GPU Metaprogramming using PyCUDA: Methods & Applications

Andreas Klöckner
Division of Applied Mathematics
Brown University

Nvidia GTC · October 2, 2009
Thanks

- Tim Warburton (Rice)
- Jan Hesthaven (Brown)
- Nicolas Pinto (MIT)
- Hendrik Riedmann (Stuttgart/Brown)
- PyCUDA contributors
- Nvidia Corporation
Outline

1. Why GPU Scripting?
2. Scripting CUDA
3. GPU Run-Time Code Generation
4. DG on GPUs
5. Perspectives
Outline

1. Why GPU Scripting?
   - Combining two Strong Tools

2. Scripting CUDA

3. GPU Run-Time Code Generation

4. DG on GPUs

5. Perspectives
Why GPU Scripting?

Scripting CUDA

GPU RTCG

DG on GPUs

Perspectives

Combining two Strong Tools

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.
Why GPU Scripting?

Combining two Strong Tools

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:

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

Program creation workflow:

1. Edit
2. Compile (crossed out)
3. Link
4. Run
Scripting: Interpreted, not Compiled

Program creation workflow:

1. Edit
2. Compile → X
3. Link → X
4. Run
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
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
  → 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

1. Why GPU Scripting?

2. Scripting CUDA
   - Whetting your Appetite
   - Under the Hood
   - Fun with GPU Arrays

3. GPU Run-Time Code Generation

4. DG on GPUs

5. Perspectives
Whetting your appetite

1. `import pycuda.driver as cuda`
2. `import pycuda.autoinit`
3. `import numpy`
4. 
5. `a = numpy.random.randn(4,4).astype(numpy.float32)`
6. `a_gpu = cuda.mem_alloc(a.nbytes)`
7. `cuda.memcpy_htod(a_gpu, a)`

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

```python
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
```
Why GPU Scripting?

WHETTING YOUR APPETITE

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

Andreas Klöckner

Applied Math · Brown University

GPU Metaprogramming using PyCUDA: Methods & Applications
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
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
PyCUDA: Completeness

PyCUDA supports every OS that CUDA supports.

- Linux
- Windows
- OS X
Why GPU Scripting?

Whetting your Appetite

PyCUDA: Documentation

Welcome to PyCuda’s documentation!

PyCuda gives you easy, Pythonic access to Nvidia’s CUDA parallel computation API. Several wrappers of the CUDA API already exist—so why the need for PyCuda?

- Object cleanup tied to lifetime of objects. This idiom, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code. PyCuda knows about dependencies, too, so (for example) it won’t detach from a context before all memory allocated in it is also freed.
- Convenience. Abstractions like `pycuda.driver`, `SourceModule` and `pycuda.gparray`, `GPUArray` make CUDA programming even more convenient than with Nvidia’s C-based runtime.
- Completeness. PyCuda puts the full power of CUDA’s driver API at your disposal, if you wish.
- Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions.
- Speed. PyCuda’s base layer is written in C++, so all the niceties above are virtually free.
- Helpful Documentation. You’re looking at it. :)"

Here’s an example, to give you an impression:

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

mod = drv.SourceModule('''
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
''')

multiply_them = mod.get_function('multiply_them')

a = numpy.random.randn(100).astype(numpy.float32)
b = numpy.random.randn(100).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(a, b, dest)
```
PyCUDA: Workflow

Edit

Run
PyCUDA: Workflow

1. Edit
2. Run

```
SourceModule("...")
```
PyCUDA: Workflow

- **Edit**
- **Run**
- `SourceModule("...")`
PyCUDA: Workflow

Edit -> Run -> SourceModule("...")

Cache?

PyCUDA
PyCUDA: Workflow

1. Edit
2. Run
3. SourceModule("...")
4. Cache?
5. nvcc
6. PyCUDA
PyCUDA: Workflow

- **Edit**
- **Run**
- **SourceModule("...")**
- **Cache?**
  - **nvcc**
  - **.cubin**

---

Andreas Klöckner

GPU Metaprogramming using PyCUDA: Methods & Applications
PyCUDA: Workflow

Edit
Run
SourceModule("...")

Cache!
nvcc
.cubin

PyCUDA
PyCUDA: Workflow

- **Edit**
- **Run**

SourceModule("...") → Cache! → .cubin

nvcc

Upload to GPU

PyCUDA
PyCUDA: Workflow

1. **Edit**
2. **Run**
3. **SourceModule("...")**
4. **Run on GPU**
5. **Cache!**
6. **nvcc**
7. **.cubin**
8. **Upload to GPU**
9. **PyCUDA**
Automatic Cleanup

- Reachable objects (memory, streams, ...) are never destroyed.
- Once unreachable, released at an unspecified future time.
- Scarce resources (memory) can be explicitly freed. (`obj.free()`)  
- Correctly deals with multiple contexts and dependencies.
**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.
Avoiding extra store-fetch cycles for elementwise math:

```python
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
```
Example: A scalar product calculation

```python
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())
```
Sparse Matrix-Vector on the GPU

- In development version: Sparse matrix-vector multiplication
- Uses “packeted format” by Garland and Bell (also includes parts of their code)
- Integrates with scipy.sparse.
- Optimized conjugate-gradients solver included
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
In GPU scripting, GPU code does not need to be a compile-time constant.
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)
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)
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)
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)
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)
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**

- Python Code
- GPU Code
- GPU Compiler
- GPU Binary
- GPU

**Good for code generation**

**In PyCUDA**

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

(Key: Code is data—it *wants* to be reasoned about at run time)
Why machine-generate code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables (→ register pressure)
- Loop Unrolling
PyCUDA: Support for Metaprogramming

- Access properties of compiled code:
  
  ```c
  func.{num_regs,shared_size_bytes,local_size_bytes}
  ```

- Exact GPU timing via events

- Can calculate hardware-dependent MP occupancy

- codepy:
  - Build C syntax trees from Python
  - Generates readable, indented C

- Or use a templating engine (many available)
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 %}
  }"")

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

smod = SourceModule(renderedTpl)
Why GPU Scripting?

Programs that write Programs

RTC G via AST Generation

```python
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)
```

Andreas Klöckner

Applied Math · Brown University

GPU Metaprogramming using PyCUDA: Methods & Applications
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 $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$. 

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

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

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

$$\nabla \cdot E = \frac{\rho}{\varepsilon},$$

$$\nabla \cdot H = 0.$$
Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

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

Integrate by parts again, substitute in basis functions, introduce elementwise differentiation and “lifting” matrices \( D \), \( L \):

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

For straight-sided simplicial elements:
Reduce \( D^{\partial_{\nu}} \) 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

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

Flux term
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

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

```python
flux = 1/2*cross(normal, h.int-h.ext -alpha*cross(normal, e.int-e.ext))
```
**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);
```

Andreas Klöckner
Applied Math  ·  Brown University

GPU Metaprogramming using PyCUDA: Methods & Applications
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

$w_p$: in parallel (amortize preparation) (exploit register space)
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
  - Thread
  - $t$

- $w_i$: “inline-parallel”
  - Thread
  - $t$

- $w_p$: in parallel
  - Thread
  - $t$
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
  - Thread

- $w_i$: “inline-parallel”
  - Thread

- $w_p$: in parallel
  - Thread

(amoortize preparation)
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

Local differentiation, matrix-in-shared, order 4, with microblocking
point size denotes $w_i \in \{1, \ldots, 4\}$

Execution time [ms]

Local differentiation, matrix-in-shared,
order 4, with microblocking
point size denotes $w_i \in \{1, \ldots, 4\}$
Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400

The chart shows the comparison between GPU and CPU performance for Nvidia GTX280 and a single core of Intel Core 2 Duo E8400. The x-axis represents the polynomial order $N$, and the y-axis shows the GFlops/s. The chart includes a line graph indicating the speedup factor, with the speedup increasing significantly at certain polynomial orders.
16 T10s vs. 64 = 8 × 2 × 4 Xeon E5472

**Flop Rates and Speedups: 16 GPUs vs 64 CPU cores**

- **GPU**
- **CPU**

**Polynomial Order** $N$
- 0
- 1000
- 2000
- 3000
- 4000

**GFlops/s**
- 0
- 5
- 10
- 15
- 20
- 25

**Speedup Factor**
- 0
- 5
- 10
- 15
- 20
- 25

---

Andreas Klöckner

Applied Math · Brown University

GPU Metaprogramming using PyCUDA: Methods & Applications
GPU DG Showcase

Eletromagnetism
Eletromagnetism

Poisson

Performance: Double Precision Poisson Solver
Unpreconditioned CG with IP DG on $K = 18068$ elements

GPU DG Showcase

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming using PyCUDA: Methods & Applications
GPU DG Showcase

- Eletromagnetism
- CFD

Graph showing performance of Double Precision Poisson Solver using Unpreconditioned CG with IP DG on $K = 18068$ elements.

Graphs comparing performance on GPU vs. CPU for different polynomial orders $N$.

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming using PyCUDA: Methods & Applications
GPU DG Showcase

Eletromagnetism

CFD

etc...

Performance: Double Precision Poisson Solver
Unpreconditioned CG with IP DG on $K=18068$ elements

GPU

CPU

Speedup

Polynomial order $N$

Iterations/s

Andreas Klöckner

Applied Math · Brown University

GPU Metaprogramming using PyCUDA: Methods & Applications
Outline

1. Why GPU Scripting?
2. Scripting CUDA
3. GPU Run-Time Code Generation
4. DG on GPUs
5. Perspectives
   ■ Conclusions
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
Introducing... PyOpenCL

Same flavor, different recipe:

```python
ctx = cl.create_context_from_type (cl.device_type.ALL)
queue = cl.CommandQueue(ctx)

a = numpy.random.rand(50000).astype(numpy.float32)
a_buf = cl.Buffer(ctx, cl.mem_flags.COPY_HOST_PTR, hostbuf=a)

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

twice_a = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, a_buf, twice_a).wait()
```
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
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

GPU RTCG
Questions?

Thank you for your attention!

http://mathema.tician.de/
Image Credits

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