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

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

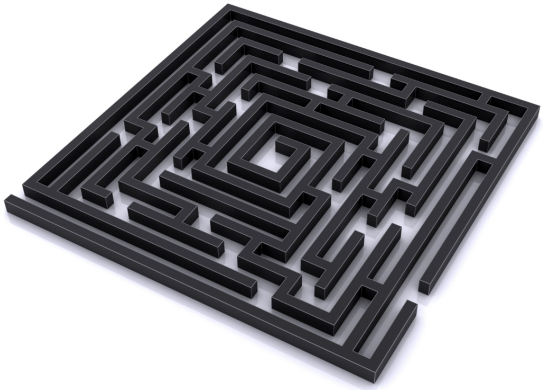
PASI: The Challenge of Massive Parallelism
Lecture 4 · January 8, 2011

Outline

Lab 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

Whetting your appetite

```
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

```
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
```

Whetting your appetite

```
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
```

Compute kernel

Whetting your appetite, Part II

Did somebody say "Abstraction is good"?

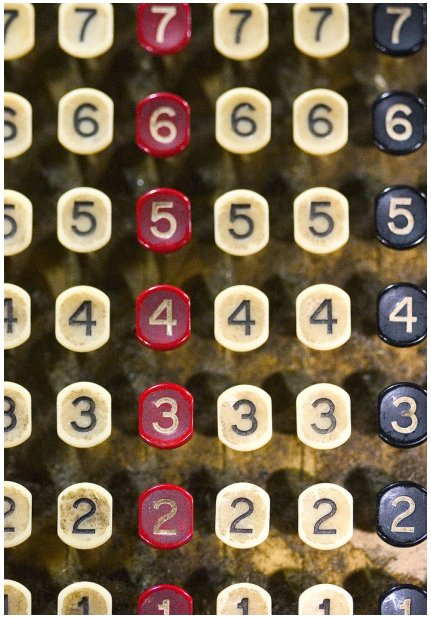
Whetting your appetite, Part II

```
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

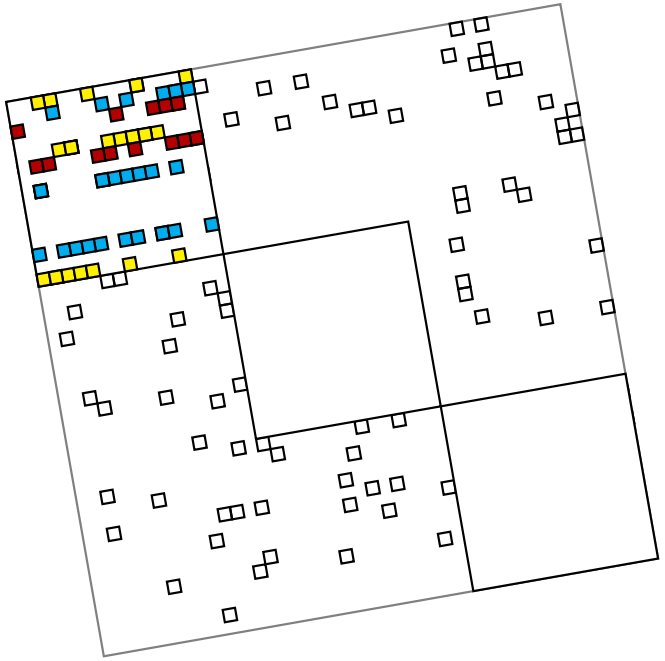
pycuda.gpuarray:

- Meant to look and feel just like numpy.
  - 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

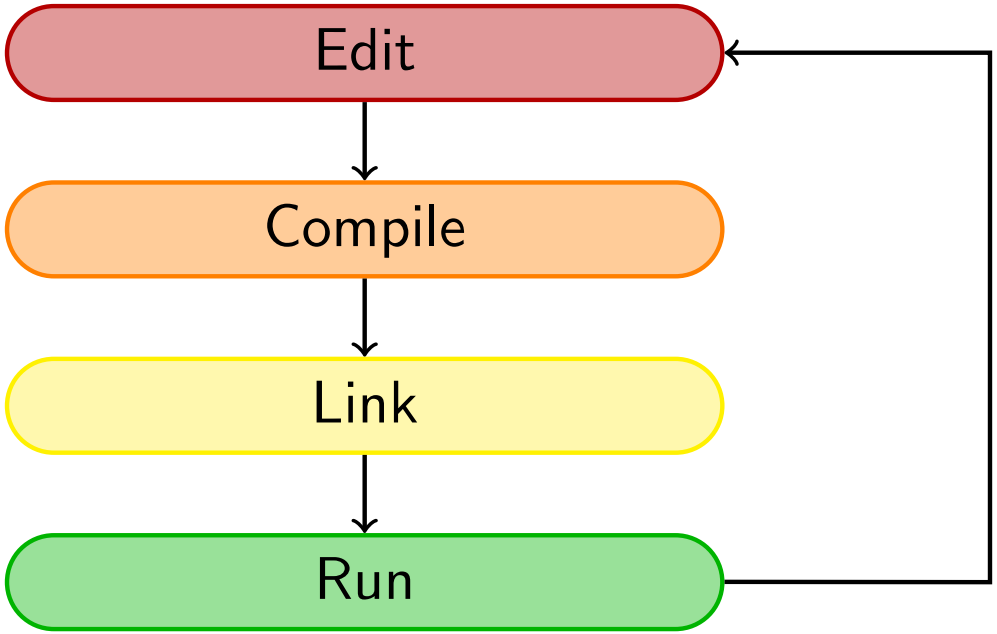


PyOpenCL ↔ PyCUDA: A (rough) dictionary

PyOpenCL	PyCUDA
Context	Context
CommandQueue	Stream
Buffer	mem_alloc / DeviceAllocation
Program	SourceModule
Kernel	Function
Event (eg. enqueue_marker)	Event

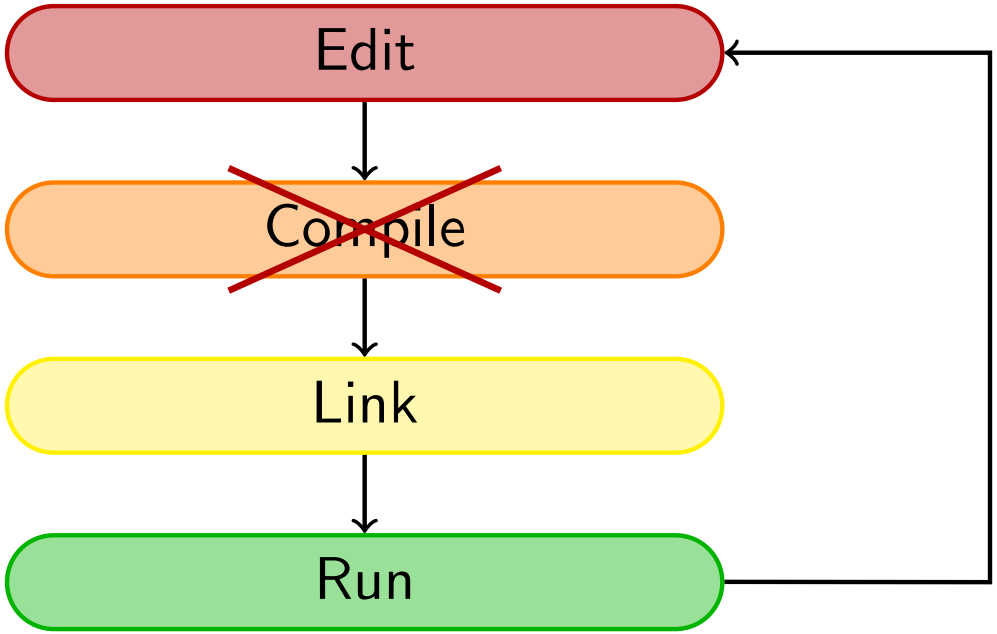
Scripting: Interpreted, not Compiled

Program creation workflow:



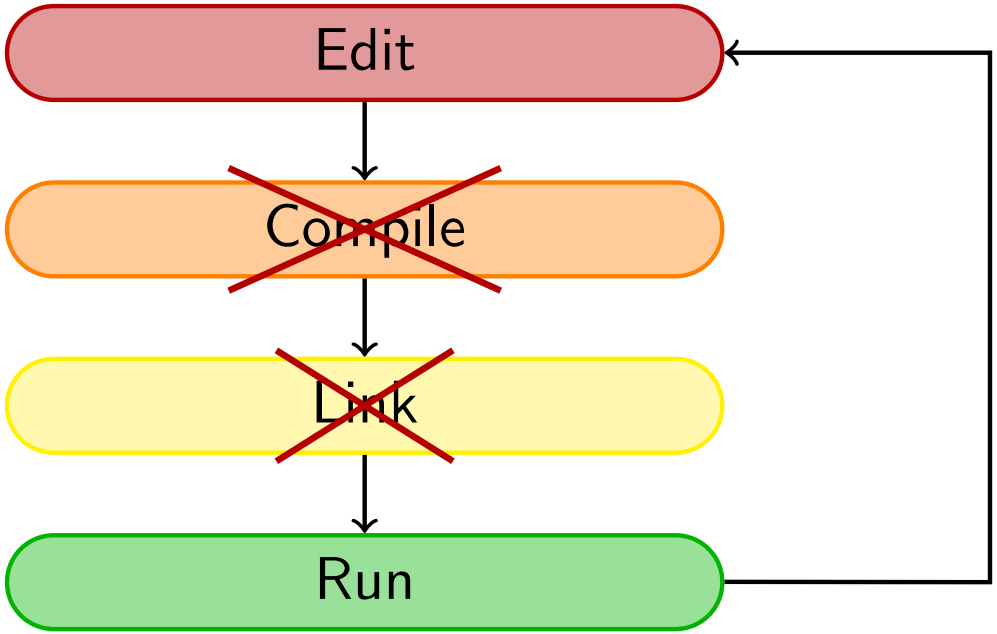
Scripting: Interpreted, not Compiled

Program creation workflow:

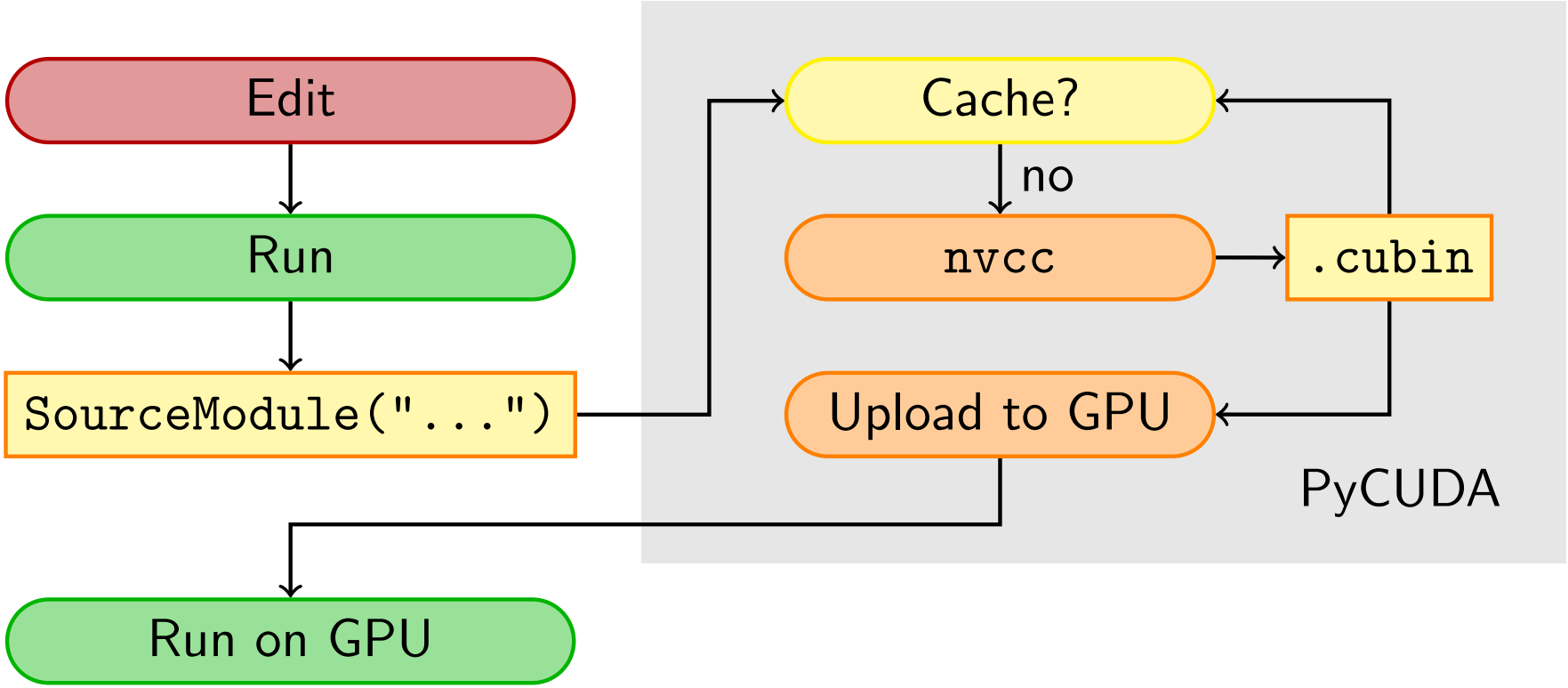


Scripting: Interpreted, not Compiled

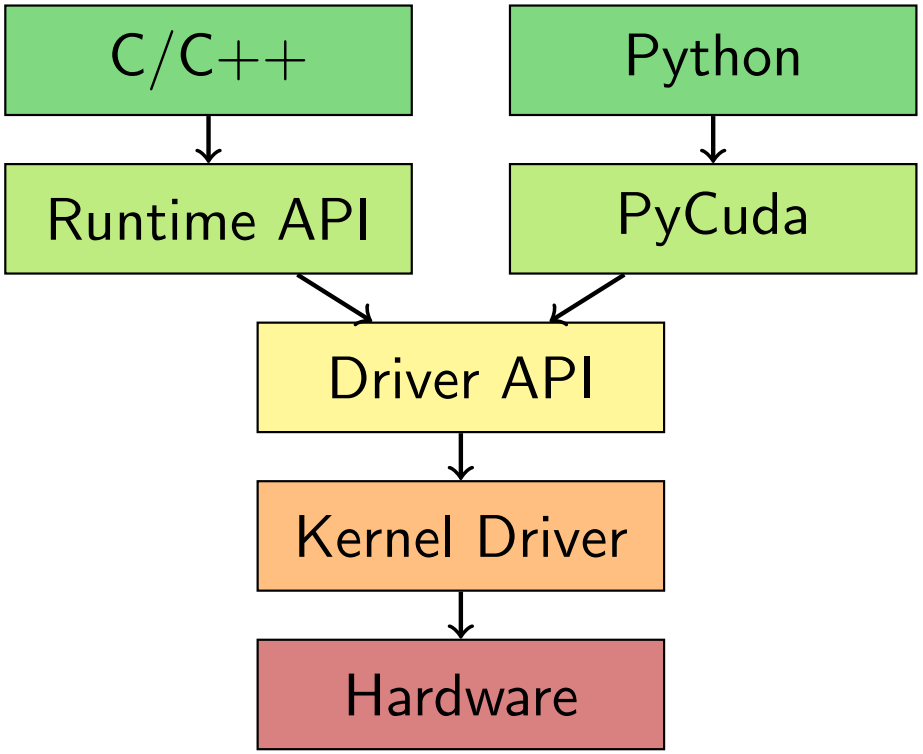
Program creation workflow:



PyCUDA: Workflow



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

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

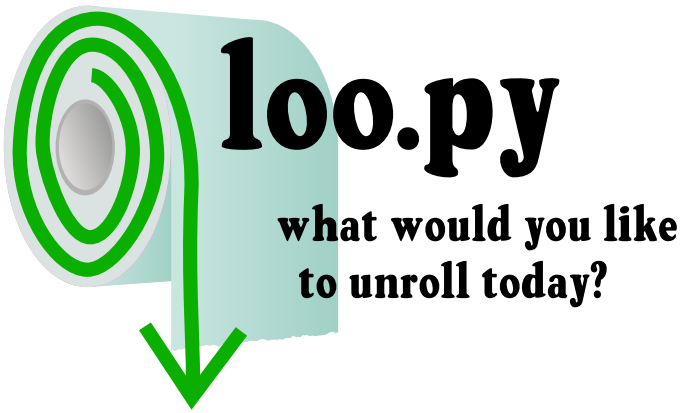
GPU programming can be time-consuming, unintuitive and error-prone.

- 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

Loo.py Example

Empirical GPU loop optimization:

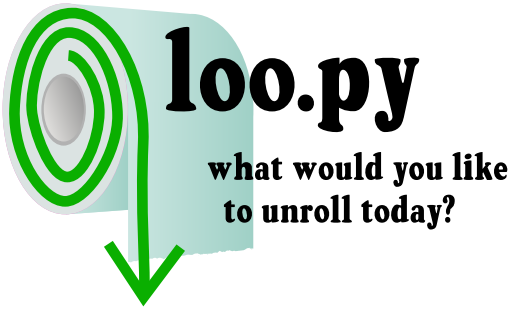
```
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel([
    LoopDimension("i", n),
    LoopDimension("j", n),
    LoopDimension("k", n),
    ], [
    (c[i+n*j], a[i+n*k]*b[k+n*j])
    ])

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



→ Ideal case: Finds 160 GF/s kernel without human intervention.

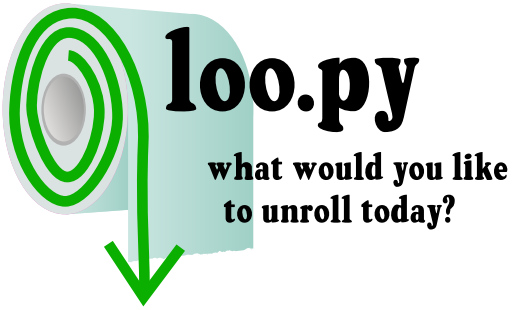
Loo.py Status

- 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

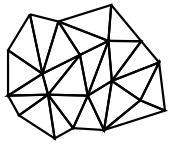
- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using "loopy" model
    (i.e. indices decompose into "output" and "reduction")
  - Enough for DG, LA, FD, ...
- Kernel compilation limits trial rate
- Non-Goal: Peak performance
- Good results currently for dense linear algebra and (some) DG subkernels



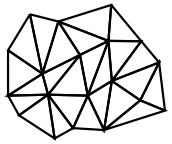
Outline

Outline

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i \mathsf{D}_k \subset \mathbb{R}^d$. 

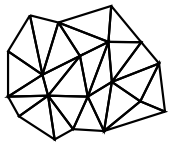
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i \mathrm{D}_k \subset \mathbb{R}^d$. 

Goal

Solve a *conservation law* on Ω: $$u_t + \nabla \cdot F(u) = 0$$

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i \mathrm{D}_k \subset \mathbb{R}^d$. 

Goal

Solve a *conservation law* on Ω: $u_t + \nabla \cdot F(u) = 0$

Example

Maxwell's Equations: EM field: $E(x,t)$, $H(x,t)$ on Ω governed by

$$\partial_t E - \frac{1}{\varepsilon}\nabla \times H = -\frac{j}{\varepsilon}, \qquad \partial_t H + \frac{1}{\mu}\nabla \times E = 0,$$

$$\nabla \cdot E = \frac{\rho}{\varepsilon}, \qquad \nabla \cdot H = 0.$$

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

$$
\begin{aligned}
0 &= \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, \mathrm{d}x \\
&= \int_{\mathrm{D}_k} u_t \varphi - F(u) \cdot \nabla \varphi \, \mathrm{d}x + \int_{\partial \mathrm{D}_k} (\hat{n} \cdot F)^* \varphi \, \mathrm{d}S_x,
\end{aligned}
$$

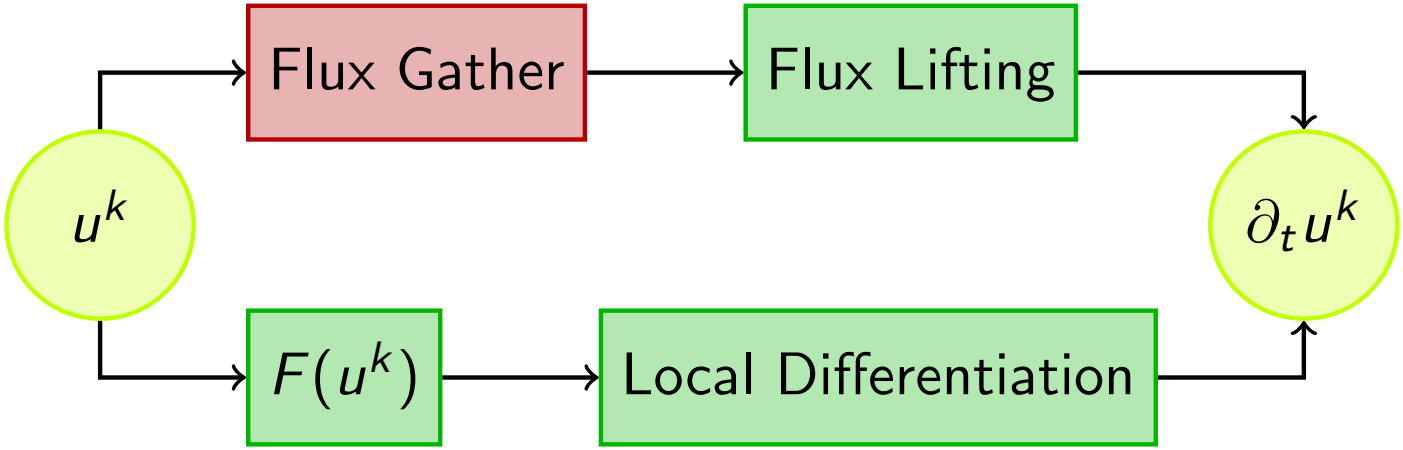
Subsitute in basis functions, introduce elementwise stiffness, mass, and surface mass matrices matrices S, M, M_A:

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

For straight-sided simplicial elements:
Reduce D^{∂_ν} and L to reference matrices.

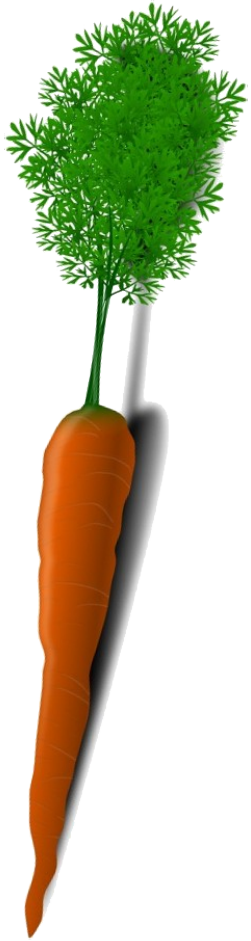
Decomposition of a DG operator into Subtasks

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



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:

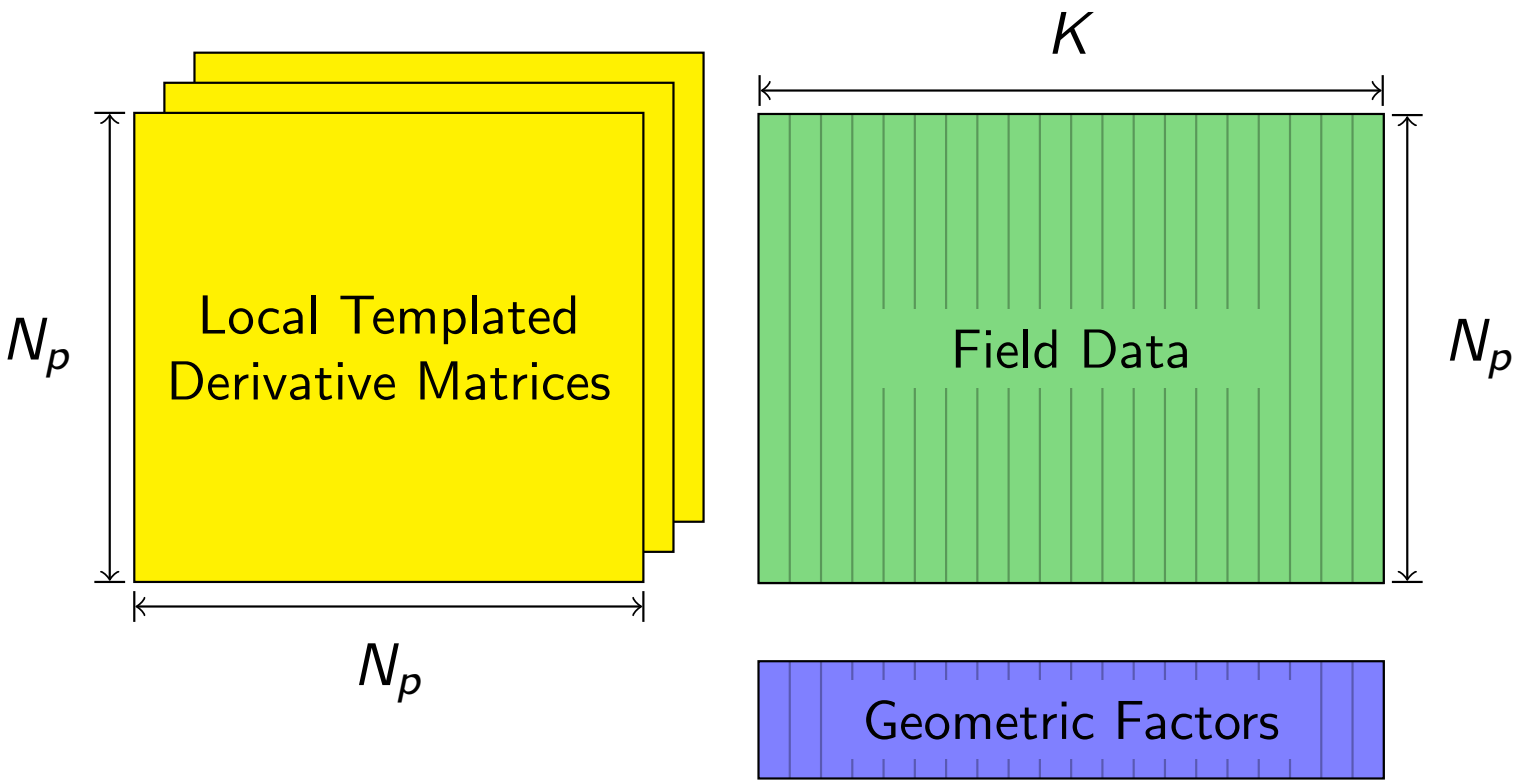
- *Main: Speed*
  Reduce need for compute-bound clusters
- *Secondary: Generality*
  Be applicable to many problems
- *Tertiary: Ease-of-Use*
  Hide complexity of GPU hardware

Setting (for now):

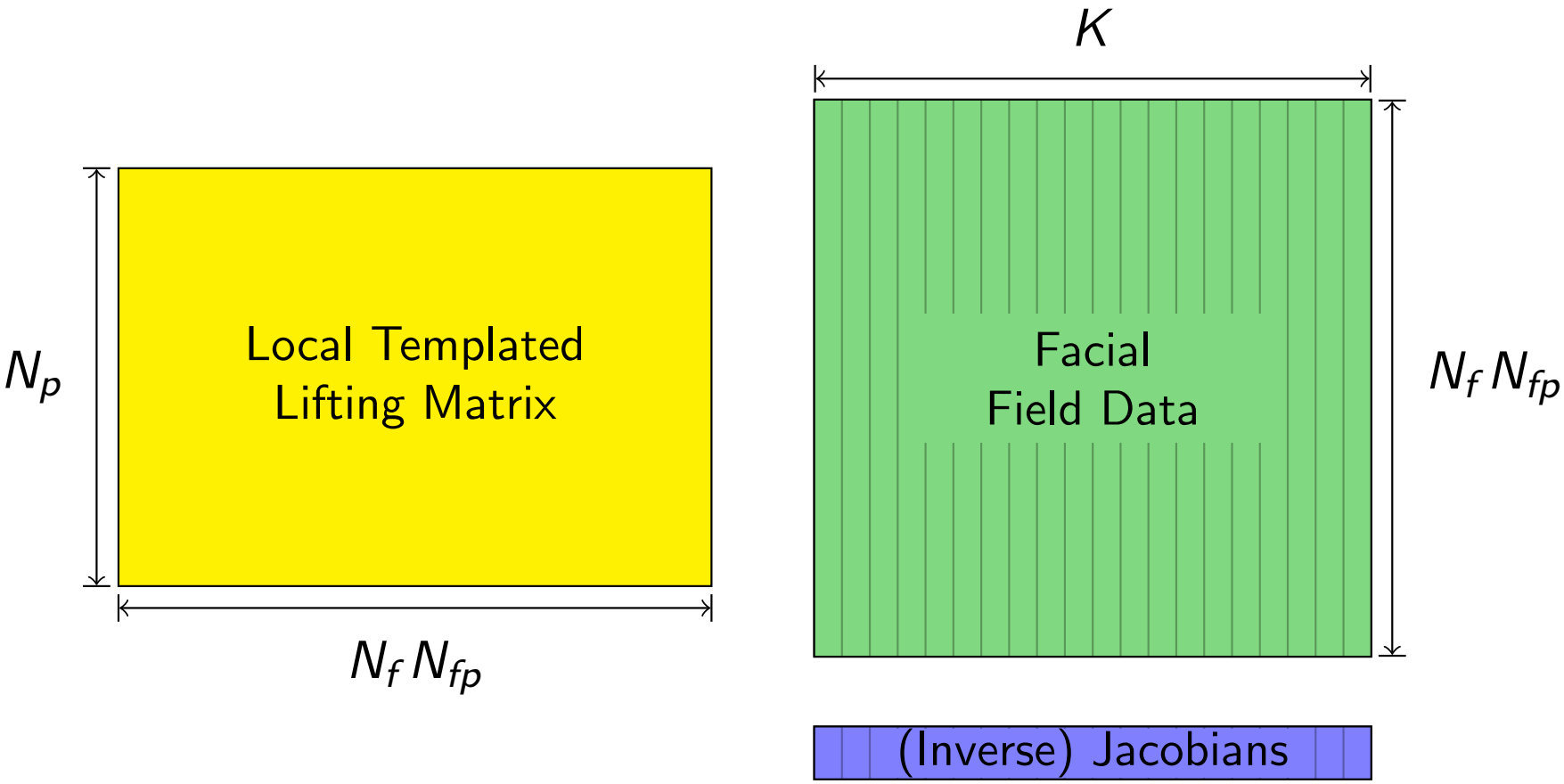
- Specialize to straight-sided simplices
- Optimize for (but don't specialize to) tetrahedra (ie. 3D)
- Optimize for "medium" order (3...5)

Outline

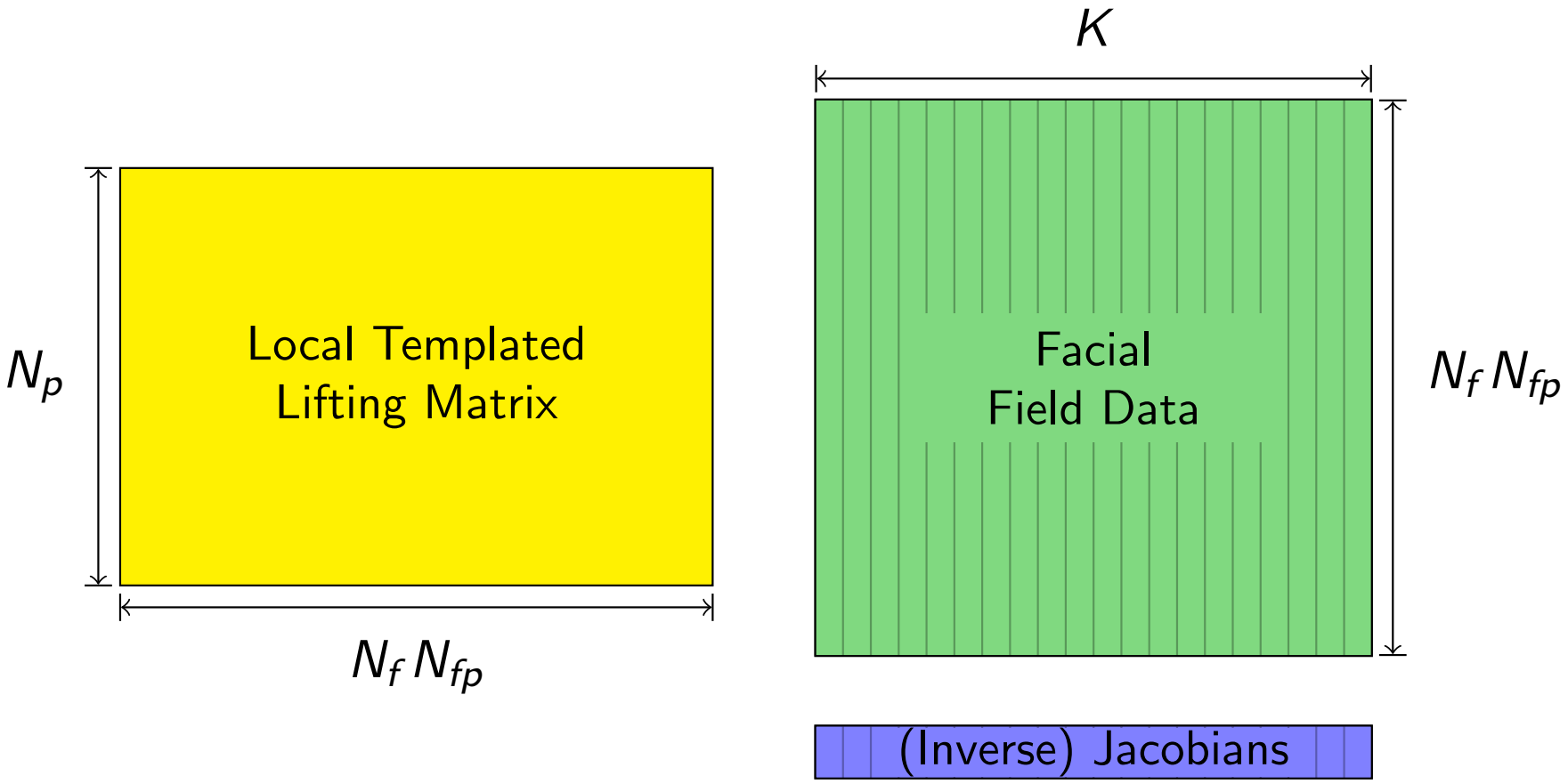
Element-Local Operations: Differentiation



Element-Local Operations: Lifting

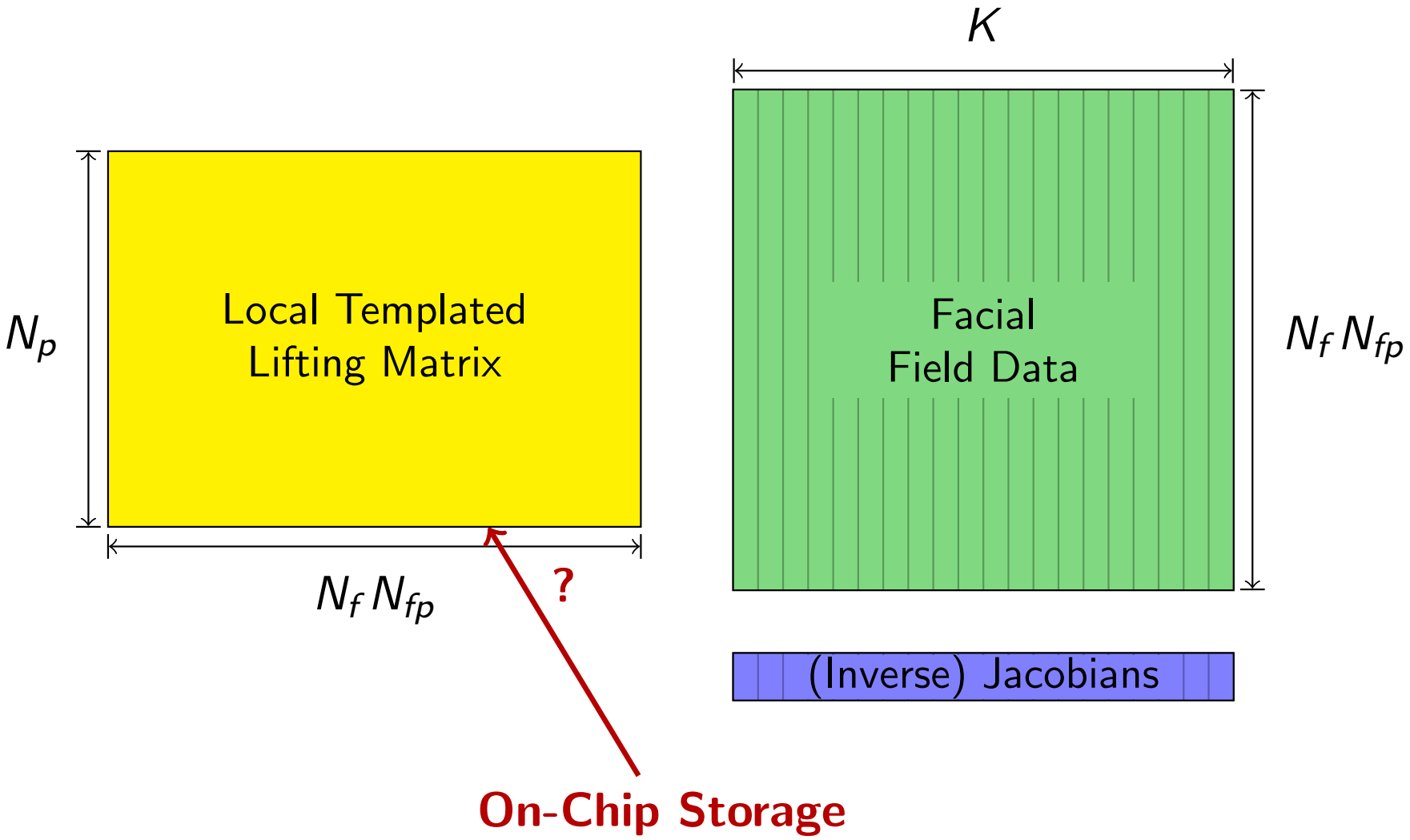


Element-Local Operations: Lifting

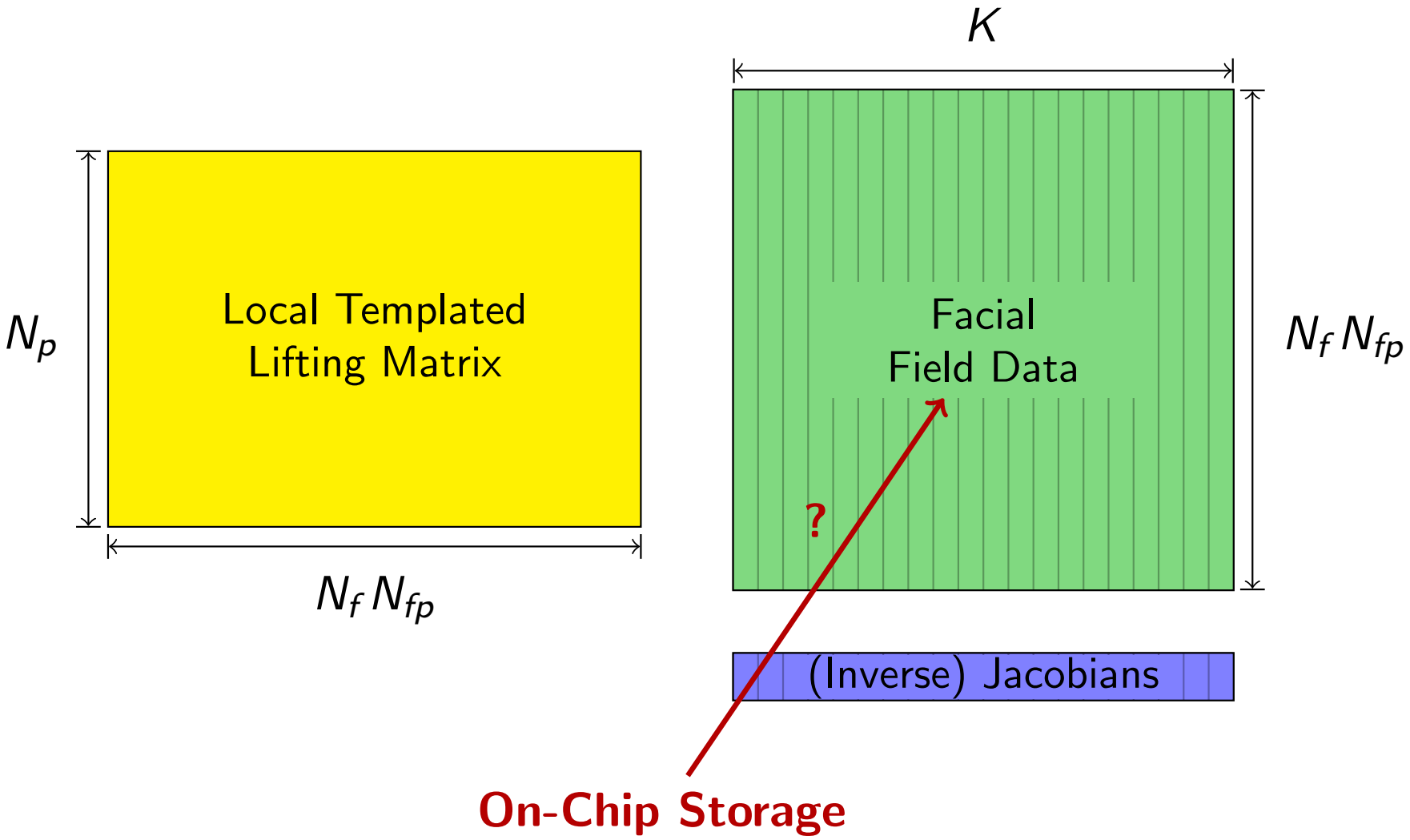


On-Chip Storage

Element-Local Operations: Lifting



Element-Local Operations: Lifting



Best use for on-chip memory?

Basic Problem

On-chip storage is scarce. . .

. . . and will be for the foreseeable future.

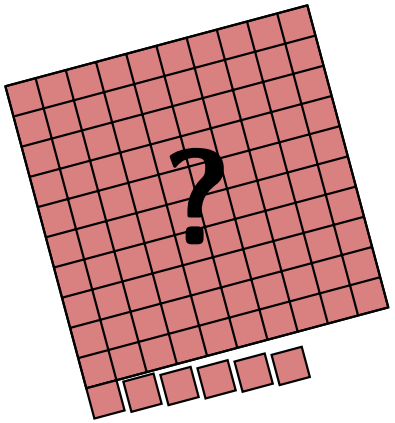
Possible uses:

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

How to decide? Does it matter?

Work Partition for Element-Local Operators

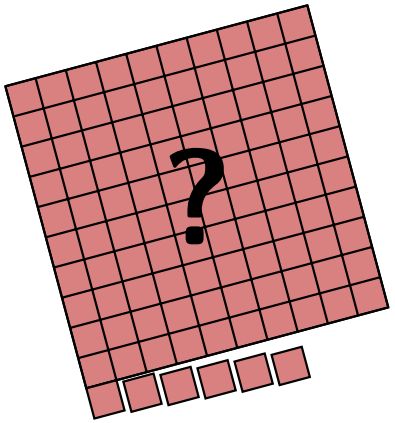
Natural Work Decomposition:
One Element per Block



Work Partition for Element-Local Operators

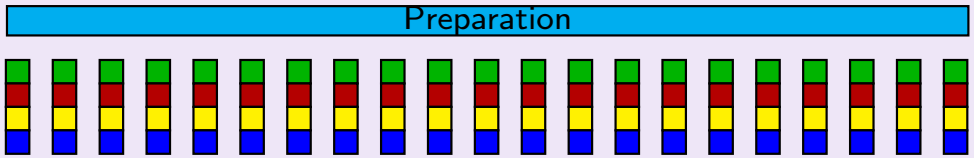
Natural Work Decomposition:
One Element per Block

- ⊕ 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

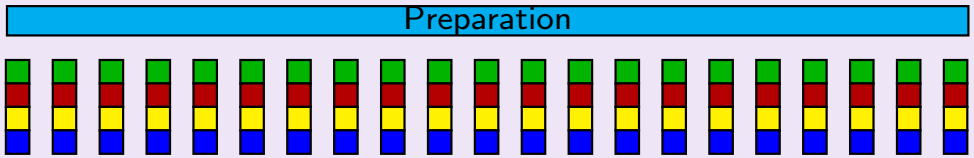
Per Block: K_L element-local mat.mult. + matrix load



Question: How should one assign work to threads?

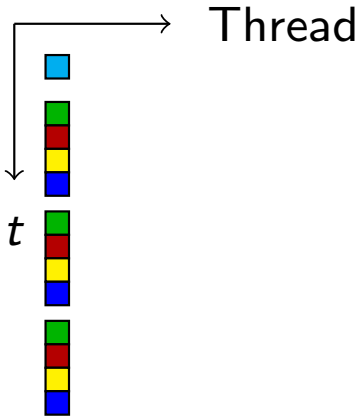
Loop Slicing for element-local parts of GPU DG

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



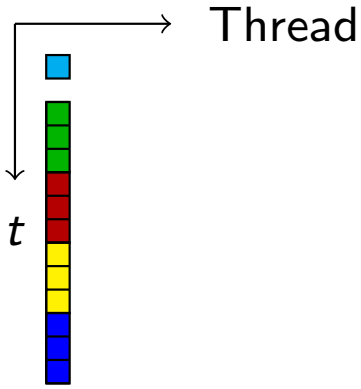
Question: How should one assign work to threads?

w_s: in sequence



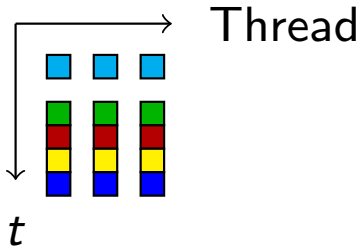
(amortize preparation)

w_i: "inline-parallel"



(exploit register space)

w_p: in parallel



Best Work Partition?

Basic Problem

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

Possible work partitions:

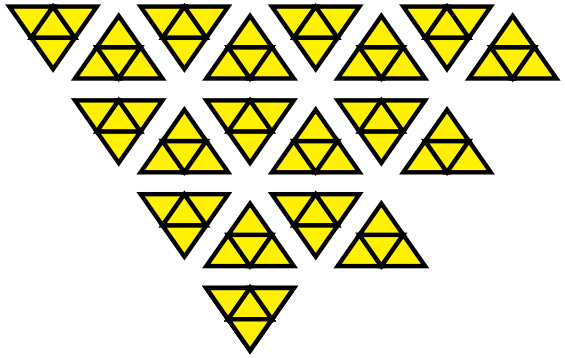
- One or multiple elements per block?
- One or multiple DOFs per thread?
  - In parallel?
  - In sequence?
  - In-line?

How to decide? Does it matter?

Work Partition for Surface Flux Evaluation

Granularity Tradeoff:

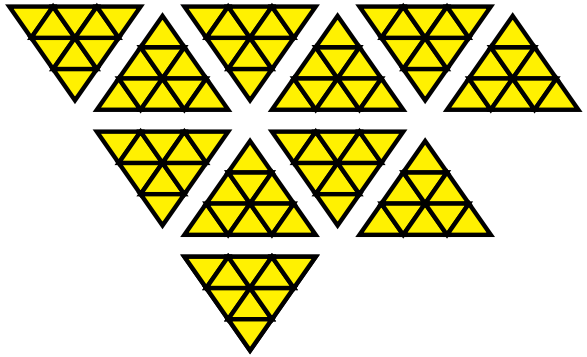
- Large Blocks:
  - ⊕ More Data Reuse
  - ⊖ Less Parallelism
  - ⊖ Less Latency Hiding
- Block Size limited by two factors:
  - Output buffer size
  - Face metadata size
- Optimal Block Size:
  not obvious ?



Work Partition for Surface Flux Evaluation

Granularity Tradeoff:

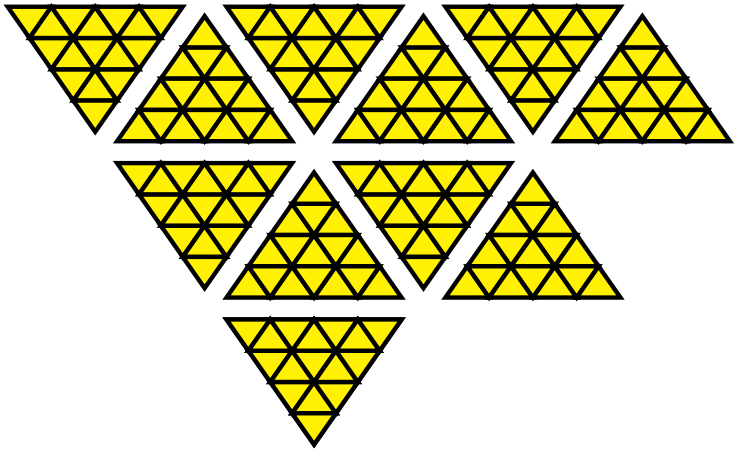
- Large Blocks:
  - ⊕ More Data Reuse
  - ⊖ Less Parallelism
  - ⊖ Less Latency Hiding
- Block Size limited by two factors:
  - Output buffer size
  - Face metadata size
- Optimal Block Size:
  not obvious ?



Work Partition for Surface Flux Evaluation

Granularity Tradeoff:

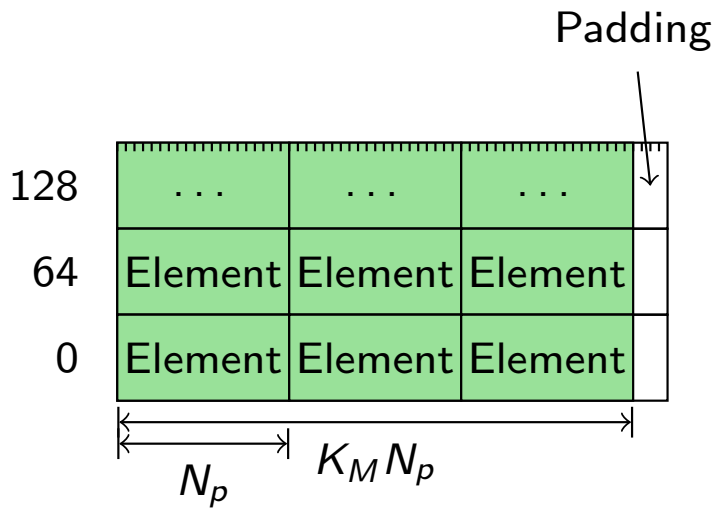
- Large Blocks:
  - ⊕ More Data Reuse
  - ⊖ Less Parallelism
  - ⊖ Less Latency Hiding
- Block Size limited by two factors:
  - Output buffer size
  - Face metadata size
- Optimal Block Size:
  not obvious ?



More than one Granularity

Different block sizes introduced so far:

- Differentiation
- Lifting
- Surface Fluxes



More than one Granularity

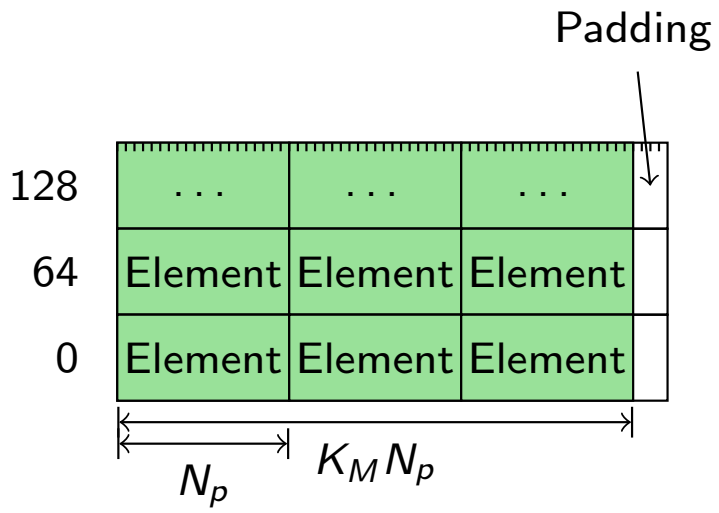
Different block sizes introduced so far:

- 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:

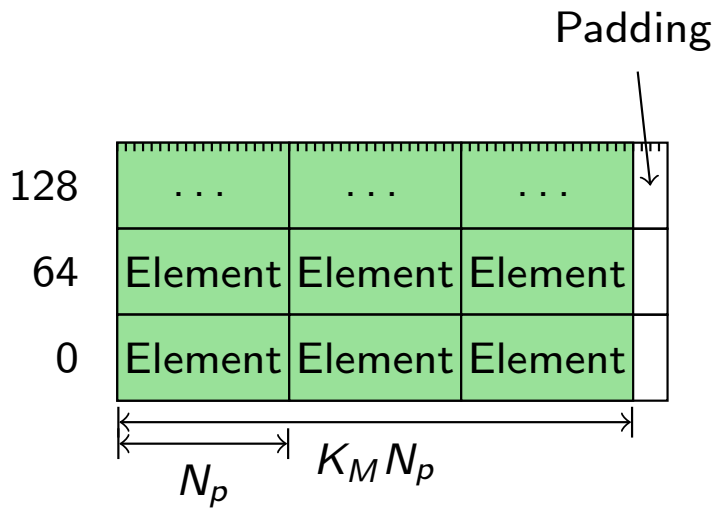
- Differentiation
- Lifting
- Surface Fluxes

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

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

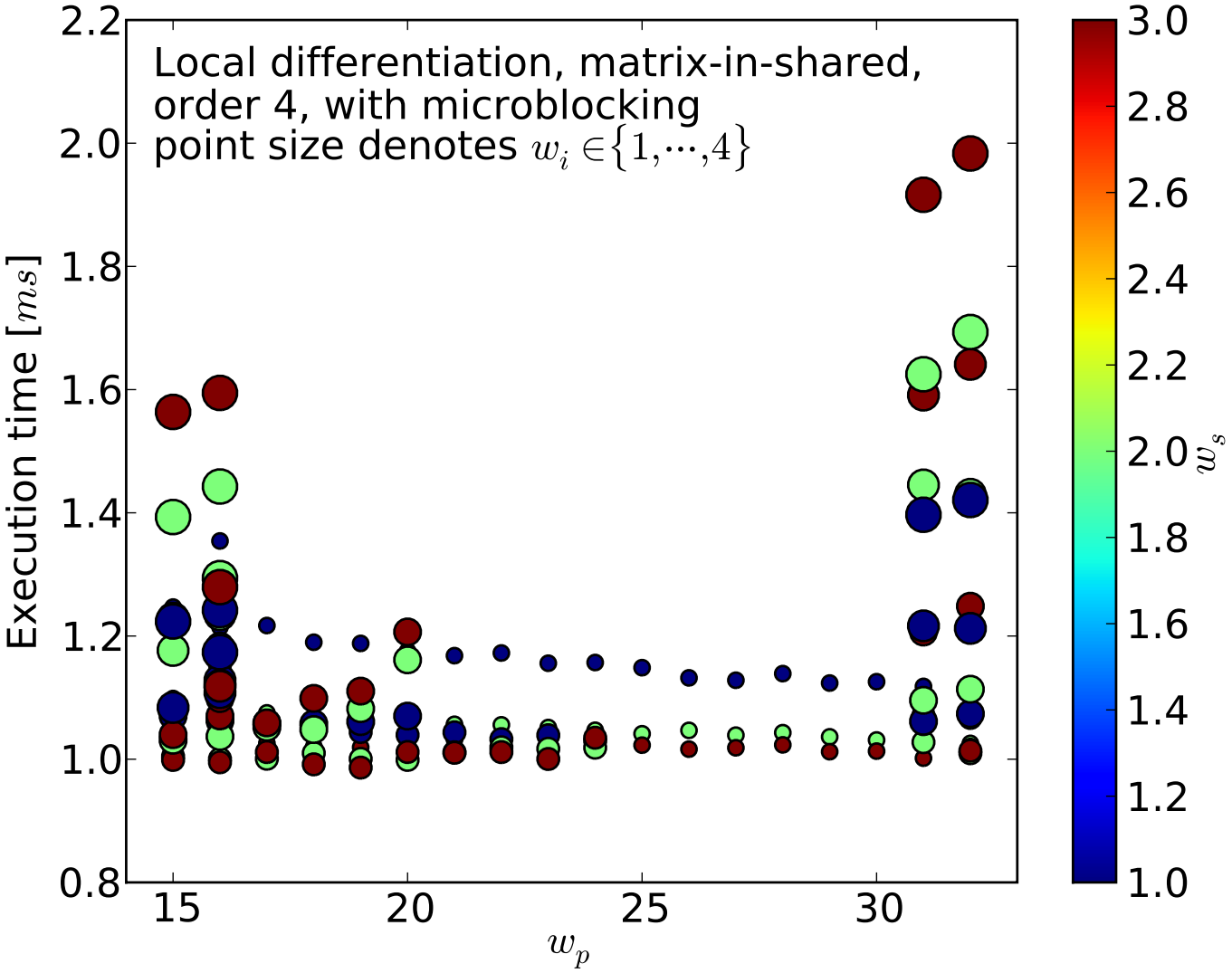
Metaprogramming for GPU-DG

- 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

Loop Slicing for Differentiation



Metaprogramming DG: Flux Terms

$$0 = \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)]\varphi \,\mathrm{d}x - \underbrace{\int_{\partial \mathrm{D}_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*]\varphi \,\mathrm{d}S_x}_{\text{Flux term}}$$

Metaprogramming DG: Flux Terms

$$0 = \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)]\varphi \,\mathrm{d}x - \underbrace{\int_{\partial \mathrm{D}_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*]\varphi \,\mathrm{d}S_x}_{\text{Flux term}}$$

Flux terms:

- vary by problem
- expression specified by user
- evaluated pointwise

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

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

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

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

User writes: Vectorial statement in math. notation

```
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

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

We generate: Scalar evaluator in C (6×)

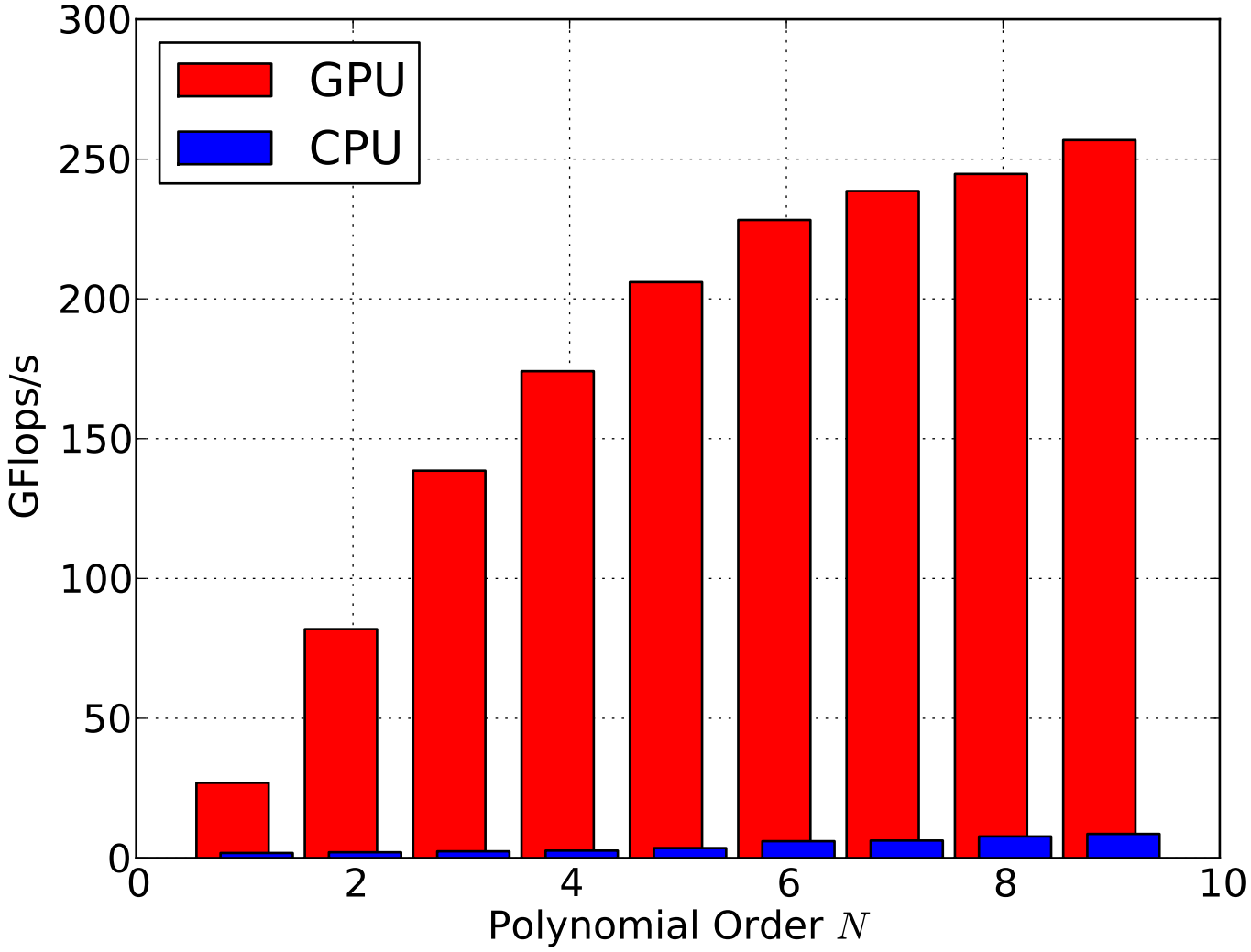
```
a_flux += (
    (((  val_a_field5  -  val_b_field5 )* fpair->normal[2]
      - ( val_a_field4  -  val_b_field4 )* fpair->normal[0])
     +  val_a_field0  -  val_b_field0 )* fpair->normal[0]
    - (((  val_a_field4  -  val_b_field4 ) * fpair->normal[1]
        - ( val_a_field1  -  val_b_field1 )*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

Hedge DG Solver

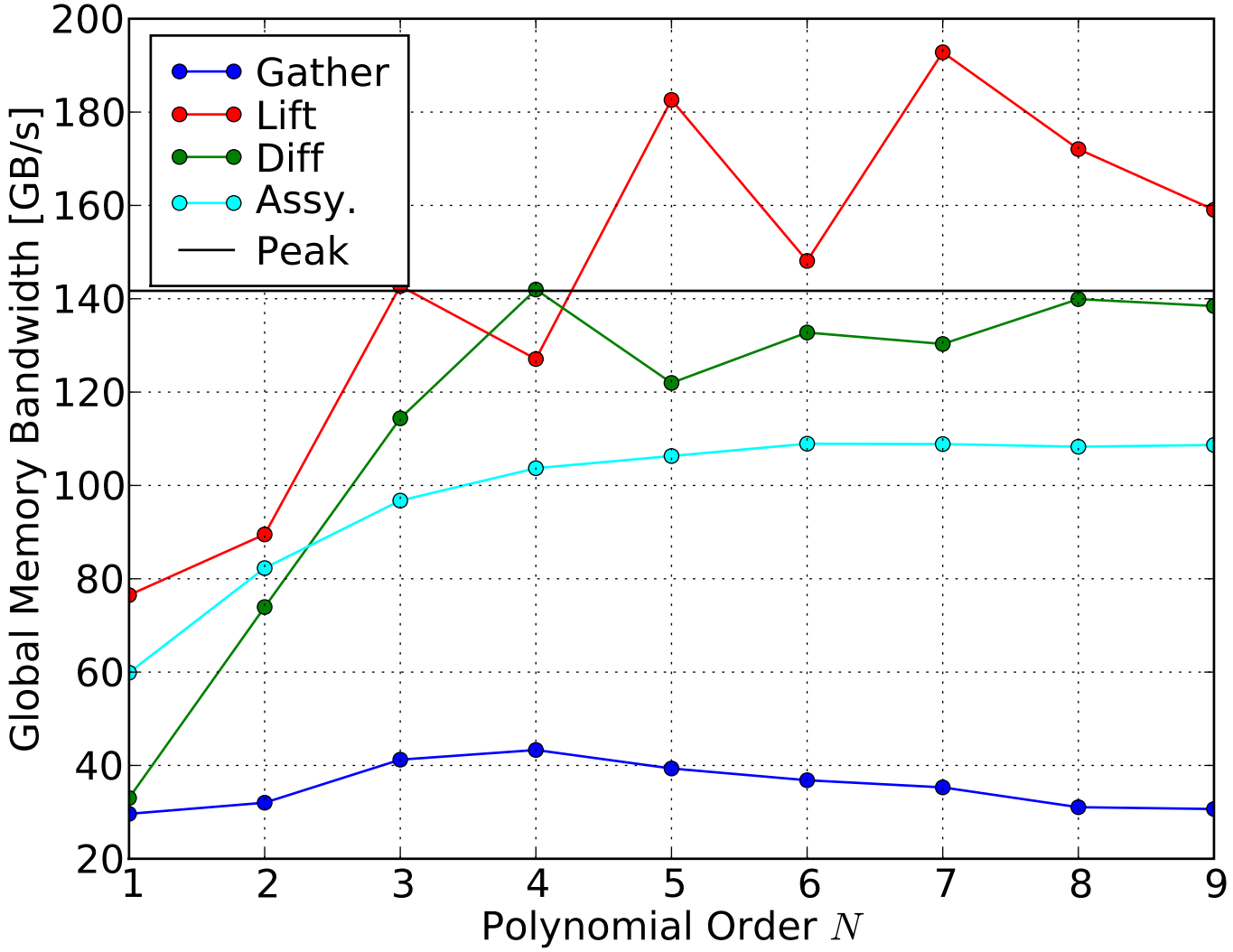
- High-Level Operator Description
  - Maxwell's
  - Euler
  - Poisson
  - Compressible Navier-Stokes, ...
- One Code runs...
  - ...on CPU, CUDA
  - ...on {CPU,CUDA}+MPI
  - ...in 1D, 2D, 3D
  - ...at any order
- Uses CPU, GPU code generation
- Open Source (GPL3)
- Written in Python,

Outline

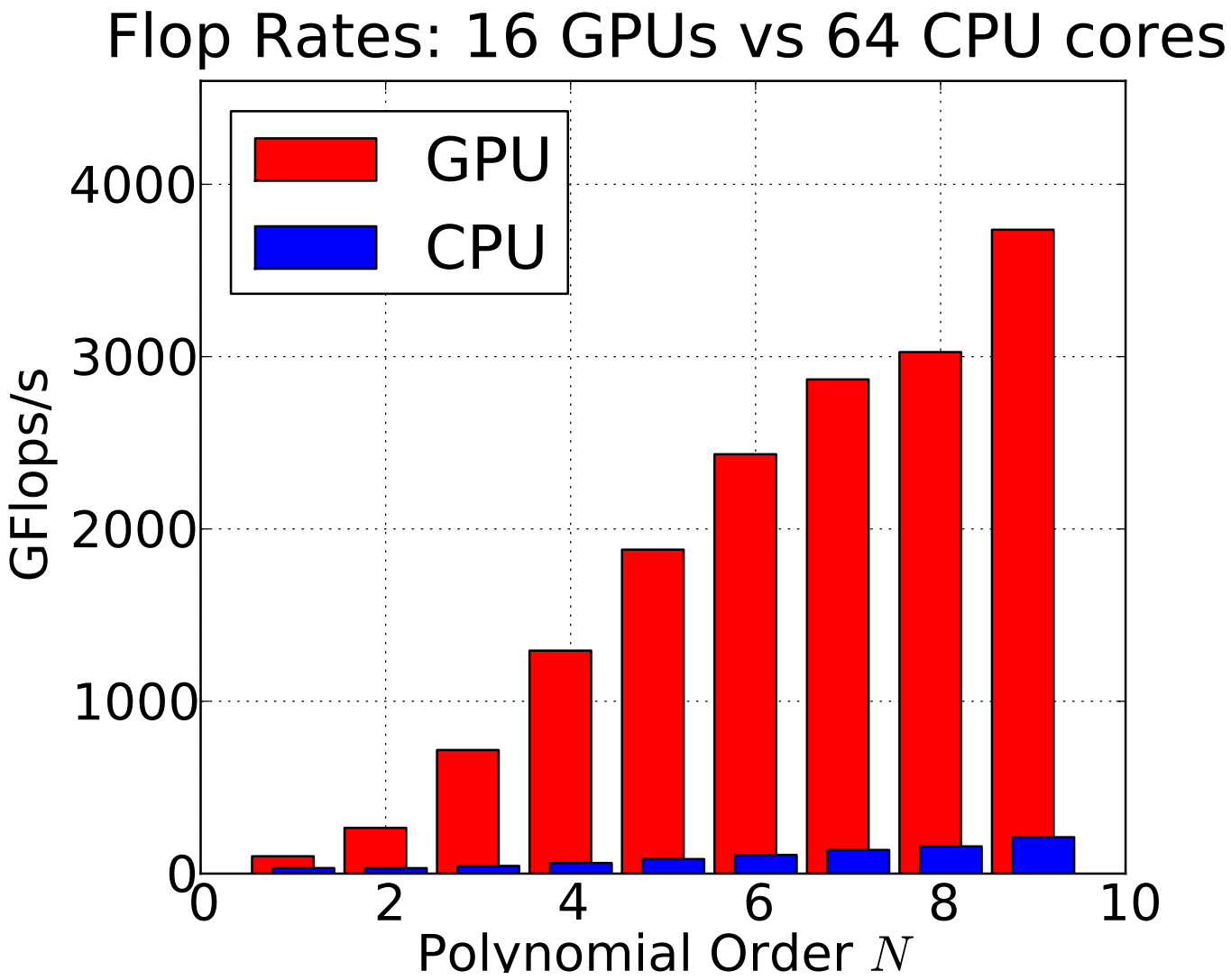
Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400



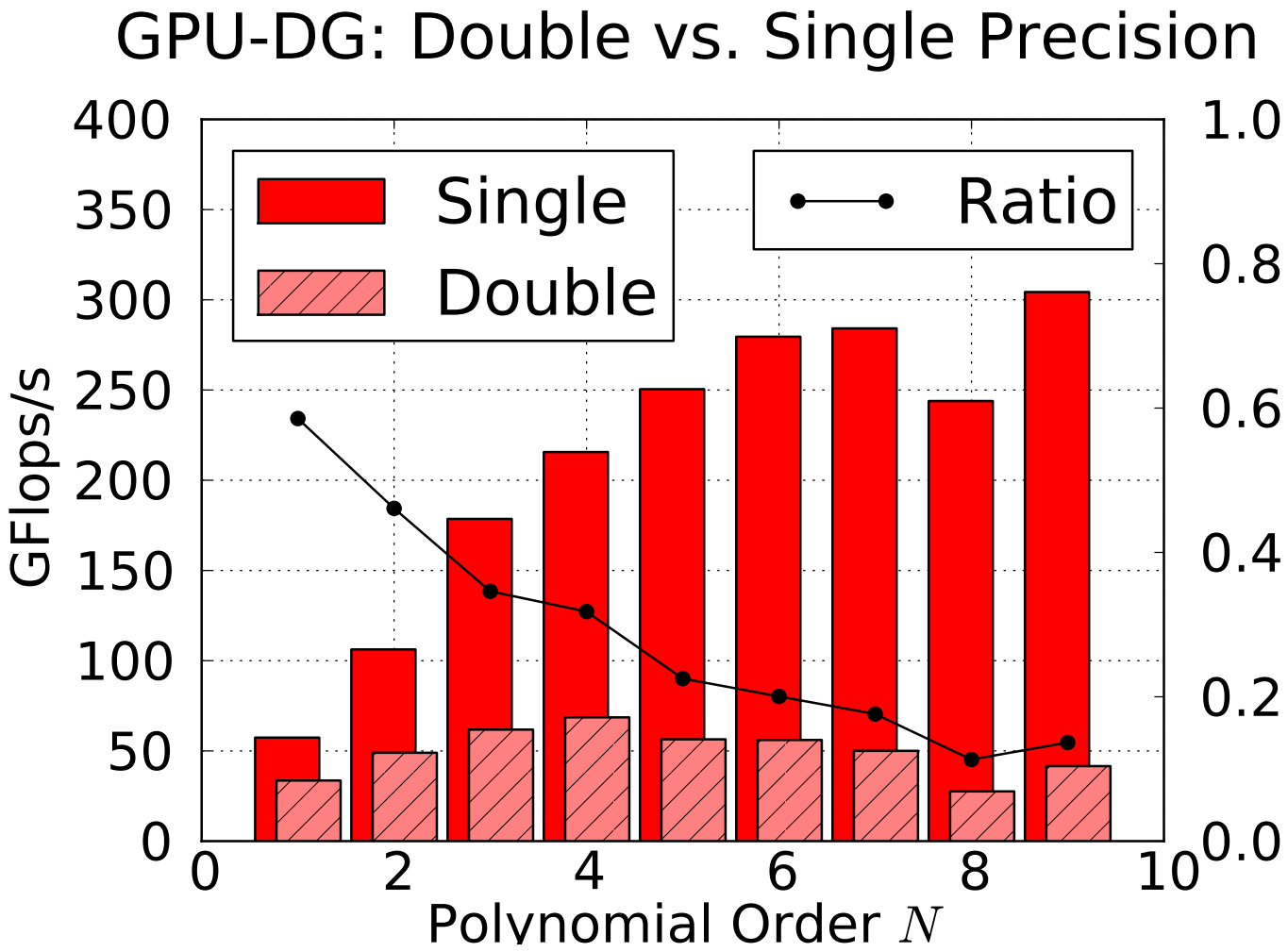
Memory Bandwidth on a GTX 280



Multiple GPUs via MPI: 16 GPUs vs. 64 CPUs

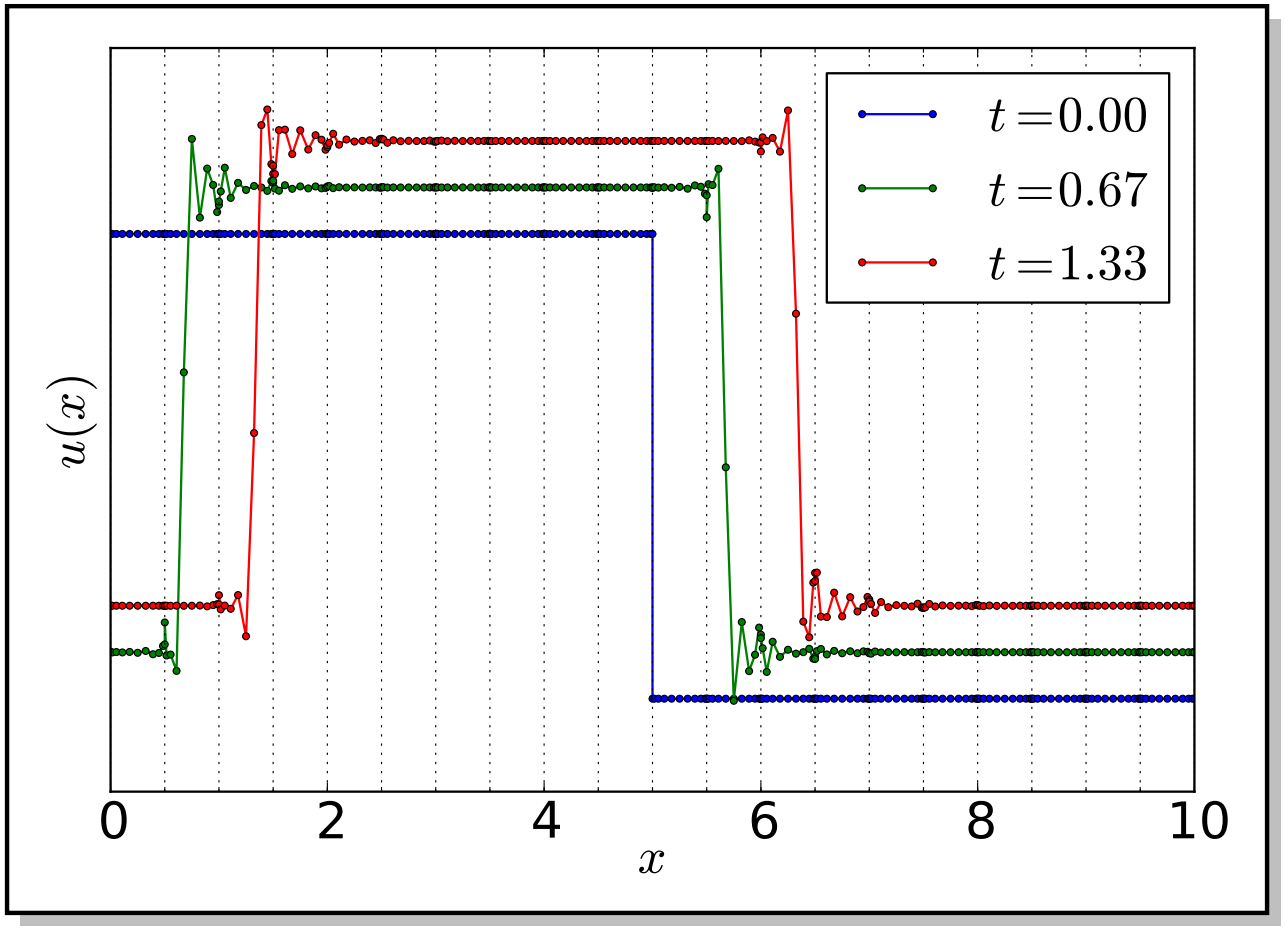


GPU-DG in Double Precision

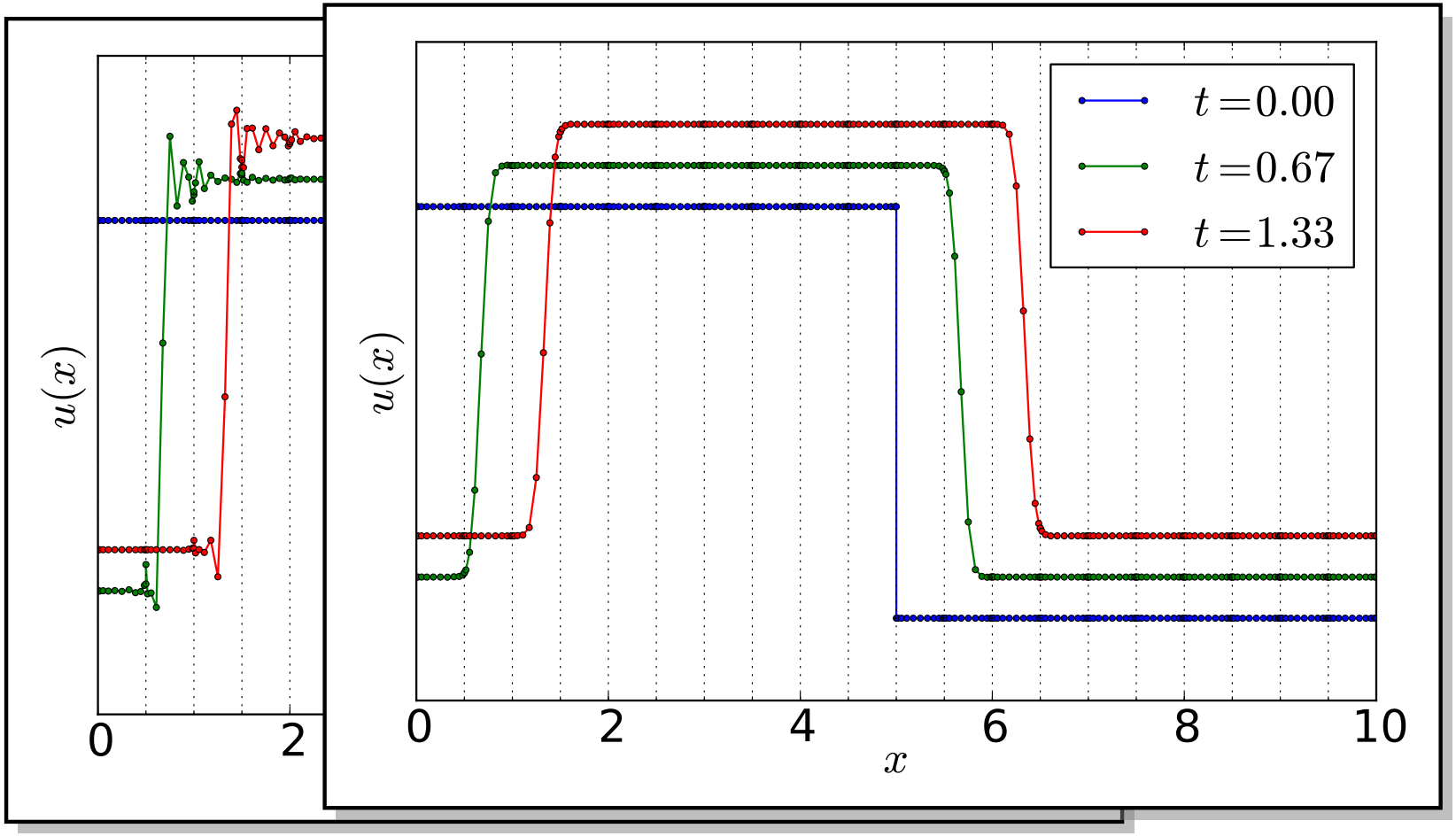


Outline

Nonlinear conservation laws → shocks?



Nonlinear conservation laws → shocks?



Nonlinear conservation laws → shocks?

1D advection:

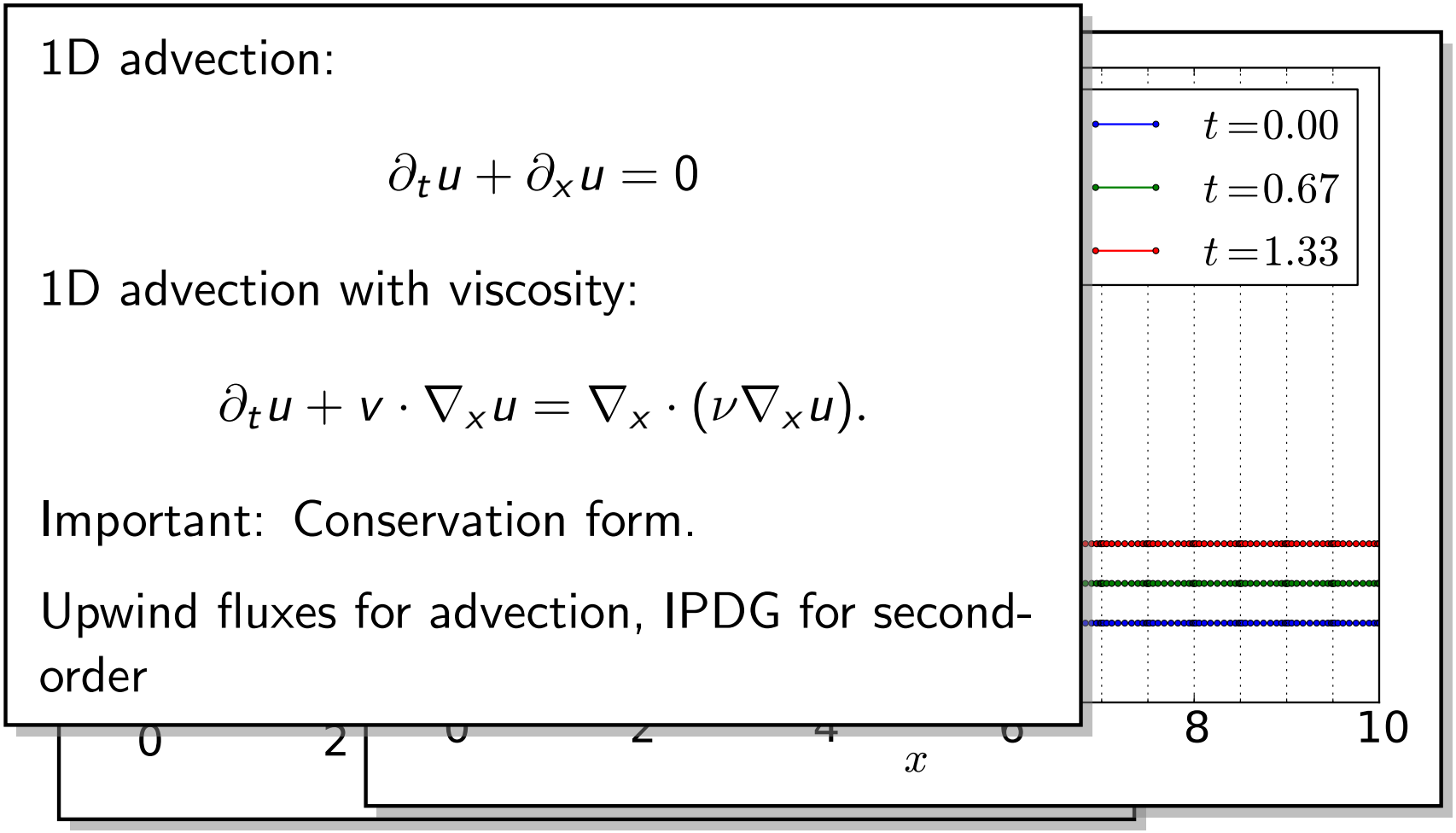
$$\partial_t u + \partial_x u = 0$$

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



Nonlinear conservation laws → shocks?

1D advection:

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

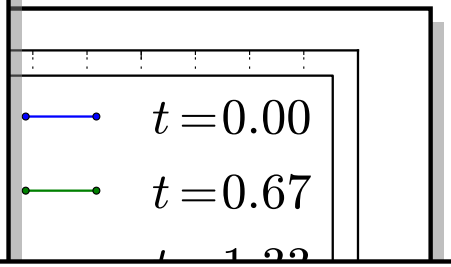
1D advection

$$\partial_t u +$$

Important: Co

Upwind fluxes
order

0 2



- Detector → ν?
  GPU-suitability? Data locality?
  Properties? [Build on work by Persson/Peraire '06]
- Time integration
  Implicit/explicit? Adaptivity? RKC for bigger Δt with viscosity?
- Accuracy?
  Near shocks? Away from them?

Results: Euler's Equations of Gas Dynamics

Euler's equations with viscosity:

$$\partial_t \rho + \nabla_x \cdot (\rho \mathbf{u}) = \nabla_x \cdot (\nu \nabla_x \rho),$$
$$\partial_t (\rho \mathbf{u}) + \nabla_x \cdot (\mathbf{u} \otimes (\rho \mathbf{u})) + \nabla_x p = \nabla_x \cdot (\nu \nabla_x (\rho \mathbf{u})),$$
$$\partial_t E + \nabla_x \cdot (\mathbf{u}(E + p)) = \nabla_x \cdot (\nu \nabla_x E).$$

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

[Persson/Peraire '06] suggest Navier-Stokes-like viscosity. No good: can't control jumps in ρ.

Rusanov fluxes for Euler, IPDG for viscosity.

Results: Euler's Equations of Gas Dynamics

Euler's equations with viscosity:

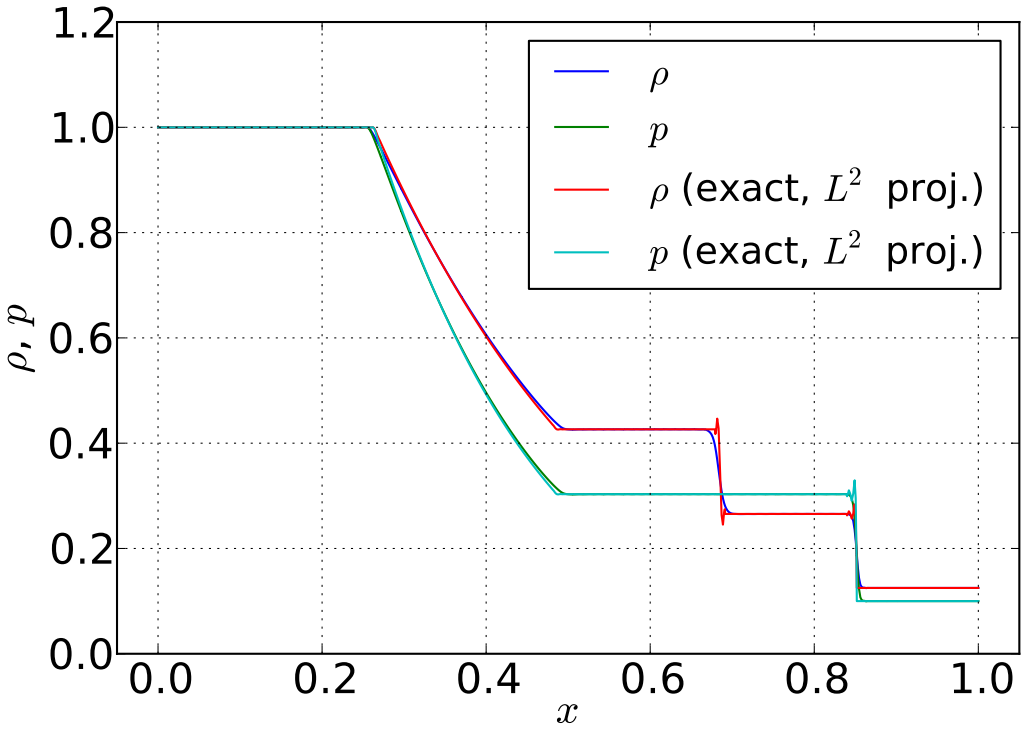
$$\partial_t \rho + \nabla_x \cdot (\rho \mathbf{u}) = \nabla_x \cdot (\nu \nabla_x \rho),$$

$$\partial_t(\rho \mathbf{u}$$))),

Again of the ot

[Perss is-cosity.

Rusan



Results: Euler's Equations of Gas Dynamics

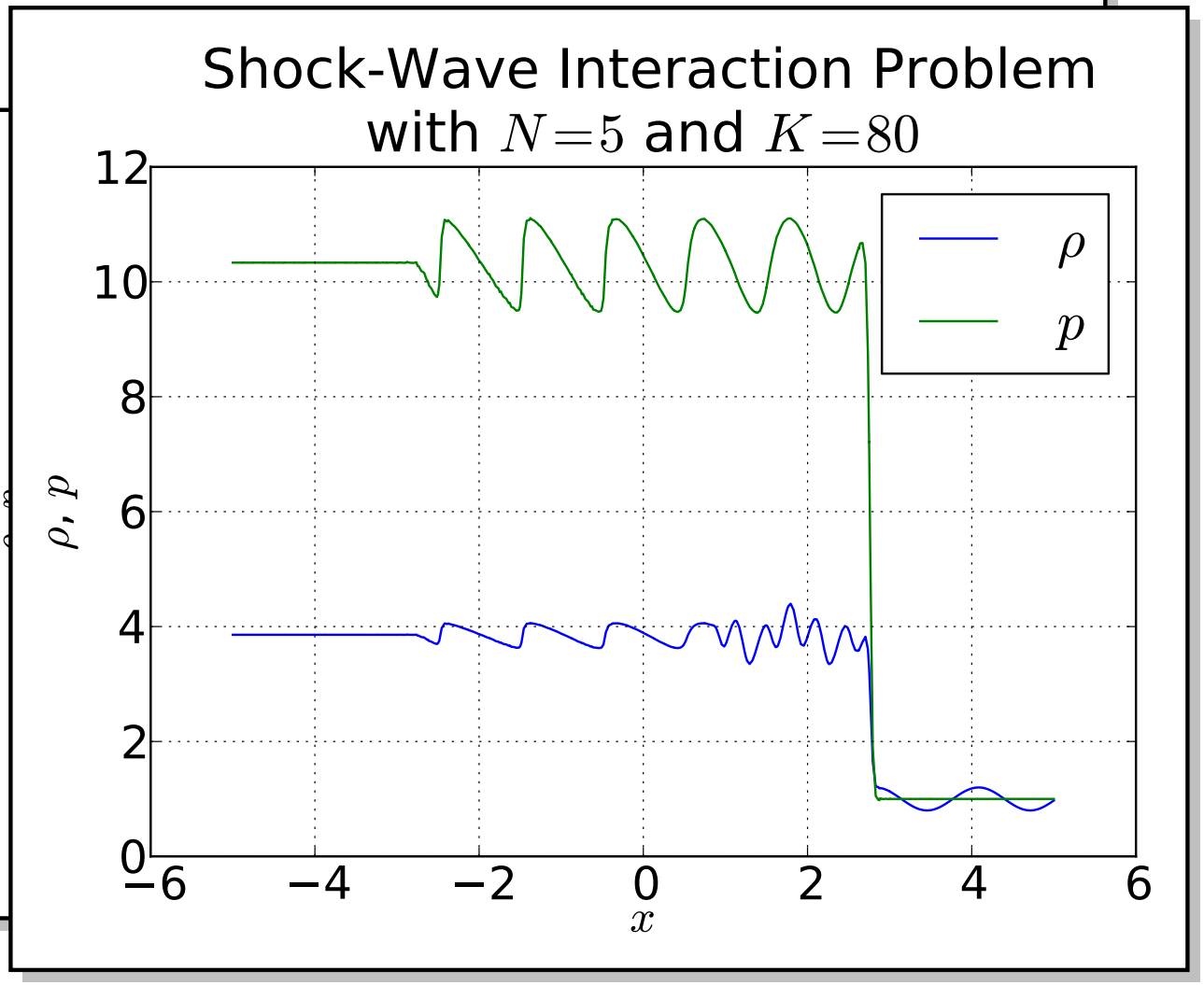
Euler's equations with viscosity:

$\partial_t(\rho \mathbf{u}$

Again
the ot

[Perss
cosity.

Rusan



Results: Euler's Equations of Gas Dynamics

Euler's equations with viscosity:

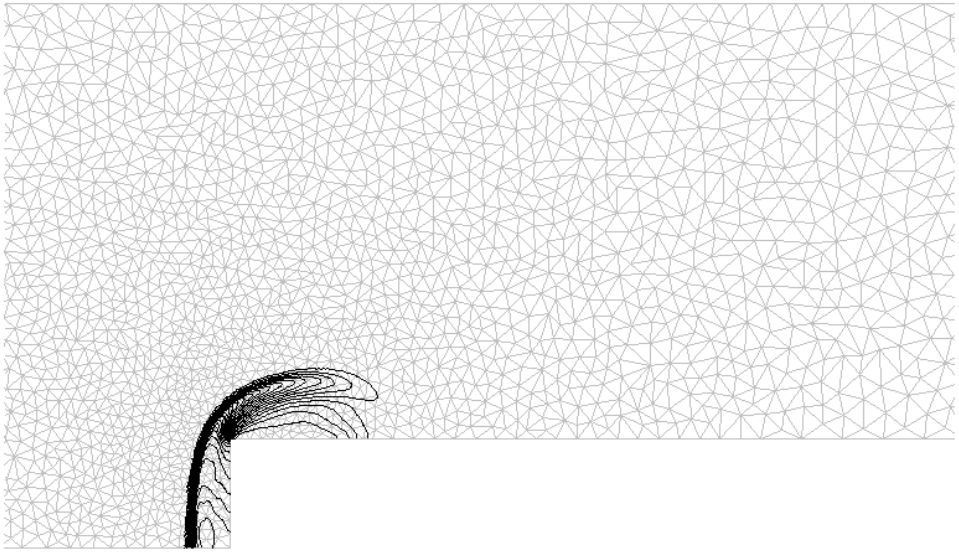
$\partial_t(\rho \mathbf{u}$

Again
the ot

[Perss
cosity.

Rusan

Shock-Wave Interaction Problem



Results: Euler's Equations of Gas Dynamics

Euler's equations with viscosity:

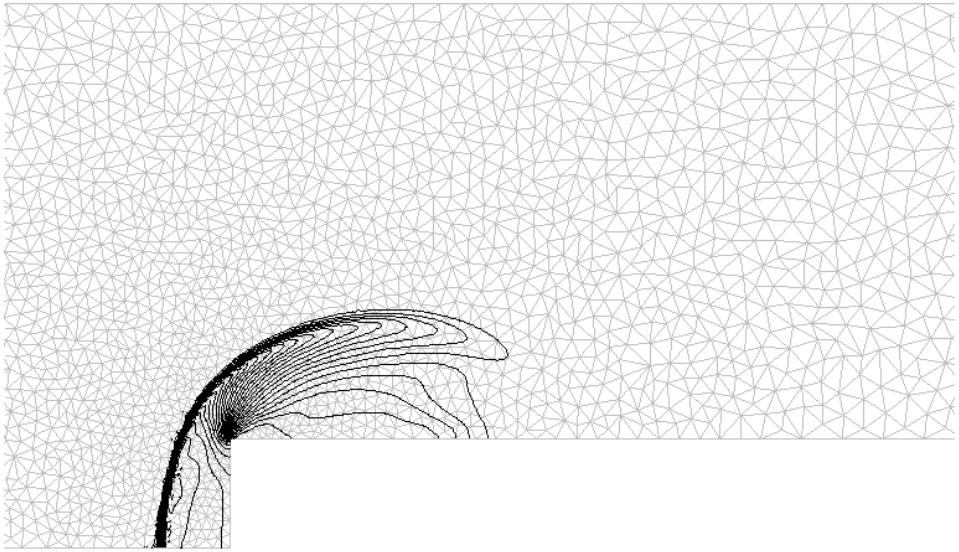
$\partial_t(\rho \mathbf{u}$

Again
the ot

[Perss
cosity.

Rusan

Shock-Wave Interaction Problem



Results: Euler's Equations of Gas Dynamics

Euler's equations with viscosity:

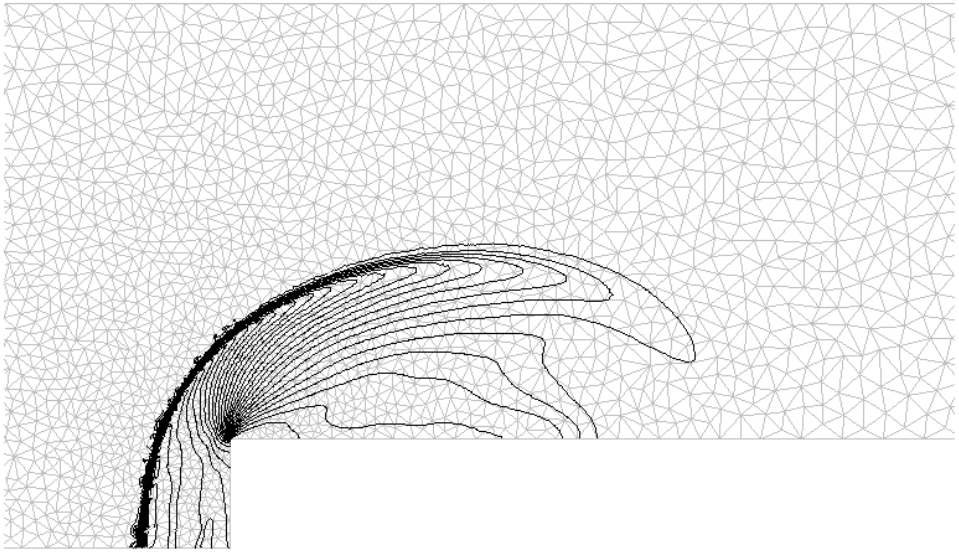
$\partial_t(\rho \mathbf{u}$

Again
the ot

[Perss
cosity.

Rusan

Shock-Wave Interaction Problem



Results: Euler's Equations of Gas Dynamics

Euler's equations with viscosity:

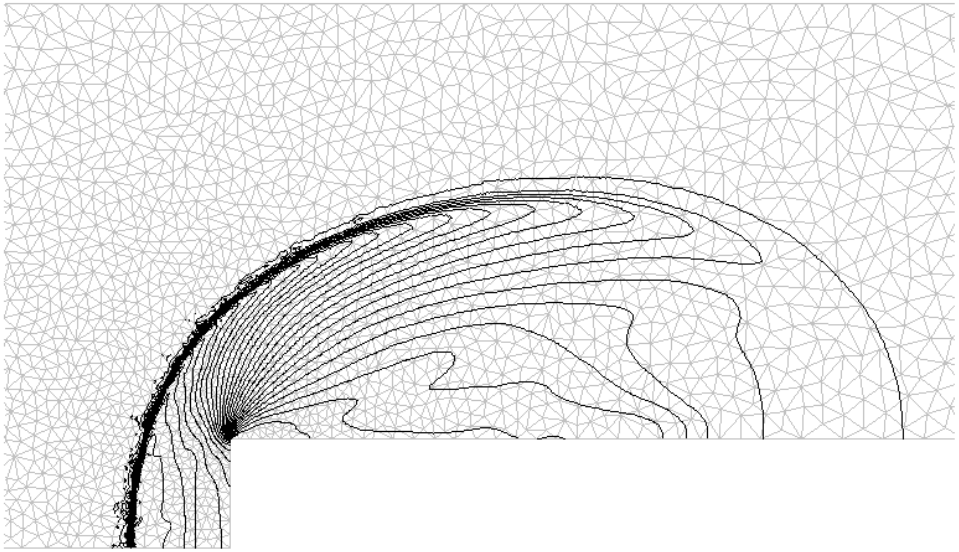
$\partial_t(\rho \mathbf{u}$

Again
the ot

[Perss
cosity.

Rusan

Shock-Wave Interaction Problem



Results: Euler's Equations of Gas Dynamics

Euler's equations with viscosity:

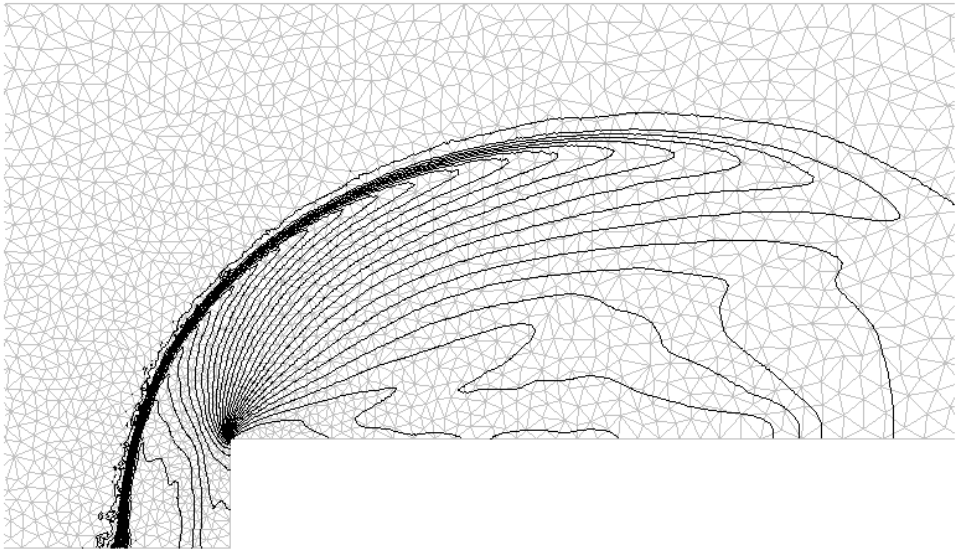
$\partial_t(\rho \mathbf{u}$

Again
the ot

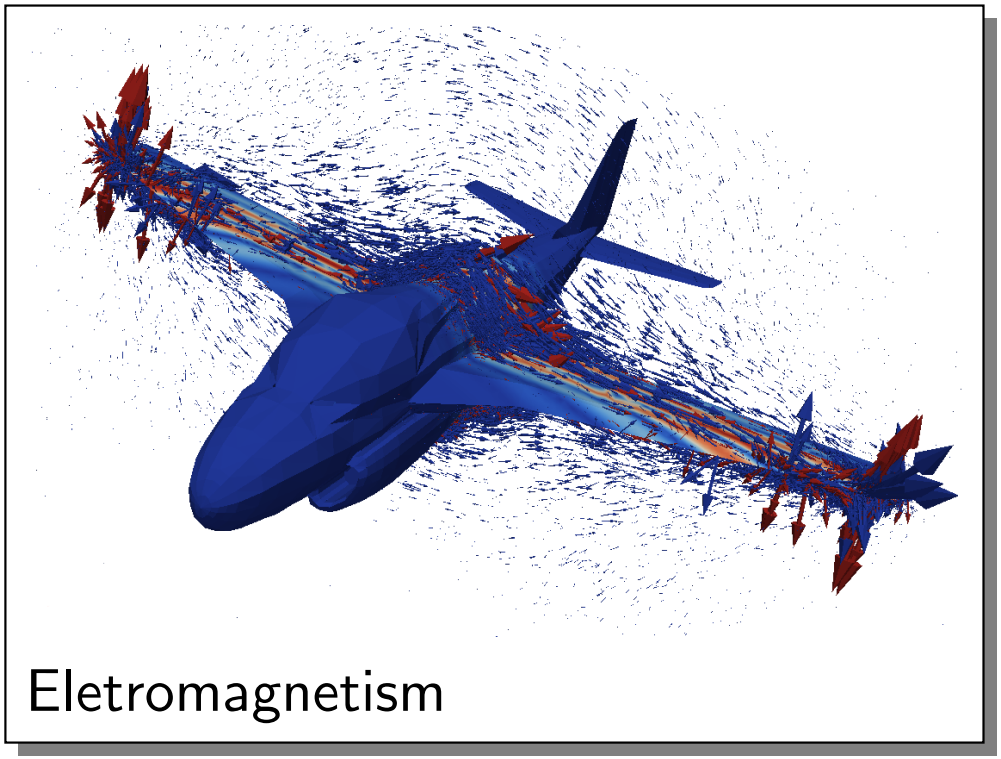
[Perss
cosity.

Rusan

Shock-Wave Interaction Problem

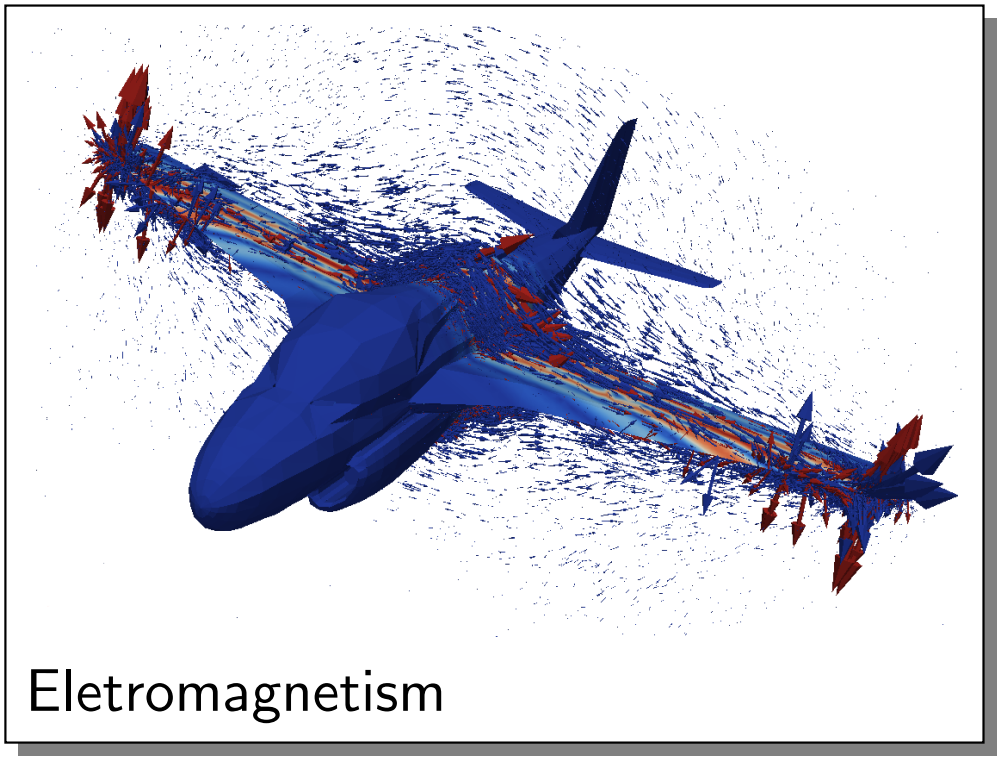


GPU DG Showcase

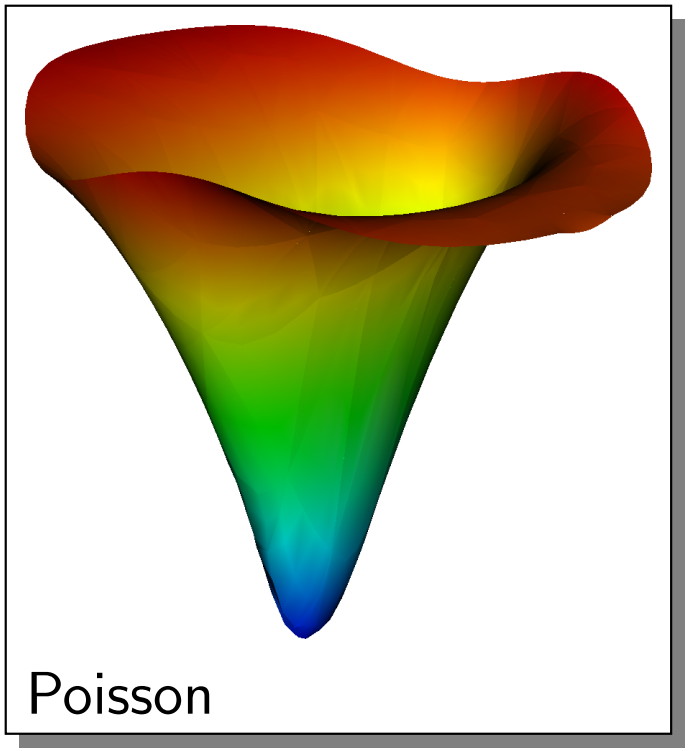


Eletromagnetism

GPU DG Showcase

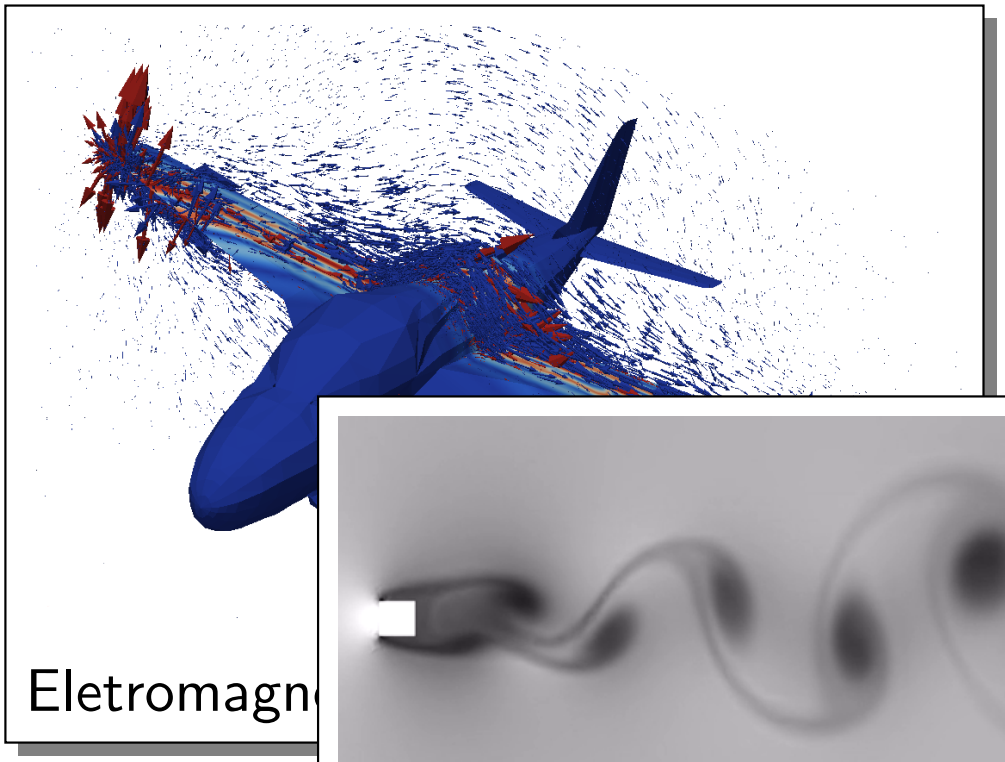


Eletromagnetism

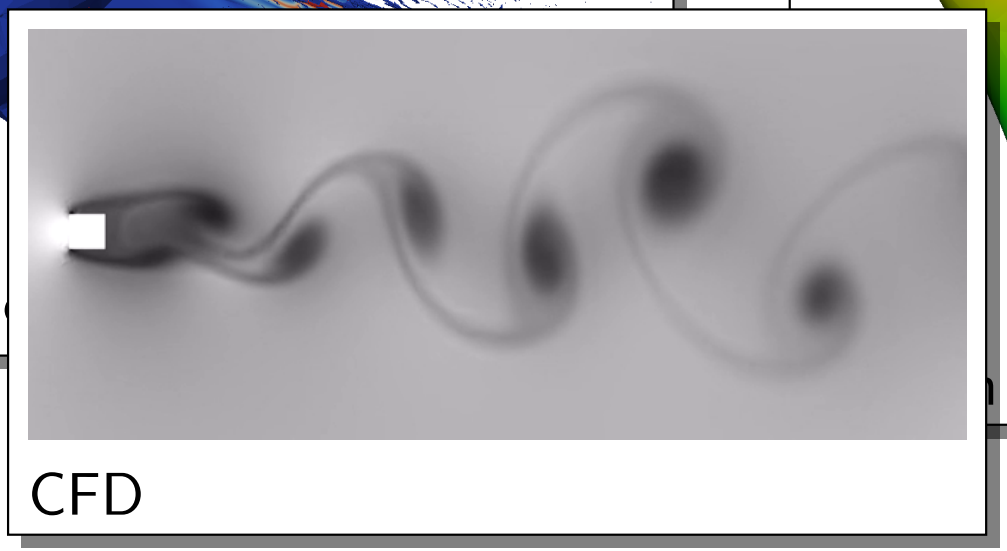


Poisson

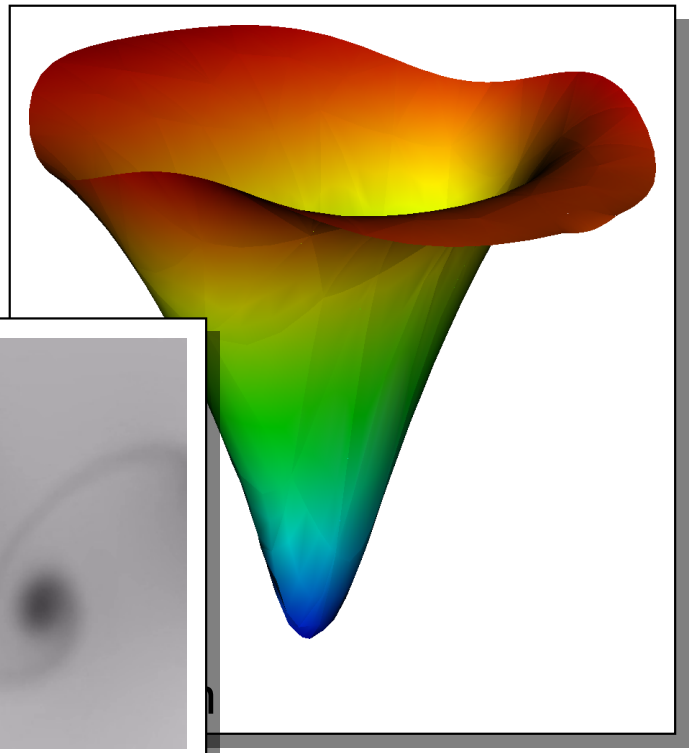
GPU DG Showcase



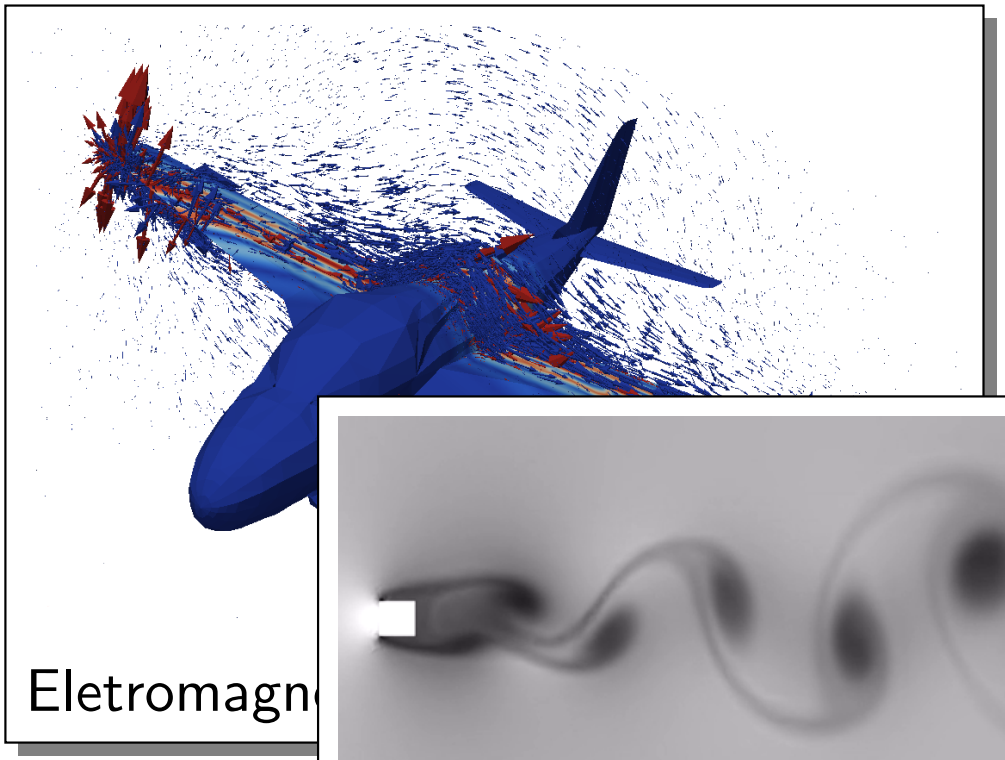
Eletromagn



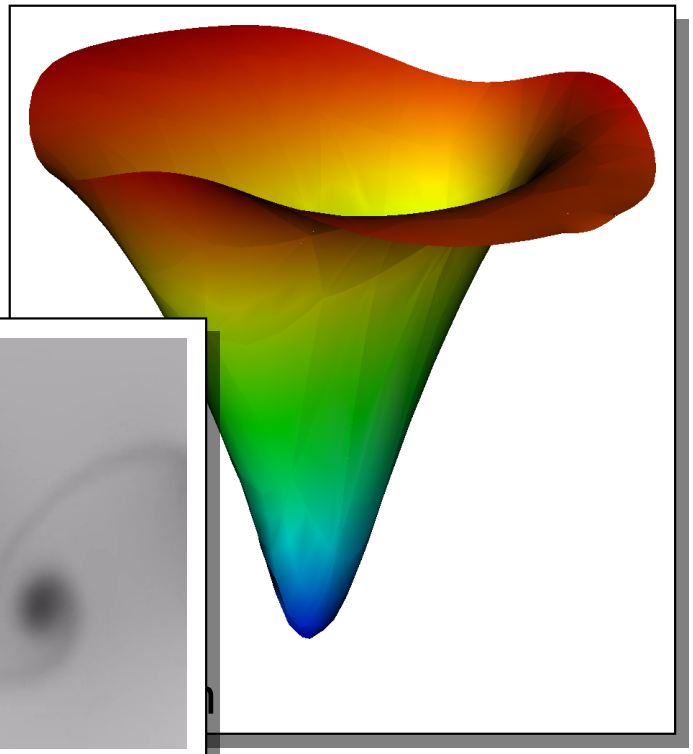
CFD

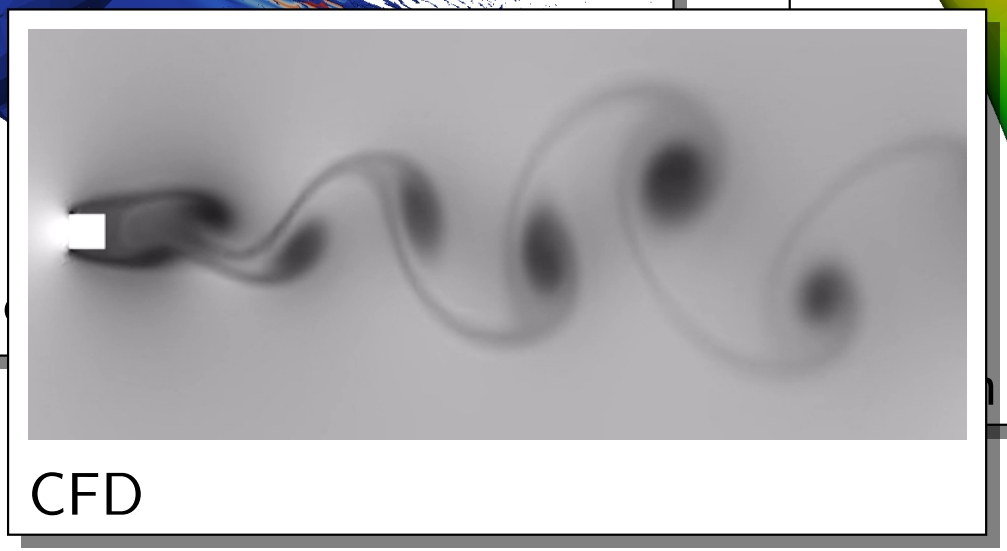


GPU DG Showcase



Eletromagn





CFD

Where to from here?

PyCUDA, PyOpenCL, hedge

→ `http://www.cims.nyu.edu/~kloeckner/`

GPU-DG Article

AK, T. Warburton, J. Bridge, J.S. Hesthaven, *"Nodal Discontinuous Galerkin Methods on Graphics Processors"*, J. Comp. Phys., 228 (21), 7863–7882.

GPU RTCG

AK, N. Pinto et al. *PyCUDA: GPU Run-Time Code Generation for High-Performance Computing*, submitted.

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

Image Credits

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