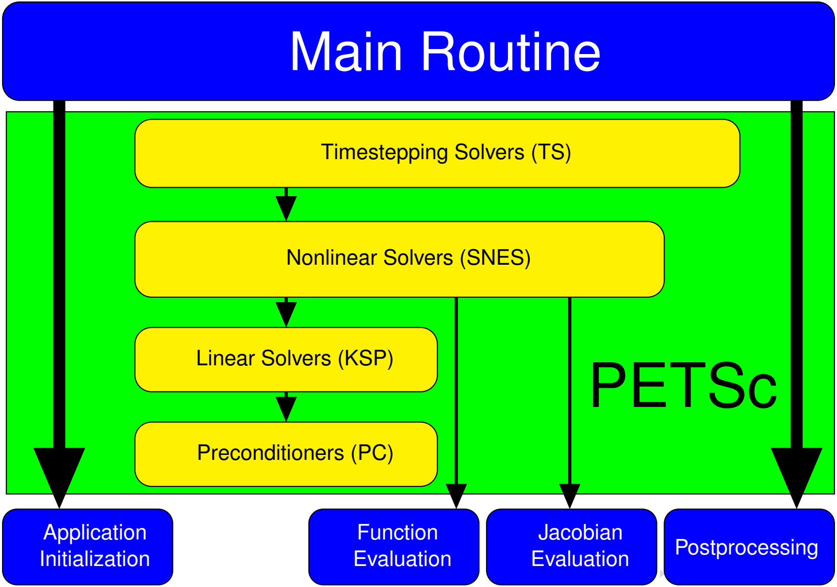
1. Parallel ICC
   - BlockSolve95 (Mark Jones and Paul Plassman, ANL)
2. Parallel ILU
   - PaStiX (Faverge Mathieu, INRIA)
3. Parallel Sparse Approximate Inverse
   - Parasails (Hypre, Edmund Chow, LLNL)
   - SPAI 3.0 (Marcus Grote and Barnard, NYU)
4. Sequential Algebraic Multigrid
   - RAMG (John Ruge and Klaus Steuben, GMD)
   - SAMG (Klaus Steuben, GMD)
5. Parallel Algebraic Multigrid
   - Prometheus (Mark Adams, PPPL)
   - BoomerAMG (Hypre, LLNL)
   - ML (Trilinos, Ray Tuminaro and Jonathan Hu, SNL)

Outline

3 Linear Algebra and Solvers

- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit

Flow Control for a PETSc Application



SNES Paradigm

The SNES interface is based upon callback functions

- FormFunction(), set by SNESSetFunction()
- FormJacobian(), set by SNESSetJacobian()

When PETSc needs to evaluate the nonlinear residual $F(x)$,

- Solver calls the **user's** function
- User function gets application state through the `ctx` variable
    - PETSc never sees application data

Topology Abstractions

- `DMDA`
  - Abstracts Cartesian grids in any dimension
  - Supports stencils, communication, reordering
  - Nice for simple finite differences
- `DMMesh`
  - Abstracts general topology in any dimension
  - Also supports partitioning, distribution, and global orders
  - Allows aribtrary element shapes and discretizations

Assembly Abstractions

- DM
  - Abstracts the logic of multilevel (multiphysics) methods
  - Manages allocation and assembly of local and global structures
  - Interfaces to PCMG solver
- PetscSection
  - Abstracts functions over a topology
  - Manages allocation and assembly of local and global structures
  - Will merge with DM somehow

SNES Function

User provided function calculates the nonlinear residual:

```
PetscErrorCode (*func)(SNES snes,Vec x,Vec r,void *ctx)
```

x: The current solution

r: The residual

ctx: The user context passed to SNESSetFunction()

- Use this to pass application information, e.g. physical constants

SNES Jacobian

User provided function calculates the Jacobian:

```
PetscErrorCode (*func)(SNES snes, Vec x, Mat *J, Mat *M, void *ctx)
```

x: The current solution

J: The Jacobian

M: The Jacobian preconditioning matrix (possibly J itself)

ctx: The user context passed to SNESSetJacobian()

- Use this to pass application information, e.g. physical constants

Alternatively, you can use

- matrix-free finite difference approximation, -snes_mf
- finite difference approximation with coloring, -snes_fd

SNES Variants

- Picard iteration
- Line search/Trust region strategies
- Quasi-Newton
- Nonlinear CG/GMRES
- Nonlinear GS/ASM
- Nonlinear Multigrid (FAS)
- Variational inequality approaches

Finite Difference Jacobians

PETSc can compute and explicitly store a Jacobian via 1st-order FD

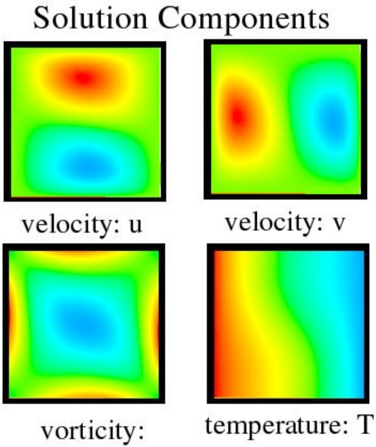
- Dense
  - Activated by `-snes_fd`
  - Computed by SNESDefaultComputeJacobian()
- Sparse via colorings (default)
  - Coloring is created by MatFDColoringCreate()
  - Computed by SNESDefaultComputeJacobianColor()

Can also use Matrix-free Newton-Krylov via 1st-order FD

- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from SNESSetJacobian()

SNES Example

Driven Cavity



- Velocity-vorticity formulation
- Flow driven by lid and/or bouyancy
- Logically regular grid
  - Parallelized with `DMDA`
- Finite difference discretization
- Authored by David Keyes

$PETSC_DIR/src/snes/examples/tutorials/ex19.c

Driven Cavity Application Context

```
typedef struct {
  /*———  basic application data  ———*/
  PetscReal lid_velocity;
  PetscReal prandtl
  PetscReal grashof;
  PetscBool draw_contours;
} AppCtx;
```

$PETSC_DIR/src/snes/examples/tutorials/ex19.c

Driven Cavity Residual Evaluation

```
Residual(SNES snes, Vec X, Vec F, void *ptr) {
  AppCtx         *user = (AppCtx *) ptr;

  /* local starting and ending grid points */
  PetscInt        istart, iend, jstart, jend;
  PetscScalar    *f; /* local vector data */
  PetscReal       grashof = user->grashof;
  PetscReal       prandtl = user->prandtl;
  PetscErrorCode  ierr;

  /* Code to communicate nonlocal ghost point data */
  VecGetArray(F, &f);
  /* Code to compute local function components */
  VecRestoreArray(F, &f);
  return 0;
}
```

$PETSC_DIR/src/snes/examples/tutorials/ex19.c

Better Driven Cavity Residual Evaluation

```
ResLocal(DMDALocalInfo *info,
         PetscScalar **x, PetscScalar **f, void *ctx)
{
  for(j = info->ys; j < info->ys+info->ym; ++j) {
    for(i = info->xs; i < info->xs+info->xm; ++i) {
      u    = x[j][i];
      uxx = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
      uyy = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
      f[j][i].u = uxx + uyy - .5*(x[j+1][i].omega-x[j-1][i].omega)*hx;
      f[j][i].v = uxx + uyy + .5*(x[j][i+1].omega-x[j][i-1].omega)*hy;
      f[j][i].omega = uxx + uyy +
          (vxp*(u - x[j][i-1].omega) + vxm*(x[j][i+1].omega - u))*hy +
          (vyp*(u - x[j-1][i].omega) + vym*(x[j+1][i].omega - u))*hx -
          0.5*grashof*(x[j][i+1].temp - x[j][i-1].temp)*hy;
      f[j][i].temp  =  uxx + uyy + prandtl*
          ((vxp*(u - x[j][i-1].temp) + vxm*(x[j][i+1].temp - u))*hy +
           (vyp*(u - x[j-1][i].temp) + vym*(x[j+1][i].temp - u))*hx);
}}}
```

$PETSC_DIR/src/snes/examples/tutorials/ex19.c

Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit

What is a DMDA?

DMDA is a topology interface on structured grids

- Handles parallel data layout
- Handles local and global indices
    - DMDAGetGlobalIndices() and DMDAGetAO()
- Provides local and global vectors
    - DMGetGlobalVector() and DMGetLocalVector()
- Handles ghost values coherence
    - DMGlobalToLocalBegin/End() and DMLocalToGlobalBegin/End()

Residual Evaluation

The **DM** interface is based upon *local* callback functions

- FormFunctionLocal()
- FormJacobianLocal()

Callbacks are registered using

- SNESSetDM(), TSSetDM()
- DMSNESSetFunctionLocal(), DMTSSetJacobianLocal()

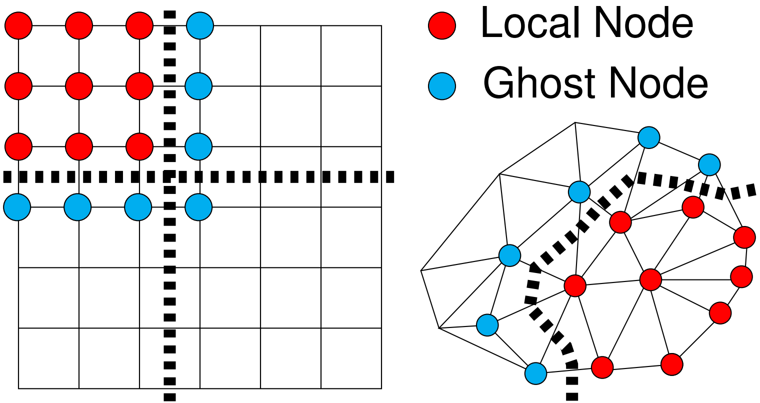
When PETSc needs to evaluate the nonlinear residual **F(x)**,

- Each process evaluates the local residual
- PETSc assembles the global residual automatically
    - Uses DMLocalToGlobal() method

Ghost Values

To evaluate a local function $f(x)$, each process requires

- its local portion of the vector x
- its ghost values, bordering portions of x owned by neighboring processes



DMDA Global Numberings

Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

PETSc numbering

DMDA Global vs. Local Numbering

- **Global**: Each vertex has a unique id belongs on a unique process
- **Local**: Numbering includes vertices from neighboring processes
  - These are called ghost vertices

Proc 2			Proc 3	
X	X	X	X	X
X	X	X	X	X
12	13	14	15	X
8	9	10	11	X
4	5	6	7	X
0	1	2	3	X
Proc 0			Proc 1	

Local numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

Global numbering

DMDA Local Function

User provided function calculates the nonlinear residual (in 2D)

```
(* lf )(DMDALocalInfo *info, PetscScalar**x, PetscScalar **r, void *ctx)
```

info: All layout and numbering information

x: The current solution (a multidimensional array)

r: The residual

ctx: The user context passed to `DMDASNESSetFunctionLocal()`

The local DMDA function is activated by calling

```
DMDASNESSetFunctionLocal(dm, INSERT_VALUES, lfunc, &ctx)
```

Bratu Residual Evaluation

$$\Delta u + \lambda e^{u} = 0$$

```
ResLocal(DMDALocalInfo *info, PetscScalar **x, PetscScalar **f, void *ctx)
for(j = info->ys; j < info->ys+info->ym; ++j) {
  for(i = info->xs; i < info->xs+info->xm; ++i) {
    u = x[j][i];
    if (i==0 || j==0 || i == M || j == N) {
      f[j][i] = 2.0*(hydhx+hxdhy)*u; continue;
    }
    u_xx    = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
    u_yy    = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
    f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
}}}
```

$PETSC_DIR/src/snes/examples/tutorials/ex5.c

DMDA Local Jacobian

User provided function calculates the Jacobian (in 2D)

```
(* ljac )(DMDALocalInfo *info, PetscScalar**x, Mat J, void *ctx)
```

info: All layout and numbering information

x: The current solution

J: The Jacobian

ctx: The user context passed to DASetLocalJacobian()

The local DMDA function is activated by calling

```
DMDASNESSetJacobianLocal(dm, ljac, &ctx)
```

Bratu Jacobian Evaluation

```c
JacLocal(DMDALocalInfo *info,PetscScalar **x,Mat jac,void *ctx) {
for(j = info->ys; j < info->ys + info->ym; j++) {
  for(i = info->xs; i < info->xs + info->xm; i++) {
    row.j = j; row.i = i;
    if (i == 0 || j == 0 || i == mx-1 || j == my-1) {
      v[0] = 1.0;
      MatSetValuesStencil(jac,1,&row,1,&row,v,INSERT_VALUES);
    } else {
      v[0] = -(hx/hy); col[0].j = j-1; col[0].i = i;
      v[1] = -(hy/hx); col[1].j = j;   col[1].i = i-1;
      v[2] = 2.0*(hy/hx+hx/hy)
             - hx*hy*lambda*PetscExpScalar(x[j][i]);
      v[3] = -(hy/hx); col[3].j = j;   col[3].i = i+1;
      v[4] = -(hx/hy); col[4].j = j+1; col[4].i = i;
      MatSetValuesStencil(jac,1,&row,5,col,v,INSERT_VALUES);
}}}}
```

$PETSC_DIR/src/snes/examples/tutorials/ex5.c

A DMDA is more than a Mesh

A DMDA contains topology, geometry, and (sometimes) an implicit Q1 discretization.

It is used as a template to create

- Vectors (functions)
- Matrices (linear operators)

DMDA Vectors

- The **DMDA** object contains only layout (topology) information
  - All field data is contained in PETSc **Vecs**
- Global vectors are parallel
  - Each process stores a unique local portion
  - DMCreateGlobalVector(DM da, Vec $*$gvec)
- Local vectors are sequential (and usually temporary)
  - Each process stores its local portion plus ghost values
  - DMCreateLocalVector(DM da, Vec $*$lvec)
  - includes ghost and boundary values!

Updating Ghosts

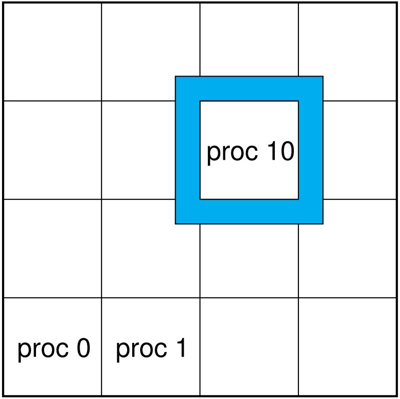
Two-step process enables overlapping computation and communication

- DMGlobalToLocalBegin(da, gvec, mode, lvec)
  - `gvec` provides the data
  - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
  - `lvec` holds the local and ghost values
- DMGlobalToLocalEnd(da, gvec, mode, lvec)
  - Finishes the communication

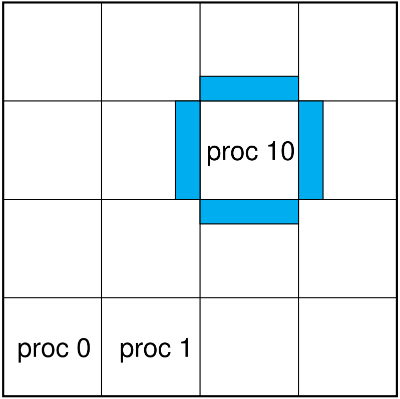
The process can be reversed with DALocalToGlobalBegin/End().

DMDA Stencils

Both the box stencil and star stencil are available.



Box Stencil



Star Stencil

Setting Values on Regular Grids

PETSc provides

```
MatSetValuesStencil(Mat A, m, MatStencil idxm[], n, MatStencil idxn[],
                    PetscScalar values[], InsertMode mode)
```

- Each row or column is actually a **MatStencil**
  - This specifies grid coordinates and a component if necessary
  - Can imagine for unstructured grids, they are *vertices*
- The values are the same logically dense block in row/col

Creating a DMDA

DMDACreate2d(comm, bdX, bdY, type, M, N, m, n, dof, s, lm[], ln[], DMDA *da)

bd: Specifies boundary behavior

- DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, or DM_BOUNDARY_PERIODIC

type: Specifies stencil

- DMDA_STENCIL_BOX or DMDA_STENCIL_STAR

M/N: Number of grid points in x/y-direction

m/n: Number of processes in x/y-direction

dof: Degrees of freedom per node

s: The stencil width

lm/n: Alternative array of local sizes

- Use NULL for the default

Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit

MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - **VecScatter** and `MatGetSubMatrix()` for efficiency
- assemble functions/operators over all physics
  - Generalizes `LocalToGlobal()` mapping
- is composable with ANY PETSc solver and preconditioner
  - This can be done recursively

MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - **VecScatter** and `MatGetSubMatrix()` for efficiency
- assemble functions/operators over all physics
  - Generalizes `LocalToGlobal()` mapping
- is composable with ANY PETSc solver and preconditioner
  - This can be done recursively

FieldSplit provides the buildings blocks for multiphysics preconditioning.

MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - **VecScatter** and `MatGetSubMatrix()` for efficiency
- assemble functions/operators over all physics
  - Generalizes `LocalToGlobal()` mapping
- is composable with ANY PETSc solver and preconditioner
  - This can be done recursively

Notice that this works in exactly the same manner as

- multiple resolutions (MG, FMM, Wavelets)
- multiple domains (Domain Decomposition)
- multiple dimensions (ADI)

Preconditioning

Several varieties of preconditioners can be supported:

- Block Jacobi or Block Gauss-Siedel
- Schur complement
- Block ILU (approximate coupling and Schur complement)
- Dave May's implementation of Elman-Wathen type PCs

which only require actions of individual operator blocks

Notice also that we may have any combination of

- "canned" PCs (ILU, AMG)
- PCs needing special information (MG, FMM)
- custom PCs (physics-based preconditioning, Born approximation)

since we have access to an algebraic interface

Outline

1. Tools and Infrastructure

2. GPU Computing

3. Linear Algebra and Solvers

4. First Lab: Assembling a Code
   - Beginning with Mercurial
   - Setting up Configure and Build
   - Running a PETSc Python example
   - Running a PETSc solver on the GPU
   - Adding a GPU kernel

5. Second Lab: Debugging and Performance Benchmarking

Outline

4 First Lab: Assembling a Code
- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel

Create a clone

```
hg clone http://petsc.cs.iit.edu/petsc/tutorials/vecfem
```

or

```
from mercurial.dispatch import dispatch

args = ['clone',
  'http://petsc.cs.iit.edu/petsc/tutorials/vecfem']
dispatch(args)
```

Make a change

```
# Make sure we are up-to-date
hg pull -u
# Create a file
touch TODO
# Schedule the file for version control
hg add TODO
# Commit to a ChangeSet
hg ci -m "Added␣TODO␣list" TODO
# Edit file
echo "Get␣configure␣working" >> TODO
# Commit edit
hg ci -m "Added␣first␣task"
# Push change to the master repository
hg push
```

Make a bugfix

```
# Create new repository
hg clone -r1 vecfem vecfem-bugfix
cd vecfem-bugfix
# Fix bug
hg ci -m "Fixed␣bug"
# Merge in changes from Master
hg pull --rebase
# Push bugfix
hg push
```

Make a bugfix (alternate)

```
# Create new repository
hg clone -r1 vecfem vecfem-bugfix
cd vecfem-bugfix
# Fix bug
hg ci -m "Fixed␣bug"
# Merge in changes from Master
hg pull
hg merge
hg ci -m "Merge"
# Push bugfix
hg push
```

Outline

4 First Lab: Assembling a Code
- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel

Basic Configuration

http://petsc.cs.iit.edu/petsc/SimpleConfigure
provides basic configure support

1. Choose projectName
2. Copy in:
   - `configure.py` to the root
   - `config/Matt/*` to `config/projectName/`
3. Change main call in `configure.py`:

```
ConfigurationManager(projectName).configure([])
```

4. Change project name in
   `config/projectName/Configure.py`

Basic Configuration

Arch

We can add support for multiple builds using `--arch`

```
class ProjectArch(object):
  def __init__(self, argDB):
    self.argDB = argDB
  @property
  def arch(self):
    return self.argDB['arch']
```

changing the member variable in `__init__()` to

```
self.arch = ProjectArch(self.argDB)
```

and adding an option to `setupHelp()`

```
help.addArgument(self.Project, '–arch=<name>',
  nargs.Arg(None,'debug','The name of this build'))
```

Basic Configuration

Arch

We can add support for multiple builds using `--arch`

```
class ProjectArch(object):
  def __init__(self, argDB):
    self.argDB = argDB
  @property
  def arch(self):
    return self.argDB['arch']
```

changing the member variable in `__init__()` to

```
self.arch = ProjectArch(self.argDB)
```

and adding an option to `setupHelp()`

```
help.addArgument(self.Project, '–arch=<name>',
  nargs.Arg(None,'debug','The name of this build'))
```

Basic Configuration

Arch

We can add support for multiple builds using `--arch`

```
class ProjectArch(object):
  def __init__(self, argDB):
    self.argDB = argDB
  @property
  def arch(self):
    return self.argDB['arch']
```

changing the member variable in `__init__()` to

```
self.arch = ProjectArch(self.argDB)
```

and adding an option to `setupHelp()`

```
help.addArgument(self.Project, '-arch=<name>',
  nargs.Arg(None,'debug','The name of this build'))
```

Basic Build

Make

We can add support for builds using PETSc make rules

```
CFLAGS    = -I.
ARCHFLAGS = -arch x86_64

meshObjs: sieveMesh.o
  ARCHFLAGS="${ARCHFLAGS}" python setupSieve.py build_ext --inplace
ex1: ex1.o
  ${CLINKER} -o ex1 ex1.o ${PETSC_LIB}
```

Building Extensions

We need a `setupSieve.py` to build an extension module

```
import os, numpy
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [Extension('petscSieve', ['petscSieve.pyx'],
  extra_objects = ['sieveMesh.o'],
  library_dirs = [
    os.path.join(os.environ['PETSC_DIR'], os.environ['PETSC_ARCH'], 'lib')
    '/usr/X11/lib',
    '/System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks
  libraries = ['petsc','X11','LAPACK','BLAS','pmpich','mpich','opa','mpl']
  include_dirs = [numpy.get_include()], language = 'c++')]
setup(
  name = 'PETSc Sieve converter',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

Outline

4 First Lab: Assembling a Code

- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel

Empty Example

```
# Update repository to new code
hg update -r2
# Execute empty run
python vecfem.py
# We can set verbosity level
./vecfem.py --verbose=2
# Python 3 should fail gracefully
python3 vecfem.py
```

Add Empty Vec, Mat, and KSP

Structured Mesh

```
hg update -r5
# Monitor solve and reason for termination
./vecfem.py -ksp_monitor -ksp_converged_reason
# Look at solve configuration and performance data
./vecfem.py -ksp_view -log_summary
# Can put performance data in a module
./vecfem.py -log_summary_python logSummary.py
```

Solve P_1 Laplace Problem

Structured Mesh

```
hg update -r7
# Solve system to high tolerance
./vecfem.py -ksp_monitor -ksp_converged_reason -ksp_rtol 1.0e-9
# Output the computed solution
#   Notice that the constant mode is removed
./vecfem.py -ksp_monitor -ksp_rtol 1.0e-9 --verbose
# We can change the size
./vecfem.py -ksp_monitor -ksp_rtol 1.0e-9 --sizes=[5,5]
# Output intermediate objects
./vecfem.py -ksp_monitor -ksp_rtol 1.0e-9 --verbose=2
```

Outline

4 First Lab: Assembling a Code
- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel

GPU Solvers
Structured Mesh

```
hg update -r8
# Specify CUDA types for linear algebra
#   Solve will automatically happen on GPU
./vecfem.py -ksp_monitor -ksp_converged_reason \
  -ksp_rtol 1.0e-9 \
  -da_vec_type cuda -da_mat_type aijcuda
```

Outline

4 First Lab: Assembling a Code

- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel

Updates

- Configure Sieve extension module
- Added Python modules for:
    - Mesh handling (Sieve)
    - Geometry
    - Discretization
    - Integration
    - Computational Modeling
    - Kernel Execution
- Extension module for Sieve has:
    - C wrapper for C++ methods
    - Cython wrapper for C methods
    - Python build script
- Added unstructured mesh support

GPU Kernel

- `femIntegration` makes kernel using templating
- PyCUDA creates module and launches
- Cython bridges gap with C support libraries
- numpy is used to transfer data

Outline

1. Tools and Infrastructure
2. GPU Computing
3. Linear Algebra and Solvers
4. First Lab: Assembling a Code
5. Second Lab: Debugging and Performance Benchmarking
   - Debugging
   - Performance Benchmarking

Outline

5 Second Lab: Debugging and Performance Benchmarking
- Debugging
- Performance Benchmarking

Valgrind

Valgrind is a debugging framework

- **Memcheck**: Check for memory overwrite and illegal use
- **Callgrind**: Generate call graphs
- **Cachegrind**: Monitor cache usage
- **Helgrind**: Check for race conditions
- **Massif**: Monitor memory usage

Valgrind

Memcheck

Memcheck will catch

- Illegal reads and writes to memory
- Uninitialized values
- Illegal frees
- Overlapping copies
- Memory leaks

Valgrind

Memcheck

Let's try a simple experiment

```
# Get the tutorial repository
hg clone http://petsc.cs.iit.edu/petsc/tutorials/SimpleTutorial
hg update -r2
# Memcheck is the default tool
valgrind --trace-children=yes --suppressions=bin/simple.supp \
  ./bin/ex5 -use_coords
# Try it for multiple processes
valgrind --trace-children=yes --suppressions=bin/simple.supp \
  $PETSC_DIR/$PETSC_ARCH/bin/mpiexec -n 2 ./bin/ex5 -use_coords
```

Valgrind

Memcheck

We get an error!

```
==13697== Invalid read of size 8
==13697==    at 0x100005263: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13697==    by 0x100004447: main (ex5.c:202)
==13697==  Address 0x103dc6fa0 is 0 bytes after a block of size 48 alloc'd
==13697==    at 0x10001ED75: malloc (vg_replace_malloc.c:236)
==13697==    by 0x1005CABC4: PetscMallocAlign(unsigned long, int, char const*, char const*, char c
==13697==    by 0x1009CC07D: VecGetArray2d(_p_Vec*, int, int, int, int, double***) (rvector.c:1739
==13697==    by 0x10030D980: DMDAVecGetArray(_p_DM*, _p_Vec*, void*) (dagetarray.c:72)
==13697==    by 0x100005102: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:38)
==13697==    by 0x100004447: main (ex5.c:202)
==13697==
==13697== Invalid read of size 8
==13697==    at 0x100005273: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13697==    by 0x100004447: main (ex5.c:202)
==13697==  Address 0x18 is not stack'd, malloc'd or (recently) free'd
==13697==
==13698== Use of uninitialised value of size 8
==13698==    at 0x10000529D: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13698==    by 0x100004447: main (ex5.c:202)
==13698==
==13698== Invalid read of size 8
==13698==    at 0x10000529D: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13698==    by 0x100004447: main (ex5.c:202)
==13698==  Address 0x6f5c300000018 is not stack'd, malloc'd or (recently) free'd
```

Valgrind

Memcheck

We can fix the error by using ghosted coordinates

```
hg update -r3
make
valgrind --trace-children=yes --suppressions=bin/simple.supp \
  $PETSC_DIR/$PETSC_ARCH/bin/mpiexec -n 2 ./bin/ex5 -use_coords
```

Valgrind

Memcheck

A *suppressions* file suppresses errors

It can be generated automatically

```
valgrind --trace-children=yes --gen-suppressions=all \
  ./vecfem.py -ksp_rtol 1.0e-9
```

and we put them into `vecfem.supp`, so that

```
hg -r9 update
valgrind --trace-children=yes --suppressions=vecfem.supp ./vecfem.py
```

produces no errors.

Valgrind

Massif

```
# Memcheck is the default tool
valgrind --tool=massif --trace-children=yes \
  --massif-out-file=vecfem.massif \
  ./vecfem --sizes=[100,100] -ksp_rtol 1.0e-9
# Turn on stack profiling
valgrind --tool=massif --trace-children=yes \
  --massif-out-file=vecfem.massif \
  ./vecfem --stacks=yes --sizes=[100,100] -ksp_rtol 1.0e-9
# Visualize output
ms_print --threshold=10.0 vecfem.massif
```

Correctness Debugging

- Automatic generation of tracebacks
- Detecting memory corruption and leaks
- Optional user-defined error handlers

Outline

5 Second Lab: Debugging and Performance Benchmarking
- Debugging
- Performance Benchmarking

Performance Debugging

- PETSc has integrated profiling
  - Option `-log_summary` prints a report on `PetscFinalize()`
- PETSc allows user-defined events
  - Events report time, calls, flops, communication, etc.
  - Memory usage is tracked by object
- Profiling is separated into stages
  - Event statistics are aggregated by stage

Using Stages and Events

- Use `PetscLogStageRegister()` to create a new stage
  - Stages are identifier by an integer handle
- Use `PetscLogStagePush/Pop()` to manage stages
  - Stages may be nested, but will not aggregate in a nested fashion
- Use `PetscLogEventRegister()` to create a new stage
  - Events also have an associated class
- Use `PetscLogEventBegin/End()` to manage events
  - Events may also be nested and will aggregate in a nested fashion
  - Can use `PetscLogFlops()` to log user flops

Adding A Logging Stage

C

```
int stageNum;

PetscLogStageRegister(&stageNum, "name");
PetscLogStagePush(stageNum);

/* Code to Monitor */

PetscLogStagePop();
```

Adding A Logging Stage

Python

```
with PETSc.LogStage('Fluid Stage') as fluidStage:
  # All operations will be aggregated in fluidStage
  fluid.solve()
```

Adding A Logging Event
C

```
static int USER_EVENT;

PetscLogEventRegister(&USER_EVENT, "name", CLS_ID);
PetscLogEventBegin(USER_EVENT,0,0,0,0);

/* Code to Monitor */

PetscLogFlops(user_event_flops);
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

Adding A Logging Event

Python

```
with PETSc.logEvent('Reconstruction') as recEvent:
  # All operations are timed in recEvent
  reconstruct(sol)
  # Flops are logged to recEvent
  PETSc.Log.logFlops(user_event_flops)
```

Matrix Memory Preallocation

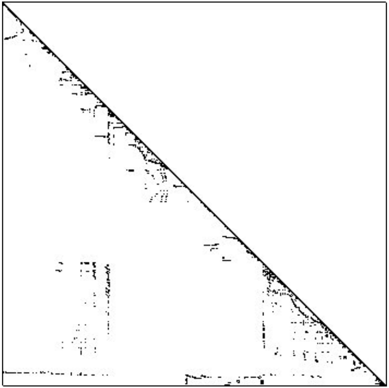
- PETSc sparse matrices are dynamic data structures
    - can add additional nonzeros freely
- Dynamically adding many nonzeros
    - requires additional memory allocations
    - requires copies
    - can kill performance
- Memory preallocation provides
    - the freedom of dynamic data structures
    - good performance
- Easiest solution is to replicate the assembly code
    - Remove computation, but preserve the indexing code
    - Store set of columns for each row
- Call preallocation rourines for all datatypes
    - `MatSeqAIJSetPreallocation()`
    - `MatMPIAIJSetPreallocation()`
    - Only the relevant data will be used

Matrix Memory Preallocation
Sequential Sparse Matrices

```
MatSeqAIJPreallocation(Mat A, int nz, int nnz[])
```

nz: expected number of nonzeros in any row

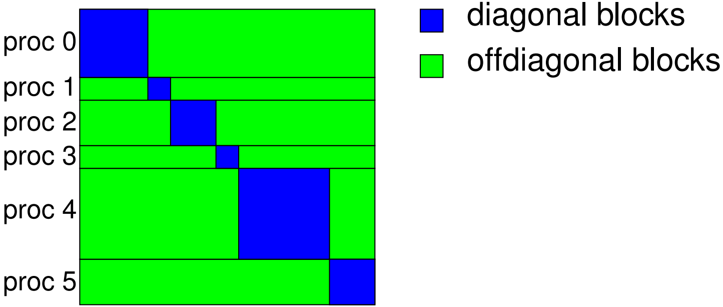
nnz(i): expected number of nonzeros in row i



Matrix Memory Preallocation

ParallelSparseMatrix

- Each process locally owns a submatrix of contiguous global rows
- Each submatrix consists of diagonal and off-diagonal parts



- `MatGetOwnershipRange(Mat A,int *start,int *end)`

`start`: first locally owned row of global matrix

`end-1`: last locally owned row of global matrix

Matrix Memory Preallocation

Parallel Sparse Matrices

```
MatMPIAIJPreallocation(Mat A, int dnz, int dnnz[],
int onz, int onnz[])
```

dnz: expected number of nonzeros in any row in the diagonal block

nnz(i): expected number of nonzeros in row i in the diagonal block

onz: expected number of nonzeros in any row in the offdiagonal portion

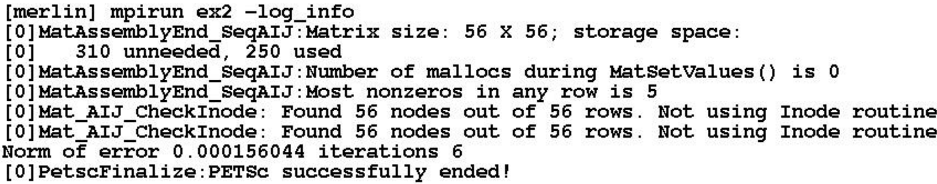
nnz(i): expected number of nonzeros in row i in the offdiagonal portion

Matrix Memory Preallocation

Verifying Preallocation

- Use runtime option `-info`
- Output:

  ```
  [proc #] Matrix size:  %d X %d; storage space:
  %d unneeded, %d used
  [proc #] Number of mallocs during MatSetValues( )
  is %d
  ```



```
[merlin] mpirun ex2 -log_info
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 56 X 56; storage space:
[0]   310 unneeded, 250 used
[0]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
[0]MatAssemblyEnd_SeqAIJ:Most nonzeros in any row is 5
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
Norm of error 0.000156044 iterations 6
[0]PetscFinalize:PETSc successfully ended!
```

Scalability is not Efficiency

Scalability is easy

Efficiency is hard

Scalability is not Efficiency

Scalability is easy

Efficiency is hard

Scalability is not Efficiency

Scalability is easy

Efficiency is hard

Scalability

Def: Computation, Communication, and Memory are in $\mathcal{O}(N)$

- Can also demand $\mathcal{O}(P)$
- Watch out for hidden constants
  - $6N$ and $6000N$ are both scalable

PDE

PDEs are scalable

- Computations are local
  - abstract data types, e.g. `Mat`
- Communication is nearest neighbor
  - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
  - `GlobalToLocalMapping`
  - `DA`, `Mesh`, `VecScatter`

PDE

PDEs are scalable

- Computations are local
  - abstract data types, e.g. `Mat`
- Communication is nearest neighbor
  - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
  - `GlobalToLocalMapping`
  - `DA`, `Mesh`, `VecScatter`

PDE

PDEs are scalable

- Computations are local
  - abstract data types, e.g. `Mat`
- Communication is nearest neighbor
  - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
  - `GlobalToLocalMapping`
  - `DA`, `Mesh`, `VecScatter`

PDE

PDEs are scalable unless you screw something up

- Prescribed data structures
  - abstract data types, e.g. `Mat`
- Fully replicated data structures
  - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
  - `GlobalToLocalMapping`
  - `DA`, `Mesh`, `VecScatter`

PDE

PDEs are scalable unless you screw something up

Mistakes:

- Prescribed data structures
  - abstract data types, e.g. `Mat`
- Fully replicated data structures
  - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
  - `GlobalToLocalMapping`
  - `DA`, `Mesh`, `VecScatter`

PDE

PDEs are scalable unless you screw something up

Mistakes:

- Prescribed data structures
  - abstract data types, e.g. `Mat`
- Fully replicated data structures
  - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
  - `GlobalToLocalMapping`
  - `DA`, `Mesh`, `VecScatter`

PDE

PDEs are scalable unless you screw something up

Mistakes:

- Prescribed data structures
  - abstract data types, e.g. `Mat`
- Fully replicated data structures
  - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
  - `GlobalToLocalMapping`
  - `DA`, `Mesh`, `VecScatter`

PDE

PDEs are scalable unless you screw something up

Mistakes:

- Prescribed data structures
    - abstract data types, e.g. `Mat`
- Fully replicated data structures
    - parallel data structures, e.g. `DA`
- Referenced arbitrary unknowns
    - `GlobalToLocalMapping`
    - `DA`, `Mesh`, `VecScatter`

Integral Equations

Integral equations can be scalable

- But, they couple all unknowns
- Need special algorithms
  - Fast Fourier Transform
  - Fast Multipole Method
  - Fast Wavelet Transform

Integral Equations

Integral equations can be scalable

- But, they couple all unknowns
- Need special algorithms
  - Fast Fourier Transform
  - Fast Multipole Method
  - Fast Wavelet Transform

Integral Equations

Integral equations can be scalable

- But, they couple all unknowns
- Need special algorithms
  - Fast Fourier Transform
  - Fast Multipole Method
  - Fast Wavelet Transform

Importance of Computational Modeling

Without a model,
performance measurements are meaningless!

Before a code is written, we should have a model of

- computation
- memory usage
- communication
- bandwidth
- achievable concurrency

This allows us to

- verify the implementation
- predict scaling behavior

Complexity Analysis

The key performance indicator, which we will call the *balance factor* β, is the ratio of flops executed to bytes transfered.

- We will designate the unit $\frac{\mathrm{flop}}{\mathrm{byte}}$ as the *Keyes*
- Using the peak flop rate r_{peak}, we can get the required bandwidth B_{req} for an algorithm
$$B_{\mathrm{req}} = \frac{r_{\mathrm{peak}}}{\beta} \tag{20}$$
- Using the peak bandwidth B_{peak}, we can get the maximum flop rate r_{max} for an algorithm
$$r_{\mathrm{max}} = \beta B_{\mathrm{peak}} \tag{21}$$

Performance Caveats

- The peak flop rate r_{peak} on modern CPUs is attained through the usage of a SIMD multiply-accumulate instruction on special 128-bit registers.
- SIMD MAC operates in the form of 4 simultaneous operations (2 adds and 2 multiplies):

$$c_1 = c_1 + a_1 * b_1 \tag{22}$$

$$c_2 = c_2 + a_2 * b_2 \tag{23}$$

  You will miss peak by the corresponding number of operations you are missing. In the worst case, you are reduced to 25% efficiency if your algorithm performs naive summation or products.
- Memory alignment is also crucial when using SSE, the instructions used to load and store from the 128-bit registers throw very costly alignment exceptions when the data is not stored in memory on 16 byte (128 bit) boundaries.

Analysis of BLAS `axpy()`

$$\vec{y} \leftarrow \alpha\vec{x} + \vec{y}$$

For vectors of length N and b-byte numbers, we have

- Computation
  - $2N$ flops
- Memory Access
  - $(3N+1)b$ bytes

Thus, our balance factor $\beta = \frac{2N}{(3N+1)b} \approx \frac{2}{3b}\mathrm{Keyes}$

Analysis of BLAS `axpy()`

$$\vec{y} \leftarrow \alpha\vec{x} + \vec{y}$$

For Matt's Laptop,

- $r_{\mathrm{peak}} = 1700\mathrm{MF/s}$

  implies that
- $B_{\mathrm{req}} = 2550b\ \mathrm{MB/s}$
  - Much greater than B_{peak}

- $B_{\mathrm{peak}} = 1122\mathrm{MB/s}$

  implies that
- $r_{\mathrm{max}} = \frac{748}{b}\ \mathrm{MF/s}$
  - 5.5% of r_{peak}

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

$$\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V}\ \mathrm{byte/flop} \tag{24}$$

or achieveable performance given a bandwith *BW*

$$\frac{Vnz}{(8V + 2)m + 6nz} BW\ \mathrm{Mflop/s} \tag{25}$$

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

Improving Serial Performance

For a single matvec with 3D FD Poisson, Matt's laptop can achieve at most

$$\frac{1}{(8+2)\frac{1}{7}+6}\ \mathrm{bytes/flop}(1122.4\ \mathrm{MB/s}) = 151\ \mathrm{MFlops/s}, \tag{26}$$

which is a dismal 8.8% of peak.

Can improve performance by

- Blocking
- Multiple vectors

but operation issue limitations take over.

Improving Serial Performance

For a single matvec with 3D FD Poisson, Matt's laptop can achieve at most

$$\frac{1}{(8+2)\frac{1}{7}+6} \text{ bytes/flop}(1122.4 \text{ MB/s}) = 151 \text{ MFlops/s}, \tag{26}$$

which is a dismal 8.8% of peak.

Better approaches:

- Unassembled operator application (Spectral elements, FMM)
  - N data, N^2 computation
- Nonlinear evaluation (Picard, FAS, Exact Polynomial Solvers)
  - N data, N^k computation

Performance Tradeoffs

We must balance storage, bandwidth, and cycles

- Assembled Operator Action
  - Trades cycles and storage for bandwidth in application
- Unassembled Operator Action
  - Trades bandwidth and storage for cycles in application
  - For high orders, storage is impossible
  - Can make use of FErari decomposition to save calculation
  - Could storage element matrices to save cycles
- Partial assembly gives even finer control over tradeoffs
  - Also allows introduction of parallel costs (load balance, . . .)