GPU Metaprogramming using PyCUDA: Methods & Applications

Andreas Klöckner

Division of Applied Mathematics
Brown University

GPU @ BU · November 12, 2009

Thanks

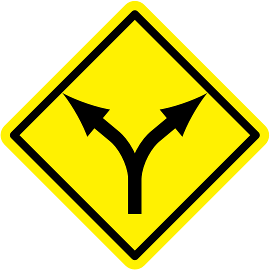
- Tim Warburton (Rice)
- Jan Hesthaven (Brown)
- Nicolas Pinto (MIT)
- PyCUDA contributors
- Nvidia Corporation

Outline

Outline

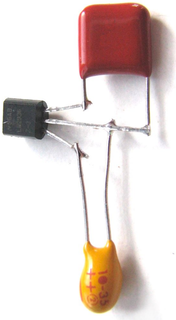
How are High-Performance Codes constructed?

- "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.



BROWN

Scripting: Means

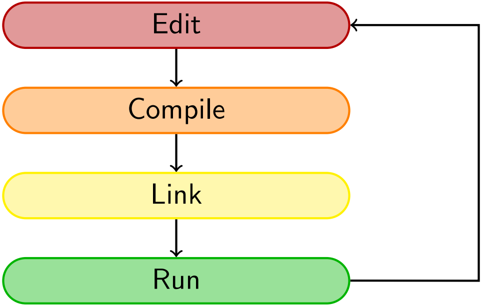


A scripting language...

- is discoverable and interactive.
- has comprehensive built-in functionality.
- manages resources automatically.
- is dynamically typed.
- works well for "gluing" lower-level blocks together.

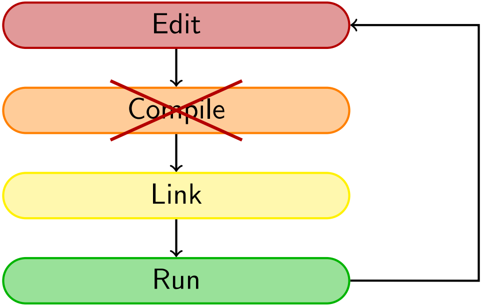
Scripting: Interpreted, not Compiled

Program creation workflow:



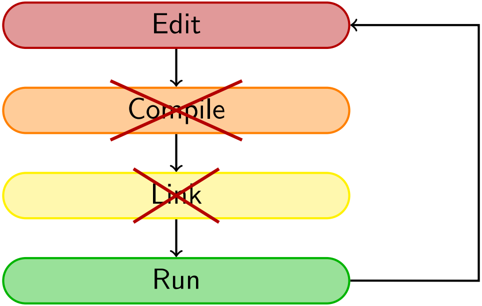
Scripting: Interpreted, not Compiled

Program creation workflow:



Scripting: Interpreted, not Compiled

Program creation workflow:



Why do Scripting for GPUs?

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

→ complement each other

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 (∼1000/sec)
  - Scripting fast enough

Why do Scripting for GPUs?

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

  → complement each other
- CPU: largely restricted to control tasks (∼1000/sec)
  - Scripting fast enough
- Python + CUDA = **PyCUDA**

Outline

BROWN

Whetting your appetite

```
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

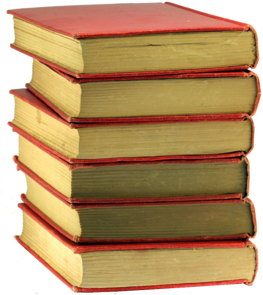
Whetting your appetite, Part II

Did somebody say "Abstraction is good"?

Whetting your appetite, Part II

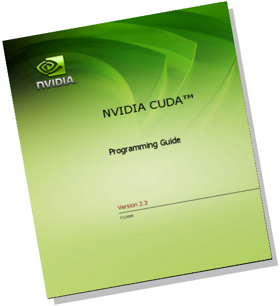
```
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

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
```

PyCUDA Philosophy



- Provide complete access
- Automatically manage resources
- Provide abstractions
- Allow interactive use
- Check for and report errors automatically
- Integrate tightly with `numpy`

PyCUDA: Completeness



PyCUDA exposes *all* of CUDA.

For example:

- Arrays and Textures
- Pagelocked host memory
- Memory transfers (asynchronous, structured)
- Streams and Events
- Device queries
- GL Interop

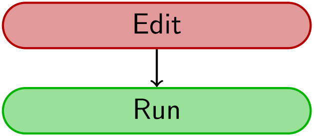
BROWN

PyCUDA: Completeness

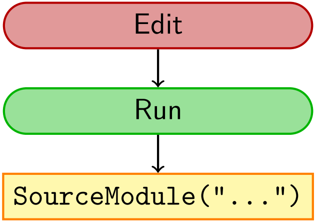
PyCUDA supports every OS that CUDA supports.

- Linux
- Windows
- OS X

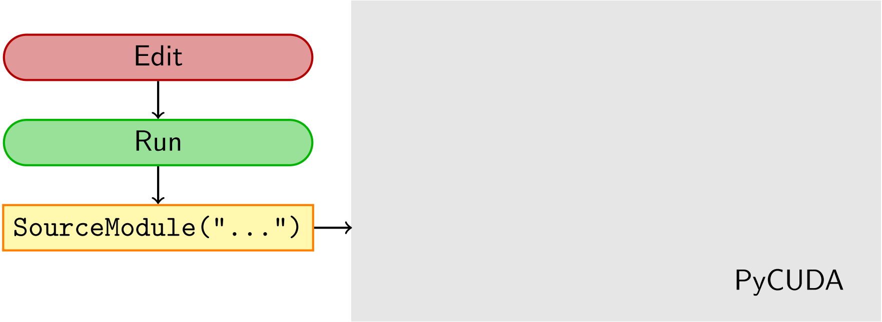
PyCUDA: Workflow



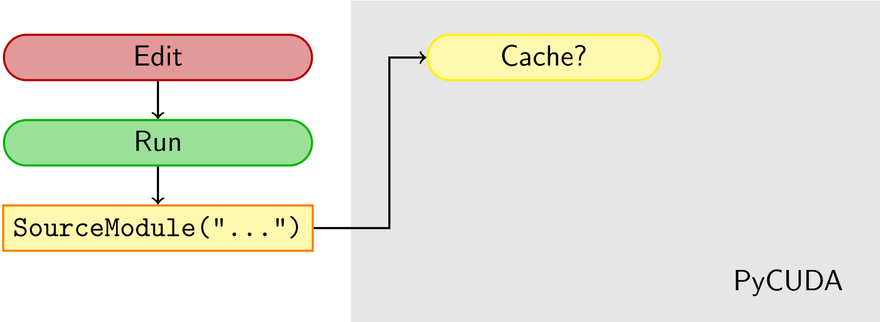
PyCUDA: Workflow



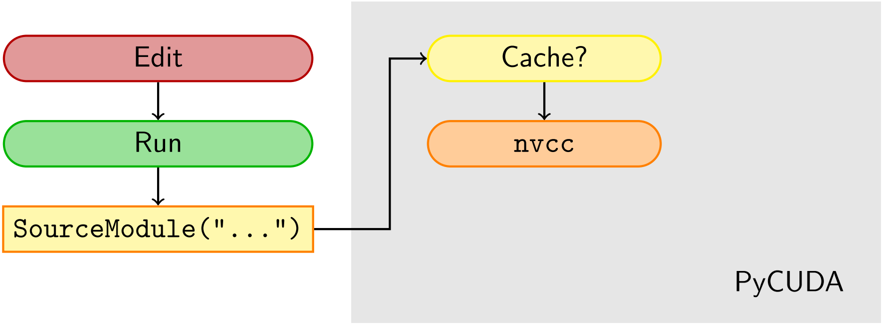
PyCUDA: Workflow



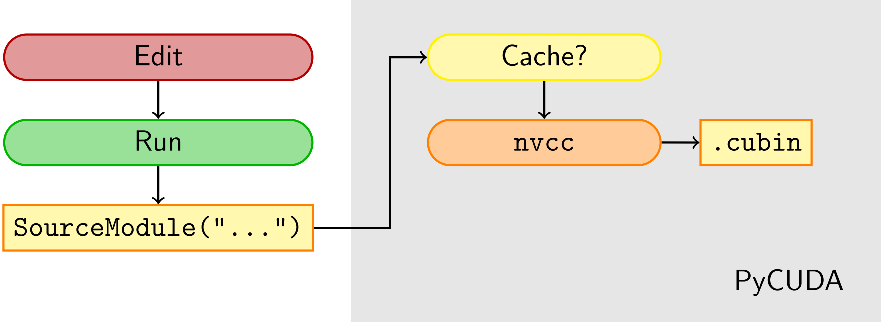
PyCUDA: Workflow



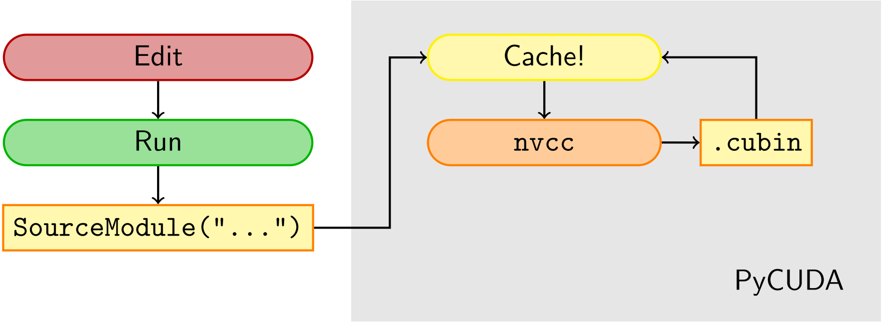
PyCUDA: Workflow



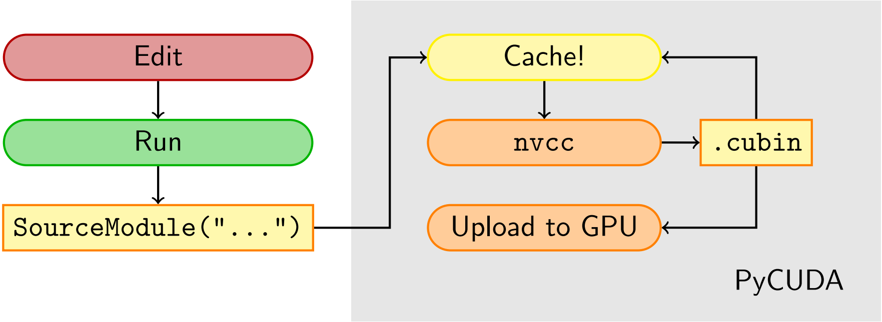
PyCUDA: Workflow



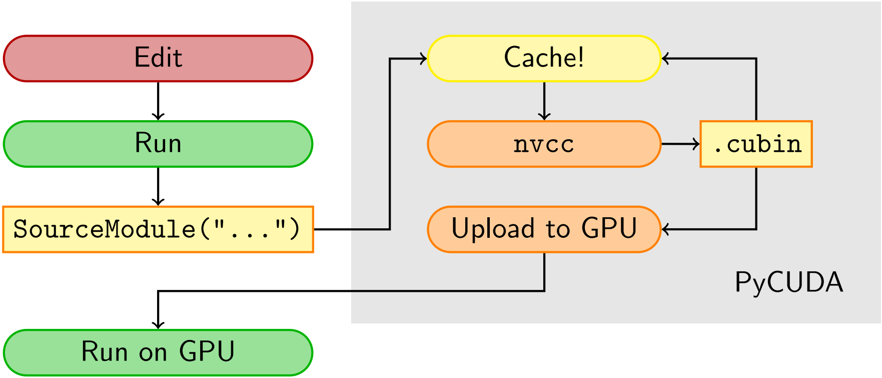
PyCUDA: Workflow



PyCUDA: Workflow



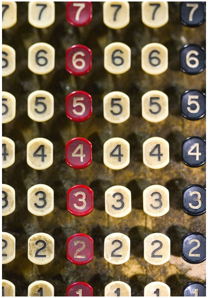
PyCUDA: Workflow



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.



BROWN

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
```

PyCUDA: Vital Information

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

Outline

1 Why GPU Scripting?

2 Scripting CUDA

3 GPU Run-Time Code Generation
- Programs that write Programs

4 DG on GPUs

5 Perspectives

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)

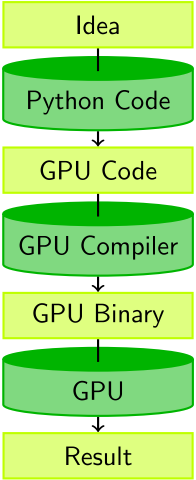
Metaprogramming

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)

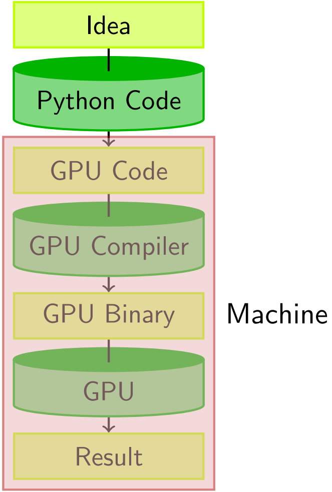
Metaprogramming



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)

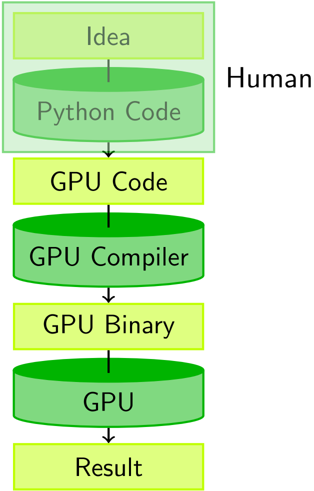
Metaprogramming



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)

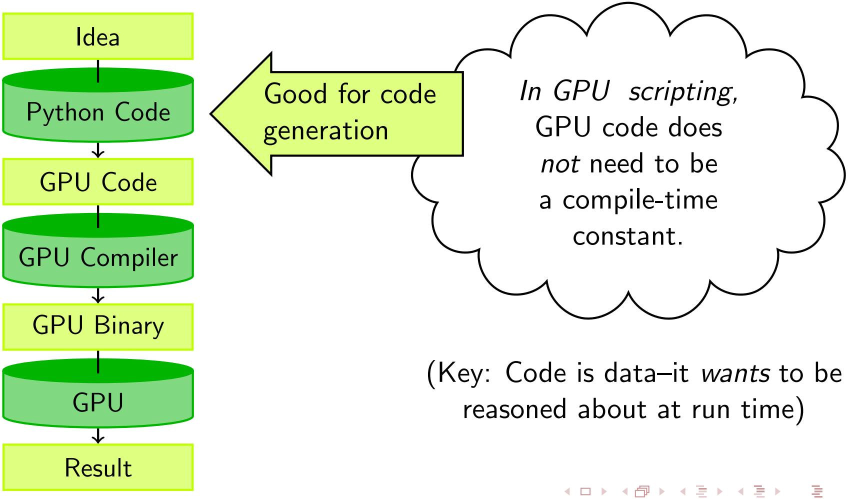
Metaprogramming



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

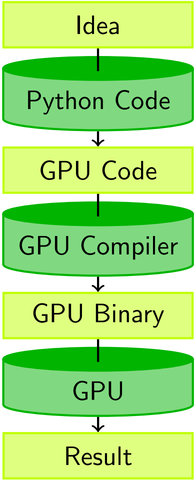
(Key: Code is data–it *wants* to be reasoned about at run time)

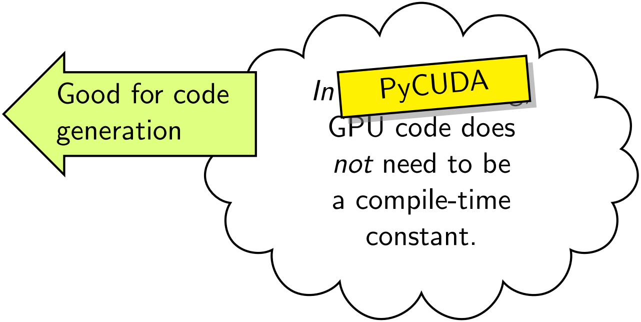
Metaprogramming



(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)

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)
```

RTCG via AST Generation

```
from codepy.cgen import *
from codepy.cgen.cuda import CudaGlobal

mod = Module([
    FunctionBody(
        CudaGlobal(FunctionDeclaration(
            Value("void", "twice"),
             arg_decls=[Pointer(POD(dtype, "tgt"))])),
        Block([
             Initializer (POD(numpy.int32, "idx"),
                "threadIdx.x + %d*blockIdx.x"
                % (thread_block_size * block_size)),
            ]+[
            Assign("tgt[idx+%d]" % (o*thread_block_size),
                "2*tgt[idx+%d]" % (o*thread_block_size))
            for o in range(block_size)]))])

smod = SourceModule(mod)
```

Outline

1 Why GPU Scripting?

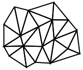
2 Scripting CUDA

3 GPU Run-Time Code Generation

4 DG on GPUs
- Introduction
- DG and Metaprogramming
- Results

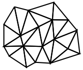
5 Perspectives

Discontinuous Galerkin Method

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

Discontinuous Galerkin Method

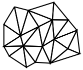
Let $\Omega := \bigcup_i \mathrm{D}_k \subset \mathbb{R}^d$.



Goal

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

Discontinuous Galerkin Method

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

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:

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

Integrate by parts again, subsitute in basis functions, introduce elementwise differentiation and "lifting" matrices D, L:

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

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

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

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

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

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

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

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

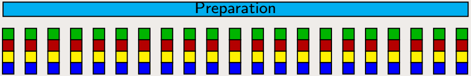
$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([\![H]\!] - \alpha \hat{n} \times [\![E]\!])]$$

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);
```

Loop Slicing on the GPU: A Pattern

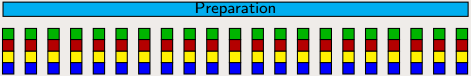
Setting: N independent work units + preparation



Question: How should one assign work units to threads?

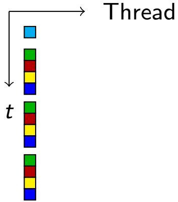
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation



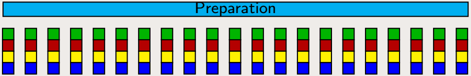
Question: How should one assign work units to threads?

w_s: in sequence



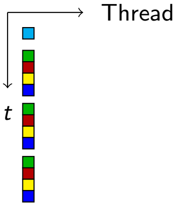
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation

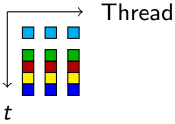


Question: How should one assign work units to threads?

w_s: in sequence

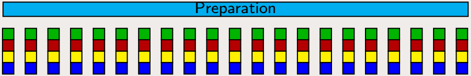


w_p: in parallel



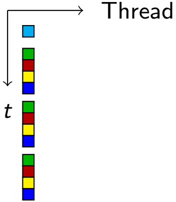
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation

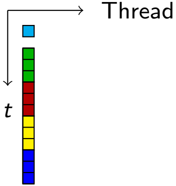


Question: How should one assign work units to threads?

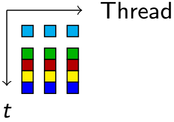
w_s: in sequence



w_i: “inline-parallel”

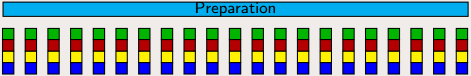


w_p: in parallel



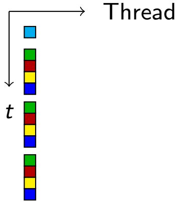
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation



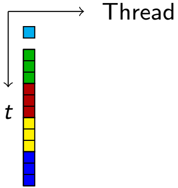
Question: How should one assign work units to threads?

w_s: in sequence

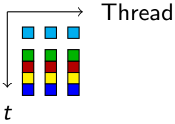


(amortize preparation)

w_i: "inline-parallel"

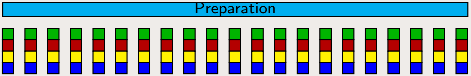


w_p: in parallel



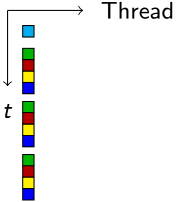
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation

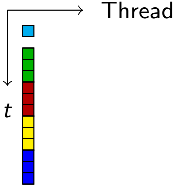


Question: How should one assign work units to threads?

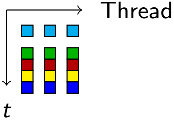
w_s: in sequence



w_i: "inline-parallel"

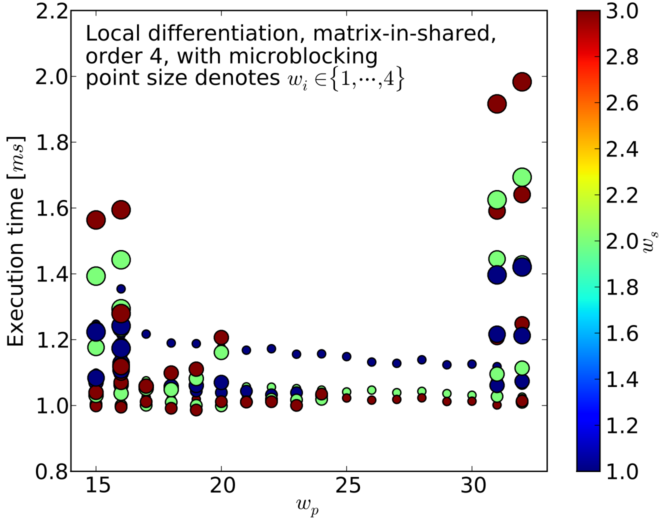


w_p: in parallel

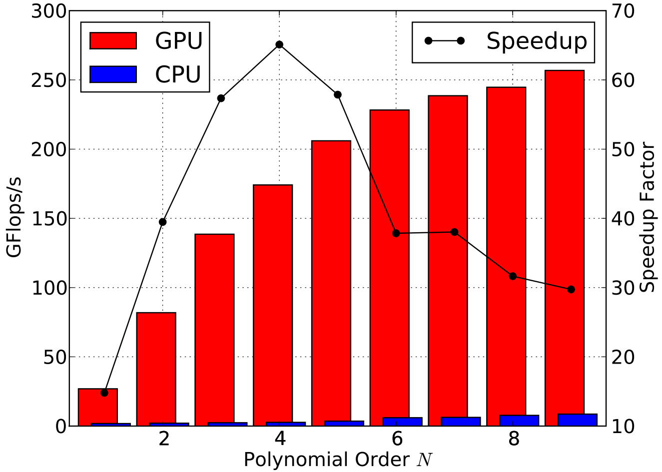


(amortize preparation) (exploit register space)

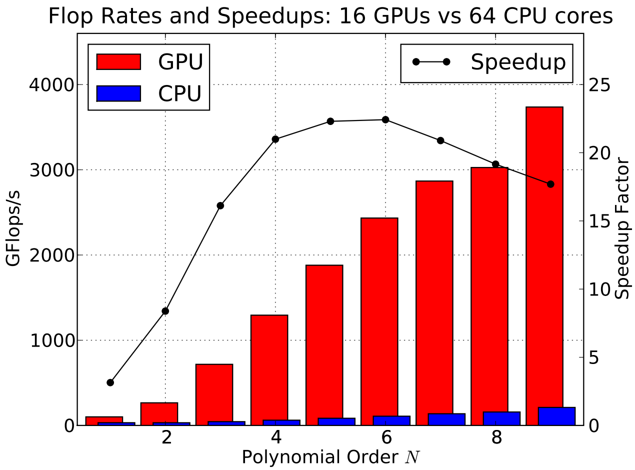
Loop Slicing for Differentiation



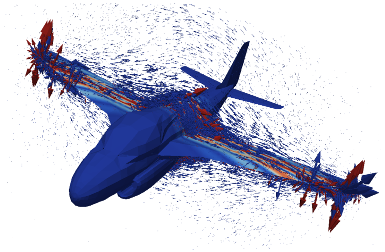
Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400



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

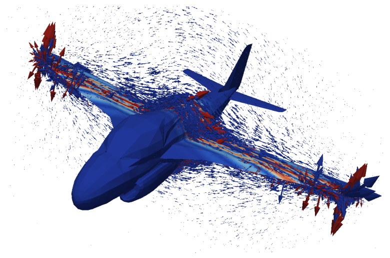


GPU DG Showcase

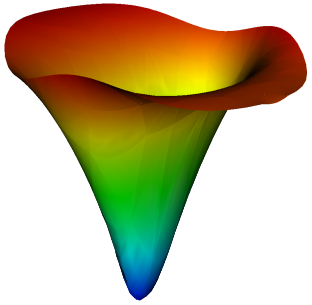


Eletromagnetism

GPU DG Showcase

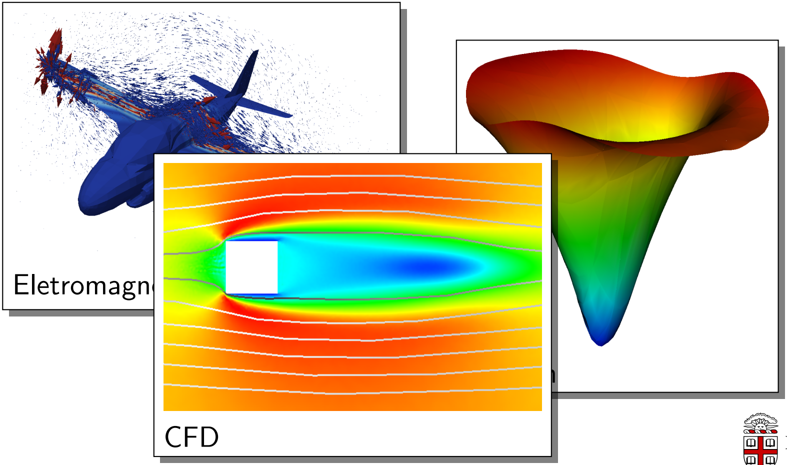


Eletromagnetism

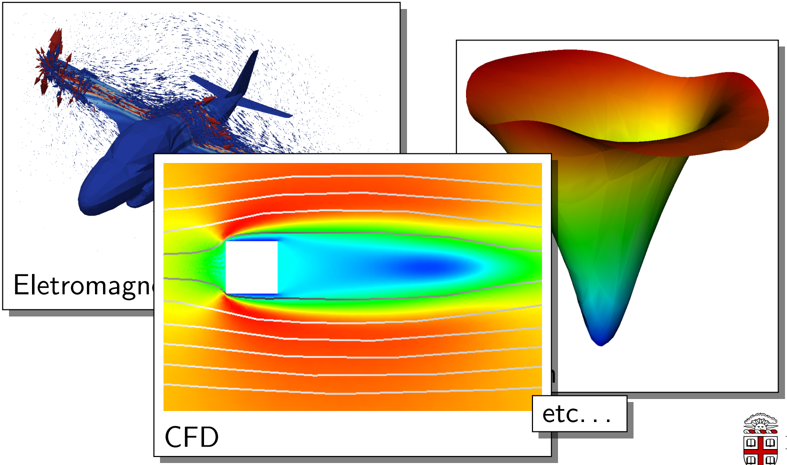


Poisson

GPU DG Showcase



GPU DG Showcase



Outline

Introducing... PyOpenCL

- PyOpenCL is "PyCUDA for OpenCL"
- Complete, mature API wrapper
- Features like PyCUDA: not yet
- Tested on all available Implementations, OSs
- `http://mathema.tician.de/software/pyopencl`

OpenCL

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

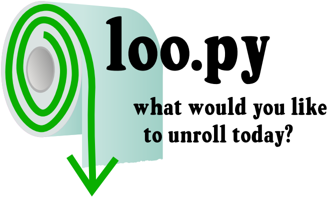
GPU programming can be time-consuming, unintuitive and error-prone.

- 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:

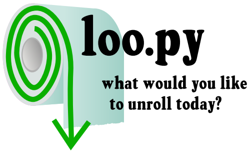
```
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel([
    LoopDimension("i", n),
    LoopDimension("j", n),
    LoopDimension("k", n),
    ], [
    (c[i+n*j], a[i+n*k]*b[k+n*j])
    ])

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



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

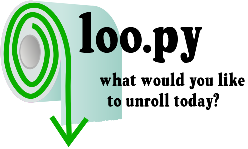
Loo.py Status

- 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



BROWN

Conclusions

- Fun time to be in computational science

Conclusions

- Fun time to be in computational science
- Use Python and PyCUDA to have even more fun :-)
  - With no compromise in performance

Conclusions

- Fun time to be in computational science
- Use Python and PyCUDA to have even more fun :-)
  - With no compromise in performance
- GPUs and scripting work well together
  - Enable Metaprogramming

Conclusions

- Fun time to be in computational science
- Use Python and PyCUDA to have even more fun :-)
  - With no compromise in performance
- GPUs and scripting work well together
  - Enable Metaprogramming
- Further work in GPU-DG:
  - Other equations (Euler, Navier-Stokes)
  - Curvilinear Elements
  - Local Time Stepping

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

- Circuitry: flickr.com/oskay (cc)
- C870 GPU: Nvidia Corp.
- Old Books: flickr.com/ppdigital (cc)
- OpenCL logo: Ars Technica/Apple Corp.
- OS Platforms: flickr.com/aOliN.Tk
- Adding Machine: flickr.com/thomashawk (cc)
- Floppy disk: flickr.com/ethanhein (cc)
- Machine: flickr.com/13521837@N00 (cc)
- OpenCL logo: Ars Technica/Apple Corp.

BROWN