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

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

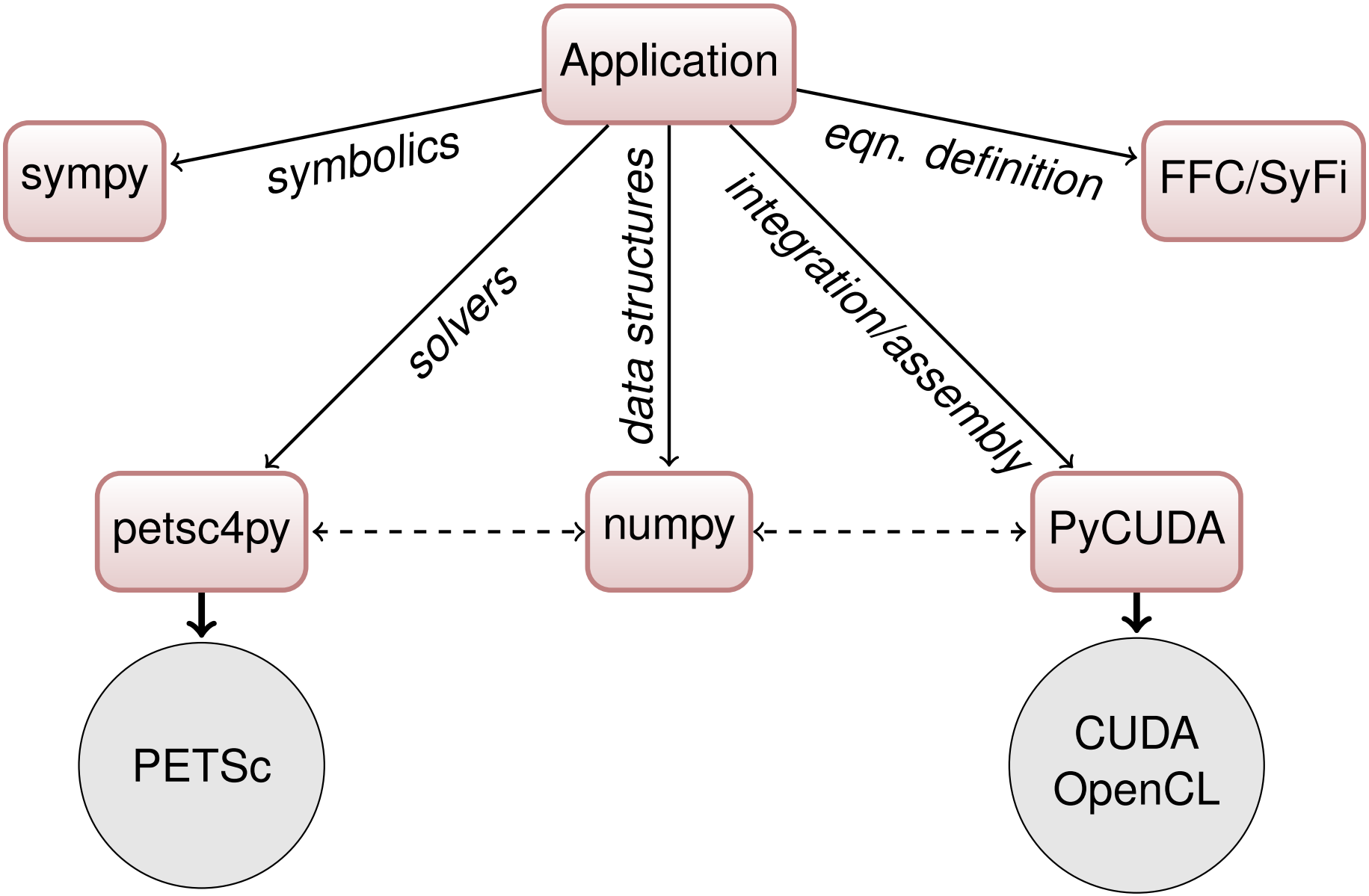
New Model for Scientific Software

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

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

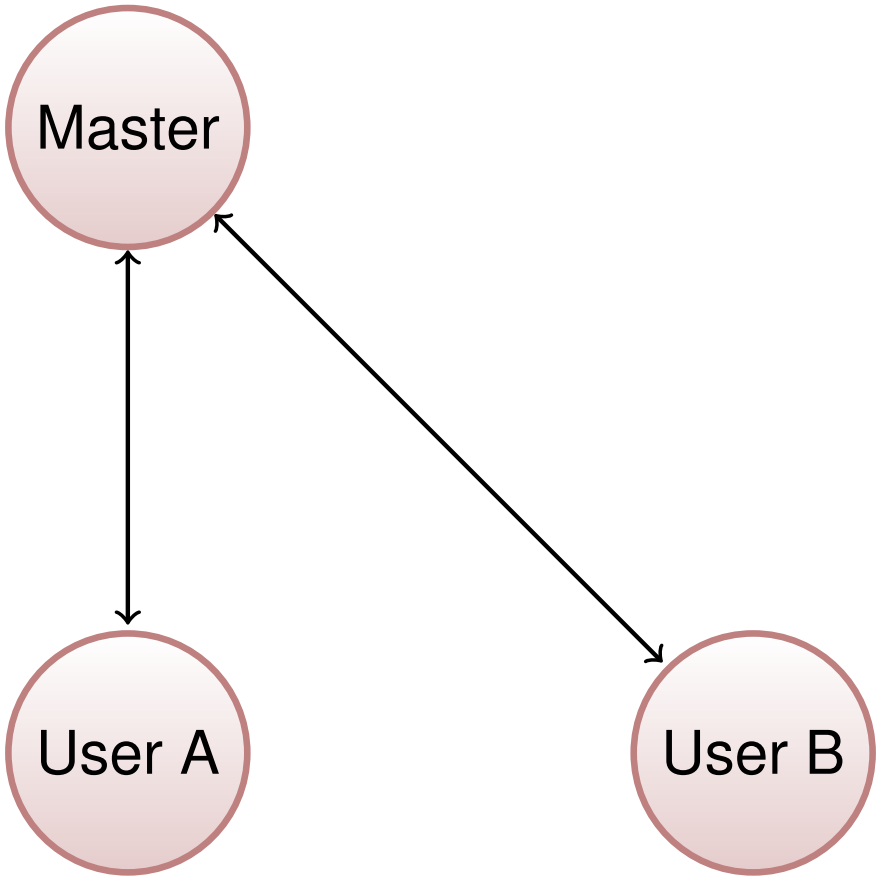


Figure: Master Repository with User Clones

Project Workflow

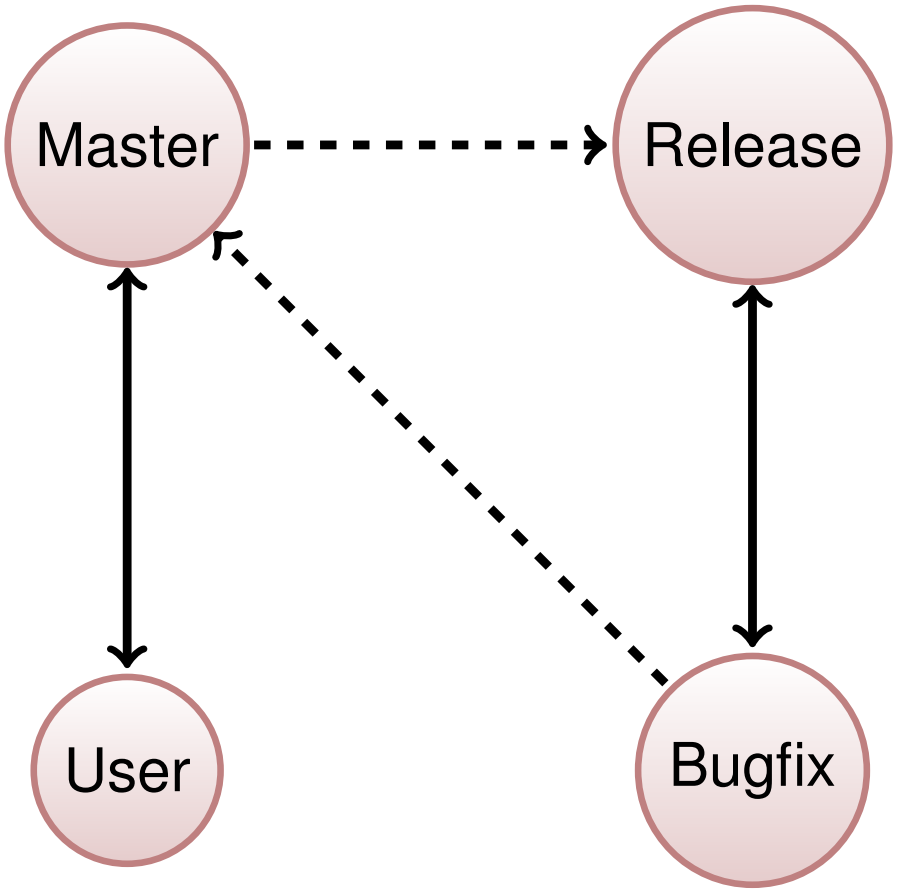


Figure: Project with Release and Bugfix Repositories

Outline

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

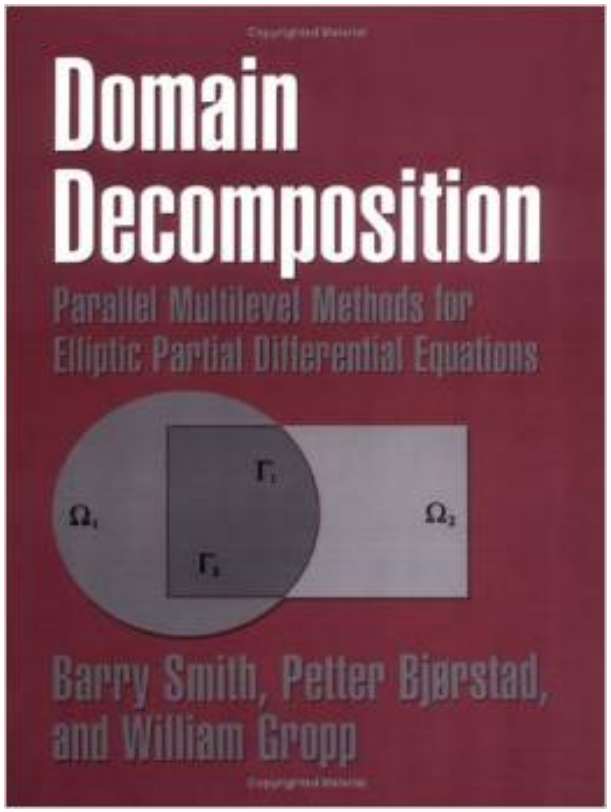
Outline

How did PETSc Originate?

PETSc was developed as a Platform for Experimentation

We want to experiment with different

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



The Role of PETSc

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

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

— Barry Smith

Advice from Bill Gropp

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

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

What is PETSc?

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

Free

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

Supported

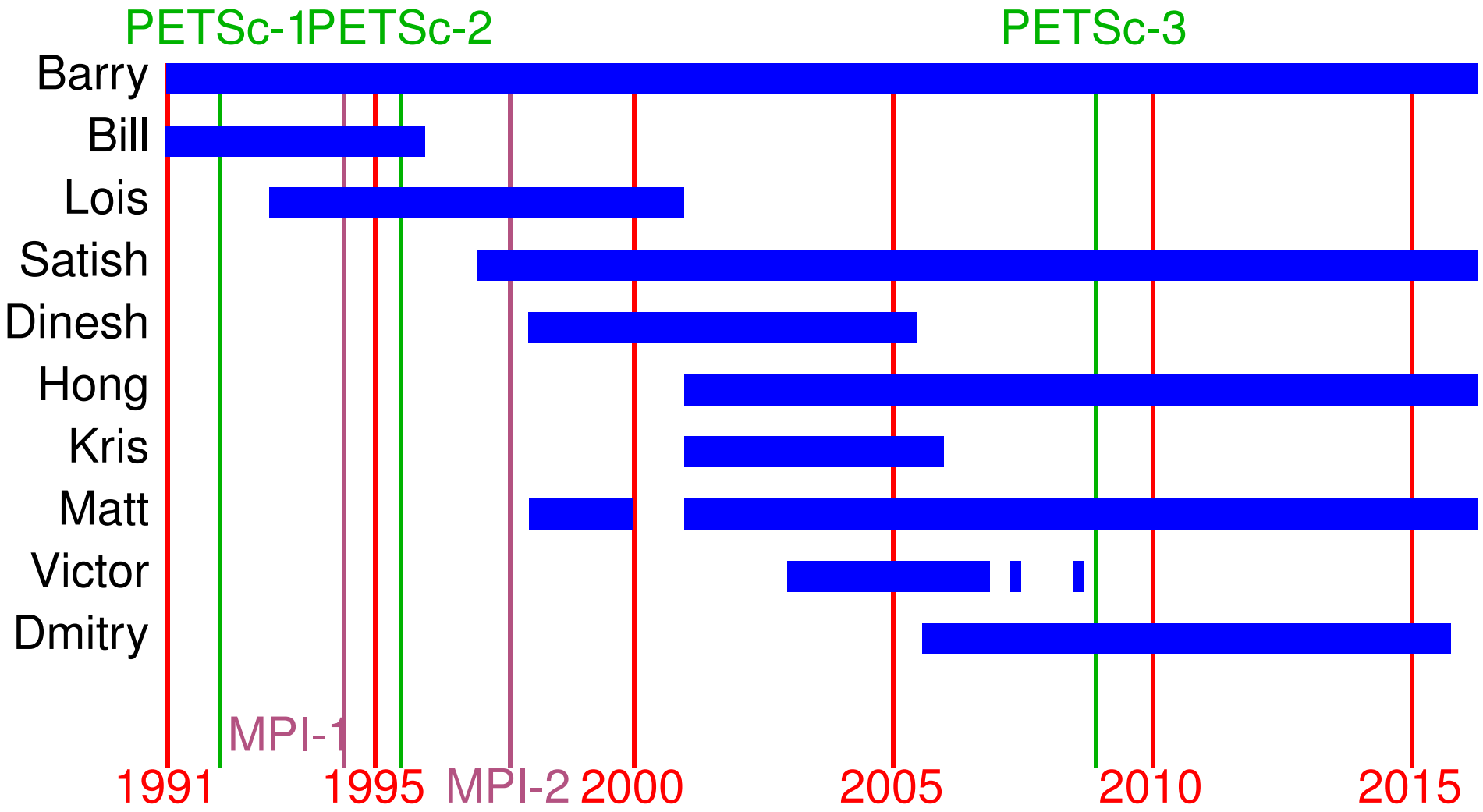
- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov

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

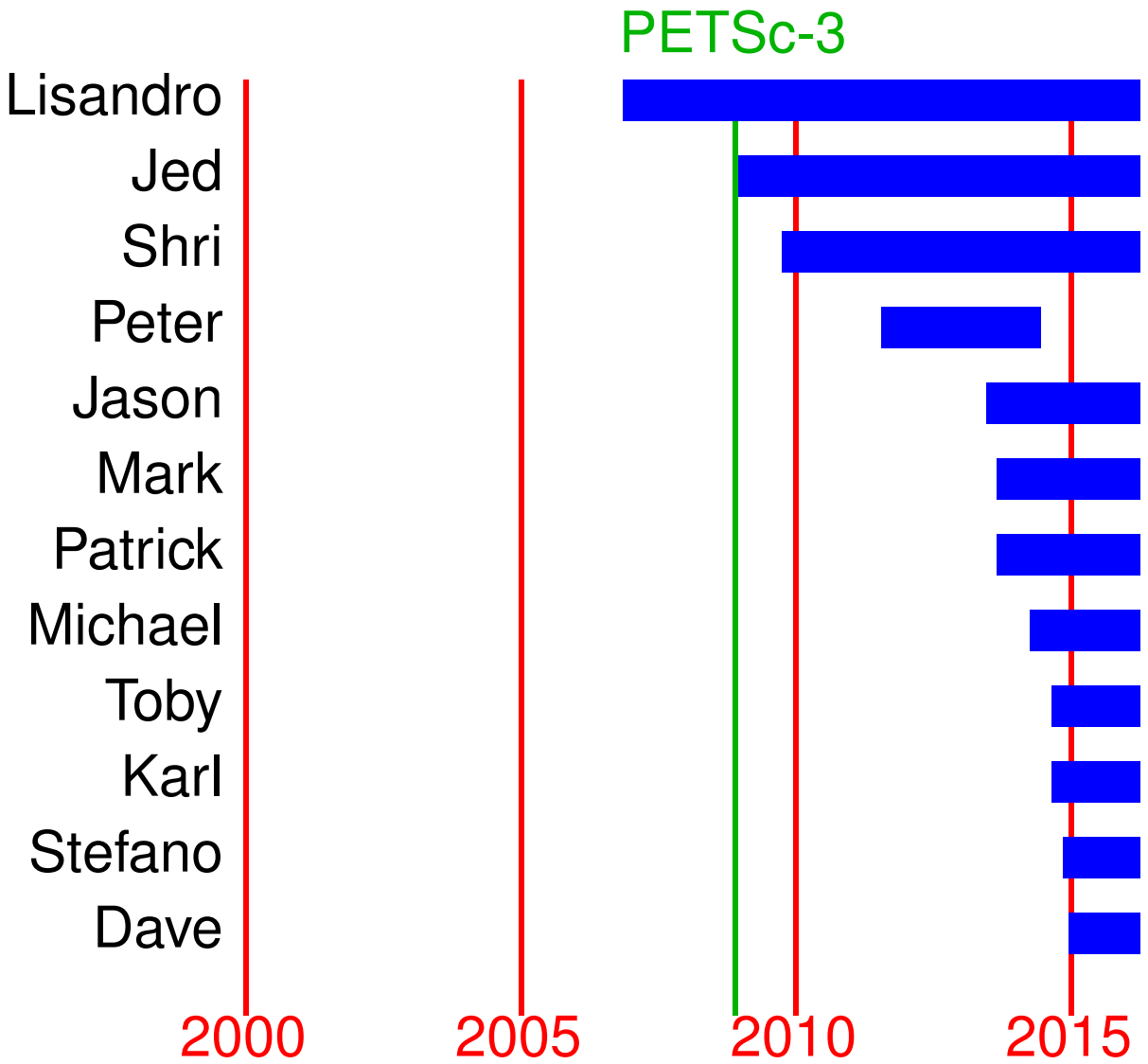
What is PETSc?

- 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
    - ECP, PSAAPIII, AMR, BES, SciDAC, MICS
  - National Science Foundation
    - CSSI, SI2, CIG, CISE
  - Intel Parallel Computing Center

Timeline (Old People)



Timeline (Young People)



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 1,500,000 cores efficiently
  - Gordon Bell Prize Mantle Convection on IBM BG/Q Sequoia
- 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 1,500,000 cores efficiently
  - Gordon Bell Prize Mantle Convection on IBM BG/Q Sequoia
- 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 1,500,000 cores efficiently
  - Gordon Bell Prize Mantle Convection on IBM BG/Q Sequoia
- PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code

Outline

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

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

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

```
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

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

```
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

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

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

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

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

Code Generation

Big Idea: **Code Generation**

- Compile time examples
  - Dense linear algebra
  - Digital Signal Processing
  - FEM Accumulation/Assembly
  - Model coupling
- Runtime support
  - Inspector-Executor
  - PGAS
  - Libraries, if written correctly, can be oblivious

Code Generation

Big Idea: **Code Generation**

- Compile time examples
  - Dense linear algebra
  - Digital Signal Processing
  - FEM Accumulation/Assembly
  - Model coupling
- Runtime support
  - Inspector-Executor
  - PGAS
  - Libraries, if written correctly, can be oblivious

Code Generation

Big Idea: **Code Generation**

- Compile time examples
  - Dense linear algebra
  - Digital Signal Processing
  - FEM Accumulation/Assembly
  - Model coupling
- Runtime support
  - Inspector-Executor
  - PGAS
  - Libraries, if written correctly, can be oblivious

Other Software

- High Order, Discontinuous Galerkin FEM
  - Hedge, Andreas Klöckner
- Cartesian, Finite Difference Multigrid
  - OpenCurrent, Jon Cohen
- Fast Multipole Method
  - PetFMM, Lorena Barba, Felipe Cruz, Matthew Knepley
- Parallel Linear Algebra and Solvers
  - PETSc, Barry Smith, et.al.
  - Cusp, Nathan Bell, et.al.
  - CUSPARSE, NVIDIA

Outline

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 analytic and geometric parts:

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{4}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} \tag{5}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| d\mathbf{x} \tag{6}$$

$$= \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} d\mathbf{x} \tag{7}$$

$$= G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma} \tag{8}$$

Coefficients are also put into the geometric part.

Form Decomposition

Additional fields give rise to multilinear forms.

$$\int_{\mathcal{T}} \phi_i(\mathbf{x}) \cdot \left(\phi_k(\mathbf{x}) \nabla \phi_j(\mathbf{x})\right) dA \tag{9}$$

$$= \int_{\mathcal{T}} \phi_i^\beta(\mathbf{x}) \left(\phi_k^\alpha(\mathbf{x}) \frac{\partial \phi_j^\beta(\mathbf{x})}{\partial x_\alpha}\right) dA \tag{10}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| dA \tag{11}$$

$$= \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} dA \tag{12}$$

$$= G^{\alpha\gamma}(\mathcal{T}) K_{\alpha\gamma}^{ijk} \tag{13}$$

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

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) dA \tag{14}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} dA \tag{15}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| dA \tag{16}$$

$$= |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k J_k^{\beta\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l J_l^{\gamma\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{17}$$

$$= J_k^{\beta\alpha} J_l^{\gamma\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{18}$$

$$= G_{kl}^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ijkl} \tag{19}$$

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

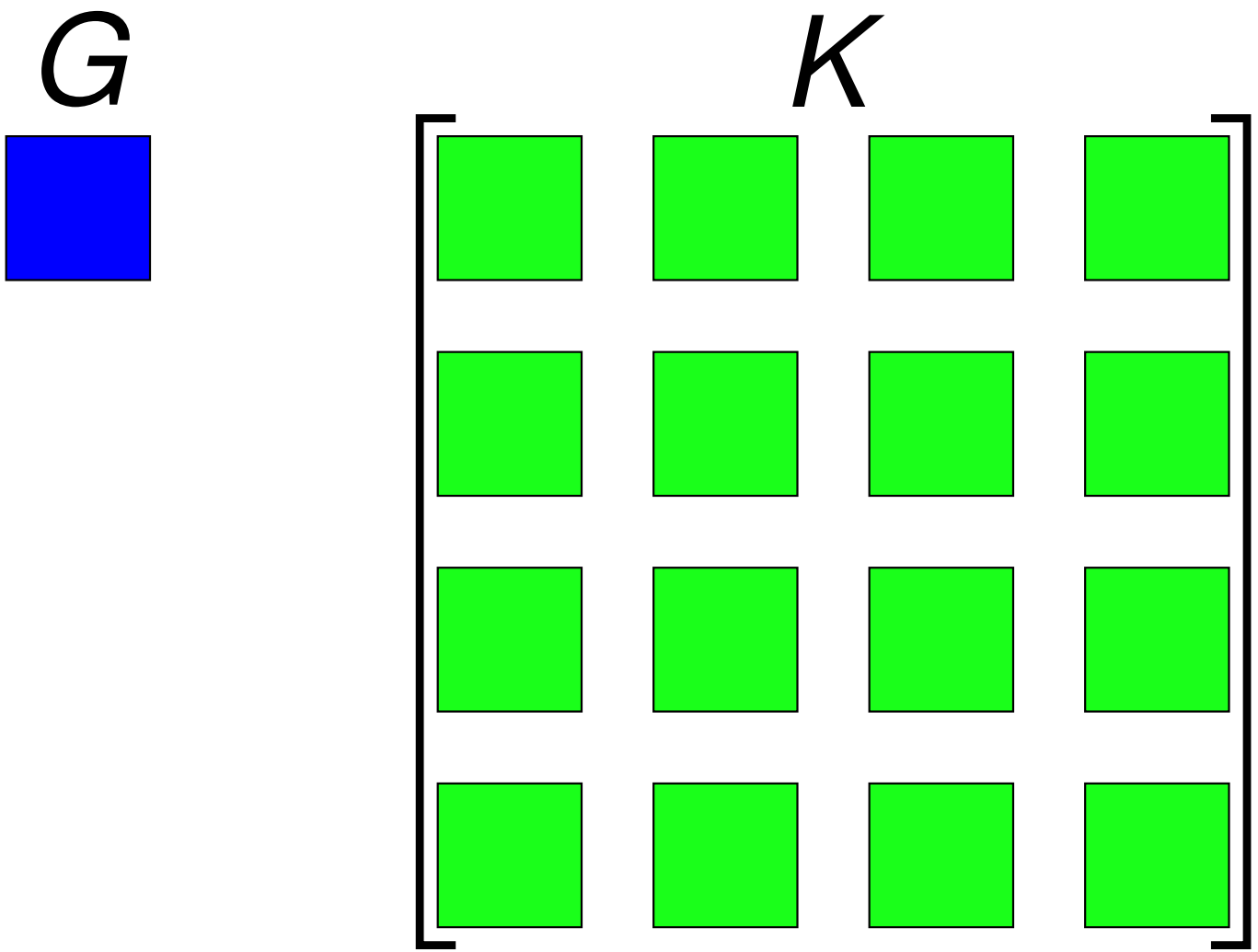


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Basic Contraction

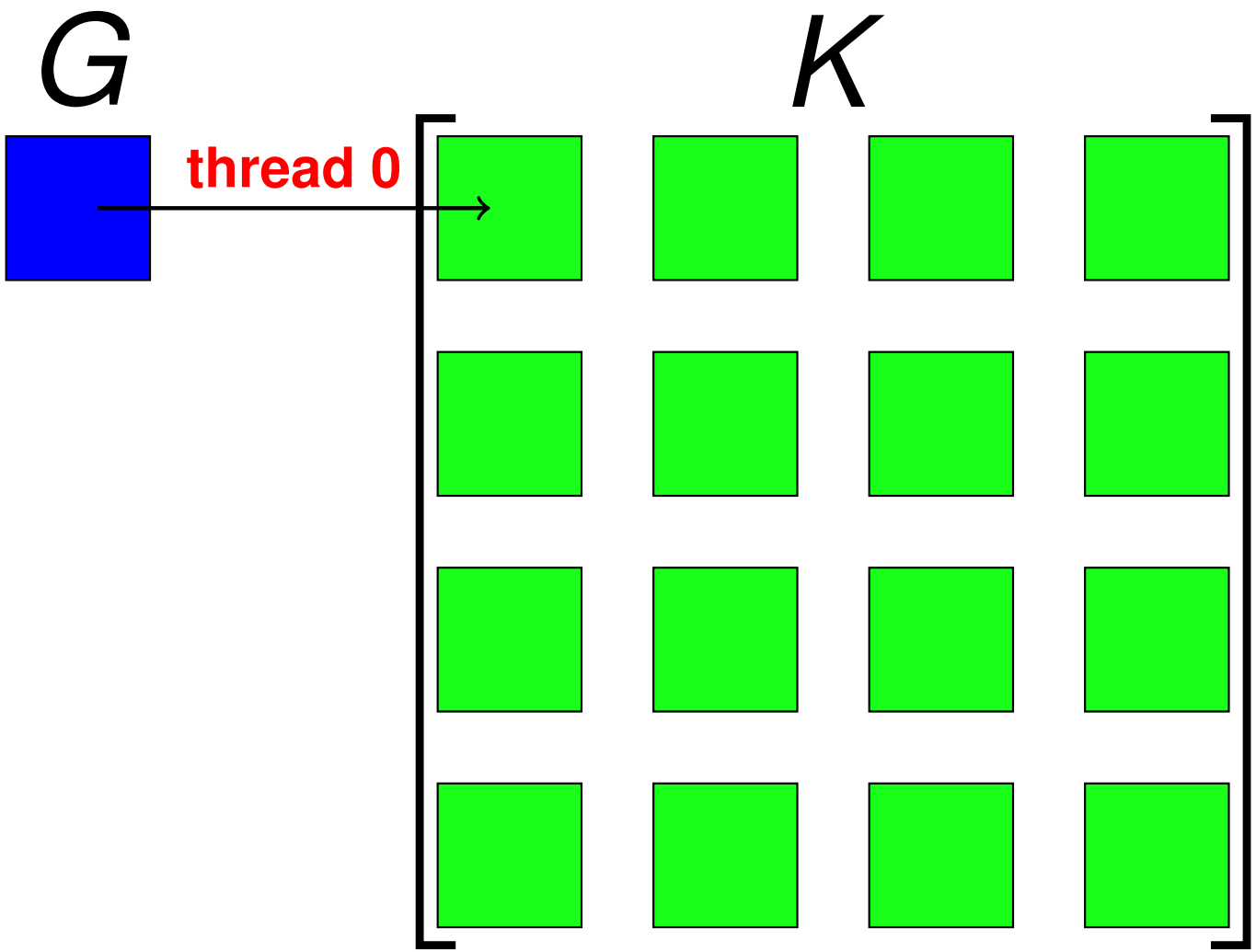


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Basic Contraction

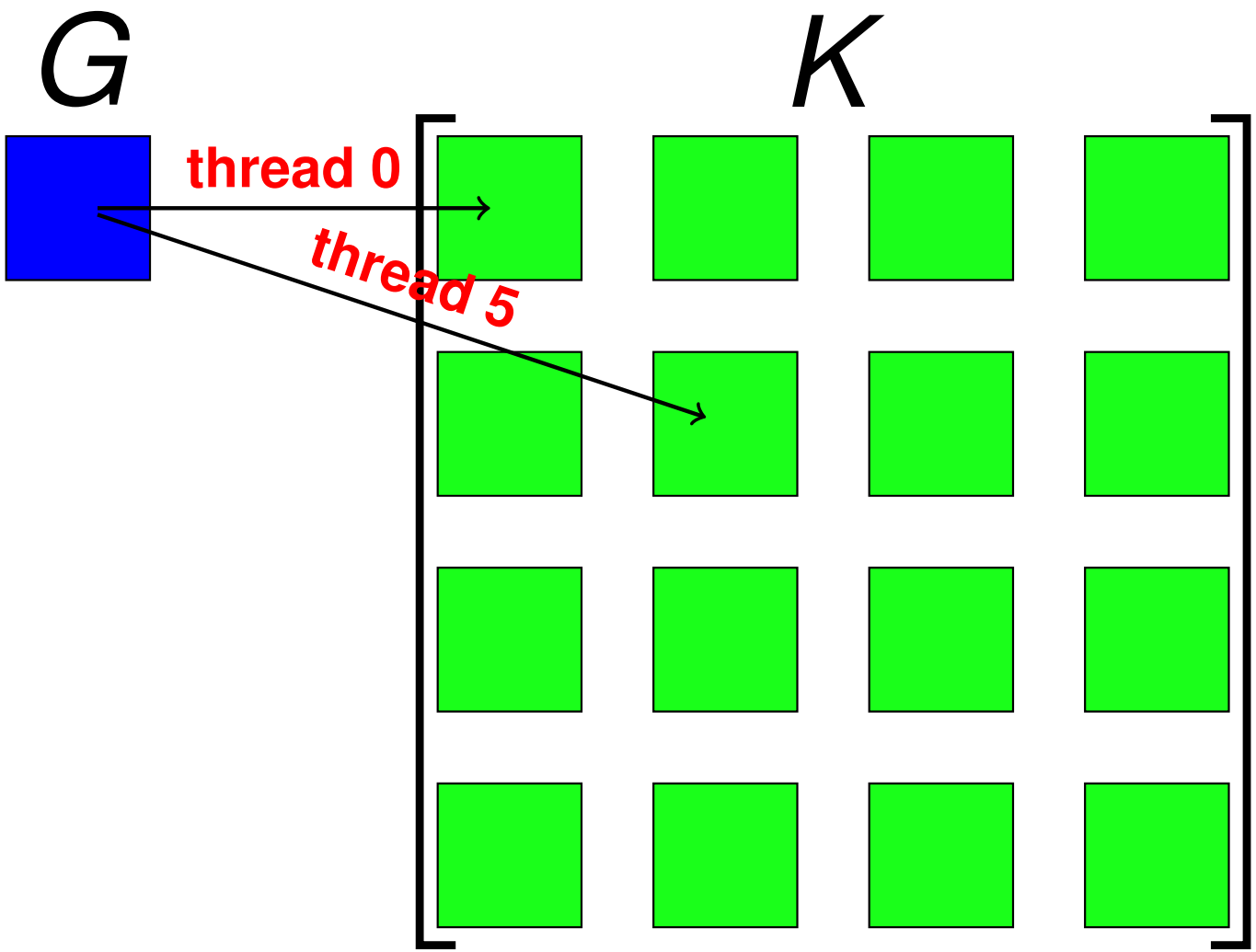


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Basic Contraction

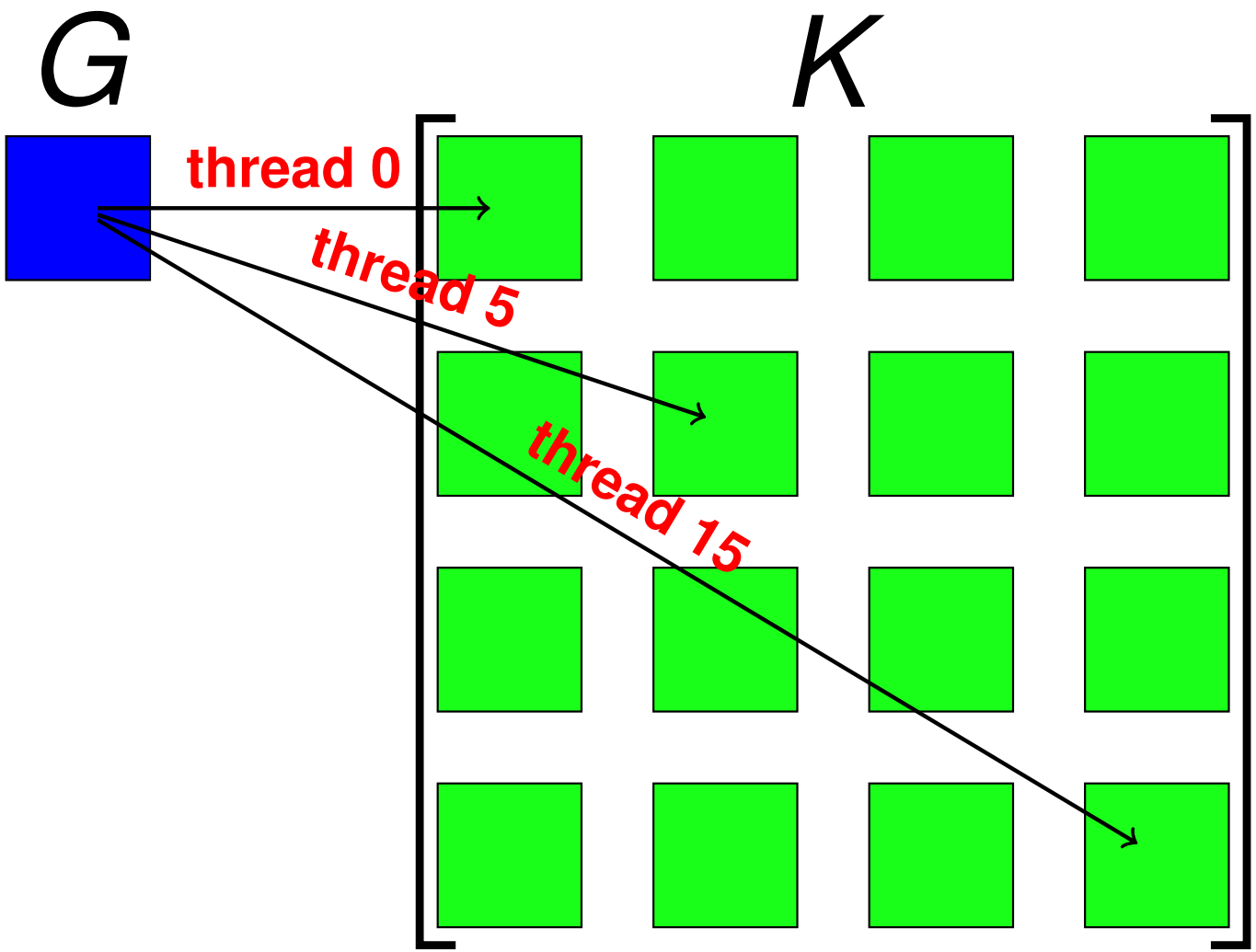


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

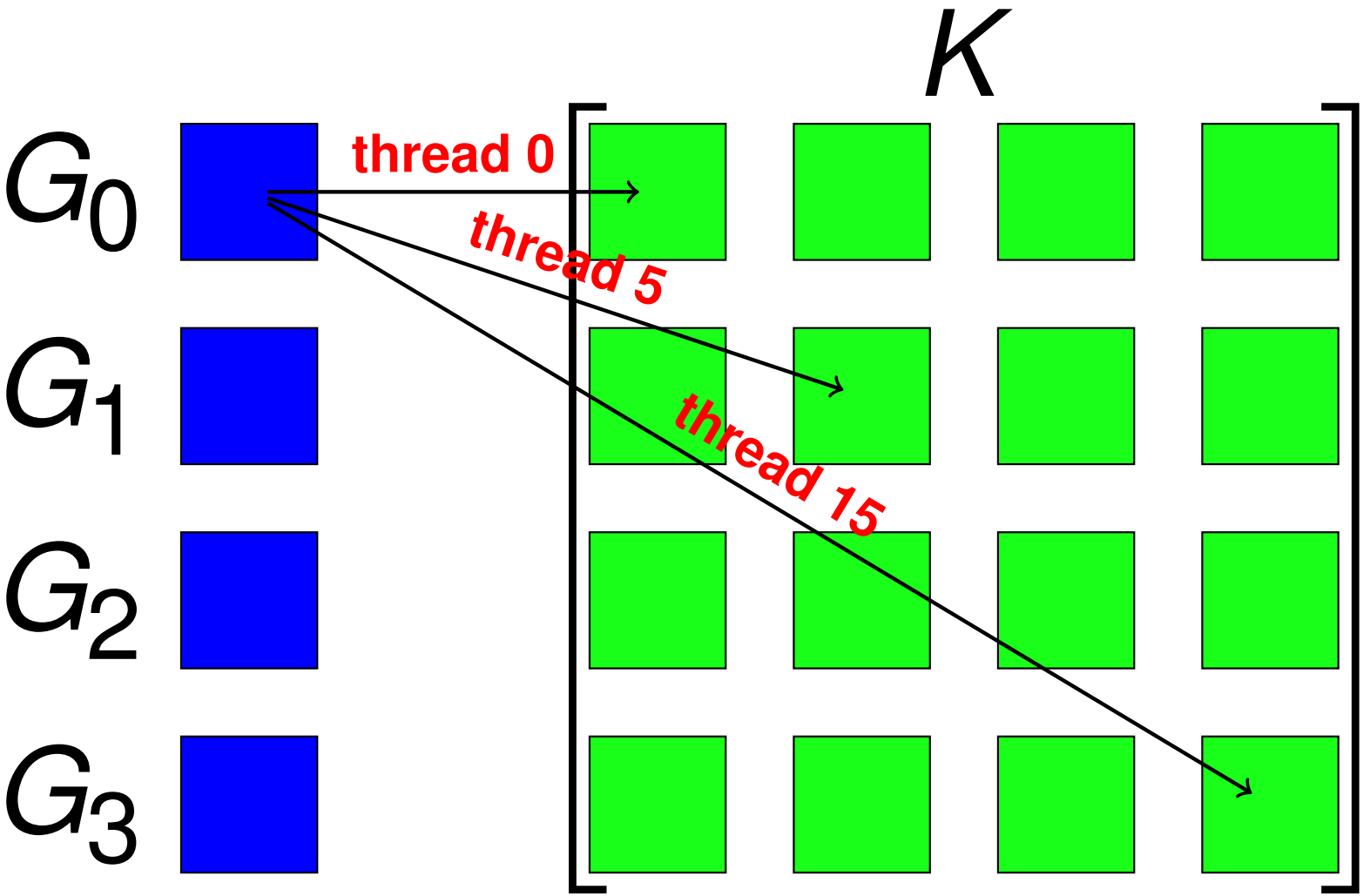


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

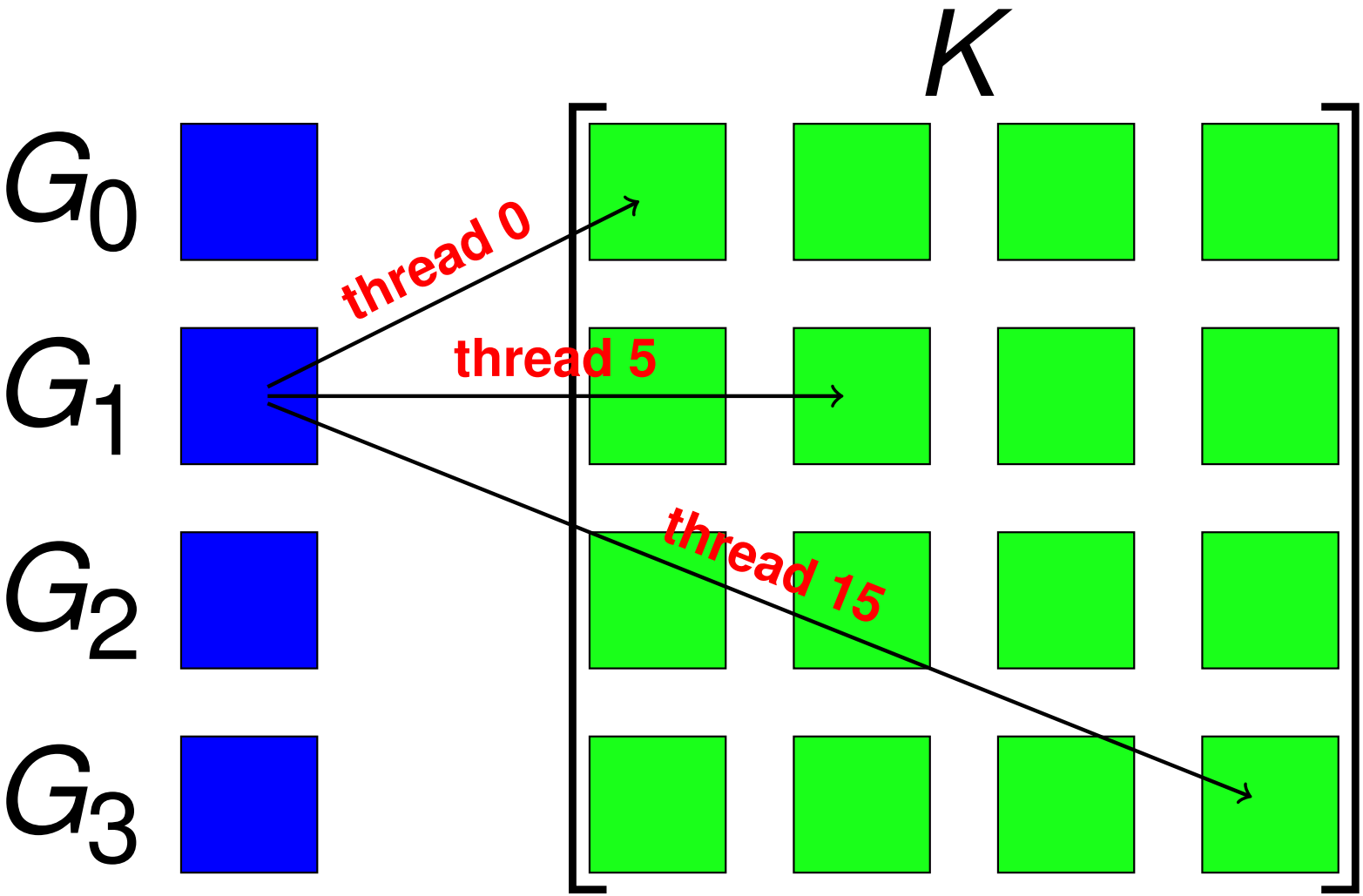


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

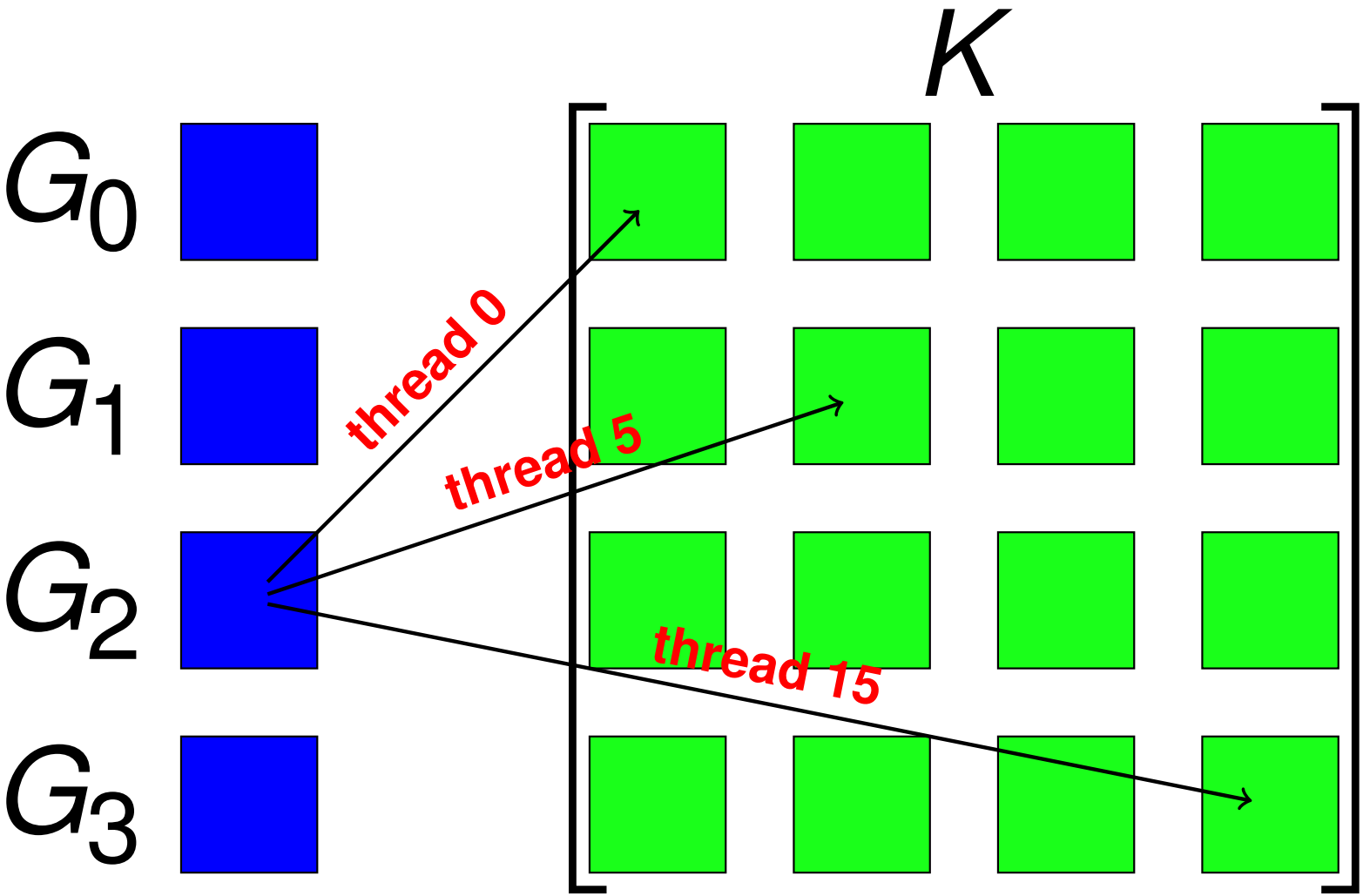


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

Computational Flexibility

Element Batch Size

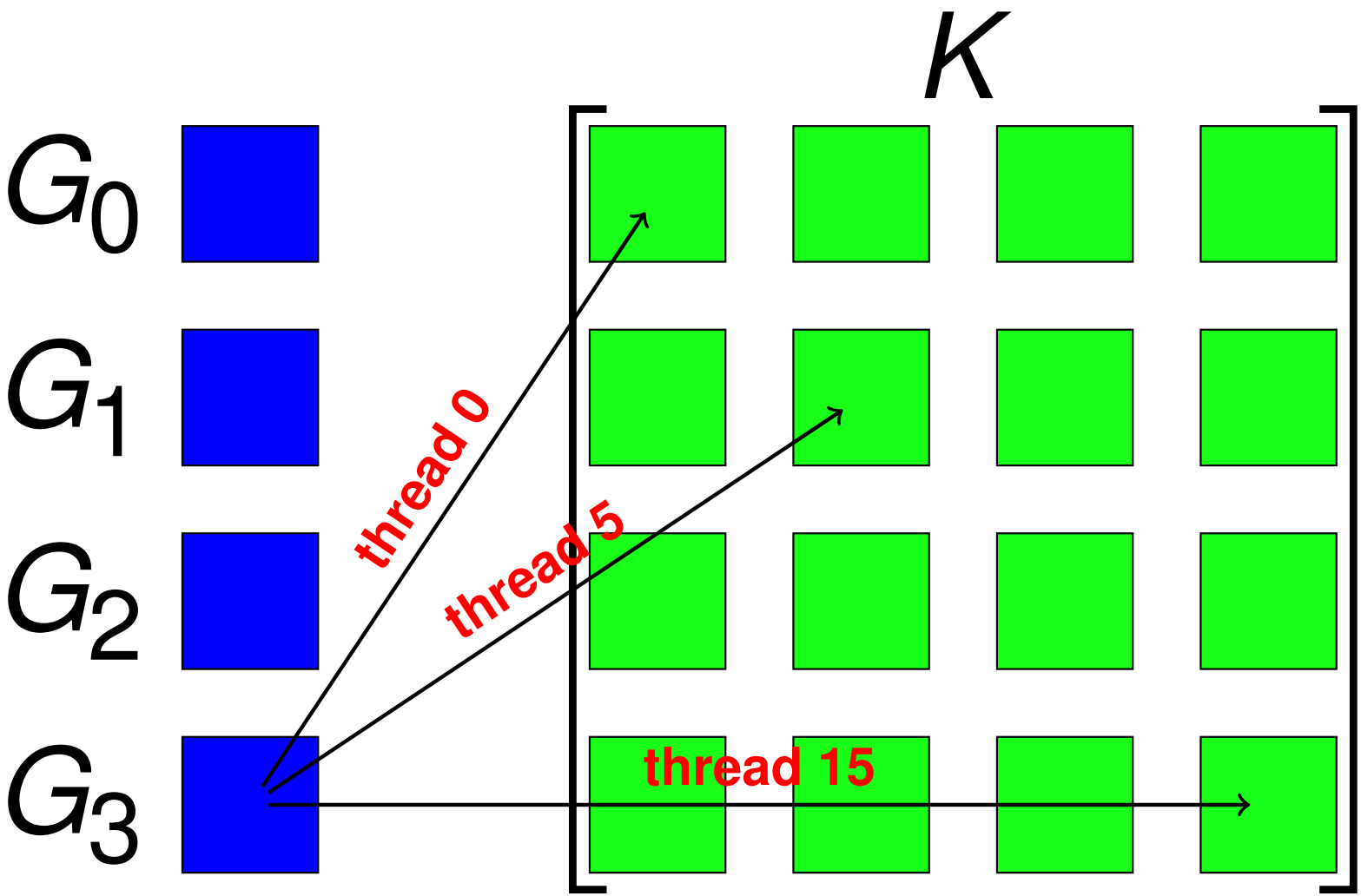
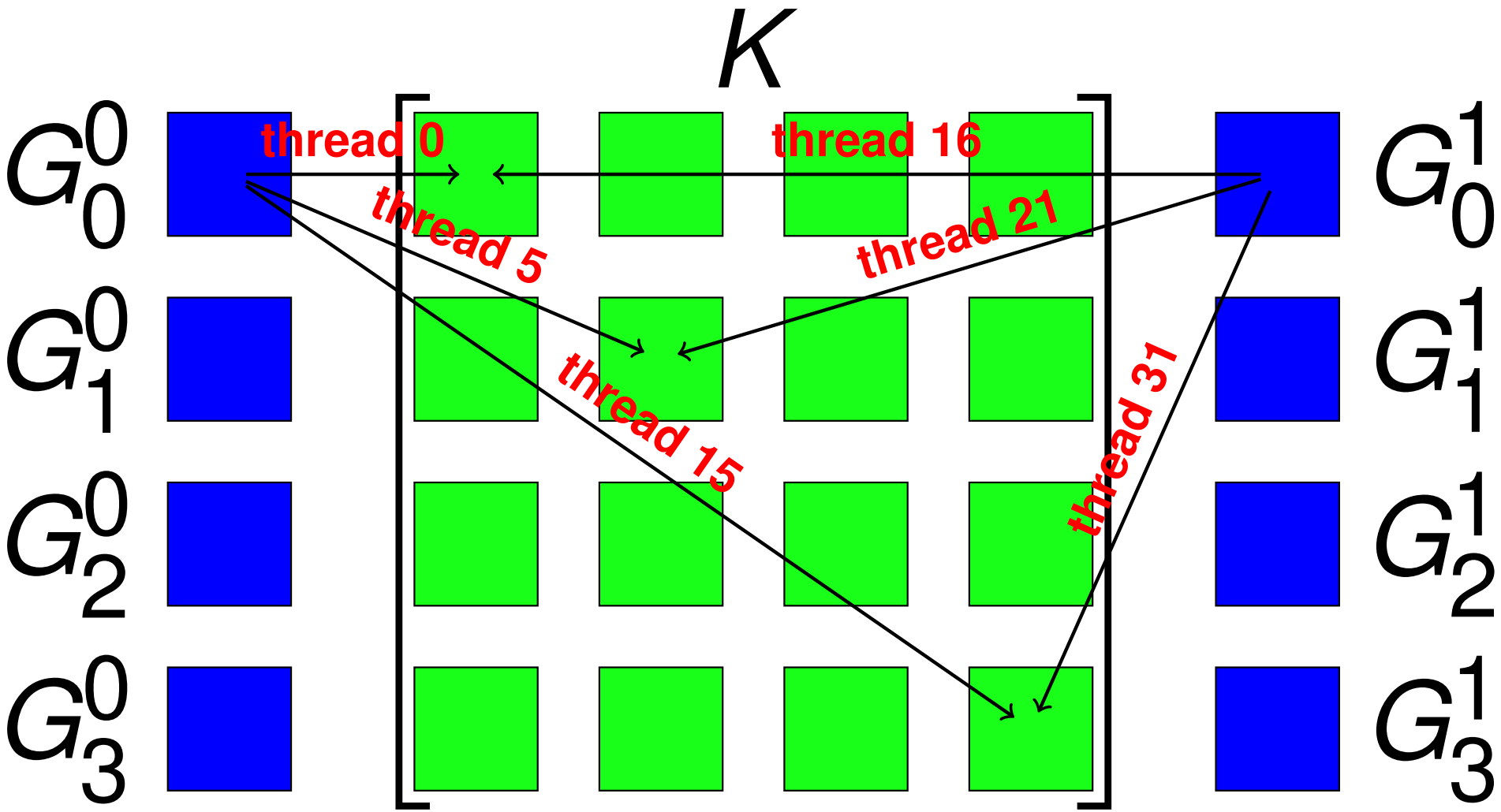


Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$

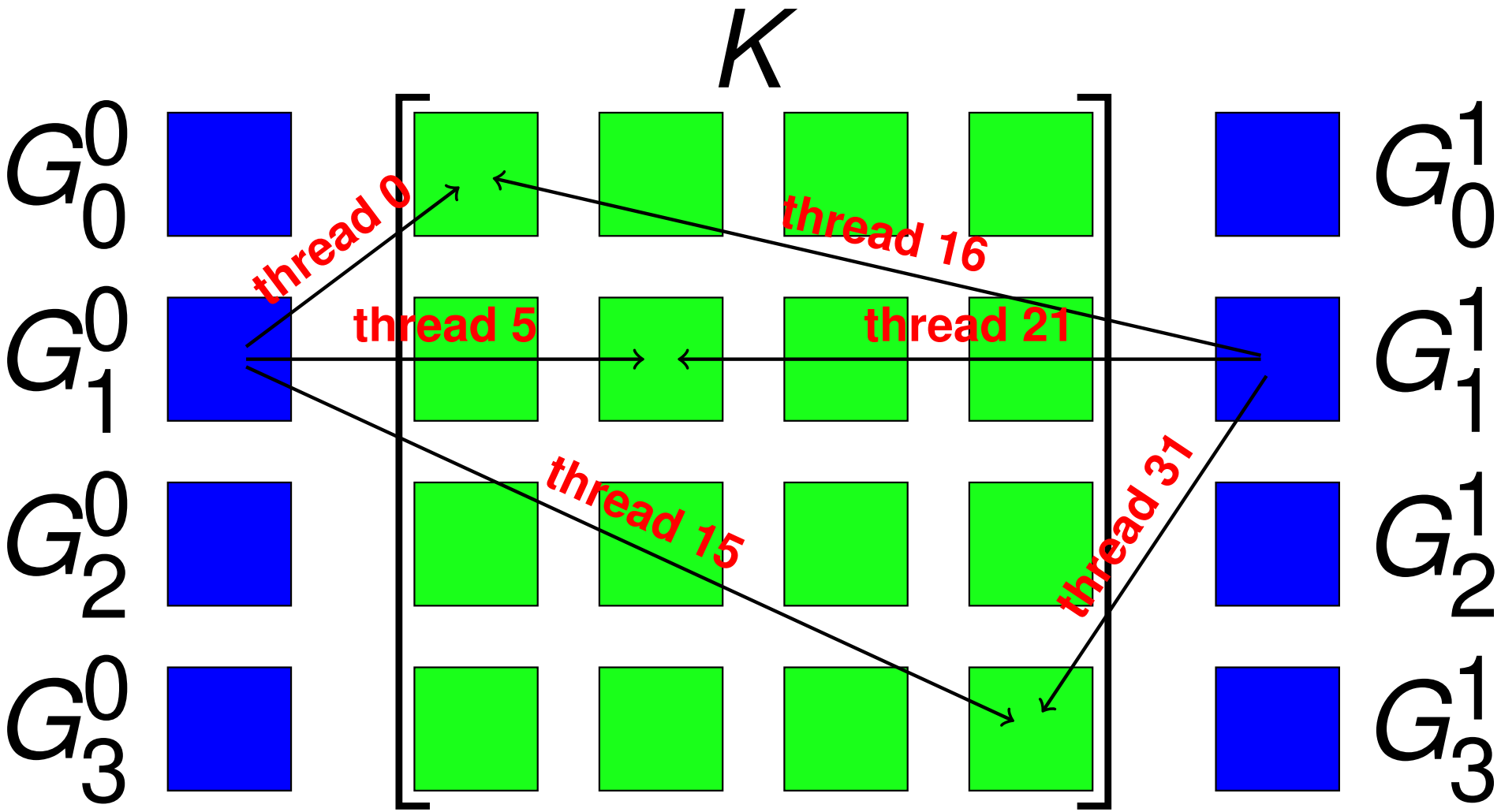
Computational Flexibility

Concurrent Elements



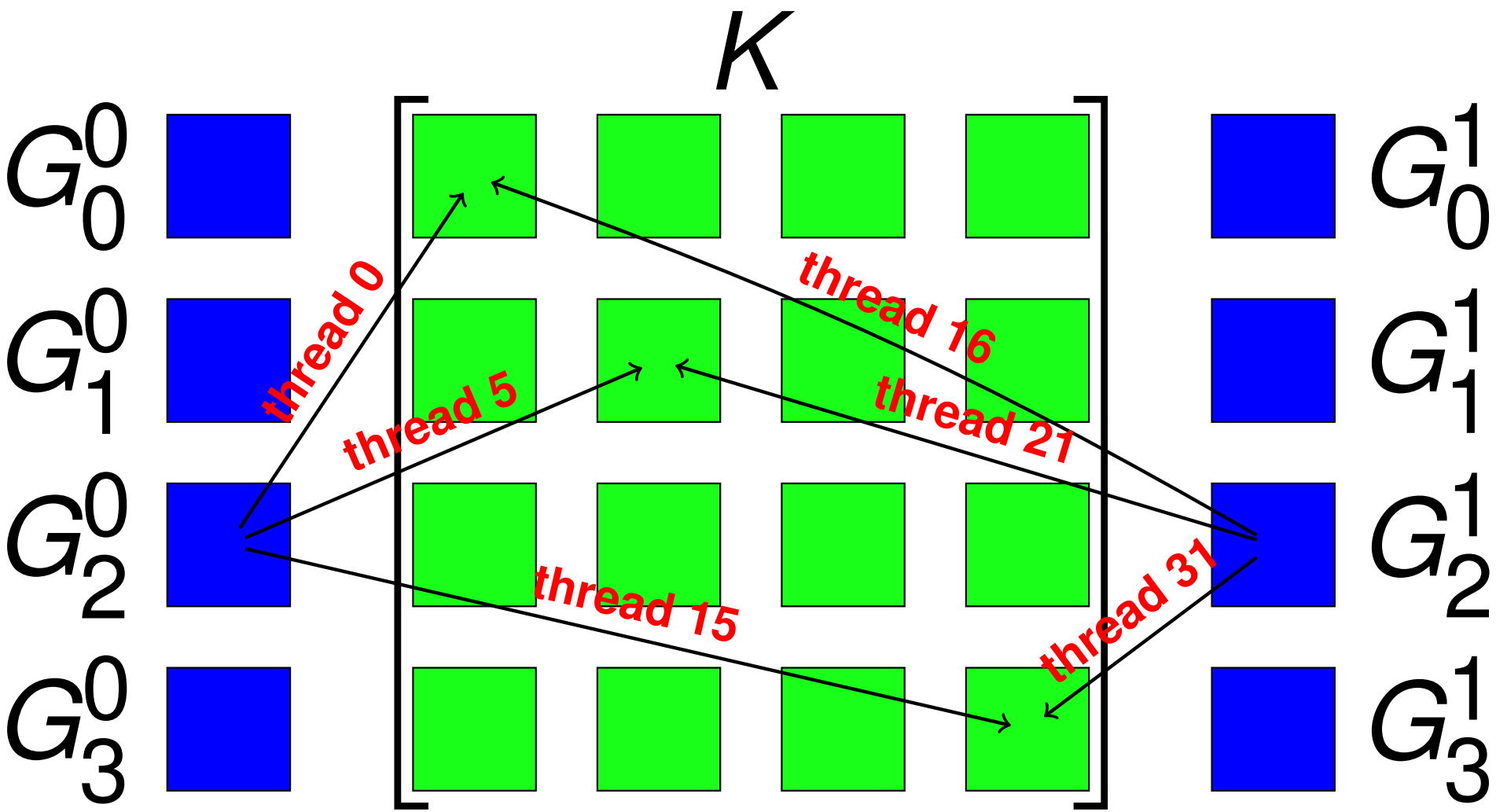
Computational Flexibility

Concurrent Elements



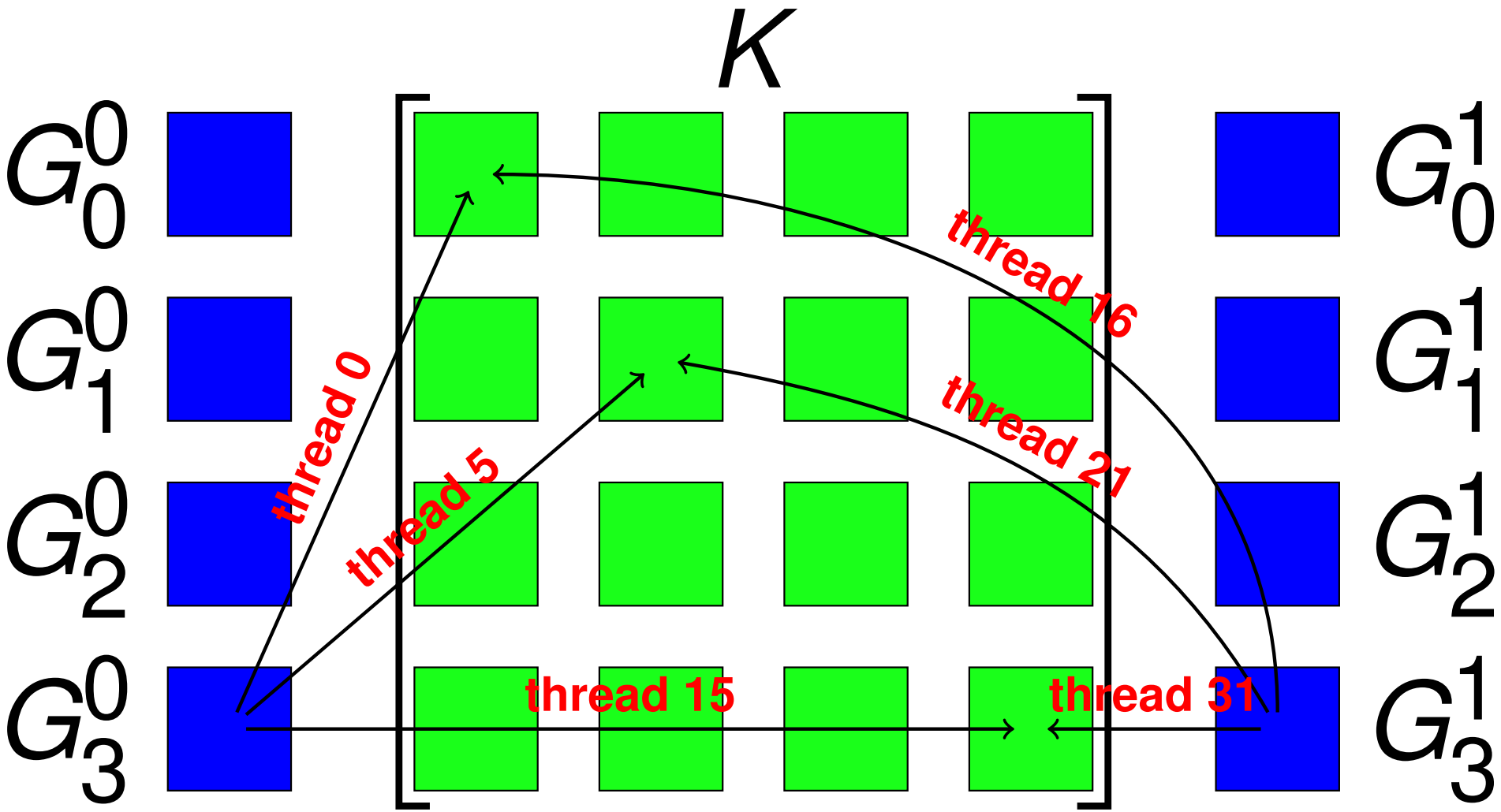
Computational Flexibility

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 $N_{\text{basis}} \times N_{\text{basis}} \times N_L$

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

$$\text{blockDim} = (N_L/N_B, 1, 1)$$

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

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

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]
```