Implications for Library Developers of GPU Hardware

Matthew Knepley[1,2]

[1]Computation Institute
University of Chicago

[2]Department of Molecular Biology and Physiology
Rush University Medical Center

Intelligent Software Workshop
Edinburgh, Scotland October 20, 2009

Outline

Biggest Changes

Multi-language programming is necessary, for at least the near future

Interfaces will have to be fluid as hardware changes rapidly.

Biggest Changes

Multi-language programming is necessary, for at least the near future

Interfaces will have to be fluid as hardware changes rapidly.

Outline

Build System

- Hardware detection during configure more difficult
  - Need a community solution
- New language (CUDA, Cell Broadband Engine)
  - Necessitates new compiler
  - Source and library segregation
  - Interaction issues with other languages/compilers/libraries
  - There are some libraries (TBB)
- Still not clear how to multiplex over different approaches
  - OpenCL is far from mature, and future is uncertain
  - `#define` is not enough to cope with different underlying builds

PETSc Configure System:
`http://petsc.cs.iit.edu/petsc/BuildSystem`

Interaction with MPI

There are several possible models:

- One process controls a single GPU
  - No extra work
- One process controls several GPUs
  - Need allocation strategy for kernels (multiple queues)
- Several processes control one GPU
  - Need standard locking mechanism
- Several processes control several GPUs
  - Just a combination of above, harder to optimize

Interaction with MPI

Do not anticipate GPU-to-GPU communication:

- At least not in the short term
- Requires hardware and/or OS changes

Partitioning will become more involved:

- Multilevel
  - MPI Processes
  - Multicore Threads
- Weighted
  - Different processing speeds
  - Different memory bandwidth

Performance and Memory Logging

- On CPU can use standard packages
  - gprof, TAU, PAPI
  - PETSc defines an extensible logging system (stages)
- For kernel, count manually
  - Might use source analysis on kernel
  - Hardware counters need better interface
- Need better modeling
  - Very large number of interacting threads

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

Outline

Robustness

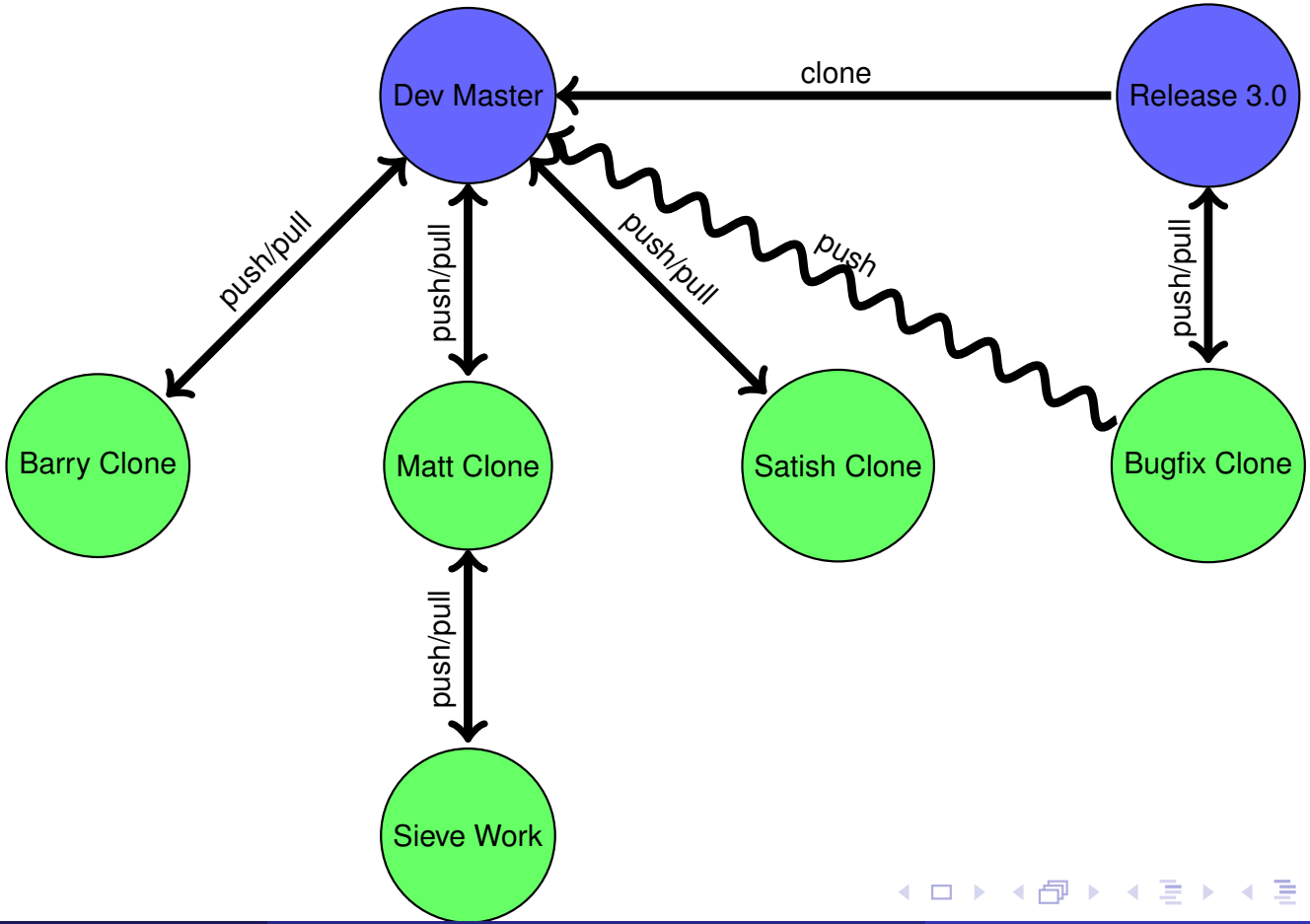
In the face of rapid interface change, we need:

- Version control
  - I recommend Mercurial, but Git is acceptable
- Unit testing
  - I recommend CppUnit, but it is not parallel
  - Also need model-based performance tests
- Regression testing
  - I recommend Buildbot
  - Performance regression is also important
- Vigorous email support
  - Every day, many developers

Robustness

In the face of rapid interface change, we need:

- Version control methodology
  - I recommend Mercurial, but Git is acceptable



Test Methodology

I see the testing proceeding in three phases:

1. Python kernel development with PyCUDA
   - Rapid prototyping
   - Easy development of benchmarking tools (petsc4py)
2. Transfer of kernels to C++ test harness
   - Replicate Python harness in C++, or
   - Use wrappers?
3. Integration into test applications
   - Regression tests
   - New support API

Outline

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  Ex: Autogenerated FEM integration
- Partitioning will be controlled by the library
  Ex: Partition for MPI and then for GPU
- Communication will be managed by the library
  Ex: Marshalling to GPU
- Assembly will be controlled by the algorithm
  Ex: Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  Ex Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  - Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  - Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  - Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  - Ex Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  Ex Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  - Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  - Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  - Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  - Ex Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  - Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  - Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  - Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  - Ex Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  - Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  - Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  - Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  - Ex Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  - Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  - Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  - Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  - Ex Substructuring (`PCFieldSplit`)

What Will Change?

More control will pass from user to library/compiler

- Kernels will be generated by the library
  - Ex Autogenerated FEM integration
- Partitioning will be controlled by the library
  - Ex Partition for MPI and then for GPU
- Communication will be managed by the library
  - Ex Marshalling to GPU
- Assembly will be controlled by the algorithm
  - Ex Substructuring (`PCFieldSplit`)

Outline

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent
  (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent
  (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)

Hierarchical Interface

We encode topological structure as a (nested) set of restrictions.

- Hierarchy is encoded by a DAG (`Sieve`)
- PETSc handles mappings and parallelism
- Allows separation of
  - analytic from topological code
  - topological from algebraic code

Mesh Algorithms for PDE with Sieve I: Mesh Distribution, Knepley and Karpeev, Sci. Prog., 17(3), 2009.

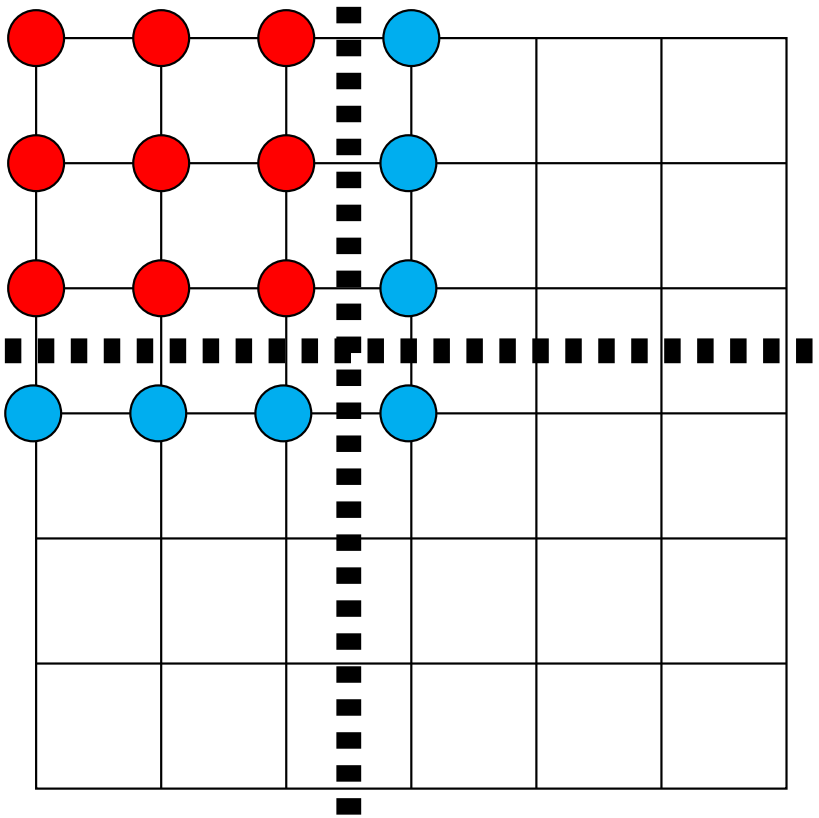
DMDA Vectors

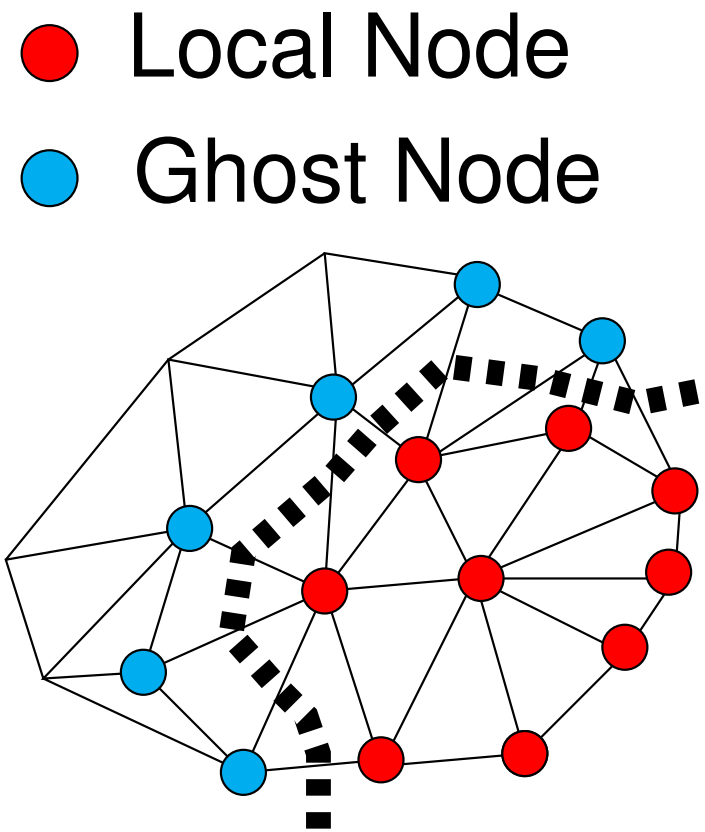
- The **DMDA** object contains only layout (topology) information
  - All field data is contained in PETSc **Vecs**
- Global vectors are parallel
  - Each process stores a unique local portion
  - DMCreateGlobalVector(DM da, Vec *gvec)
- Local vectors are sequential (and usually temporary)
  - Each process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

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

```
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

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().

Mesh Interfaces

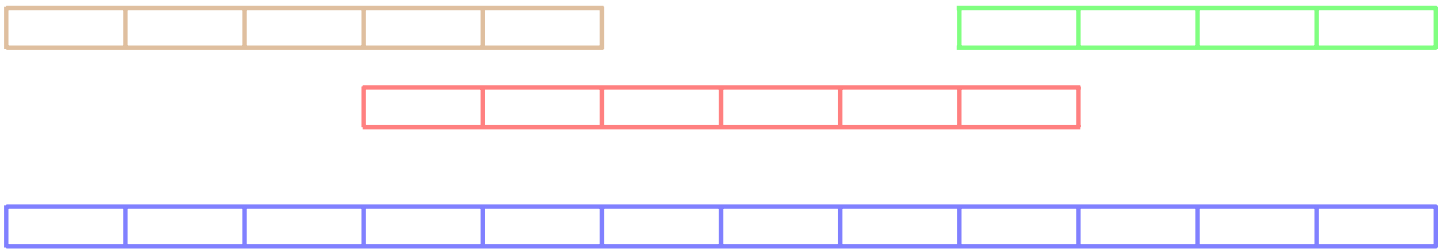
Global

- `Vec`
- Unique storage
- Global numbering
- For solver interaction

Mesh Interfaces

Local

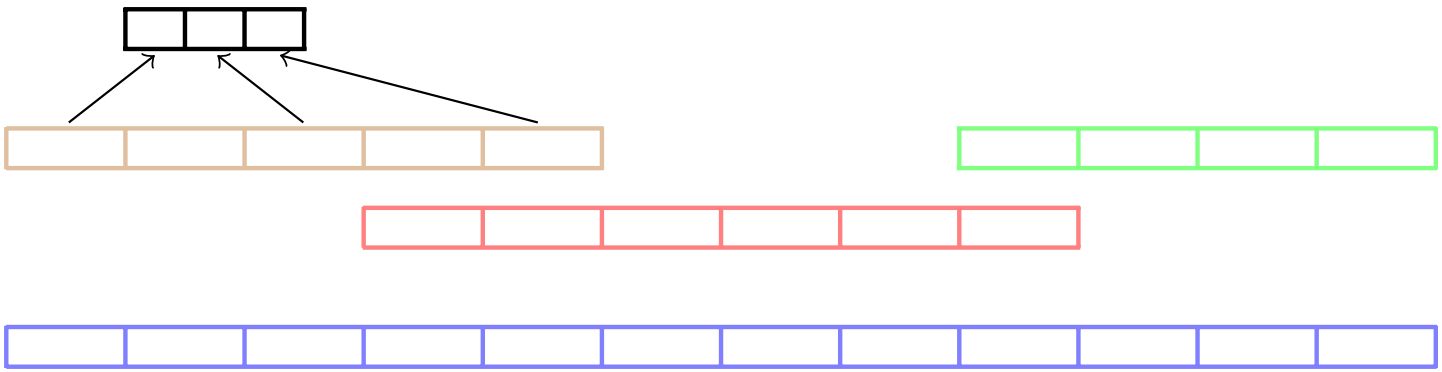
- `Section`
- Redundant storage
- For accumulation, more general fusion interface



Mesh Interfaces

Cell

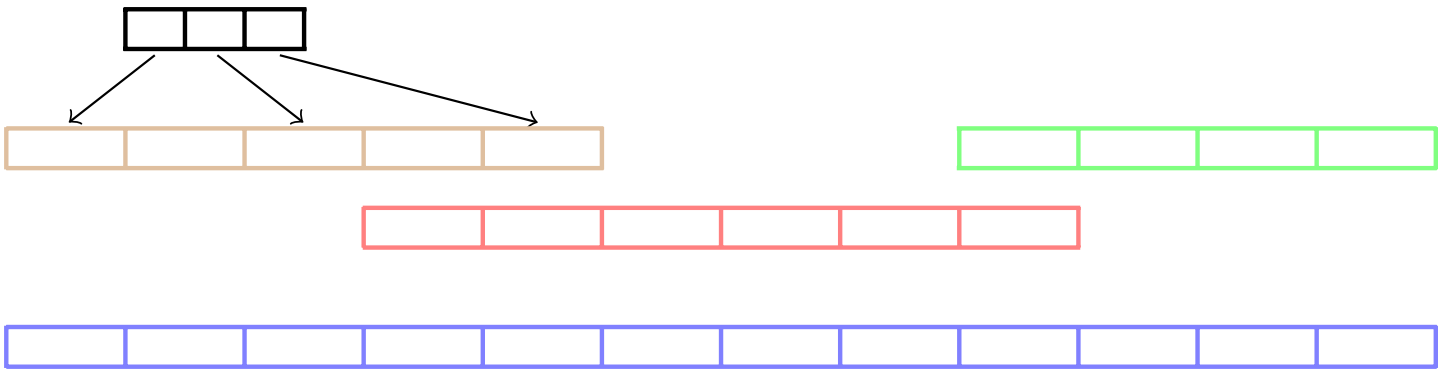
- Accesses as raw `double []` from `restrict()`
- Use `update()` to get back to local storage
- Redundant storage
- For user interaction



Mesh Interfaces

Cell

- Accesses as raw `double []` from `restrict()`
- Use `update()` to get back to local storage
- Redundant storage
- For user interaction



GPU Interaction

Analytic routines become GPU kernels.

Kernels can be

- FD Stencils
- FEM and FV Integrals
- Domain Cells for Integral Equations

Storage can be reached by appropriate `restrict()` call

- Usually includes the closure
- Building block for marshalling

GPU programming in General

- What design ideas are useful?
- How do we customize them for GPUs?
- Can we show an example?

Reorder for Locality

Exploits “nearby” operations to aggregate computation

- Can be *temporal* or *spatial*
- Usually exploits a cache
- Difficult to predict/model on a modern processor

Reorder for Locality
GPU Differences

We have to manage our “cache” explicitly

- The NVIDIA 1060C shared memory is only 16K for 32 threads
- We must also manange “main memory” explicitly
  - Need to move data to/from GPU
- Must be aware of limited precision when reordering
- Can be readily modeled
- Need tools for automatic data movement (marshalling)

Reorder for Locality
Example

Data-Aware Work Queue

- A work queue manages many small tasks
  - Dependencies are tracked with a DAG
  - Queue should manage a single computational phase (supertask)
- Nodes also manage an input and output data segment
  - Specific classes can have known sizes
  - Can hold main memory locations for segments
- Framework manages marshalling:
  - Allocates contiguous data segments
  - Calculates segment offsets for tasks
  - Marshalls (moves) data
  - Passes offsets to supertask execution

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

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

FIAT Integration

The `quadrature.fiat` file contains:

- An element (usually a family and degree) defined by FIAT
- A quadrature rule

It is run

- automatically by `make`, or
- independently by the user

It can take arguments

- `-element_family` and `-element_order`, or
- `make` takes variables `ELEMENT` and `ORDER`

Then `make` produces `bratu_quadrature.h` with:

- Quadrature points and weights
- Basis function and derivative evaluations at the quadrature points
- Integration against dual basis functions over the cell
- Local dofs for Section allocation

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

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

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
```

What Is Most Important?

- Multiple Languages will be Necessary
  - Build systems need the most work
- Users will give up more Control
  - Move toward a hierarchical paradigm

What Is Most Important?

- Multiple Languages will be Necessary
  - Build systems need the most work
- Users will give up more Control
  - Move toward a hierarchical paradigm

What Is Most Important?

- Multiple Languages will be Necessary
  - Build systems need the most work
- Users will give up more Control
  - Move toward a hierarchical paradigm

What Is Most Important?

- Multiple Languages will be Necessary
  - Build systems need the most work
- Users will give up more Control
  - Move toward a hierarchical paradigm

Change alone is unchanging
— Heraclitus, 544–483 BC