High-Productivity Supercomputing: Metaprogramming GPUs

Andreas Klöckner
Applied Mathematics, Brown University

January 28, 2009
Thanks

- Jan Hesthaven (Brown)
- Tim Warburton (Rice)
- Nico Gödel (HSU Hamburg)
- Lucas Wilcox (UT Austin)
- Akil Narayan (Brown)
- PyCuda contributors
Outline

1. Scripting Languages
2. Scripting CUDA
3. Metaprogramming CUDA
4. Discontinuous Galerkin on CUDA
Outline

1. Scripting Languages
   - Scripting: what and why?

2. Scripting CUDA

3. Metaprogramming CUDA

4. Discontinuous Galerkin on CUDA
Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
- Encourage experimentation
Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
- Encourage experimentation
- Eliminate sources of error
Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
- Encourage experimentation
- Eliminate sources of error
- Encourage abstraction wherever possible
Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
- Encourage experimentation
- Eliminate sources of error
- Encourage abstraction wherever possible
- Value programmer time over computer time
Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
- Encourage experimentation
- Eliminate sources of error
- Encourage abstraction wherever possible
- Value programmer time over computer time

Think about the tools you use. Use the right tool for the job.
Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
- Encourage experimentation
- Eliminate sources of error
- Encourage abstraction wherever possible
- Value programmer time over computer time

Think about the tools you use. Use the right tool for the job.

How are these goals achieved?
A scripting language...  
- is discoverable and interactive.
Scripting Languages

Scripting: what and why?

Scripting: Means

A scripting language... 

- is discoverable and interactive.
- is interpreted, not compiled.
Scripting: Means

A scripting language. . .

- is discoverable and interactive.
- is interpreted, not compiled.
- has comprehensive built-in functionality.
A scripting language... 

- is discoverable and interactive.
- is interpreted, not compiled.
- has comprehensive built-in functionality.
- manages resources automatically.
Scripting: Means

A scripting language...

- is discoverable and interactive.
- is interpreted, not compiled.
- has comprehensive built-in functionality.
- manages resources automatically.
- is dynamically typed.
A scripting language. . .

- is discoverable and interactive.
- is interpreted, not compiled.
- 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. **Link**
3. **Run**

No compilation step.
Scripting: Interpreted, not Compiled

Program creation workflow:

- Edit
- Compile
- Link
- Run
Scripting languages come with “batteries included” (or easily available):

- Data structures: Lists, Sets, Dictionaries
- Linear algebra: Vectors, Matrices
- OS Interface: Files, Networks, Databases
- Persistence: Store, send and retrieve objects
- Defined, usable C interface
Typing Discipline

“If it walks like a duck and quacks like a duck, it is a duck.”

```python
def print_all(iterable):
    for i in iterable:
        print(i)

print_all([6, 7, 19])
print_all({1: "a", 2: "b", 3: "c"})
```
For this talk, Python is the scripting language of choice.
Scripting: Python

For this talk, Python is the scripting language of choice.

- Mature language
For this talk, Python is the scripting language of choice.

- Mature language
- Has a large and active community
For this talk, Python is the scripting language of choice.

- Mature language
- Has a large and active community
- Emphasizes readability
Scripting: Python

For this talk, Python is the scripting language of choice.

- Mature language
- Has a large and active community
- Emphasizes readability
- Written in widely-portable C
Scripting: Python

For this talk, Python is the scripting language of choice.

- Mature language
- Has a large and active community
- Emphasizes readability
- Written in widely-portable C
- A ‘multi-paradigm’ language
Scripting: Speed

- Speed(C) \gg Speed(Python)
Scripting: Speed

- Speed(C) \gg Speed(Python)
- For most code, it does not matter.
Scripting: Speed

- Speed(C) ≫ Speed(Python)
- For most code, it does not matter.
- It does matter for inner loops.
Scripting: Speed

- Speed(C) >> Speed(Python)
- For most code, it does not matter.
- It does matter for inner loops.
- One solution: hybrid (“glued”) code.
Scripting: Speed

- Speed(C) $\gg$ Speed(Python)
- For most code, it does not matter.
- It does matter for inner loops.
- One solution: hybrid ("glued") code.

Python + CUDA hybrids?
Scripting: Speed

- Speed(C) ≫ Speed(Python)
- For most code, it does not matter.
- It does matter for inner loops.
- One solution: hybrid (“glued”) code.

Python + CUDA hybrids? **PyCuda**!
Questions?
Outline

1. Scripting Languages

2. Scripting CUDA
   - Whetting your Appetite
   - Working with PyCuda
   - A peek under the hood

3. Metaprogramming CUDA

4. Discontinuous Galerkin on CUDA
Whetting your appetite

```python
1 import pycuda.driver as cuda
2 import pycuda.autoinit
3 import numpy
4 a = numpy.random.randn(4,4).astype(numpy.float32)
5 a_gpu = cuda.mem_alloc(a.size * a.dtype.itemsize)
6 cudaMemcpy_h2d(a_gpu, a)
```

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

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

func = mod.get_function("doublify")
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
Whetting your appetite

```python
mod = cuda.SourceModule(""

__global__ void doublify(float *a)
{
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
}
"")

func = mod.get_function("doublify")
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
Did somebody say “Abstraction is good”?
import numpy
import pycuda.autoinit
import pycuda.gpucarray as gpucarray

a_gpu = gpucarray.to_gpu(
    numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()

print a_doubled
print a_gpu
Outline

1. Scripting Languages

2. Scripting CUDA
   - Whetting your Appetite
   - Working with PyCuda
   - A peek under the hood

3. Metaprogramming CUDA

4. Discontinuous Galerkin on CUDA
PyCuda Philosophy

- Provide complete access
PyCuda Philosophy

- Provide complete access
- Automatically manage resources
PyCuda Philosophy

- Provide complete access
- Automatically manage resources
- Provide abstractions
PyCuda Philosophy

- Provide complete access
- Automatically manage resources
- Provide abstractions
- Allow interactive use
PyCuda Philosophy

- Provide complete access
- Automatically manage resources
- Provide abstractions
- Allow interactive use
- Check for and report errors automatically
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.
PyCuda exposes *all* of CUDA.

For example:

- Arrays and Textures
- Pagelocked host memory
- Memory transfers (asynchronous, structured)
- Streams and Events
- Device queries
PyCuda: Completeness

PyCuda supports every OS that CUDA supports.
PyCuda supports every OS that CUDA supports.

- Linux
- Windows
- OS X
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 idioms, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code. PyCuda knows about dependencies, too, (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.gpudarray`, `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.autotinit
import pycuda.driver as drv
import numy

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

multiply_them = mod.get_function('multiply_them')

a = numy.random.randn(1000).astype(numy.float32)
b = numy.random.randn(1000).astype(numy.float32)
dest = numy.zeros_like(a)
multiply_them(a, b, dest, 0)
```

PyCuda: Workflow

Edit ➔ Run
PyCuda: Workflow

Edit -> Run

SourceModule("...")
PyCuda: Workflow

1. Edit
2. Run
3. SourceModule("...")
PyCuda: Workflow

- Edit
- Run
- SourceModule("...")
- Cache?

PyCuda
PyCuda: Workflow

Edit → Run → SourceModule("...")

Cache? → nvcc → PyCuda
PyCuda: Workflow

Edit

Run

SourceModule("...")

Cache?

nvcc

.cubin

PyCuda
PyCuda: Workflow

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

Cache! -> nvcc -> .cubin

PyCuda
PyCuda: Workflow

Edit

Run

SourceModule("...")

Cache!

nvcc .cubin

Upload to GPU

PyCuda

Andreas Klöckner
Applied Math, Brown
High-Productivity Supercomputing: Metaprogramming GPUs
PyCuda: Workflow

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

---

**Andreas Klöckner**

High-Productivity Supercomputing: Metaprogramming GPUs
Kernel Invocation

```python
mod = pycuda.driver.SourceModule(
    "__global__ my_func(int x, float *y){...}"
)
func = mod.get_function("my_func")
mem = pycuda.driver.mem_alloc(20000)
```

Two ways:
- Immediate:
  ```python
  func(numpx.int32(17), mem, block=(tx,ty,tz), grid=(bx,by))
  ```
- Prepared:
  ```python
  func.prepare("iP", block=(tx,ty,tz))
  func.prepared_call((bx,by), 17, mem)
  ```

Fast, Safe
Convenient :-)

Andreas Klöckner
Applied Math, Brown
High-Productivity Supercomputing: Metaprogramming GPUs
Kernel Invocation

```python
mod = pycuda.driver.SourceModule("__global__ my_func(int x, float *y){...}")
func = mod.get_function("my_func")
mem = pycuda.driver.mem_alloc(20000)

Two ways:

Immediate:
func(numpy.int32(17), mem, block=(tx,ty,tz), grid=(bx,by))

Prepared:
func.prepare("iP", block=(tx,ty,tz))
# see: pycuda struct
func.prepared_call((bx,by), 17, mem)
```

Fast, Safe
Convenient :-)

Andreas Klöckner
Applied Math, Brown

High-Productivity Supercomputing: Metaprogramming GPUs
Kernel Invocation

```python
mod = pycuda.driver.SourceModule("__global__ my_func(int x, float y)
                                 {...}")
func = mod.get_function("my_func")
mem = pycuda.driver.mem_alloc(20000)

Two ways:
- Immediate:
  ```python
  func(numpy.int32(17), mem, block=(tx,ty,tz), grid=(bx,by))
  ```
- Prepared:
  ```python
  func.prepare("ip")
  func.prepared_call((bx,by), 17, mem)
  ```

Fast, Safe
Convenient :-)
```
Kernel Invocation

```python
mod = pycuda.driver.SourceModule("__global__ my_func(int x, float *y){...}"")
func = mod.get_function("my_func")
mem = pycuda.driver.mem_alloc(20000)

Two ways:

- Immediate:
  ```python
  func(numpy.int32(17), mem, block=(tx,ty,tz), grid=(bx,by))
  ```

- Prepared:
  ```python
  func.prepare("iP", block=(tx, ty, tz))  # see: pydoc struct
  func.prepared_call((bx,by), 17, mem)
  ```
```
Kernel Invocation

```python
mod = pycuda.driver.SourceModule("__global__ my_func(int x, float *y){...}"")
func = mod.get_function("my_func")
mem = pycuda.driver.mem_alloc(20000)
```

Two ways:

- **Immediate:**
  ```python
  func(numpy.int32(17), mem, block=(tx,ty,tz), grid=(bx,by))
  ```

- **Prepared:**  
  ```python
  func.prepare("iP", block=(tx, ty, tz))  # see: pydoc struct
  func.prepared_call((bx,by), 17, mem)
  ```

Fast, Safe
Kernel Invocation

```python
mod = pycuda.driver.SourceModule("__global__ my_func(int x, float *y){...}"")
func = mod.get_function("my_func")
mem = pycuda.driver.mem_alloc(20000)

Two ways:
- Immediate:
  ```python
  func(numpy.int32(17), mem, block=(tx,ty,tz), grid=(bx,by))
  ```
- Prepared:
  ```python
  func.prepare("iP", block=(tx, ty, tz))  # see: pydoc struct
  func.prepared_call((bx,by), 17, mem)
  ```
```
Kernel Invocation: Automatic Copies

```python
mod = pycuda.driver(SourceModule(
    "__global__ my_func(float *out, float *in){...}"))
func = mod.get_function("my_func")

src = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.empty_like(src)
```

InOut exists, too.

Only for immediate invocation style.
Kernel Invocation: Automatic Copies

```
mod = pycuda.driver.SourceModule(
    "__global__ my_func(float *out, float *in){...}"
)
func = mod.get_function("my_func")

src = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.empty_like(src)

my_func(
    cuda.Out(dest),
    cuda.In(src),
    block=(400,1,1))
```
Kernel Invocation: Automatic Copies

mod = pycuda.driver.SourceModule("__global__ my_func(float *out, float *in){...}")
func = mod.get_function("my_func")

src = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.empty_like(src)

my_func(
    cuda.Out(dest),
    cuda.In(src),
    block=(400,1,1))

- "InOut" exists, too.
Kernel Invocation: Automatic Copies

```python
mod = pycuda.driver.SourceModule(
    '__global__ my_func(float *out, float *in){...}')
func = mod.get_function('my_func')

src = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.empty_like(src)

my_func(
    cuda.Out(dest),
    cuda.In(src),
    block=(400,1,1))
```

- "InOut" exists, too.
- Only for immediate invocation style.
Automatic Cleanup

- Reachable objects (memory, streams, ...) are never destroyed.
Automatic Cleanup

- Reachable objects (memory, streams, ...) are never destroyed.
- Once unreachable, released at an unspecified future time.
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()`) (partially true now, in VC and next release)
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()) (partially true now, in VC and next release)
- Correctly deals with multiple contexts and dependencies.
Working with Textures

```python
mem = cuda.mem_alloc(size)
```

![GPU Memory](image)
Working with Textures

\[
\text{mem} = \text{cuda.mem\_alloc(size)}
\]
Working with Textures

```python
mem = cuda.mem_alloc(size)
```

GPU Memory

```python
mod = cuda.SourceModule("...
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...entry, 2)
tr.set_args(...
f = mod.get_function("f")
f.prepare(argument_types",
blob=(bx, by, bz), texrefs=[tr])
f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.setArgs(...)
f = mod.get_function("f")
f.prepare(arg_types=",",
block=(bx, by, bz), texrefs=[tr])
f()
```

GPU Memory

SourceModule

Textures in the code:

```c
__global__ void f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format...(float, 2)
tr.set_args(...)
f = mod.get_function("f")
f.prepare...(arg_types=",", block=(bx,bx,bz), texrefs=[tr])
f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(..., 2)
tr.set_args(...)
f = mod.get_function("f")
f.prepare(arity_types=",",
block=(bx, by, bz), texrefs=[tr])
f()
```

SourceModule

```
texture<float2> my_tex
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(..., 2)
tr.set_args(...)
f = mod.get_function("f")
f.prepare(..., block=(bx, by, bz), texrefs=[tr])
f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(float, 2)
tr.set_args(...)
f = mod.get_function("f")
f.prepare(arg_types",
    block=(bx, by, bz), texrefs=[tr])
f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(float, 2)
tr.set_args(...)

f = mod.get_function("f")
rep = f.argument_types()
block=(bx, by, bz), texrefs=[tr]
```

SourceModule

```
texture<float2> my_tex
__global__ void f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_args(...)  
f = mod.get_function("f")
f.prepare(args='', block=(bx, by, bz), texrefs=[tr])
f()
```

SourceModule

```cpp
__global__ void f()
{
    texture<float2> my_tex
    ...
}
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
```

SourceModule

```python
__global__ void f()
```

GPU Memory

```python
texture<float2> my_tex
```
Working with Textures

```python
cuda.mem_alloc(size)
cuda.SourceModule("...")
mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_args(...)  # Assume the format and arguments are correctly set
mod.get_function("f")
f.prepare(args=..., block=(bx, by, bz), texrefs=[tr])
f()  # Call the function
```

```
texture<float2> my_tex
global void f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(float, 2)
tr.set_flags(...)

f = mod.get_function("f")
f.prepare("
    block=(bx, by, bz),
    texrefs=[tr]
"
)
f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)
f = mod.get_function("f")
f.prepare(arg_types="",
block=(bx, by, bz), texrefs=[tr])
f()
```
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get Texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)
f = mod.get_function("f")
f.prepare( args=","
block=(bx, by, bz), texrefs=[tr]
)
f()
```
Scripting Languages

Scripting CUDA

Metaprogramming CUDA

Discontinuous Galerkin on CUDA

Working with PyCuda

Working with Textures

```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)
f = mod.get_function("f")
f.prepare(arg_types=[],
    block=(bx,by,bz),
    texrefs=[tr])
```

GPU Memory

SourceModule

texture<float2> my_tex

__global__ void f()
Working with Textures

\[
\begin{align*}
\text{mem} &= \text{cuda.mem_alloc(size)} \\
\text{mod} &= \text{cuda.SourceModule("...")} \\
\text{tr} &= \text{mod.get_texref("my\_tex")} \\
\text{tr.set_address}(&\text{mem, size}) \\
\text{tr.set_format}(\ldots\text{float, 2}) \\
\text{tr.set_flags}(\ldots) \\
\text{f} &= \text{mod.get_function("f")} \\
\text{f.prepare}(\text{arg\_types"f"}, \\
\quad \text{block=} (bx,by,bz), \text{texrefs=} [\text{tr}])
\end{align*}
\]
Working with Textures

```python
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)
f = mod.get_function("f")
f.prepare(arg_types="",
    block=(bx,by,bz), texrefs=[tr])
f()
```
Working withTextures

```python
mem = cuda.mem Alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)
f = mod.get_function("f")
f.prepare( arg_types "", 
    block=(bx,by,bz), texrefs=[tr])
f()
```
Scripting Languages
Scripting CUDA
Metaprogramming CUDA
Discontinuous Galerkin on CUDA

Working with PyCuda

`gpuarray`: Simple Linear Algebra

`pycuda.gparray`:
- Meant to look and feel just like `numpy`.

Memory behind `gparray` available as `.gpudata` attribute.

Use as kernel arguments, textures, etc.

Control concurrency through streams.

Andreas Klöckner
High-Productivity Supercomputing: Metaprogramming GPUs
Working with PyCuda

**gpua rray:** Simple Linear Algebra

**pycuda.gpua rray:**

- Meant to look and feel just like *numpy*.
  - `gpua rray.to_gpu(numpy_array)`
  - `numpy_array = gpua rray.get()`
Scripting Languages
Scripting CUDA
Metaprogramming CUDA
Discontinuous Galerkin on CUDA

Working with PyCuda

**gpuarray: Simple Linear Algebra**

**pycuda.gpuarray:**

- Meant to look and feel just like numpy.
  - `gpuarray.to_gpu(numpy_array)`
  - `numpy_array = gpuarray.get()`
- No: indexing, slicing, etc. (yet)
**gpuarray**: Simple Linear Algebra

`pycuda.gpuarray`:

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

- No: indexing, slicing, etc. (yet)
- Yes: `+`, `-`, `*`, `/`, fill, `sin`, `exp`, `log`, `rand`, ...
gpuarray: Simple Linear Algebra

pycuda.gpuarray:

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

- No: indexing, slicing, etc. (yet)
- Yes: +, -, *, /, fill, sin, exp, log, rand, ...
- print gpuarray for debugging.
gpuarray: Simple Linear Algebra

`pycuda.gpuarray`:

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

- No: indexing, slicing, etc. (yet)
- Yes: `+`, `-`, `*`, `/`, fill, `sin`, `exp`, `log`, `rand`, ...
- `print gpuarray` for debugging.
- Memory behind `gpuarray` available as `.gpudata` attribute.
  - Use as kernel arguments, textures, etc.
gputarray: Simple Linear Algebra

pycuda.gputarray:

- Meant to look and feel just like numpy.
  - gputarray.to_gpu(numpy_array)
  - numpy_array = gputarray.get()
- No: indexing, slicing, etc. (yet)
- Yes: +, -, *, /, fill, sin, exp, log, rand, …
- print gputarray for debugging.
- Memory behind gputarray available as .gpudata attribute.
  - Use as kernel arguments, textures, etc.
- Control concurrency through streams.
PyCuda: Vital Information

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

1. Scripting Languages

2. Scripting CUDA
   - Whetting your Appetite
   - Working with PyCuda
   - A peek under the hood

3. Metaprogramming CUDA

4. Discontinuous Galerkin on CUDA
CUDA APIs

C/C++

Runtime API

Python

PyCuda

Driver API

Kernel Driver

Hardware

Andreas Klöckner
Applied Math, Brown

High-Productivity Supercomputing: Metaprogramming GPUs
CUDA APIs

CUDA has two Programming Interfaces:

- “Runtime”
- “Driver”
CUDA has two Programming Interfaces:

- "Runtime" high-level
- "Driver" low-level
CUDA has two Programming Interfaces:

- “Runtime” high-level (libcudart.so, in the “toolkit”)
- “Driver” low-level
CUDA has two Programming Interfaces:

- "Runtime" high-level (libcuda.so, in the "toolkit")
- "Driver" low-level (libcuda.so, comes with GPU driver)
CUDA has two Programming Interfaces:

- “Runtime” high-level
  (libcudart.so, in the “toolkit”)

- “Driver” low-level
  (libcuda.so, comes with GPU driver)

(mutually exclusive)
Runtime vs. Driver API

Runtime ↔ Driver differences:

- Explicit initialization.
Runtime vs. Driver API

Runtime ↔ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.
Runtime ↔ Driver differences:

- Explicit initialization.
- Code objects (“Modules”) become programming language objects.
- Texture handling requires slightly more work.
Runtime vs. Driver API

Runtime ↔ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.
- Texture handling requires slightly more work.
- Only needs `nvcc` for compiling GPU code.
Runtime vs. Driver API

Runtime ↔ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.
- Texture handling requires slightly more work.
- Only needs `nvcc` for compiling GPU code.

Driver API:

- Conceptually cleaner
- Less sugar-coating (provide in Python)
- Not very different otherwise
PyCuda: API Tracing

With ./configure --cuda-trace=1:
PyCuda: API Tracing

With ./configure --cuda-trace=1:

```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.size * a.dtype.itemsize)
cuda.memcpy_htod(a_gpu, a)

mod = cuda.SourceModule('''
    __global__ void doublify(float *a)
    {
        int idx = threadIdx.x + threadIdx.y*4;
        a[idx] *= 2;
    }
'''
)

func = mod.get_function('doublify')
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)
```

Andreas Klöckner
High-Productivity Supercomputing: Metaprogramming GPUs
Questions?
Outline

1. Scripting Languages

2. Scripting CUDA

3. Metaprogramming CUDA
   - Programs that write Programs

4. Discontinuous Galerkin on CUDA

Andreas Klöckner
High-Productivity Supercomputing: Metaprogramming GPUs
In PyCuda, CUDA C code does not need to be a compile-time constant.
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(Unlike the CUDA Runtime API)
In PyCuda, CUDA C code does not need to be a compile-time constant.

(Unlike the CUDA Runtime API)
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(Unlike the CUDA Runtime API)
In PyCuda, CUDA C code does *not* need to be a compile-time constant.

(Unlike the CUDA Runtime API)
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(unlike the CUDA Runtime API)
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(unlike the CUDA Runtime API)
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(Unlike the CUDA Runtime API)
In PyCuda, CUDA C code does not need to be a compile-time constant.

(Unlike the CUDA Runtime API)
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(unlike the CUDA Runtime API)

Idea

Python Code

CUDA C Code

nvcc

.cubin

GPU

Result

Machine

Andreas Klöckner

High-Productivity Supercomputing: Metaprogramming GPUs
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(unlike the CUDA Runtime API)
Metaprogramming

In PyCuda, CUDA C code does not need to be a compile-time constant.

(Unlike the CUDA Runtime API)
Machine-generated Code

Why machine-generate code?

- Automated Tuning
  (cf. ATLAS, FFTW)
Machine-generated Code

Why machine-generate code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types
Why machine-generate code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
Machine-generated Code

Why machine-generated code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables (→ register pressure)
Why machine-generated 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.{registers,lmem,smem}
  ```
PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  
  ```python
  func.{registers,lmem,smem}
  ```

- Exact GPU timing via events
PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  - `func.{registers, lmem, smem}`
- Exact GPU timing via events
- Can calculate hardware-dependent MP occupancy
PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  `func.{registers,lmem,smem}`
- Exact GPU timing via events
- Can calculate hardware-dependent MP occupancy
- `codepy:`
PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  \texttt{func\{\texttt{registers,lmem,smem}\}}

- Exact GPU timing via events

- Can calculate hardware-dependent MP occupancy

- codepy:
  - Build C syntax trees from Python
PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  
  ```
  func.{registers, lmem, smem}
  ```

- Exact GPU timing via events

- Can calculate hardware-dependent MP occupancy

- codepy:
  - Build C syntax trees from Python
  - Generates readable, indented C
PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  func.{registers, lmem, smem}
- Exact GPU timing via events
- Can calculate hardware-dependent MP occupancy
- codepy:
  - Build C syntax trees from Python
  - Generates readable, indented C
  - Also: CPU metaprogramming (so far Linux only)
**PyCuda: Support for Metaprogramming**

- Access properties of compiled code:
  
  ```
  func.{registers,lmem,smem}
  ```

- Exact GPU timing via events

- Can calculate hardware-dependent MP occupancy

- codepy:
  - Build C syntax trees from Python
  - Generates readable, indented C
  - Also: CPU metaprogramming (so far Linux only)
  - Unreleased (but in public VC—ask me)
Questions?
Outline

1. Scripting Languages
2. Scripting CUDA
3. Metaprogramming CUDA
4. Discontinuous Galerkin on CUDA
   - Introduction
   - Results
   - Conclusions
Let $\Omega := \bigcup_{i} D_k \subset \mathbb{R}^d$. 

**Discontinuous Galerkin Method**

Goal: Solve a conservation law on $\Omega$:

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

Example: Maxwell's Equations: Field:

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

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

Andreas Klöckner

High-Productivity Supercomputing: Metaprogramming GPUs

Applied Math, Brown
Discontinuous Galerkin Method

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

Goal
Solve a conservation law on $\Omega$: $u_t + \nabla \cdot F(u) = 0$
Discontinuous Galerkin Method

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

Goal

Solve a conservation law on $\Omega$:

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

Example

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

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

Multiply by test function, integrate by parts:

\[ 0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx \]
Multiply by test function, integrate by parts:

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

$$\partial_t u^k = - \sum_{\nu} D^{\partial \nu, k} [F(u^k)] + L^k [\hat{n} \cdot F - (\hat{n} \cdot F)^*]|_{A \subset \partial D_k}.$$
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.
Decomposition of a DG operator into Subtasks

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

1. Flux Gather
2. Flux Lifting
3. Local Differentiation
4. $F(u^k)$
5. $\partial_t u^k$

Green: Element-local parts of the DG operator.

**Note:** Explicit timestepping.
DG: Properties

Flexible:

- Variable order of accuracy
DG: Properties

Flexible:

- Variable order of accuracy
- Unstructured discretizations
DG: Properties

Flexible:

- Variable order of accuracy
- Unstructured discretizations
- Usable for many types of equations
DG: Properties

Flexible:
- Variable order of accuracy
- Unstructured discretizations
- Usable for many types of equations

Implementation-friendly:

Andreas Klöckner
Applied Math, Brown

High-Productivity Supercomputing: Metaprogramming GPUs
DG: Properties

Flexible:
- Variable order of accuracy
- Unstructured discretizations
- Usable for many types of equations

Implementation-friendly:
- Good stability properties
Flexible:

- Variable order of accuracy
- Unstructured discretizations
- Usable for many types of equations

Implementation-friendly:

- Good stability properties
- Parallelizes well
DG: Properties

Flexible:
- Variable order of accuracy
- Unstructured discretizations
- Usable for many types of equations

Implementation-friendly:
- Good stability properties
- Parallelizes well
- Simple (compared to other high-order unstructured methods)
Why do DG on Graphics Cards?

DG on GPUs: Why?
Why do DG on Graphics Cards?

DG on GPUs: Why?

- GPUs have deep Memory Hierarchy
- The majority of DG is local.
Why do DG on Graphics Cards?

DG on GPUs: Why?
- GPUs have deep Memory Hierarchy
  - The majority of DG is local.
- Compute Bandwidth $\gg$ Memory Bandwidth
  - DG is arithmetically intense.
Why do DG on Graphics Cards?

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 local workloads.
  - DG has very limited communication.
Why do DG on Graphics Cards?

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 local workloads.
  - DG has very limited communication.

“A match made in heaven?”
Outline

1. Scripting Languages
2. Scripting CUDA
3. Metaprogramming CUDA
4. Discontinuous Galerkin on CUDA
   - Introduction
   - Results
   - Conclusions
GTX280 vs. single core of Intel Core 2 Duo E8400

Andreas Klöckner
High-Productivity Supercomputing: Metaprogramming GPUs
Memory Bandwidth on a GTX 280

```
Polynomial Order N
20 40 60 80 100 120 140 160 180 200

Global Memory Bandwidth [GB/s]
Gather
Lift
Diff
Assy.
Peak
```
“Real-World” Scattering Calculation

Order $N = 4$, 78745 elements, 2.7M · 6 DOFs, single Tesla C1060.
Outline

1. Scripting Languages
2. Scripting CUDA
3. Metaprogramming CUDA
4. Discontinuous Galerkin on CUDA
   - Introduction
   - Results
   - Conclusions
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 :-)

Andreas Klöckner
High-Productivity Supercomputing: Metaprogramming GPUs
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
- CUDA tuning too tedious? Need more speed?  
  - Automate it: Metaprogramming
Conclusions

- Fun time to be in computational science
- Use Python and PyCuda to have even more fun :-)  
  - With no compromise in performance
- CUDA tuning too tedious? Need more speed?  
  - Automate it: Metaprogramming
- Further work in CUDA-DG:  
  - Multi-GPU  
  - Other equations (Euler, Poisson, possibly Navier-Stokes?)  
  - Double Precision
Where to from here?

PyCuda Homepage
(also these slides, tonight)
→ http://mathema.tician.de/software/pycuda

CUDA-DG Preprint
→ http://arxiv.org/abs/0901.1024
Questions?

? Thank you for your attention!

http://mathema.tician.de/software/pycuda

http://arxiv.org/abs/0901.1024