Easy, Effective, Efficient: GPU Programming in Python with PyOpenCL and PyCUDA

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

PASI: The Challenge of Massive Parallelism
Lecture 4 · January 8, 2011
Outline

1. PyCUDA
2. Automatic GPU Programming
3. GPU-DG: Challenges and Solutions
Lab solutions:

- Lab 1 yesterday:
  Sorry, posted wrong tarball (I think)

- Will post lab solutions after second lab today:
  http://tiker.net/tmp/
  pasi-lab-solution.tar.gz
Outline

1. PyCUDA
2. Automatic GPU Programming
3. GPU-DG: Challenges and Solutions
import pycuda.driver as cuda
import pycuda.autoinit, pycuda.compiler
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)

[This is examples/demo.py in the PyCUDA distribution.]
Whetting your appetite

```python
mod = pycuda.compiler.SourceModule(""
    __global__  void twice( float *a)
    {
        int idx = threadIdx.x + threadIdx.y*4;
        a[idx] *= 2;
    }
""
)

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```
mod = pycuda.compiler.SourceModule('''
__global__ void twice(float *a)
{
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
}
''')

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
print a
Whetting your appetite, Part II

Did somebody say “Abstraction is good”?
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

a_gpu = gpuarray.to_gpu(
    numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
gpuarray: Simple Linear Algebra

pycuda.gpuarray:

- Meant to look and feel just like numpy.
  
  ```python
  gpuarray.to_gpu(numpy_array)
  numpy_array = gpuarray.get()
  ```

- +, -, *, /, fill, sin, exp, rand, basic indexing, norm, inner product, ...

- Mixed types (int32 + float32 = float64)

- print gpuarray for debugging.

- Allows access to raw bits
  
  - Use as kernel arguments, textures, etc.
Sparse Matrix-Vector on the GPU

- New feature in 0.94:
  Sparse matrix-vector multiplication

- Uses “packeted format” by Garland and Bell (also includes parts of their code)

- Integrates with scipy.sparse.

- Conjugate-gradients solver included
  - Deferred convergence checking
## PyOpenCL ↔ PyCUDA: A (rough) dictionary

<table>
<thead>
<tr>
<th>PyOpenCL</th>
<th>PyCUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Context</td>
</tr>
<tr>
<td>CommandQueue</td>
<td>Stream</td>
</tr>
<tr>
<td>Buffer</td>
<td>mem_alloc / DeviceAllocation</td>
</tr>
<tr>
<td>Program</td>
<td>SourceModule</td>
</tr>
<tr>
<td>Kernel</td>
<td>Function</td>
</tr>
<tr>
<td>Event (eg. enqueue_marker)</td>
<td>Event</td>
</tr>
</tbody>
</table>
Scripting: Interpreted, not Compiled

Program creation workflow:

1. Edit
2. Compile
3. Link
4. Run
Scripting: Interpreted, not Compiled

Program creation workflow:

1. Edit
2. Link
3. Run

- No Compile step
Scripting: Interpreted, not Compiled

Program creation workflow:

Edit → Compile → Link → Run
PyCUDA: Workflow

1. **Edit**
2. **SourceModule("...")**
3. **Run**
4. **Upload to GPU**
5. **Run on GPU**

**Normalization:**
- **Cache?**
  - **no**
  - **nvcc**
    - **.cubin**
PyCUDA in the CUDA ecosystem

CUDA has two Programming Interfaces:

- "Runtime" high-level (separate install)
- "Driver" low-level (libcuda.so, comes with GPU driver)
PyCUDA: Vital Information

- http://mathema.tician.de/software/pycuda
- Complete documentation
- X Consortium License
  (no warranty, free for all use)
- Convenient abstractions
  Array, Fast Vector Math, Reductions
- Requires: numpy, Python 2.4+
  (Win/OS X/Linux)
Outline

1. PyCUDA
2. Automatic GPU Programming
3. GPU-DG: Challenges and Solutions
Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers
Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- **Obvious idea**: Let the computer do it.
- **One way**: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Dumb enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware
Empirical GPU loop optimization:

```python
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel(
    [LoopDimension("i", n),
     LoopDimension("j", n),
     LoopDimension("k", n),
    ],
    [(c[i+n*j], a[i+n*k]*b[k+n*j])])

gen_kwargs = {
    "min_threads": 128,
    "min_blocks": 32,
}

→ Ideal case: Finds 160 GF/s kernel without human intervention.
```
Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using “loopy” model
    (i.e. indices decompose into “output” and “reduction”)
  - Enough for DG, LA, FD, …
Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using “loopy” model
    (i.e. indices decompose into “output” and “reduction”)
  - Enough for DG, LA, FD, ...

- Kernel compilation limits trial rate
- Non-Goal: Peak performance
- Good results currently for dense linear algebra and (some) DG subkernels
Outline

1. PyCUDA

2. Automatic GPU Programming

3. GPU-DG: Challenges and Solutions
   - Introduction
   - Challenges
   - Benefits of Metaprogramming
   - GPU-DG: Performance and Generality
   - Viscous Shock Capture
Introduction

Challenges

Benefits

Performance

Shocks

Outline

1. PyCUDA

2. Automatic GPU Programming

3. GPU-DG: Challenges and Solutions
   - Introduction
   - Challenges
   - Benefits of Metaprogramming
   - GPU-DG: Performance and Generality
   - Viscous Shock Capture
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$. 

\[
\frac{\partial u}{\partial t} + \nabla \cdot F(u) = 0
\]

Example

Maxwell's Equations:

$E(x, t), H(x, t)$ on $\Omega$ governed by

\[
\partial_t E - \frac{1}{\varepsilon} \nabla \times H = -j\varepsilon, \quad \partial_t H + \frac{1}{\mu} \nabla \times E = 0,
\]

$\nabla \cdot E = \rho \varepsilon, \quad \nabla \cdot H = 0.$

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Discontinuous Galerkin Method

Let \( \Omega := \bigcup_{i} D_k \subset \mathbb{R}^d \).

Goal

Solve a conservation law on \( \Omega \):

\[
    u_t + \nabla \cdot F(u) = 0
\]
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$.

Goal

Solve a conservation law on $\Omega$:

$$u_t + \nabla \cdot F(u) = 0$$

Example

Maxwell’s Equations: EM field: $E(x, t), H(x, t)$ on $\Omega$ governed by

$$\partial_t E - \frac{1}{\varepsilon} \nabla \times H = -\frac{j}{\varepsilon},$$
$$\nabla \cdot E = \frac{\rho}{\varepsilon},$$
$$\partial_t H + \frac{1}{\mu} \nabla \times E = 0,$$
$$\nabla \cdot H = 0.$$
Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

\[
0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx \\
= \int_{D_k} u_t \varphi - F(u) \cdot \nabla \varphi \, dx + \int_{\partial D_k} (\hat{n} \cdot F)\ast \varphi \, dS_x,
\]

Substitute in basis functions, introduce elementwise stiffness, mass, and surface mass matrices matrices $S$, $M$, $M_A$:

\[
\partial_t u^k = - \sum_{\nu} D^{\partial \nu, k}[F(u^k)] + L^k[\hat{n} \cdot F - (\hat{n} \cdot F)\ast]|_{A \subset \partial D_k}.
\]

For straight-sided simplicial elements: Reduce $D^{\partial \nu}$ and $L$ to reference matrices.
Decomposition of a DG operator into Subtasks

DG’s execution decomposes into two (mostly) separate branches:

- Flux Gather
- Flux Lifting
- \( F(u^k) \)
- Local Differentiation
- \( \partial_t u^k \)

**Green:** Element-local parts of the DG operator.
DG on GPUs: Possible Advantages

DG on GPUs: Why?

- GPUs have deep Memory Hierarchy
  - The majority of DG is local.
- Compute Bandwidth $\gg$ Memory Bandwidth
  - DG is arithmetically intense.
- GPUs favor dense data.
  - Local parts of the DG operator are dense.
DG on the GPU: What are we trying to achieve?

Objectives:

- **Main: Speed**
  Reduce need for compute-bound clusters

- **Secondary: Generality**
  Be applicable to many problems

- **Tertiary: Ease-of-Use**
  Hide complexity of GPU hardware

Setting (for now):

- Specialize to straight-sided simplices
- Optimize for (but don’t specialize to) tetrahedra (ie. 3D)
- Optimize for “medium” order (3...5)
Outline

1 PyCUDA

2 Automatic GPU Programming

3 GPU-DG: Challenges and Solutions
   - Introduction
   - Challenges
   - Benefits of Metaprogramming
   - GPU-DG: Performance and Generality
   - Viscous Shock Capture
Element-Local Operations: Differentiation

Local Templated Derivative Matrices

Field Data

Geometric Factors

$N_p$ $N_p$ $K$ $N_p$
Element-Local Operations: Lifting

Local Templated Lifting Matrix

Facial Field Data

(Inverse) Jacobians

$N_p$  

$N_f N_{fp}$  

$K$  

$N_f N_{fp}$
Element-Local Operations: Lifting

Local Templated Lifting Matrix

Facial Field Data

On-Chip Storage

(Inverse) Jacobians
Element-Local Operations: Lifting

Local Templated Lifting Matrix

Facial Field Data

$N_p$

$N_f N_{fp}$

$K$

$(Inverse) Jacobians$

On-Chip Storage

(Inverse) Jacobians

$N_f N_{fp}$
Element-Local Operations: Lifting

Local Templated Lifting Matrix

Facial Field Data

$(Inverse)$ Jacobians

On-Chip Storage
Best use for on-chip memory?

Basic Problem

On-chip storage is scarce . . .

. . . and will be for the foreseeable future.

Possible uses:

- Matrix/Matrices
- Part of a matrix
- Field Data
- Both

How to decide? Does it matter?
Work Partition for Element-Local Operators

Natural Work Decomposition:
One Element per Block
Natural Work Decomposition:
One Element per Block

- Straightforward to implement
- No granularity penalty
- Cannot fill wide SIMD: unused compute power for small to medium elements
- Data alignment: Padding wastes memory
- Cannot amortize cost of preparation steps (e.g. fetching)
Loop Slicing for element-local parts of GPU DG

**Per Block:** $K_L$ element-local mat.mult. + matrix load

**Question:** How should one assign work to threads?
Loop Slicing for element-local parts of GPU DG

**Per Block:** \( K_L \) element-local mat.mult. + matrix load

**Question:** How should one assign work to threads?

- \( w_s \): in sequence
  - Thread
  - \( t \)
  - (amortize preparation)

- \( w_i \): “inline-parallel”
  - Thread
  - \( t \)
  - (exploit register space)

- \( w_p \): in parallel
  - Thread
  - \( t \)
Best Work Partition?

Basic Problem

Additional tier in parallelism offers additional choices... ...but very little in the way of guidance.

Possible work partitions:

- One or multiple elements per block?
- One or multiple DOFs per thread?
  - In parallel?
  - In sequence?
  - In-line?

How to decide? Does it matter?
Granularity Tradeoff:

- **Large Blocks:**
  - More Data Reuse
  - Less Parallelism
  - Less Latency Hiding

- **Block Size limited by two factors:**
  - Output buffer size
  - Face metadata size

- **Optimal Block Size:**
  not obvious
Work Partition for Surface Flux Evaluation

Granularity Tradeoff:

- Large Blocks:
  - More Data Reuse
  - Less Parallelism
  - Less Latency Hiding

- Block Size limited by two factors:
  - Output buffer size
  - Face metadata size

- Optimal Block Size:
  not obvious 😕

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Granularity Tradeoff:

- Large Blocks:
  - More Data Reuse
  - Less Parallelism
  - Less Latency Hiding

- Block Size limited by two factors:
  - Output buffer size
  - Face metadata size

- Optimal Block Size: not obvious
More than one Granularity

Different block sizes introduced so far:

- Differentiation
- Lifting
- Surface Fluxes

And introduce another, smaller block size to satisfy SIMD width and alignment constraints. ("Microblock")

And demand other block sizes be a multiple of this new size

How big? Not obvious.
More than one Granularity

Different block sizes introduced so far:
- Differentiation
- Lifting
- Surface Fluxes

Idea
Introduce another, smaller block size to satisfy SIMD width and alignment constraints. ("Microblock")
- And demand other block sizes be a multiple of this new size
More than one Granularity

Different block sizes introduced so far:

- Differentiation
- Lifting
- Surface Fluxes

**Idea**

Introduce another, smaller block size to satisfy SIMD width and alignment constraints. ("Microblock")

- And demand other block sizes be a multiple of this new size

How big? Not obvious. 🤔
DG on GPUs: Implementation Choices

- Many difficult questions
- Insufficient heuristics
- Answers are hardware-specific and have no lasting value
DG on GPUs: Implementation Choices

- Many difficult questions
- Insufficient heuristics
- Answers are hardware-specific and have no lasting value

**Proposed Solution:** Tune automatically for hardware at computation time, cache tuning results.

- Decrease reliance on knowledge of hardware internals
- Shift emphasis from tuning *results* to tuning *ideas*
Outline

1. PyCUDA
2. Automatic GPU Programming
3. GPU-DG: Challenges and Solutions
   - Introduction
   - Challenges
   - Benefits of Metaprogramming
   - GPU-DG: Performance and Generality
   - Viscous Shock Capture
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms

- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms

- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity

- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity
- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes
- Loop Unrolling
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms (*)
- Automated Tuning:
  - Memory layout
  - Loop slicing (*)
  - Gather granularity
- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes
- Loop Unrolling
Local differentiation, matrix-in-shared, order 4, with microblocking point size denotes $w_i \in \{1, \ldots, 4\}$
Metaprogramming DG: Flux Terms

\[ 0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx - \int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, dS_x \]

Flux term

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Metaprogramming DG: Flux Terms

\[ 0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx - \int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, dS_x \]

Flux terms:
- vary by problem
- expression specified by user
- evaluated pointwise
Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell’s Equations

\[ \hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([H] - \alpha \hat{n} \times [E])] \]
Example: Fluxes for Maxwell’s Equations

\[ \hat{n} \cdot (F - F^*)_E := \frac{1}{2} \left[ \hat{n} \times (\|H\| - \alpha \hat{n} \times \|E\|) \right] \]

User writes: Vectorial statement in math. notation

```python
flux = 1/2*cross(normal, h.int-h.ext
          - alpha*cross(normal, e.int-e.ext))
```
Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell’s Equations

\[ \hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([H] - \alpha \hat{n} \times [E])] \]

**We generate:** Scalar evaluator in C (6×)

```c
a_flux += (( val_a_field5  -  val_b_field5 )* fpair  ->  normal[2]
      - ( val_a_field4  -  val_b_field4 )* fpair  ->  normal[0])
+ val_a_field0  -  val_b_field0 )* fpair  ->  normal[0]
- ((( val_a_field4  -  val_b_field4 ) * fpair  ->  normal[1]
      - ( val_a_field1  -  val_b_field1 )* fpair  ->  normal[2])
+ val_a_field3  -  val_b_field3 ) * fpair  ->  normal[1]
)*value_type (0.5);
```
Hedge DG Solver

- High-Level Operator Description
  - Maxwell’s
  - Euler
  - Poisson
  - Compressible Navier-Stokes, ...

- One Code runs...
  - ...on CPU, CUDA
  - ...on \{CPU,CUDA\}+MPI
  - ...in 1D, 2D, 3D
  - ...at any order

- Uses CPU, GPU code generation
- Open Source (GPL3)
- Written in Python,
Outline

1. PyCUDA

2. Automatic GPU Programming

3. GPU-DG: Challenges and Solutions
   - Introduction
   - Challenges
   - Benefits of Metaprogramming
   - GPU-DG: Performance and Generality
   - Viscous Shock Capture
Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400

Andreas Klöckner  
GPU-Python with PyOpenCL and PyCUDA
Memory Bandwidth on a GTX 280

![Graph showing memory bandwidth vs polynomial order for different operations on a GTX 280 GPU.](image_url)
Multiple GPUs via MPI: 16 GPUs vs. 64 CPUs

Flop Rates: 16 GPUs vs 64 CPU cores

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
GPU-DG in Double Precision

![Graph showing GPU-DG: Double vs. Single Precision](image)

- **Single**
- **Double**

- **Ratio**

Polynomial Order $N$ vs. GFlops/s for Single and Double Precision GPU-DG.
Outline

1. PyCUDA

2. Automatic GPU Programming

3. GPU-DG: Challenges and Solutions
   - Introduction
   - Challenges
   - Benefits of Metaprogramming
   - GPU-DG: Performance and Generality
   - Viscous Shock Capture
Nonlinear conservation laws $\rightarrow$ shocks?

\[ u(x) \]

\[ \frac{\partial}{\partial t} u(x) + \frac{\partial}{\partial x} u(x) = 0 \]

1D advection with viscosity:

\[ \frac{\partial}{\partial t} u(x) + \mathbf{v} \cdot \nabla u(x) = \nabla \cdot (\nu \nabla u(x)). \]

Important: Conservation form.

Upwind fluxes for advection, IPDG for second-order

Detector $\rightarrow \nu$?

GPU-suitability? Data locality?

Properties? [Build on work by Persson/Peraire '06]

Time integration

Implicit/explicit? Adaptivity? RKC for bigger $\Delta t$ with viscosity?

Accuracy?

Near shocks? Away from them?

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Nonlinear conservation laws $\rightarrow$ shocks?

\[ \frac{\partial}{\partial t} u(x) + \frac{\partial}{\partial x} u(x) = 0 \]
\[ \frac{\partial}{\partial t} u(x) + v \cdot \nabla x u(x) = \nabla x \cdot \left( \nu \nabla x u(x) \right) \]

Important: Conservation form.

Detector $\rightarrow$ $\nu$? GPU-suitability? Data locality? Properties? [Build on work by Persson/Peraire '06]


Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Nonlinear conservation laws $\rightarrow$ shocks?

1D advection:

$$\partial_t u + \partial_x u = 0$$

1D advection with viscosity:

$$\partial_t u + v \cdot \nabla_x u = \nabla_x \cdot (\nu \nabla_x u).$$

Important: Conservation form.

Upwind fluxes for advection, IPDG for second-order
Nonlinear conservation laws → shocks?

1D advection:

\[ \partial_t u + \partial_x u = 0 \]

1D advection with viscosity:

\[ \partial_t u + v \cdot \nabla_x u = \nabla_x \cdot (\nu \nabla_x u) \]

Important: Conservation form.

Upwind fluxes for advection, IPDG for second-order

Detector → \( \nu \)?

GPU-suitability? Data locality? Properties? [Build on work by Persson/Peraire ‘06]

Time integration

Implicit/explicit? Adaptivity? RKC for bigger \( \Delta t \) with viscosity?

Accuracy?

Near shocks? Away from them?
Results: Euler’s Equations of Gas Dynamics

Euler’s equations with viscosity:

$$\begin{align*}
\frac{\partial t}{\partial t} \rho + \nabla_x \cdot (\rho \mathbf{u}) &= \nabla_x \cdot (\nu \nabla_x \rho), \\
\frac{\partial t}{\partial t} (\rho \mathbf{u}) + \nabla_x \cdot (\mathbf{u} \otimes (\rho \mathbf{u})) + \nabla_x p &= \nabla_x \cdot (\nu \nabla_x (\rho \mathbf{u})), \\
\frac{\partial t}{\partial t} E + \nabla_x \cdot (\mathbf{u} (E + p)) &= \nabla_x \cdot (\nu \nabla_x E).
\end{align*}$$

Again: Single $\nu$, sensed on $\rho$. $\rightarrow$ Undue pollution of the other field?

[Persson/Peraire ‘06] suggest Navier-Stokes-like viscosity. No good: can’t control jumps in $\rho$.

Rusanov fluxes for Euler, IPDG for viscosity.
Results: Euler’s Equations of Gas Dynamics

Euler’s equations with viscosity:

\[ \partial_t \rho + \nabla_x \cdot (\rho \mathbf{u}) = \nabla_x \cdot (\nu \nabla_x \rho), \]

\[ \partial_t (\rho \mathbf{u}) + \nabla_x \cdot (\rho \mathbf{u} \otimes \rho \mathbf{u}) + \nabla_x p = \nabla_x \cdot (\nu \nabla_x (\rho \mathbf{u})), \]

\[ \partial_t E + \nabla_x \cdot (\mathbf{u} (E + p)) = \nabla_x \cdot (\nu \nabla_x E). \]

Again, single \( \nu \), sensed on \( \rho \).

[Persson/Peraire '06] suggest Navier-Stokes-like viscosity. No good: can’t control jumps in \( \rho \).

Rusanov fluxes for Euler, IPDG for viscosity.

\[ 0.0 \quad 0.2 \quad 0.4 \quad 0.6 \quad 0.8 \quad 1.0 \]

\[ x \]

\[ 0.0 \quad 0.2 \quad 0.4 \quad 0.6 \quad 0.8 \quad 1.0 \]

\[ \rho, p \]

Sod's Problem with \( N = 5 \) and \( K = 80 \)

\[ \rho \quad p \]

\[ \rho \text{ (exact, } L^2 \text{ proj.)} \quad p \text{ (exact, } L^2 \text{ proj.)} \]
Results: Euler’s Equations of Gas Dynamics

\[ \begin{align*}
\partial_t (\rho u) + \nabla \cdot (\rho u \otimes u) + \nabla p &= \nabla \cdot (\nu \nabla \rho), \\
\partial_t E + \nabla \cdot (u(E+p)) &= \nabla \cdot (\nu \nabla E). 
\end{align*} \]

Again: Single \( \nu \), sensed on \( \rho \). \[ \text{[Persson/Peraire '06]} \] suggest Navier-Stokes-like viscosity. No good: can’t control jumps in \( \rho \).

Rusanov fluxes for Euler, IPDG for viscosity.
Results: Euler’s Equations of Gas Dynamics

Euler’s equations with viscosity:

\[ \begin{align*}
\partial_t (\rho u) & + \nabla \cdot (\rho u u) + \nabla \cdot (\nu \nabla \rho) = 0, \\
\partial_t (\rho u) & + \nabla \cdot (u \otimes (\rho u)) + \nabla p = \nabla \cdot (\nu \nabla (\rho u)), \\
\partial_t E & + \nabla \cdot (u (E + p)) = \nabla \cdot (\nu \nabla E).
\end{align*} \]

Again: Single \( \nu \), sensed on \( \rho \).

\[ \text{Undue pollution of the other field?} \]

[Persson/Periare '06] suggest Navier-Stokes-like viscosity. No good: can't control jumps in \( \rho \).

Rusanov fluxes for Euler, IPDG for viscosity.

\[ \text{Sod's Problem with } N = 5 \text{ and } K = 80 \]

\[ \text{Shock-Wave Interaction Problem with } N = 5 \text{ and } K = 80 \]
Results: Euler’s Equations of Gas Dynamics

Euler’s equations with viscosity:

\[ \partial_t (\rho u) + \nabla \cdot (\rho u u) + \nabla \cdot (\nu \nabla \rho) = 0, \]

\[ \partial_t (\rho u) + \nabla \cdot (u \otimes (\rho u)) + \nabla p = \nabla \cdot (\nu \nabla (\rho u)), \]

\[ \partial_t E + \nabla \cdot (u (E + p)) = \nabla \cdot (\nu \nabla E). \]

Again: Single \( \nu \), sensed on \( \rho \).
→ Undue pollution of the other field?

[Persson/Peraire '06] suggest Navier-Stokes-like viscosity. No good: can’t control jumps in \( \rho \).

Rusanov fluxes for Euler, IPDG for viscosity.

Shock-Wave Interaction Problem
Results: Euler’s Equations of Gas Dynamics

Euler’s equations with viscosity:

\[
\begin{align*}
\partial_t (\rho u) + \nabla \cdot (\rho u u) + \nabla \cdot p & = \nabla \cdot (\nu \nabla \rho), \\
\partial_t E + \nabla \cdot (u (E + p)) & = \nabla \cdot (\nu \nabla E).
\end{align*}
\]

Again: Single \( \nu \), sensed on \( \rho \).
→ Undue pollution of the other field?

[Persson/Peraire '06] suggest Navier-Stokes-like viscosity. No good: can’t control jumps in \( \rho \).
Rusanov fluxes for Euler, IPDG for viscosity.

Shock-Wave Interaction Problem

\( Sod's \) Problem with \( N = 5 \) and \( K = 80 \)

\( \rho, p \) (exact, \( L_2 \) proj.)

\( \rho \) and \( p \) (exact, \( L_2 \) proj.)
Results: Euler’s Equations of Gas Dynamics

Euler’s equations with viscosity:

\[
\partial_t (\rho u) + \nabla \cdot (\rho u u) + \nabla \cdot (u \otimes \rho u) + \nabla p = \nabla \cdot (\nu \nabla \rho u)
\]

\[
\partial_t E + \nabla \cdot (u(E + p)) = \nabla \cdot (\nu \nabla E)
\]

Again: Single \( \nu \), sensed on \( \rho \).

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Results: Euler’s Equations of Gas Dynamics

Euler’s equations with viscosity:

\[
\begin{align*}
\partial_t (\rho u) + \nabla \cdot (\rho u u) + \nabla \cdot (\nu \nabla \rho) &= 0, \\
\partial_t \rho u + \nabla \cdot (u \otimes u) + \nabla p &= \nabla \cdot (\nu \nabla u), \\
\partial_t E + \nabla \cdot (u (E + p)) &= \nabla \cdot (\nu \nabla E).
\end{align*}
\]

Again: Single \( \nu \), sensed on \( \rho \).
→ Undue pollution of the other field?

[Persson/Peraire '06] suggest Navier-Stokes-like viscosity. No good: can’t control jumps in \( \rho \).
Rusanov fluxes for Euler, IPDG for viscosity.
GPU DG Showcase

Eletromagnetism
GPU DG Showcase

Eletromagnetism

Poisson
GPU DG Showcase

Eletromagnetics

CFD
GPU DG Showcase

Eletromagnetism

CFD
Where to from here?

PyCUDA, PyOpenCL, hedge

→ http://www.cims.nyu.edu/~kloeckner/

GPU-DG Article


GPU RTCG

Conclusions

- GPUs and scripting work surprisingly well together
  - Enable Run-Time Code Generation
- GPU-DG is significantly faster than CPU-DG
  - Method well-suited a priori
  - Numerous tricks enable good performance
- Further work in GPU-DG:
  - Curvilinear Elements (T. Warburton)
  - Local Time Stepping
  - Shock Capturing for Nonlinear Equations
Questions?

Thank you for your attention!

http://www.cims.nyu.edu/~kloeckner/
Image Credits

- Exclamation mark: sxc.hu/cobrasoft
- Adding Machine: flickr.com/thomashawk (CC)
- Floppy disk: flickr.com/ethanhein (CC)
- Carrot: OpenClipart.org
- Dart board: sxc.hu/195617
- Question Mark: sxc.hu/svilen001
- Question Mark: sxc.hu/svilen001
- ?/! Marks: sxc.hu/svilen001