PyCUDA: Even Simpler GPU Programming with Python

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

Nvidia GTC · September 22, 2010

Thanks

- Jan Hesthaven (Brown)
- Tim Warburton (Rice)
- Leslie Greengard (NYU)
- PyCUDA contributors
- PyOpenCL contributors
- Nvidia Corporation

Outline

Outline

Whetting your appetite

```python
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

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

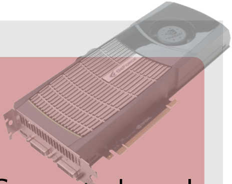
[This is `examples/demo.py` in the PyCUDA distribution.]

Whetting your appetite

```
mod = cuda.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 = cuda.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

Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough
- Python + CUDA = **PyCUDA**
- Python + OpenCL = **PyOpenCL**

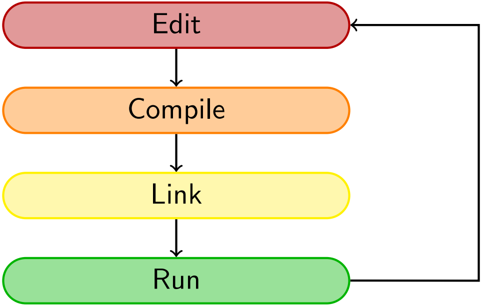
Scripting: Python

One example of a scripting language: Python

- Mature
- Large and active community
- Emphasizes readability
- Written in widely-portable C
- A 'multi-paradigm' language
- Rich ecosystem of sci-comp related software

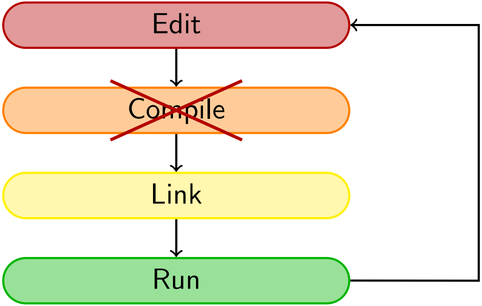
Scripting: Interpreted, not Compiled

Program creation workflow:



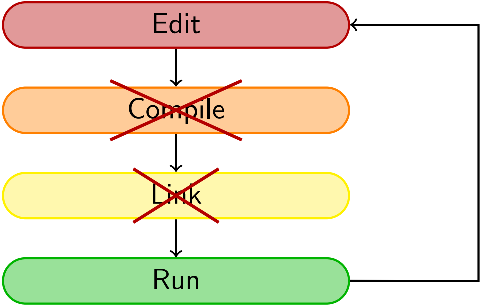
Scripting: Interpreted, not Compiled

Program creation workflow:

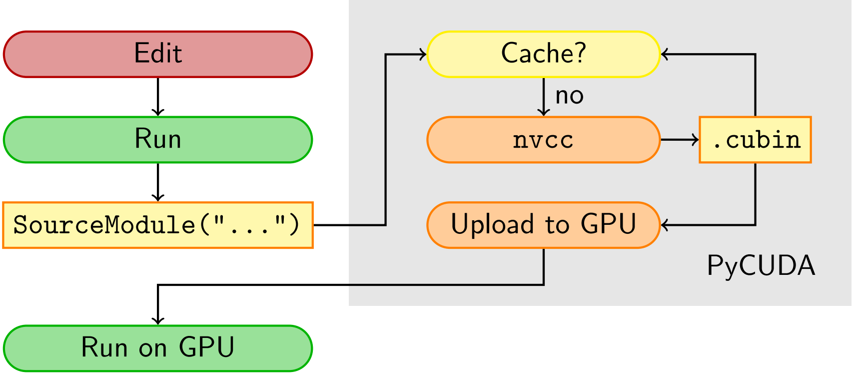


Scripting: Interpreted, not Compiled

Program creation workflow:

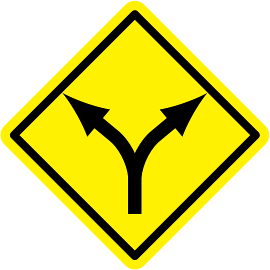


PyCUDA: Workflow

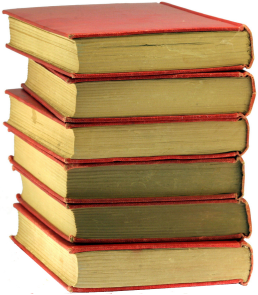


How are High-Performance Codes constructed?

- "Traditional" Construction of High-Performance Codes:
  - C/C++/Fortran
  - Libraries
- "Alternative" Construction of High-Performance Codes:
  - Scripting for 'brains'
  - GPUs for 'inner loops'
- Play to the strengths of each programming environment.



PyCUDA Philosophy



- Provide complete access
- Automatically manage resources
- Provide abstractions
- Check for and report errors automatically
- Full documentation
- Integrate tightly with `numpy`

What's this "`numpy`", anyway?

Numpy: package for large, multi-dimensional arrays.

- Vectors, Matrices, . . .
- `A+B`, `sin(A)`, `dot(A,B)`
- `la.solve(A, b)`, `la.eig(A)`
- `cube[:, :, n-k:n+k]`, `cube+5`

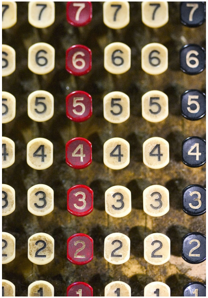
All much faster than functional equivalents in Python.

"Python's MATLAB":
Basis for SciPy, plotting, . . .

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.



Whetting your appetite, Part II

```python
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: Elementwise expressions

Avoiding extra store-fetch cycles for elementwise math:

```
from pycuda.curandom import rand as curand
a_gpu = curand((50,))
b_gpu = curand((50,))

from pycuda.elementwise import ElementwiseKernel
lin_comb = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *z",
        "z[i] = a*x[i] + b*y[i]")

c_gpu = gpuarray.empty_like(a_gpu)
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)

assert la.norm((c_gpu - (5*a_gpu+6*b_gpu)).get()) < 1e-5
```

gpuarray: Reduction made easy

Example: A scalar product calculation

```
from pycuda.reduction import ReductionKernel
dot = ReductionKernel(dtype_out=numpy.float32, neutral="0",
        reduce_expr="a+b", map_expr="x[i]*y[i]",
        arguments="const float *x, const float *y")

from pycuda.curandom import rand as curand
x = curand((1000*1000), dtype=numpy.float32)
y = curand((1000*1000), dtype=numpy.float32)

x_dot_y = dot(x, y).get()
x_dot_y_cpu = numpy.dot(x.get(), y.get())
```

PyCUDA: Vital Information

- `http://mathema.tician.de/software/pycuda`
- Complete documentation
- MIT License
  (no warranty, free for all use)
- Requires: numpy, Python 2.4+
  (Win/OS X/Linux)
- Support via mailing list

Outline

OpenCL's perception problem

OpenCL does not presently get the credit it deserves.

- Single abstraction works well for GPUs, CPUs
- Vendor-independence
- Compute Dependency DAG
- A JIT C compiler baked into a library

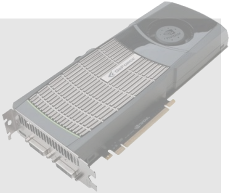
Introducing... PyOpenCL

- PyOpenCL is “PyCUDA for OpenCL”
- Complete, mature API wrapper
- Has: Arrays, elementwise operations, RNG, ...
- Near feature parity with PyCUDA
- Tested on all available Implementations, OSs
- `http://mathema.tician.de/software/pyopencl`

OpenCL

Introducing... PyOpenCL

Same flavor, different recipe:



```
import pyopencl as cl, numpy

a = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a_buf = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_buf, a)

prg = cl.Program(ctx, """
    __kernel void twice(__global float *a)
    {
      int gid = get_global_id(0);
      a[gid] *= 2;
    }""").build()

prg.twice(queue, a.shape, None, a_buf).wait()
```

Outline

Step 1: Download

Hot off the presses:

- PyCUDA 0.94.1
- PyOpenCL 0.92

All the goodies from this talk, plus

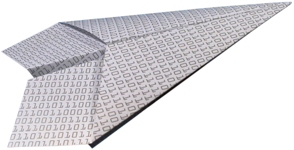
- Supports all new features in CUDA 3.0, 3.1, 3.2rc, OpenCL 1.1
- Allows `printf()` (see example in Wiki)

New stuff shows up in git very quickly.
Still needed: better release schedule.

Step 2: Installation

- PyCUDA and PyOpenCL no longer depend on Boost C++
- Eliminates major install obstacle
- Easier to depend on PyCUDA and PyOpenCL
- `easy_install pyopencl` works on Macs out of the box
- Boost is still there–just not user-visible by default.

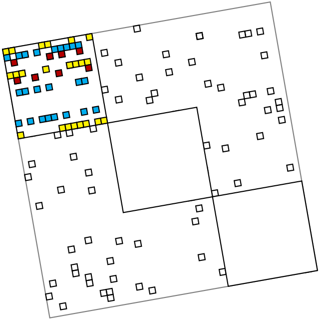
Step 3: Usage



- Complex numbers
  - ...in `GPUArray`
  - ...in user code (`pycuda-complex.hpp`)
- If/then/else for `GPUArrays`
- Support for custom device pointers
- Smarter device picking/context creation
- PyFFT: FFT for PyOpenCL and PyCUDA
- scikits.cuda: CUFFT, CUBLAS, CULA

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



Step 4: Debugging

New in 0.94.1: Support for CUDA gdb:

```
$ cuda-gdb --args python -m
pycuda.debug demo.py
```

Automatically:

- Sets Compiler flags
- Retains source code
- Disables compiler cache

Outline

GPU Programming: Implementation Choices

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

GPU Programming: Implementation Choices

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

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

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

Metaprogramming

In GPU scripting, GPU code does *not* need to be a compile-time constant.

Metaprogramming

In GPU scripting,
GPU code does
not need to be
a compile-time
constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

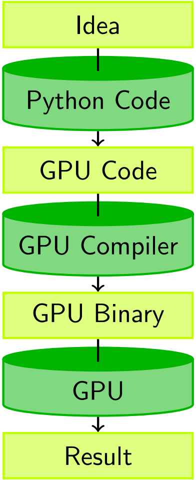
Metaprogramming

Idea

In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

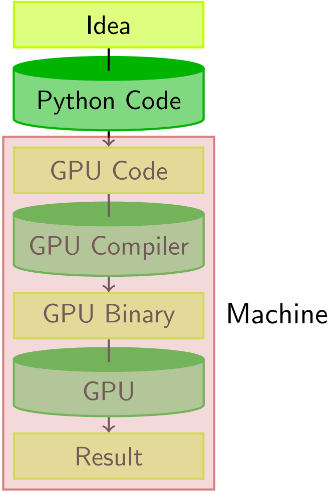
Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

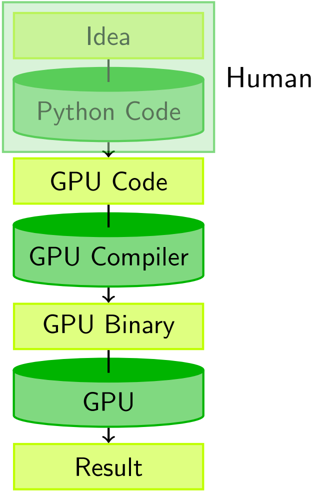
Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

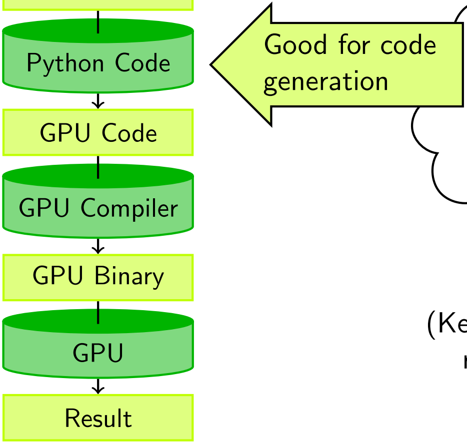
Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

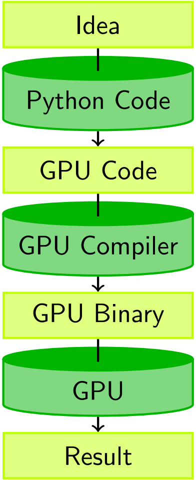
Metaprogramming

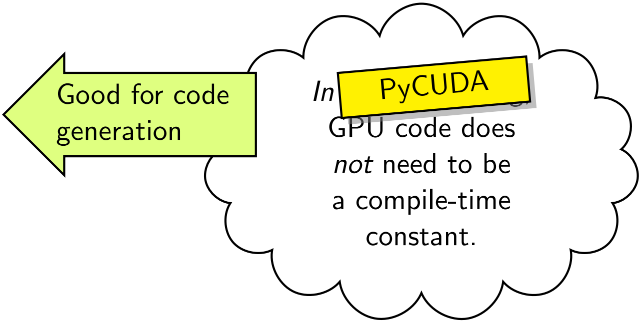


In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

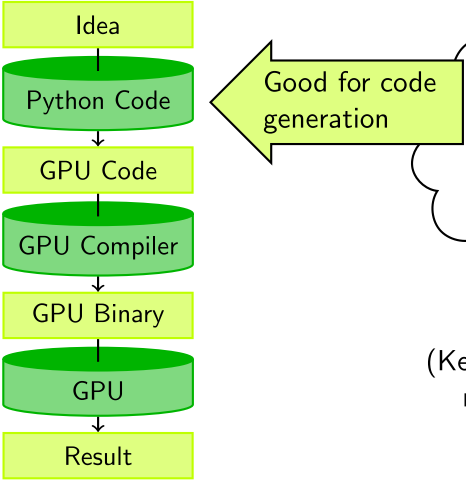
Metaprogramming





(Key: Code is data–it *wants* to be reasoned about at run time)

Metaprogramming



In PyOpenCL GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

Machine-generated Code

Why machine-generate code?

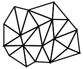
- Automated Tuning (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables ($\rightarrow$ register pressure)
- Loop Unrolling

RTCG via Templates

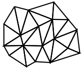
```
from jinja2 import Template
tpl = Template("""
    __global__ void twice({{ type_name }} *tgt)
    {
      int idx = threadIdx.x +
        {{ thread_block_size }} * {{block_size}}
        * blockIdx.x;

      {% for i in range( block_size ) %}
          {% set offset = i* thread_block_size %}
          tgt[idx + {{ offset }}] *= 2;
      {% endfor %}
    }""")

rendered_tpl = tpl.render(
    type_name="float", block_size=block_size,
     thread_block_size=thread_block_size)

smod = SourceModule(rendered_tpl)
```

Outline

Discontinuous Galerkin Method

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

Discontinuous Galerkin Method

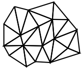
Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$. 

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

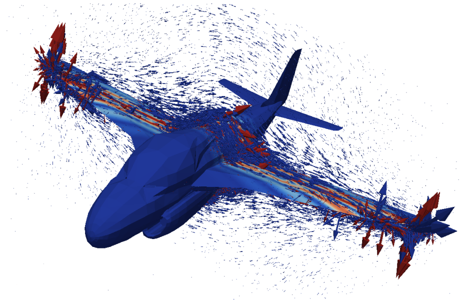
Example

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

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

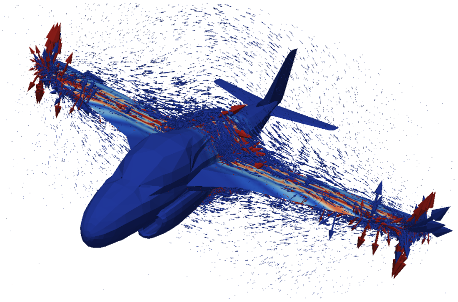
$$\nabla \cdot E = \frac{\rho}{\varepsilon}, \qquad \nabla \cdot H = 0.$$

GPU DG Showcase

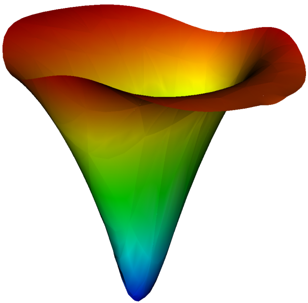


Eletromagnetism

GPU DG Showcase

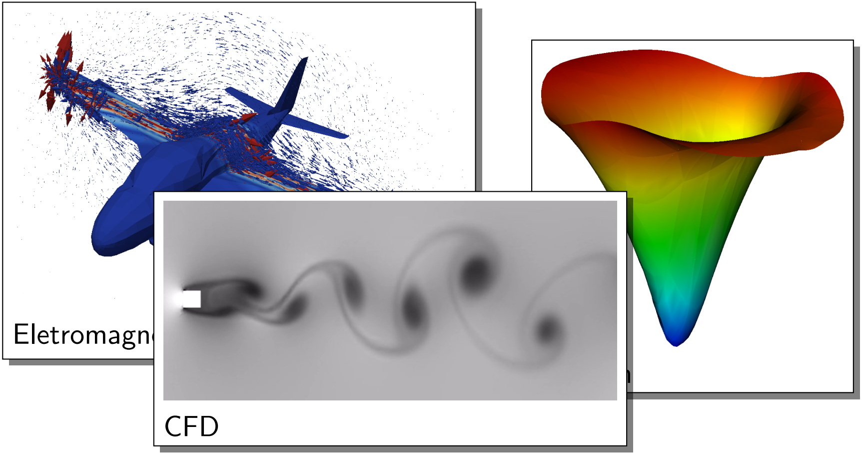


Eletromagnetism

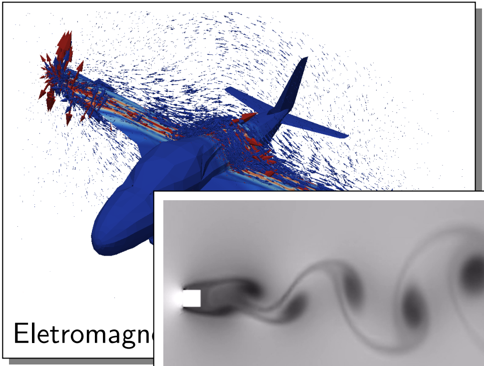


Poisson

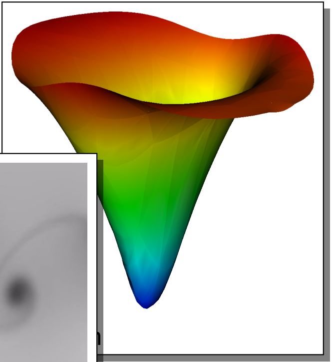
GPU DG Showcase

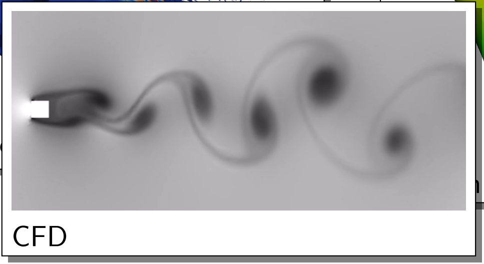


GPU DG Showcase



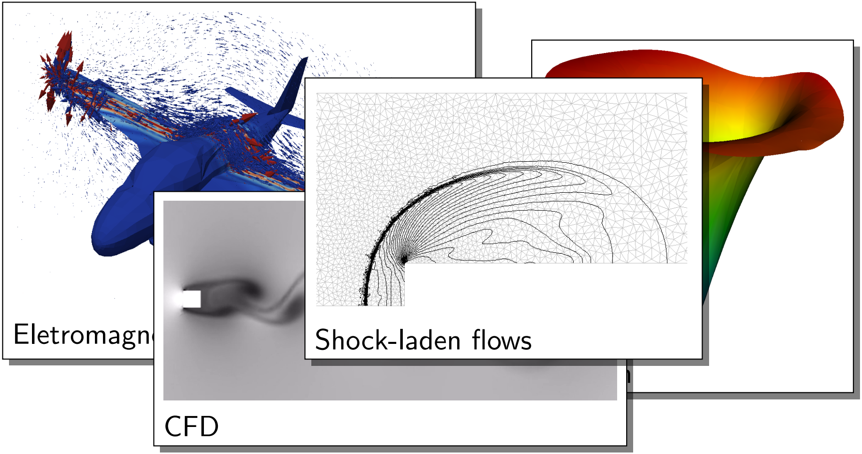
Eletromagn



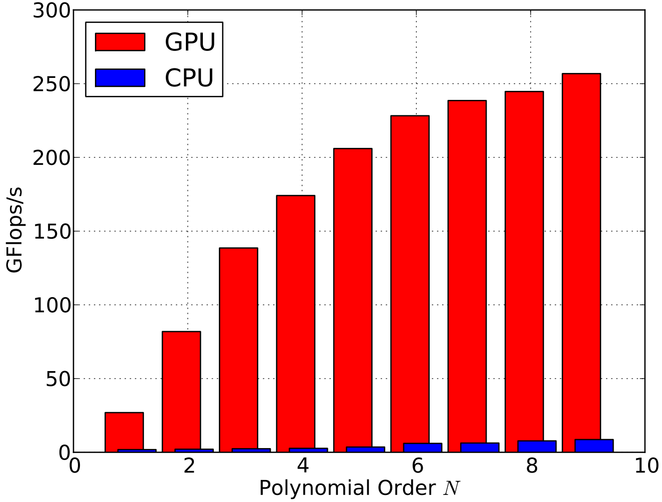


CFD

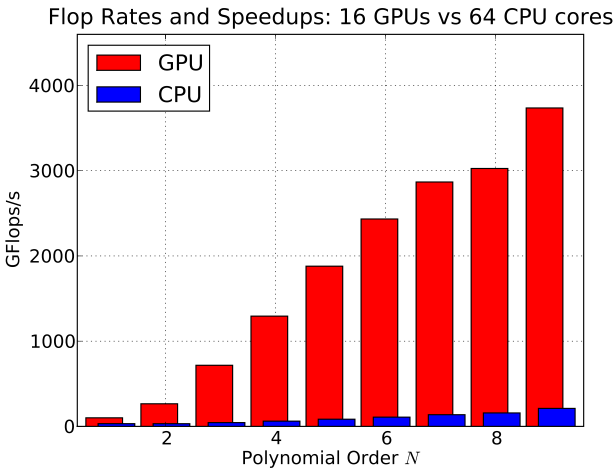
GPU DG Showcase



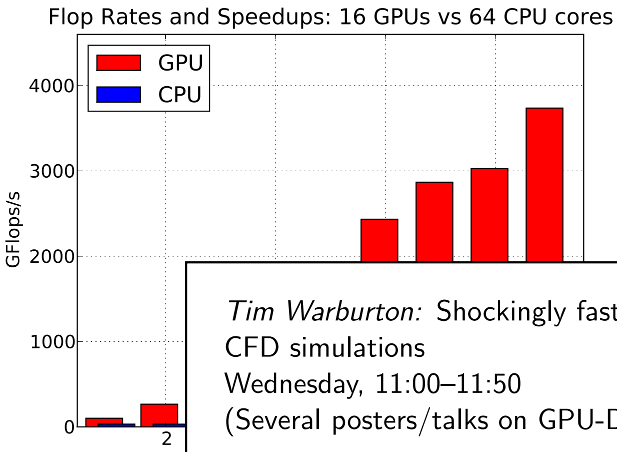
GPU-DG: Performance on GTX280



16 T10s vs. $64 = 8 \times 2 \times 4$ Xeon E5472



16 T10s vs. $64 = 8 \times 2 \times 4$ Xeon E5472



Tim Warburton: Shockingly fast and accurate CFD simulations
Wednesday, 11:00–11:50
(Several posters/talks on GPU-DG at GTC.)

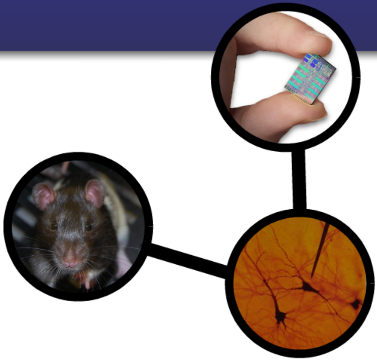
Computational Visual Neuroscience

A High-Throughput Approach to Discovering Good Forms of Visual Representation

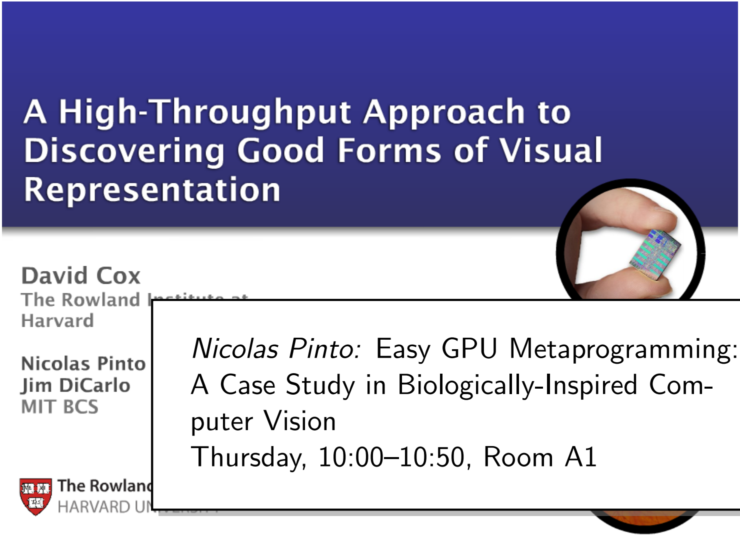
David Cox
The Rowland Institute at Harvard

Nicolas Pinto
Jim DiCarlo
MIT BCS

The Rowland Institute at Harvard
HARVARD UNIVERSITY



Computational Visual Neuroscience



Nicolas Pinto: Easy GPU Metaprogramming: A Case Study in Biologically-Inspired Computer Vision
Thursday, 10:00–10:50, Room A1

Copperhead

```
from copperhead import *
import numpy as np

@cu
def axpy(a, x, y):
    return [a * xi + yi for xi, yi in zip(x, y)]

x = np.arange(100, dtype=np.float64)
y = np.arange(100, dtype=np.float64)

with places.gpu0:
    gpu = axpy(2.0, x, y)

with places.here:
    cpu = axpy(2.0, x, y)
```

Copperhead

```
from copperhead import *
import numpy as np

@cu
def axpy(a, x, y):
  return [a * xi + yi for xi, yi in zip(x, y)]

x = np.arange(100, dtype=np.float64)
y = np.arange(100, dtype=np.float64)

with places.gpu(
  gpu = axpy(2.(

with places.here
  cpu = axpy(2.(
```

Bryan Catanzaro: Copperhead: Data-Parallel Python for the GPU
Wednesday, 15:00–15:50 (next slot!), Room N

Conclusions

- Fun time to be in computational science
- Even more fun with Python and Py{CUDA,OpenCL}
  - With no compromise in performance
- GPUs and scripting work well together
  - Enable Metaprogramming
- The "Right" way to develop computational codes
  - Bake all runtime-available knowledge into code

Where to from here?

More at...

→ `http://mathema.tician.de/`

CUDA-DG

AK, T. Warburton, J. Bridge, J.S. Hesthaven, *"Nodal Discontinuous Galerkin Methods on Graphics Processors"*, J. Comp. Phys., 2009.

GPU RTCG

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

Questions?

?

Thank you for your attention!

`http://mathema.tician.de/`

image credits

Image Credits

- Fermi GPU: Nvidia Corp.
- C870 GPU: Nvidia Corp.
- Python logo: python.org
- Old Books: flickr.com/ppdigital (cc)
- Adding Machine: flickr.com/thomashawk (cc)
- Floppy disk: flickr.com/ethanhein (cc)
- Thumbs up: sxc.hu/thiagofest
- OpenCL logo: Ars Technica/Apple Corp.
- Newspaper: sxc.hu/brandcore
- Boost C++ logo: The Boost C++ project
- ?/! Marks: sxc.hu/svilen001
- Machine: flickr.com/13521837@N00 (cc)