

GPUs in Computational Science

Matthew Knepley¹

¹Computation Institute
University of Chicago

Colloquium
Simula Research
Oslo, Norway, September 1, 2010



Chicago Automated Scientific Computing Group:

- Prof. Ridgway Scott
 - Dept. of Computer Science, University of Chicago
 - Dept. of Mathematics, University of Chicago
- Peter Brune, developer of DFT for biology
 - Dept. of Computer Science, University of Chicago
- Andy Terrel, developer of Rheagen and automated FEM
 - Dept. of Computer Science and TACC, University of Texas at Austin

Outline

- 1 Introduction
- 2 Tools
- 3 FEM on the GPU
- 4 PETSc-GPU
- 5 Conclusions

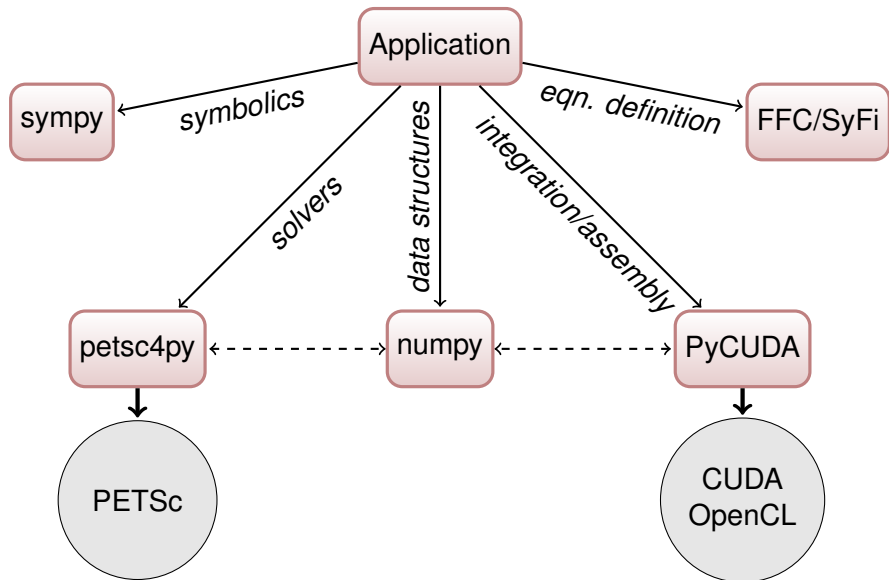
New Model for Scientific Software

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 Introduction
- 2 Tools**
 - numpy
 - petsc4py
 - PyCUDA
 - FEniCS
- 3 FEM on the GPU
- 4 PETSc-GPU
- 5 Conclusions

Outline

2 Tools

- numpy
- petsc4py
- PyCUDA
- FEniCS

`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

2 Tools

- numpy
- **petsc4py**
- PyCUDA
- FEniCS

`petsc4py` 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

- Configure PETSc using `-download-petsc4py`
 - Can also use `-download-mpi4py`
- Downloaded to `externalpackages/petsc4py-version`
 - Demo code is here
- Installed into PETSc lib directory
- Add `$PETSC_DIR/$PETSC_ARCH/lib` to **PYTHONPATH**

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

2 Tools

- numpy
- 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')
```

GPU:

```
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

2 Tools

- numpy
- petsc4py
- PyCUDA
- **FEniCS**

Weak Form Definition

Laplacian

```
# Pk element
element = FiniteElement("Lagrange",
                        domains[self.dim], k)
v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(element)

a = inner(grad(v), grad(u))*dx
L = v*f*dx
```

Form Decomposition

Element integrals are decomposed into analytic and geometric parts:

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

$$= \int_{\mathcal{T}} \frac{\partial \phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial \phi_j(\mathbf{x})}{\partial x_\alpha} dA \quad (2)$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |J| dA \quad (3)$$

$$= \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} dA \quad (4)$$

$$= \mathbf{G}^{\beta\gamma}(\mathcal{T}) \mathbf{K}_{\beta\gamma}^{ij} \quad (5)$$

Coefficients are also put into the geometric part.

Weak Form Processing

```
from ffc.analysis import analyze_forms
from ffc.compiler import compute_ir

parameters = ffc.default_parameters()
parameters['representation'] = 'tensor'
analysis = analyze_forms([a,L], {}, parameters)
ir = compute_ir(analysis, parameters)

a_K = ir[2][0]['AK'][0][0]
a_G = ir[2][0]['AK'][0][1]

K = a_K.A0.astype(numpy.float32)
G = a_G
```

Outline

- 1 Introduction
- 2 Tools
- 3 FEM on the GPU**
- 4 PETSc-GPU
- 5 Conclusions

Form Decomposition

Element integrals are decomposed into analytic and geometric parts:

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

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

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

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

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

Coefficients are also put into the geometric part.

Form Decomposition

Additional fields give rise to multilinear forms.

$$\int_{\mathcal{T}} \phi_i(\mathbf{x}) \cdot (\phi_k(\mathbf{x}) \nabla \phi_j(\mathbf{x})) \, dA \quad (11)$$

$$= \int_{\mathcal{T}} \phi_i^\beta(\mathbf{x}) \left(\phi_k^\alpha(\mathbf{x}) \frac{\partial \phi_j^\beta(\mathbf{x})}{\partial x_\alpha} \right) \, dA \quad (12)$$

$$= \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| \, dA \quad (13)$$

$$= \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} \, dA \quad (14)$$

$$= \mathbf{G}^{\alpha\gamma}(\mathcal{T}) \mathbf{K}_{\alpha\gamma}^{ijk} \quad (15)$$

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

$$\int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) dA \quad (16)$$

$$= \int_{\mathcal{T}} \frac{\partial \phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial \phi_j(\mathbf{x})}{\partial x_\alpha} dA \quad (17)$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |\mathbf{J}| dA \quad (18)$$

$$= |\mathbf{J}| \int_{\mathcal{T}_{\text{ref}}} \phi_k \mathbf{J}_k^{\beta\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \phi_l \mathbf{J}_l^{\gamma\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} dA \quad (19)$$

$$= \mathbf{J}_k^{\beta\alpha} \mathbf{J}_l^{\gamma\alpha} |\mathbf{J}| \int_{\mathcal{T}_{\text{ref}}} \phi_k \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \phi_l \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} dA \quad (20)$$

$$= \mathbf{G}_{kl}^{\beta\gamma}(\mathcal{T}) \mathbf{K}_{\beta\gamma}^{ijkl} \quad (21)$$

A different space could also be used for Jacobians

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

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

Reduction

Choose strategies to minimize reductions

- Only reduction 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;
}

```

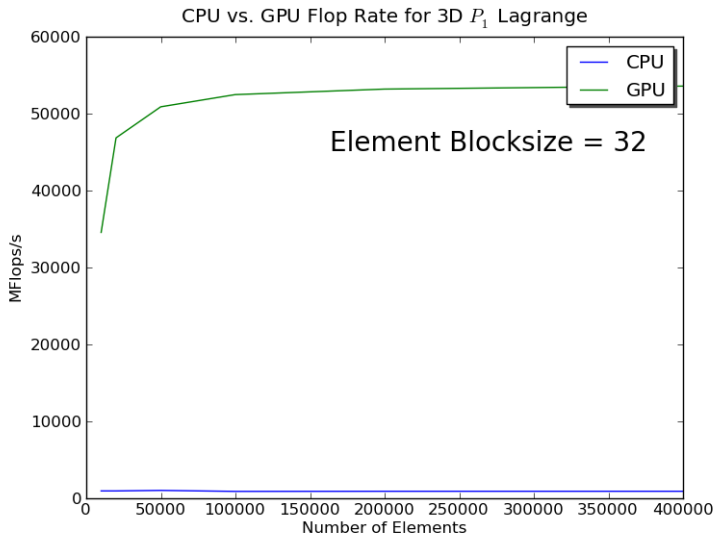
Strategy #2

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]
```

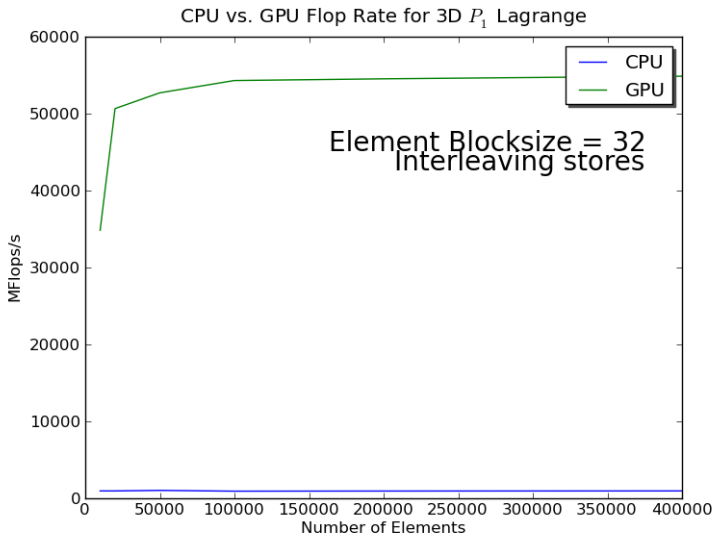
Results

GTX 285



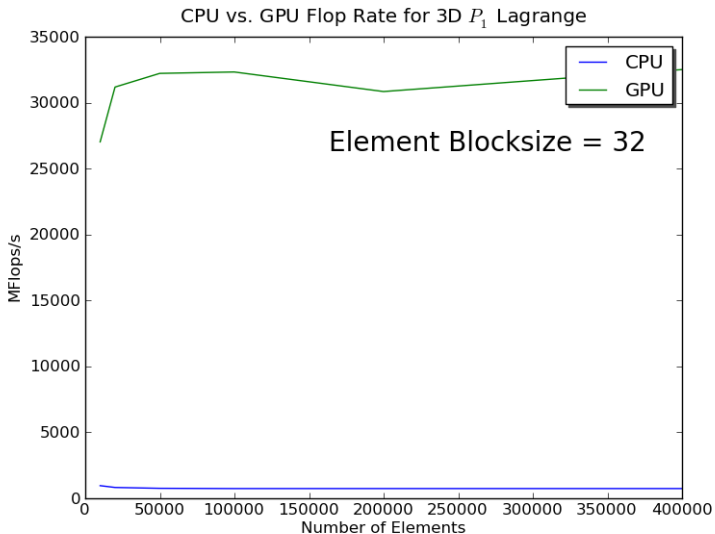
Results

GTX 285



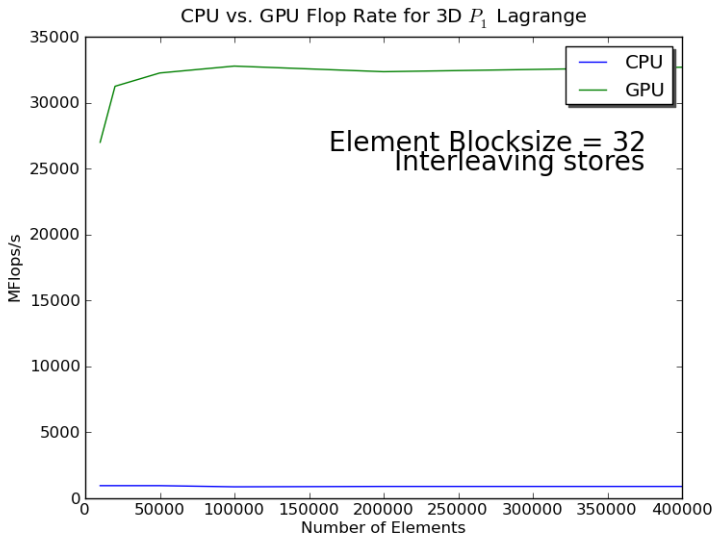
Results

GTX 285, 2 Simultaneous Elements



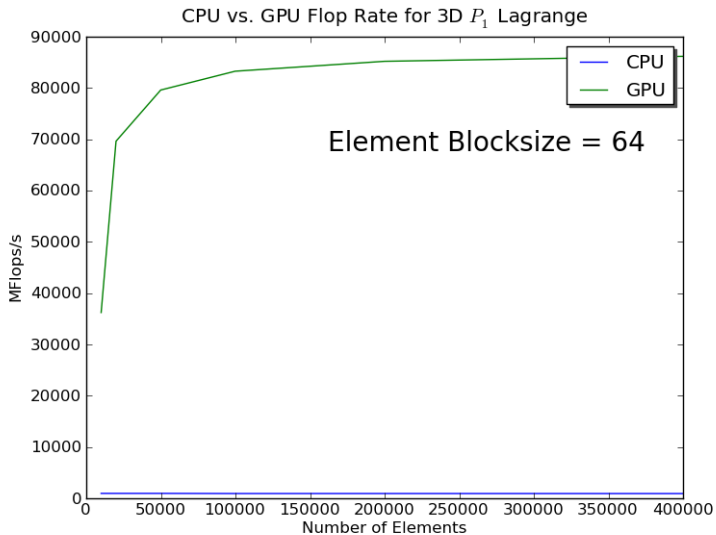
Results

GTX 285, 2 Simultaneous Elements



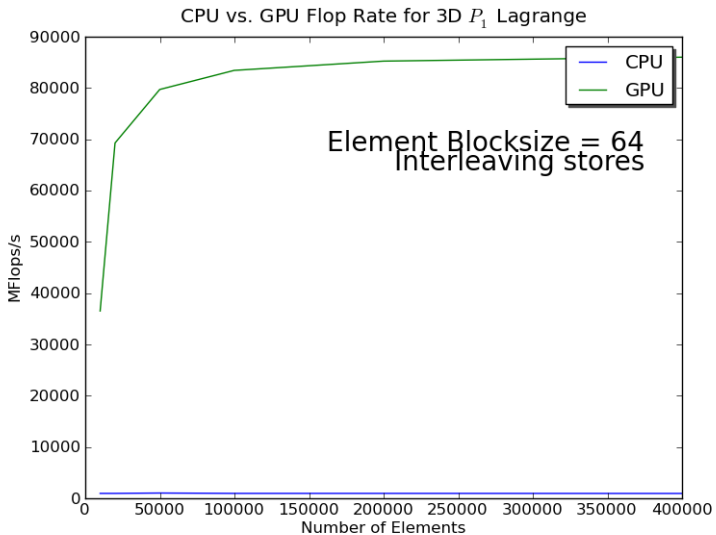
Results

GTX 285



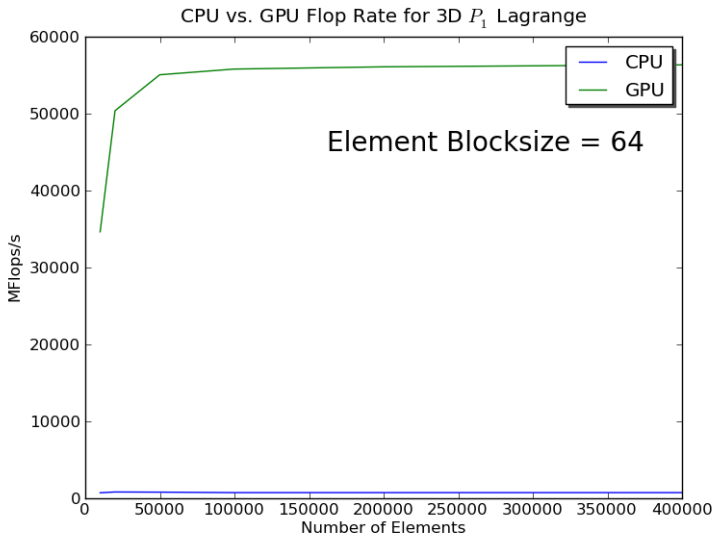
Results

GTX 285



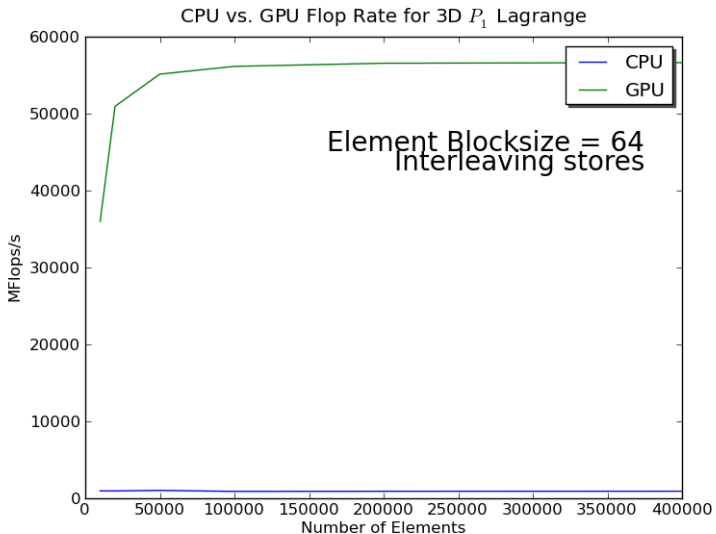
Results

GTX 285, 2 Simultaneous Elements



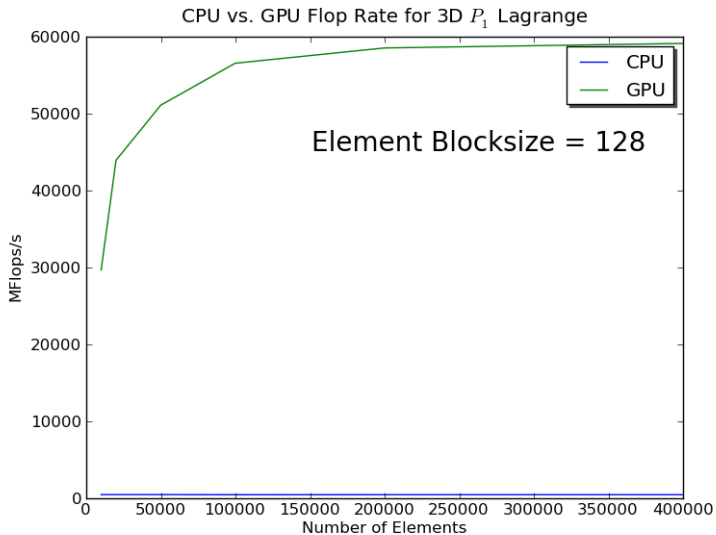
Results

GTX 285, 2 Simultaneous Elements



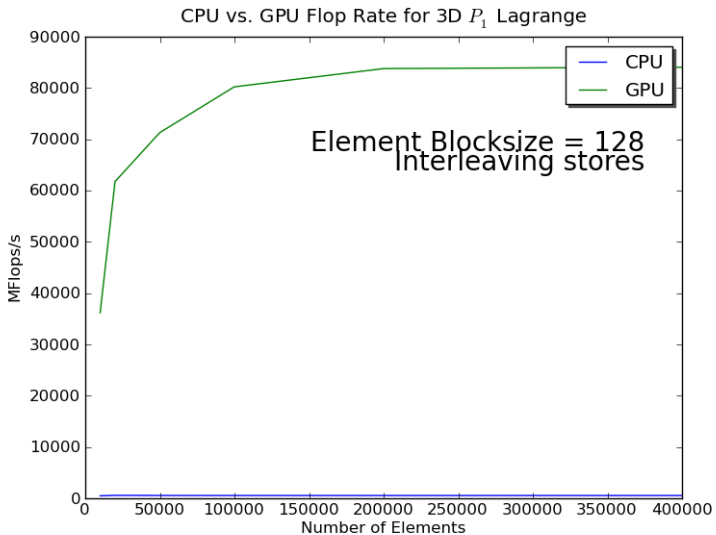
Results

GTX 285



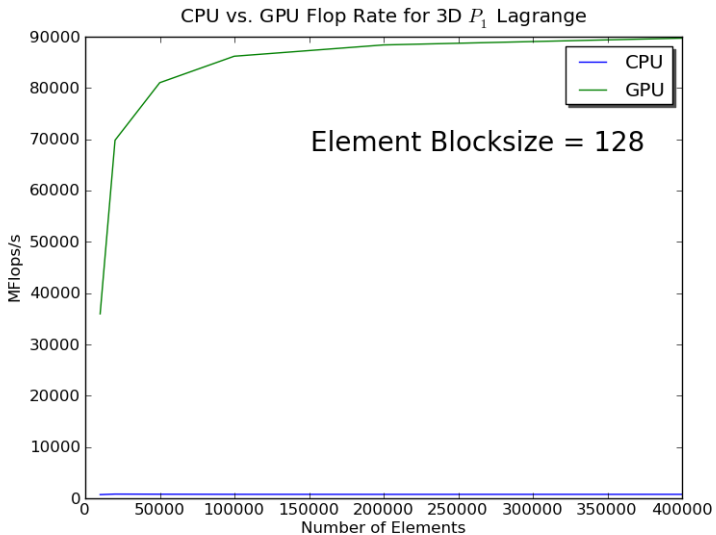
Results

GTX 285



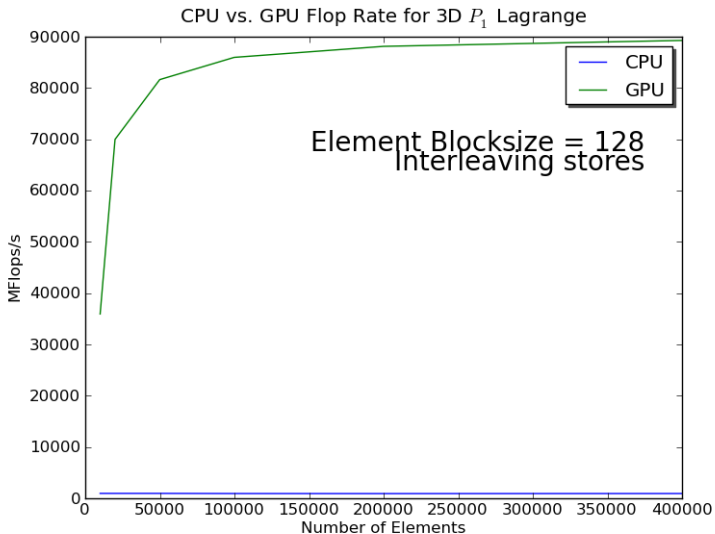
Results

GTX 285, 2 Simultaneous Elements



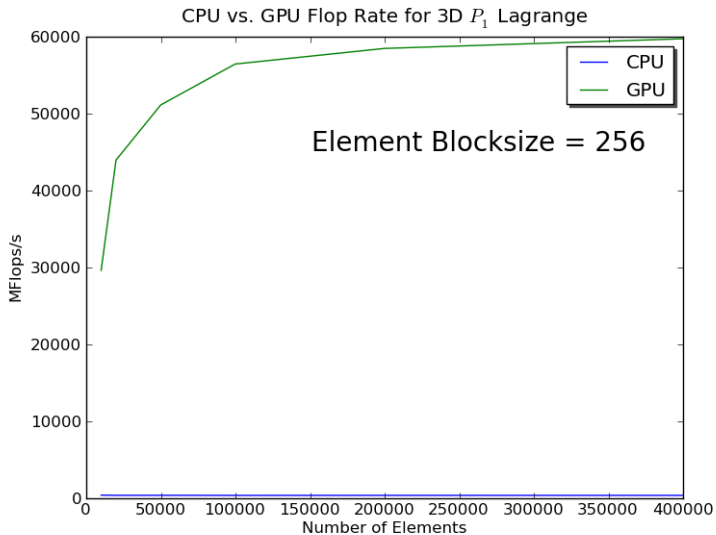
Results

GTX 285, 2 Simultaneous Elements



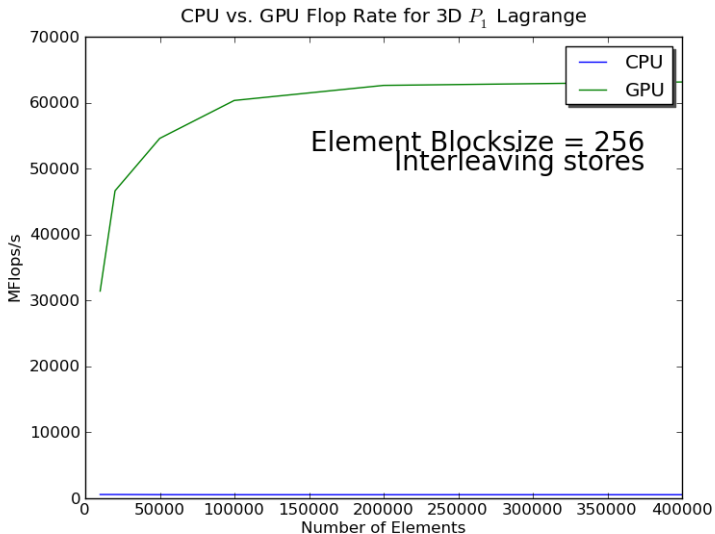
Results

GTX 285, 2 Simultaneous Elements



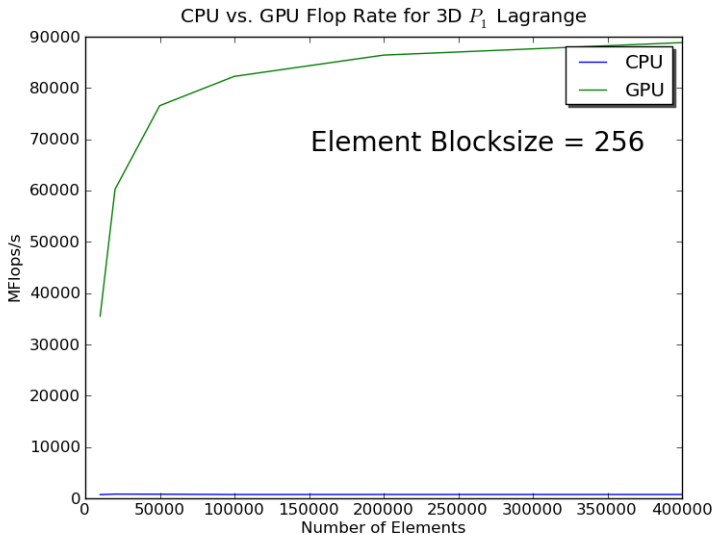
Results

GTX 285, 2 Simultaneous Elements



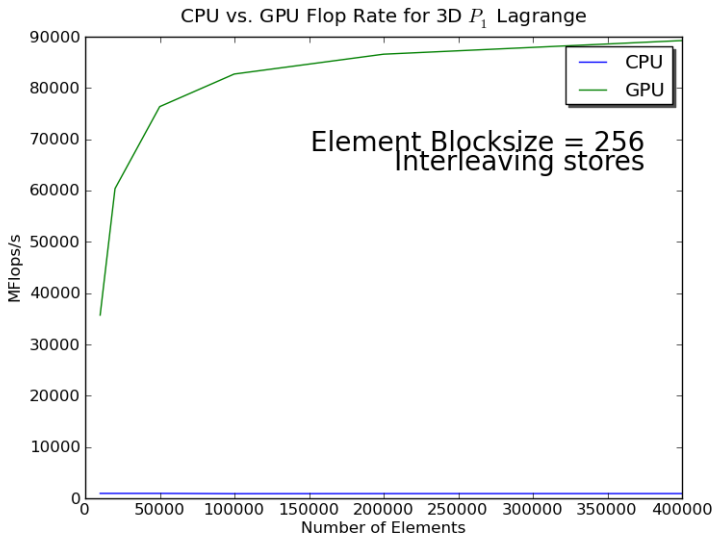
Results

GTX 285, 4 Simultaneous Elements



Results

GTX 285, 4 Simultaneous Elements



Outline

- 1 Introduction
- 2 Tools
- 3 FEM on the GPU
- 4 PETSc-GPU**
- 5 Conclusions

Thrust

Thrust is a CUDA library of parallel algorithms

- Interface similar to C++ Standard Template Library
- Containers (`vector`, `list`, `map`) on both host and device
- Algorithms: `sort`, `reduce`, `scan`
- Freely available, and part of PETSc configure
(`-with-thrust-dir`)

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, and part of PETSc configure (`-with-cusp-dir`)

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.

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 come for Free

- 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
--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

```
ex19 -da_vec_type seqcuda
      -da_mat_type aijcuda -mat_no_inode # Setup types
      -da_grid_x 100 -da_grid_y 100     # Set grid size
      -pc_type none -dmmg_nlevels 1     # Setup solver
      -preload off -cuda_synchronize   # Setup run
      -log_summary
```

Outline

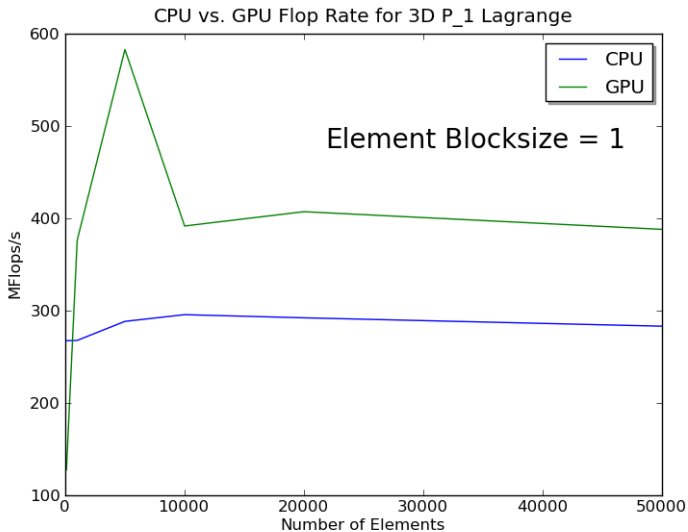
- 1 Introduction
- 2 Tools
- 3 FEM on the GPU
- 4 PETSc-GPU
- 5 Conclusions**

How Will Algorithms Change?

- **Massive concurrency** is necessary
 - Mix of vector and thread paradigms
 - Demands new analysis
- More attention to **memory management**
 - Blocks will only get larger
 - Determinant of performance

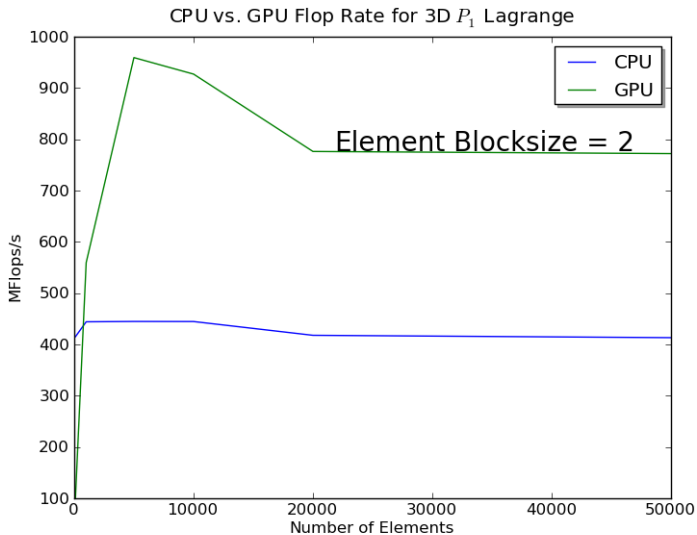
Results

9400M



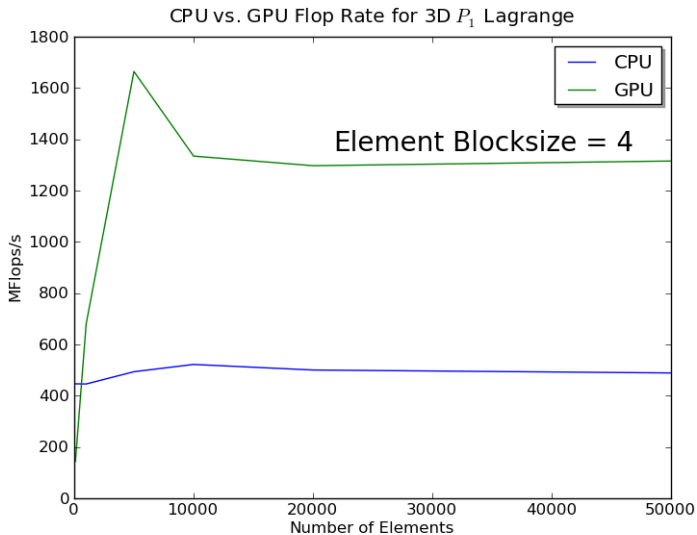
Results

9400M



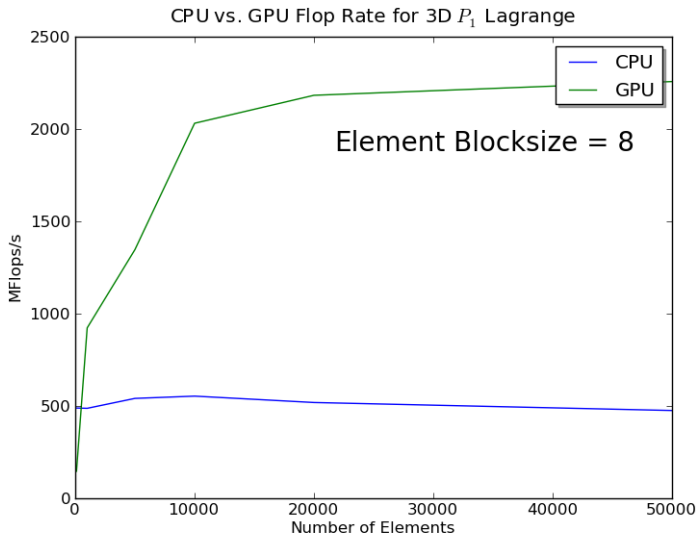
Results

9400M



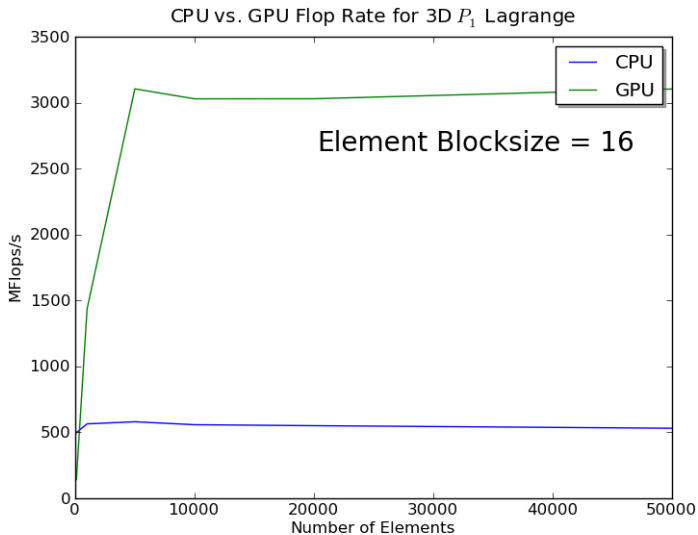
Results

9400M



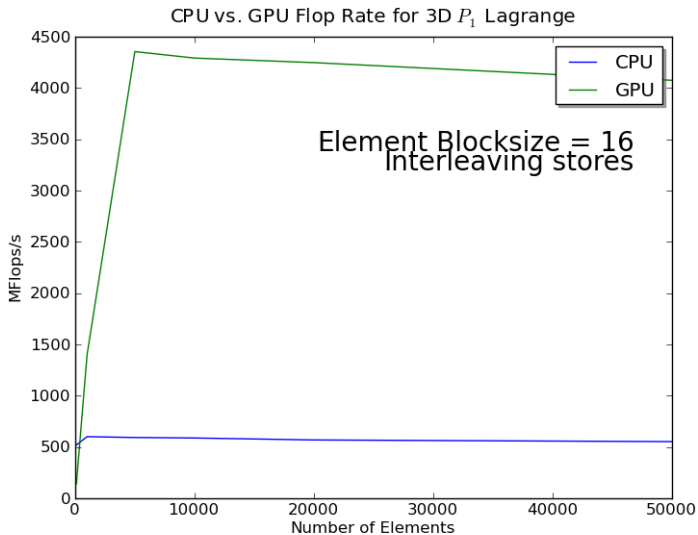
Results

9400M



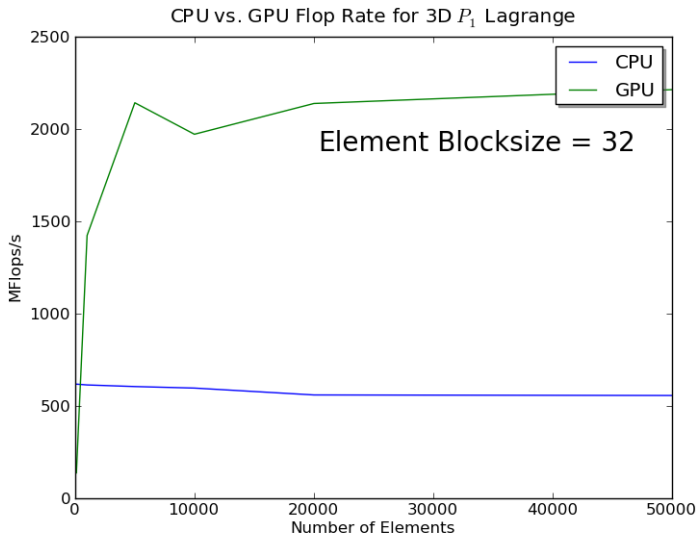
Results

9400M



Results

9400M



Results

9400M

