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

BROWN

Outline

BROWN

Outline

BROWN

Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge

BROWN

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: Goals

Scripting languages aim to reduce the load on the programmer:

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

BROWN

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.

BROWN

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?

BROWN

Scripting: Means

A scripting language...

- is discoverable and interactive.

BROWN

Scripting: Means

A scripting language...

- is discoverable and interactive.
- is interpreted, not compiled.

BROWN

Scripting: Means

A scripting language...

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

BROWN

Scripting: Means

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.

Scripting: Means

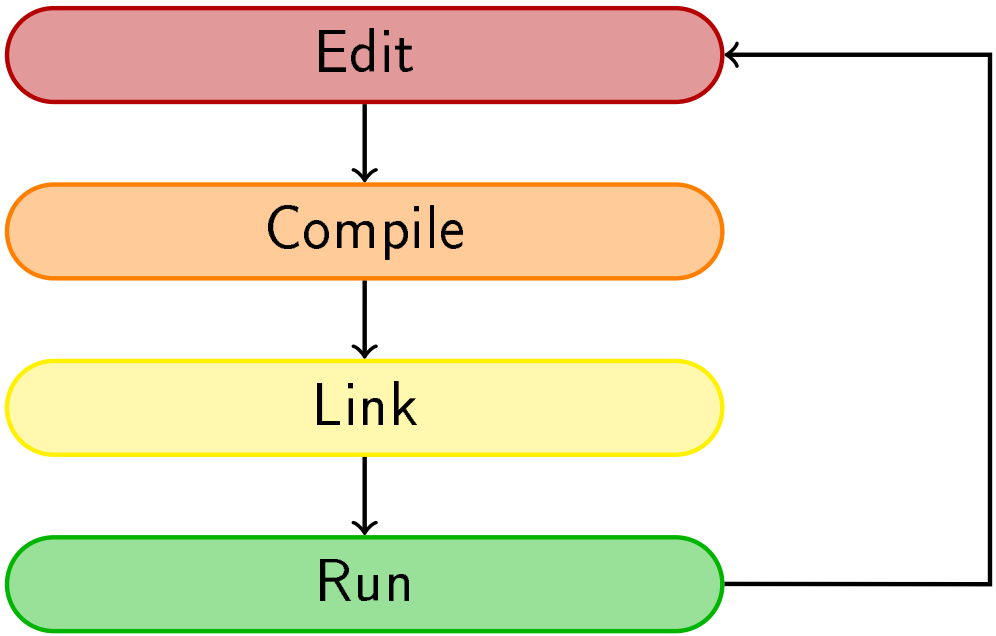
A scripting language...

- 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.

BROWN

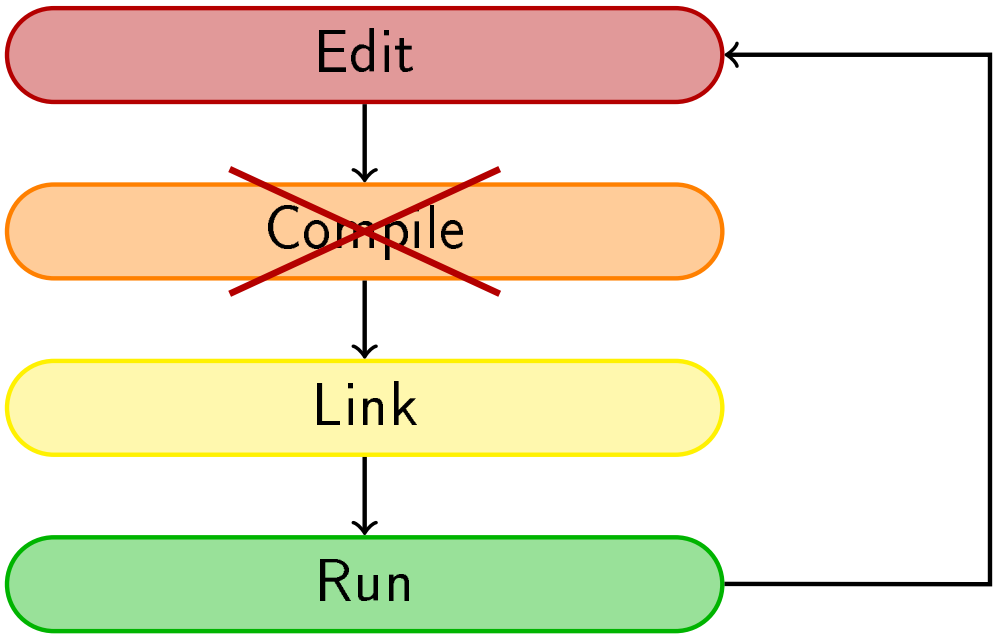
Scripting: Interpreted, not Compiled

Program creation workflow:



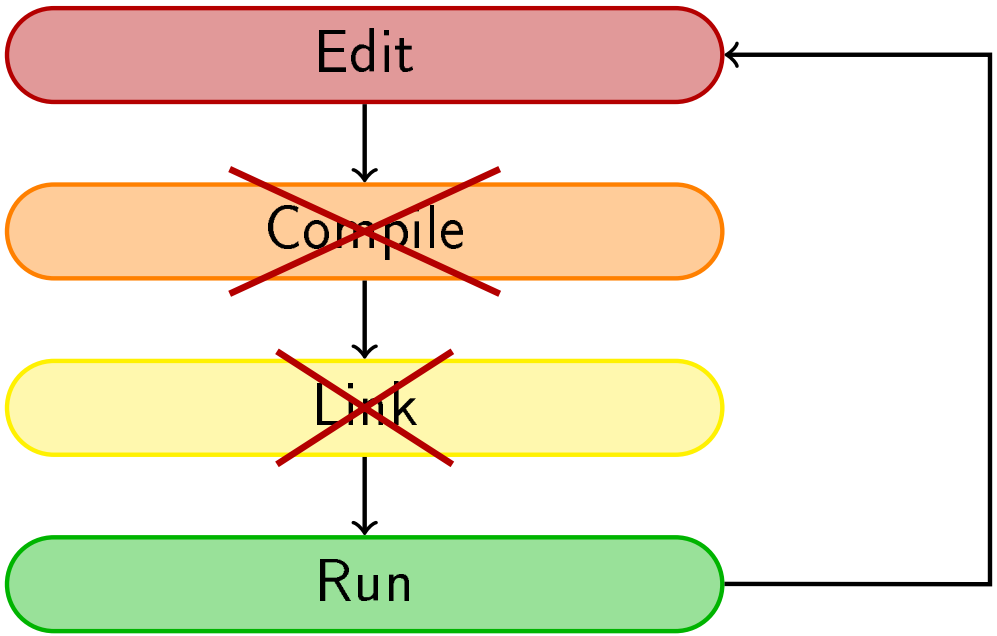
Scripting: Interpreted, not Compiled

Program creation workflow:



Scripting: Interpreted, not Compiled

Program creation workflow:



Batteries Included

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

BROWN

Scripting: Run-Time Typing

Typing Discipline

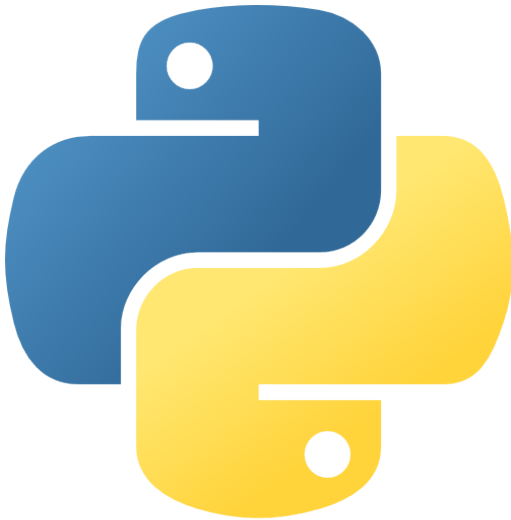
"If it walks like a duck and quacks like a duck, it *is* a duck."

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

print_all([6, 7, 19])
print_all({1: "a", 2: "b", 3: "c"})
```

BROWN

Scripting: Python

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

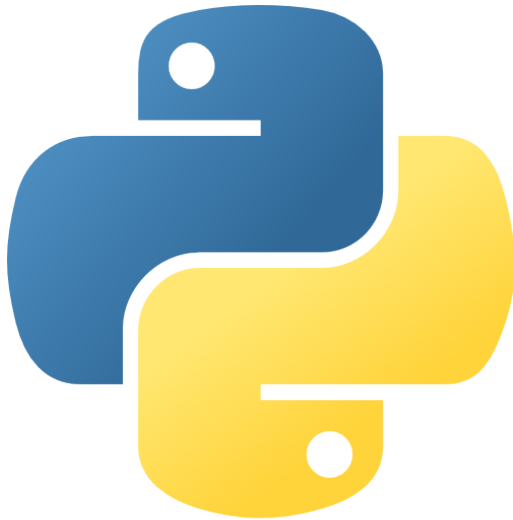


BROWN

Scripting: Python

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

- Mature language

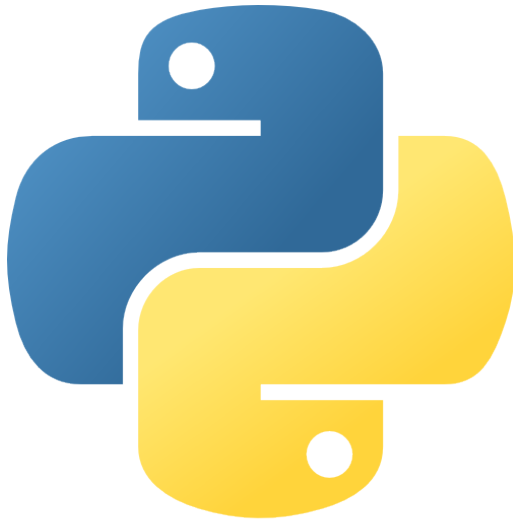


BROWN

Scripting: Python

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

- Mature language
- Has a large and active community

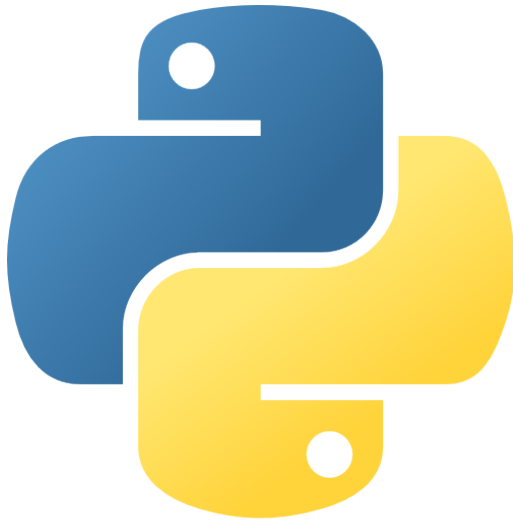


BROWN

Scripting: Python

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

- Mature language
- Has a large and active community
- Emphasizes readability

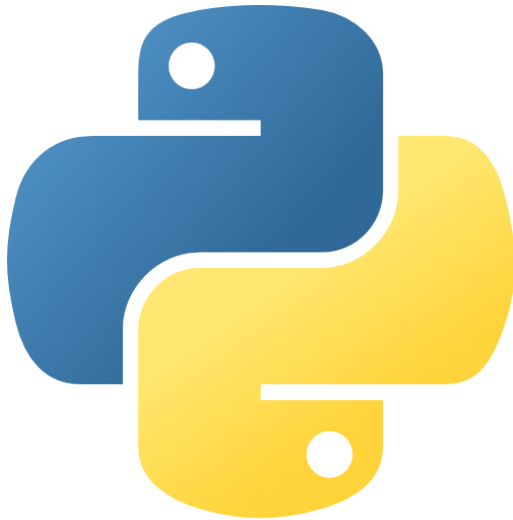


BROWN

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

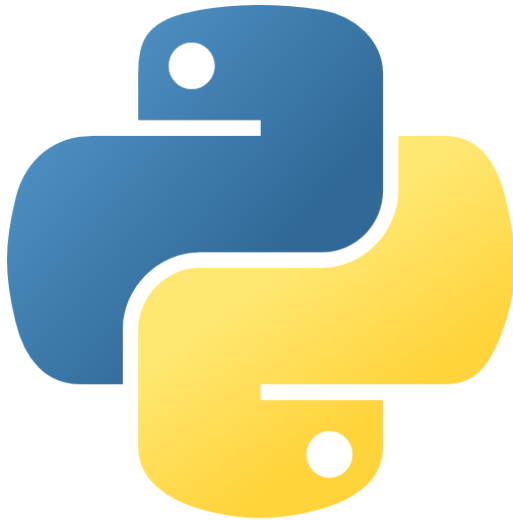


BROWN

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



BROWN

Scripting: Speed

- Speed(C) $\gg$ Speed(Python)

Scripting: Speed

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

BROWN

Scripting: Speed

- Speed(C) $\gg$ Speed(Python)
- For most code, it does not matter.
- It does matter for inner loops.

BROWN

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.

BROWN

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?

BROWN

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? **PyCuda!**

BROWN

Questions?

?

BROWN

Outline

BROWN

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

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

BROWN

Whetting your appetite

```
 9  mod = cuda.SourceModule("""
10      __global__ void doublify(float *a)
11      {
12        int idx = threadIdx.x + threadIdx.y*4;
13        a[idx] *= 2;
14      }
15      """)
16
17  func = mod.get_function("doublify")
18  func(a_gpu, block=(4,4,1))
19
20  a_doubled = numpy.empty_like(a)
21  cuda.memcpy_dtoh(a_doubled, a_gpu)
22  print a_doubled
23  print a
```

Whetting your appetite

```
 9  mod = cuda.SourceModule("""
10      __global__ void doublify(float *a)
11      {
12        int idx = threadIdx.x + threadIdx.y*4;
13        a[idx] *= 2;
14      }
15      """)
16
17  func = mod.get_function("doublify")
18  func(a_gpu, block=(4,4,1))
19
20  a_doubled = numpy.empty_like(a)
21  cuda.memcpy_dtoh(a_doubled, a_gpu)
22  print a_doubled
23  print a
```

Compute kernel

Whetting your appetite, Part II

Did somebody say "Abstraction is good"?

BROWN

Whetting your appetite, Part II

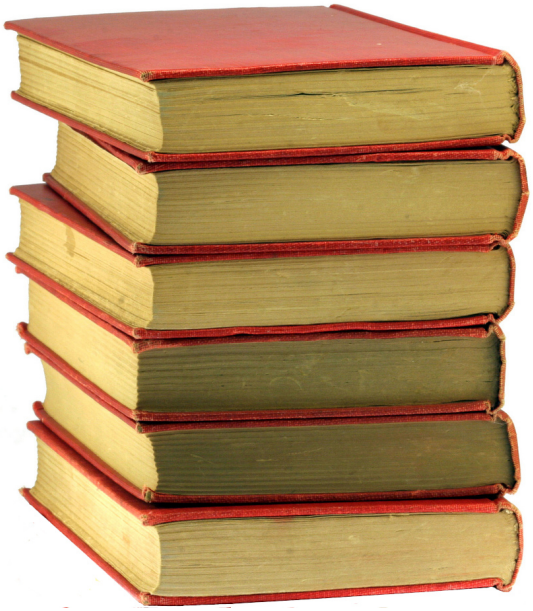
```
1  import numpy
2  import pycuda.autoinit
3  import pycuda.gpuarray as gpuarray
4
5  a_gpu = gpuarray.to_gpu(
6      numpy.random.randn(4,4).astype(numpy.float32))
7  a_doubled = (2*a_gpu).get()
8  print a_doubled
9  print a_gpu
```

BROWN

Outline

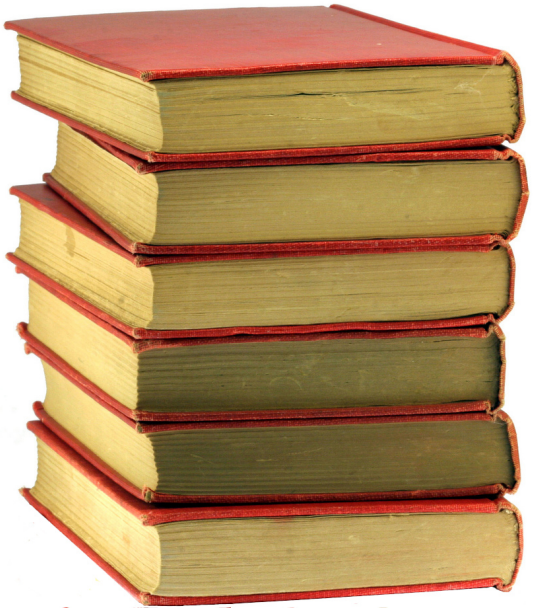
BROWN

PyCuda Philosophy



- Provide complete access

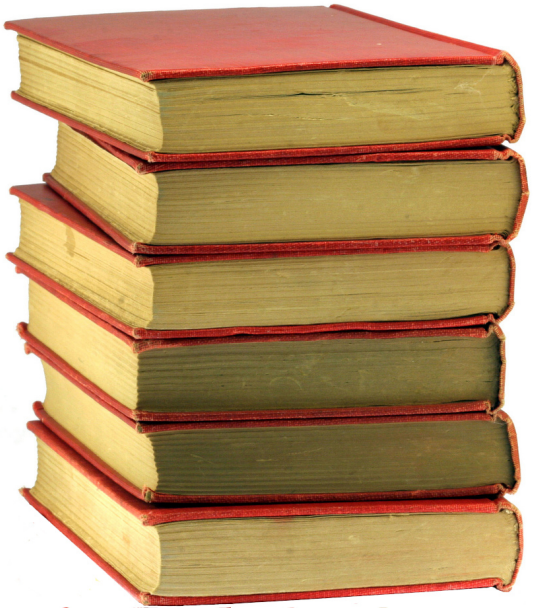
PyCuda Philosophy



- Provide complete access
- Automatically manage resources

BROWN

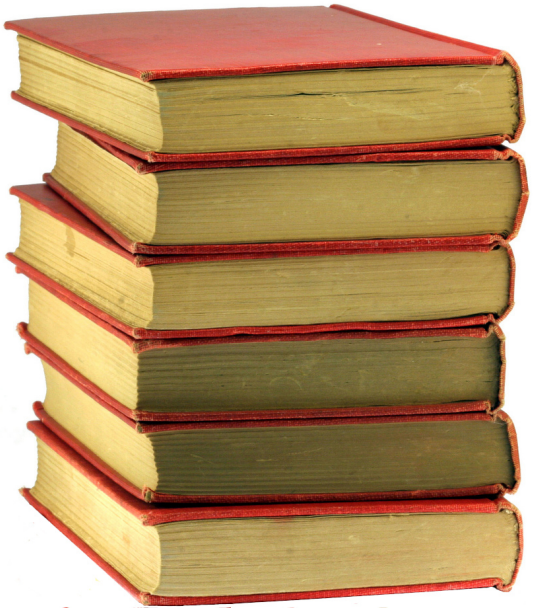
PyCuda Philosophy



- Provide complete access
- Automatically manage resources
- Provide abstractions

BROWN

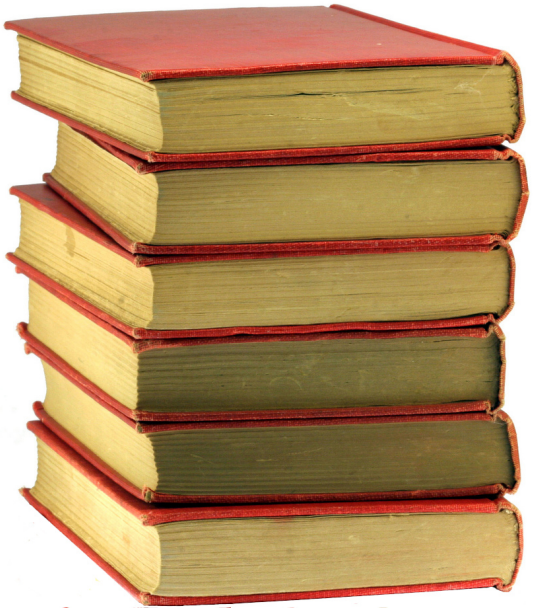
PyCuda Philosophy



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

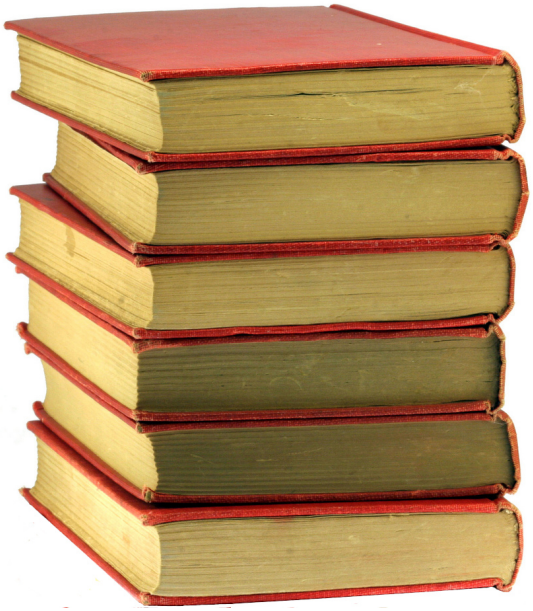
BROWN

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: Completeness

PyCuda exposes *all* of CUDA.

For example:

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

BROWN

PyCuda: Completeness

PyCuda supports every OS that CUDA supports.

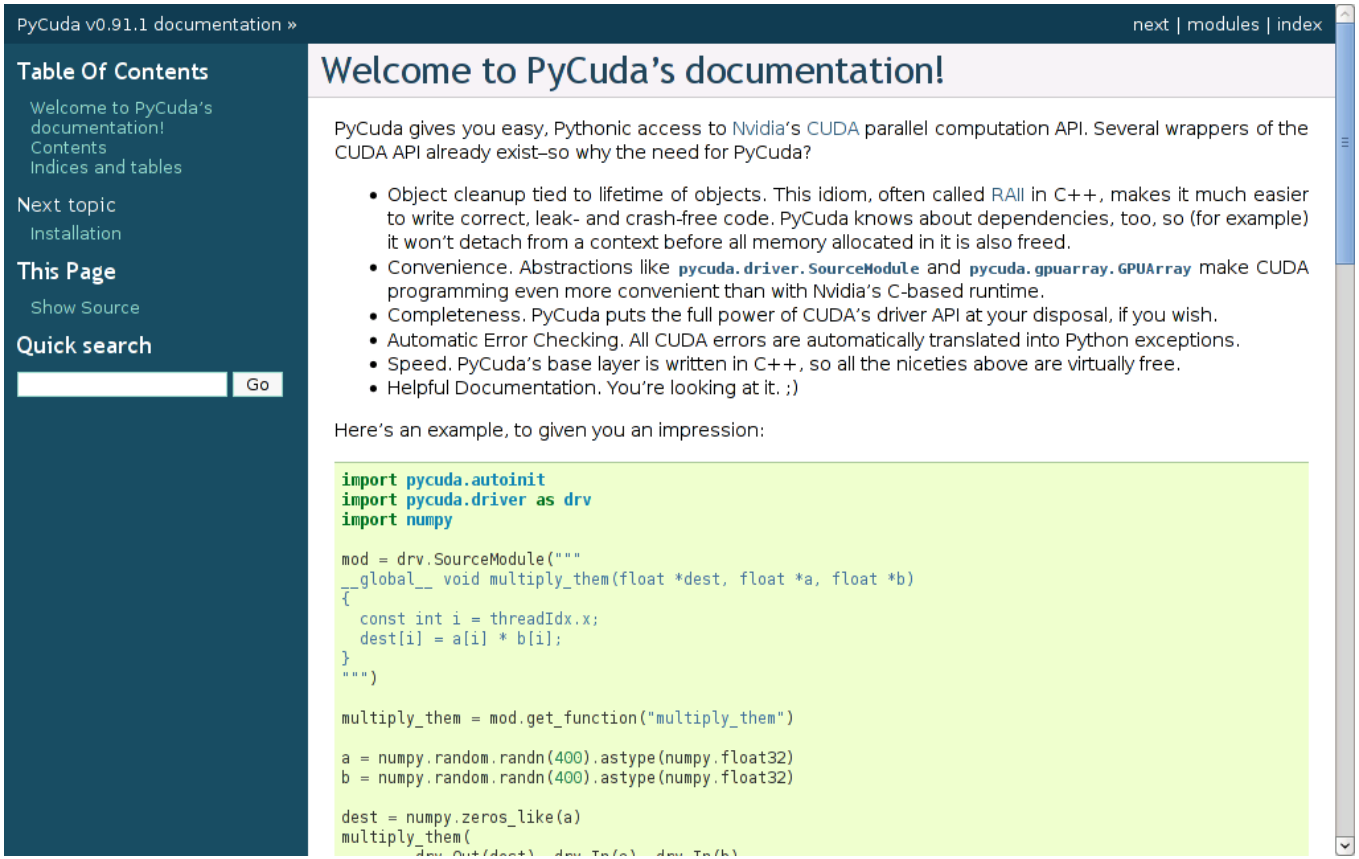
BROWN

PyCuda: Completeness

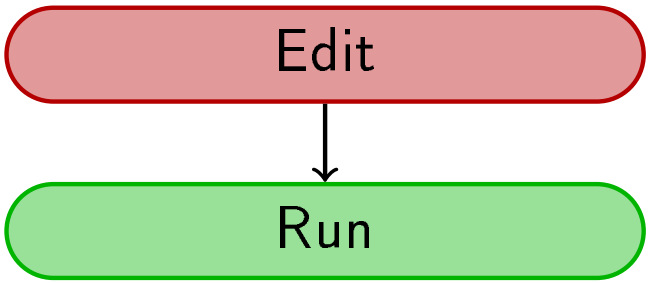
PyCuda supports every OS that CUDA supports.

- Linux
- Windows
- OS X

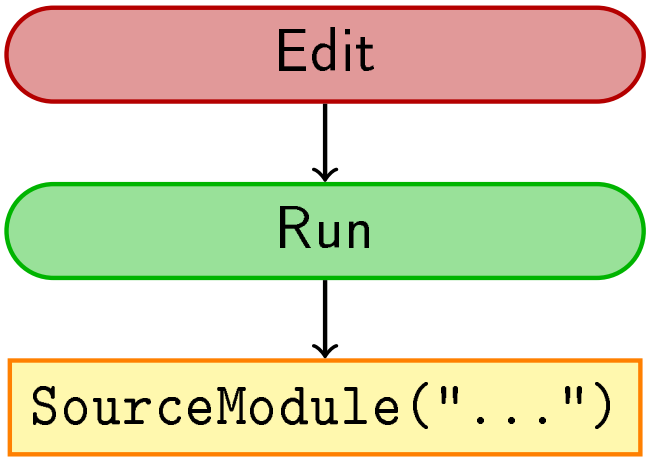
PyCuda: Documentation



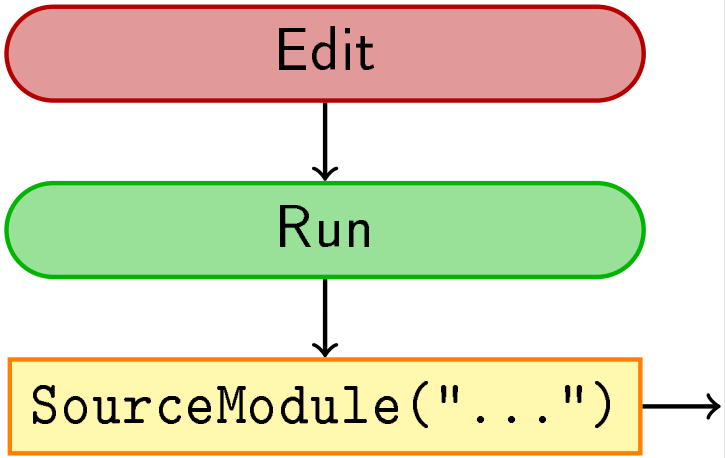
PyCuda: Workflow



PyCuda: Workflow



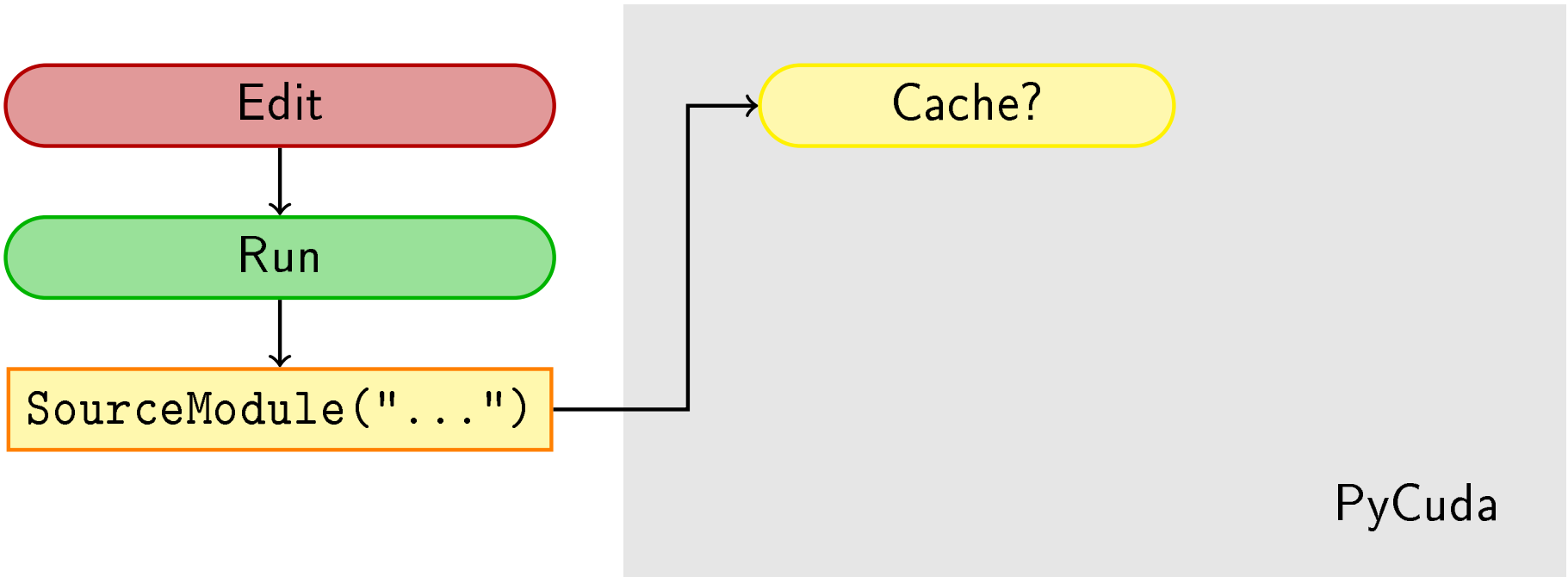
PyCuda: Workflow



PyCuda

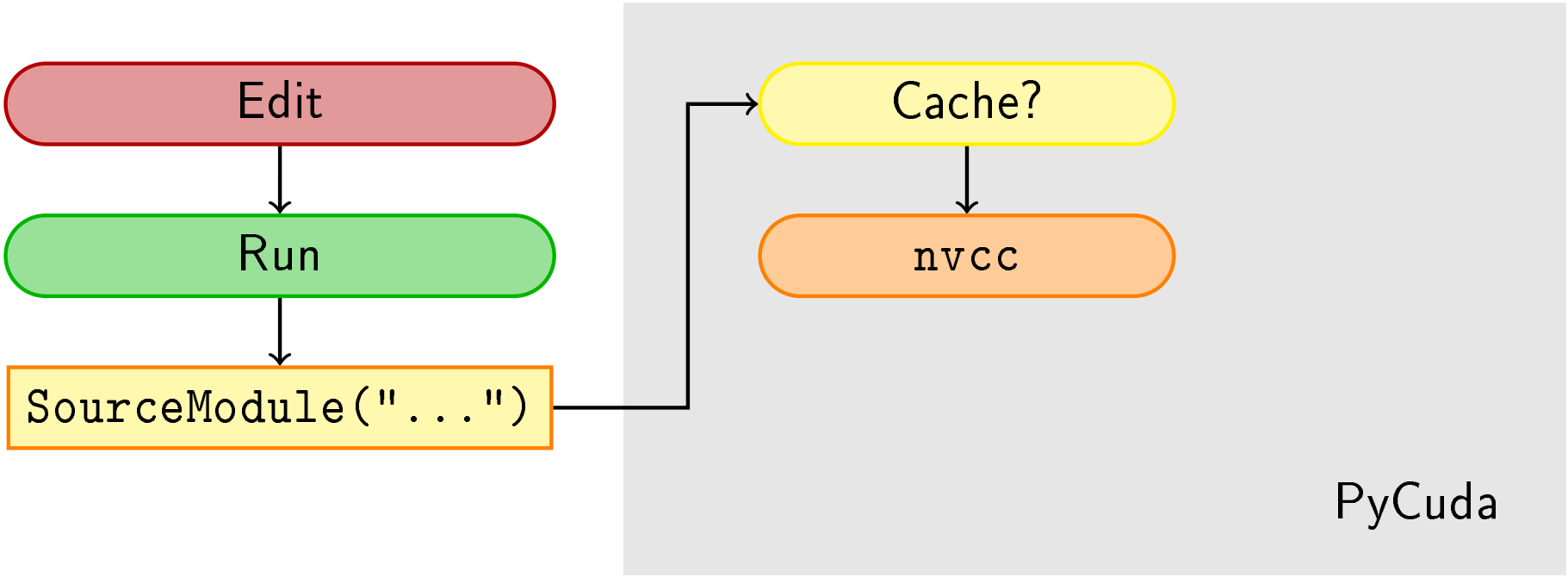
BROWN

PyCuda: Workflow

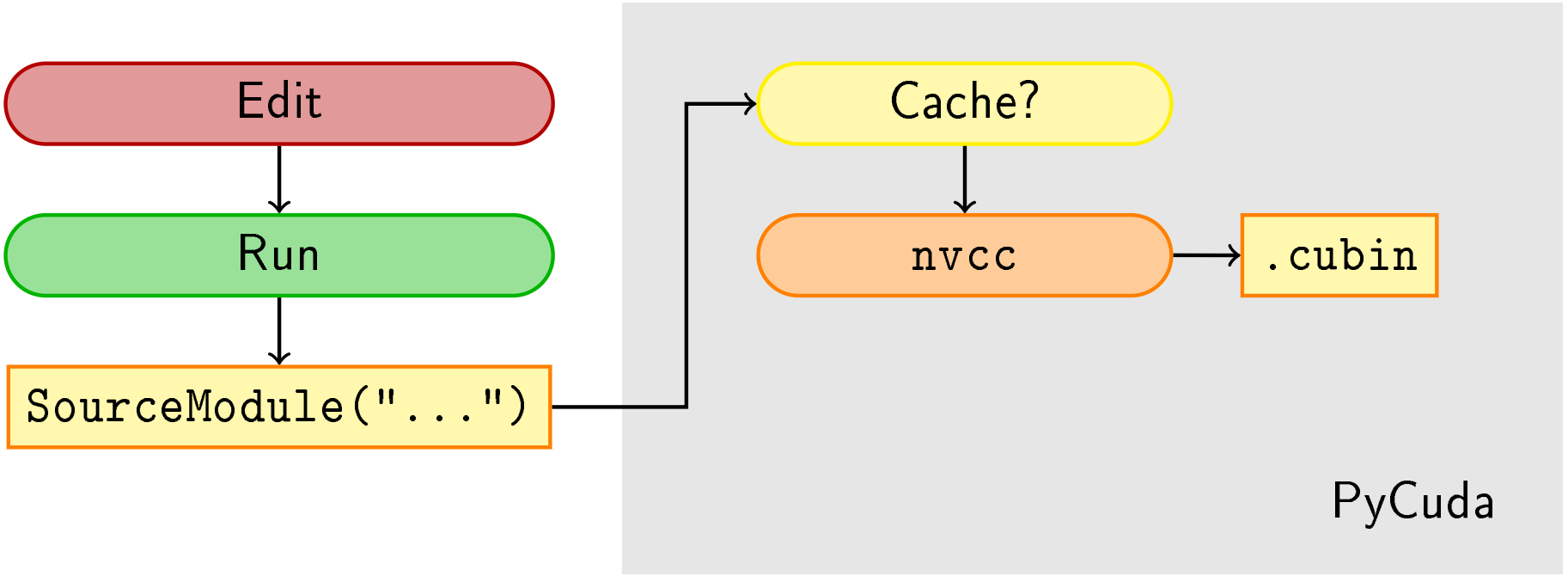


BROWN

PyCuda: Workflow

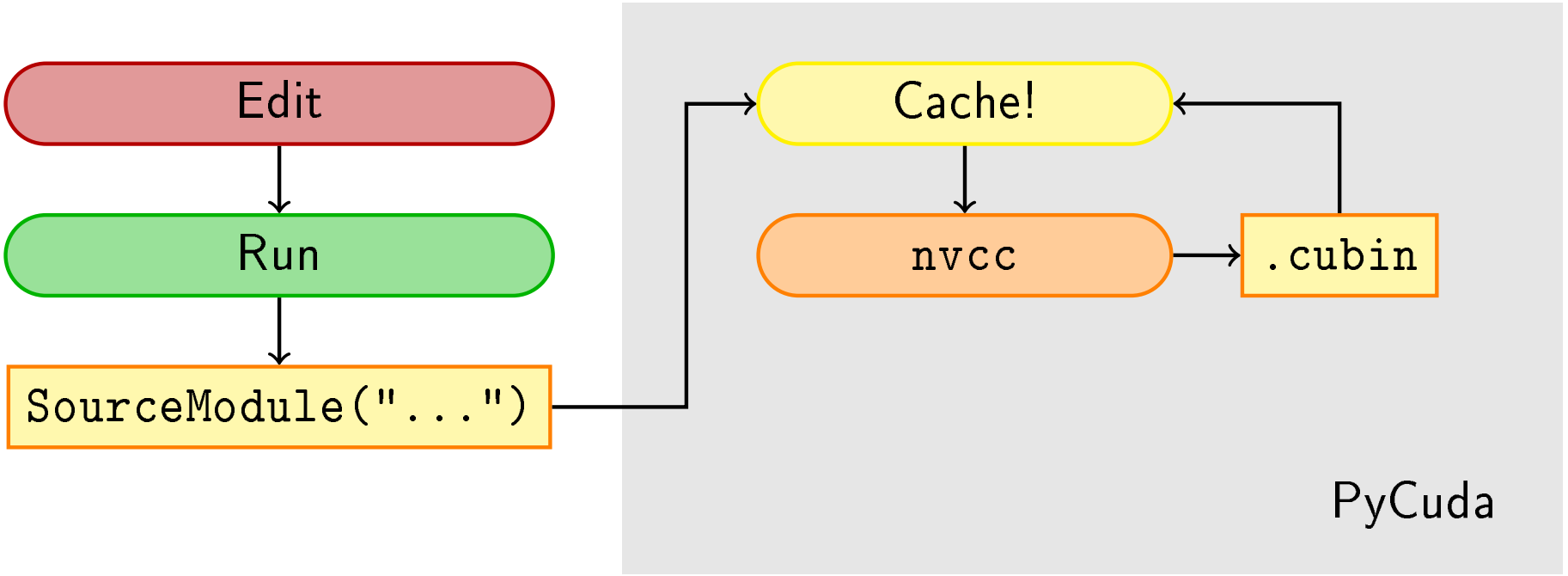


PyCuda: Workflow

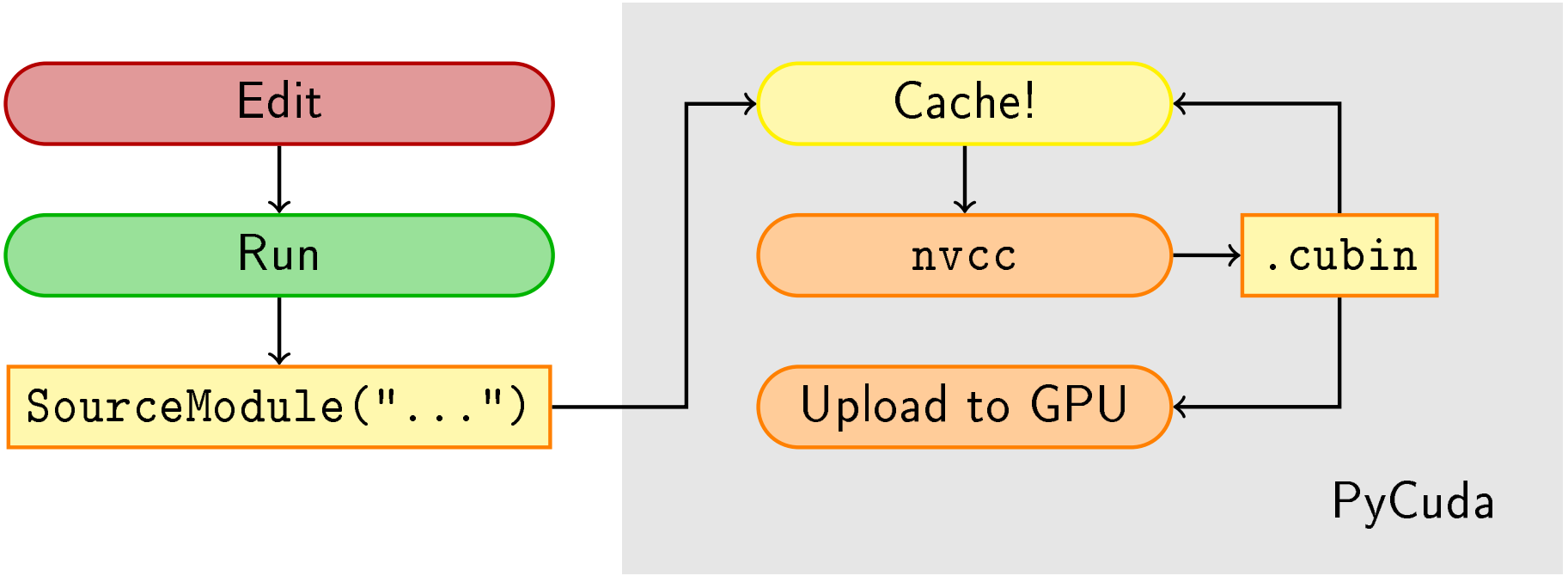


BROWN

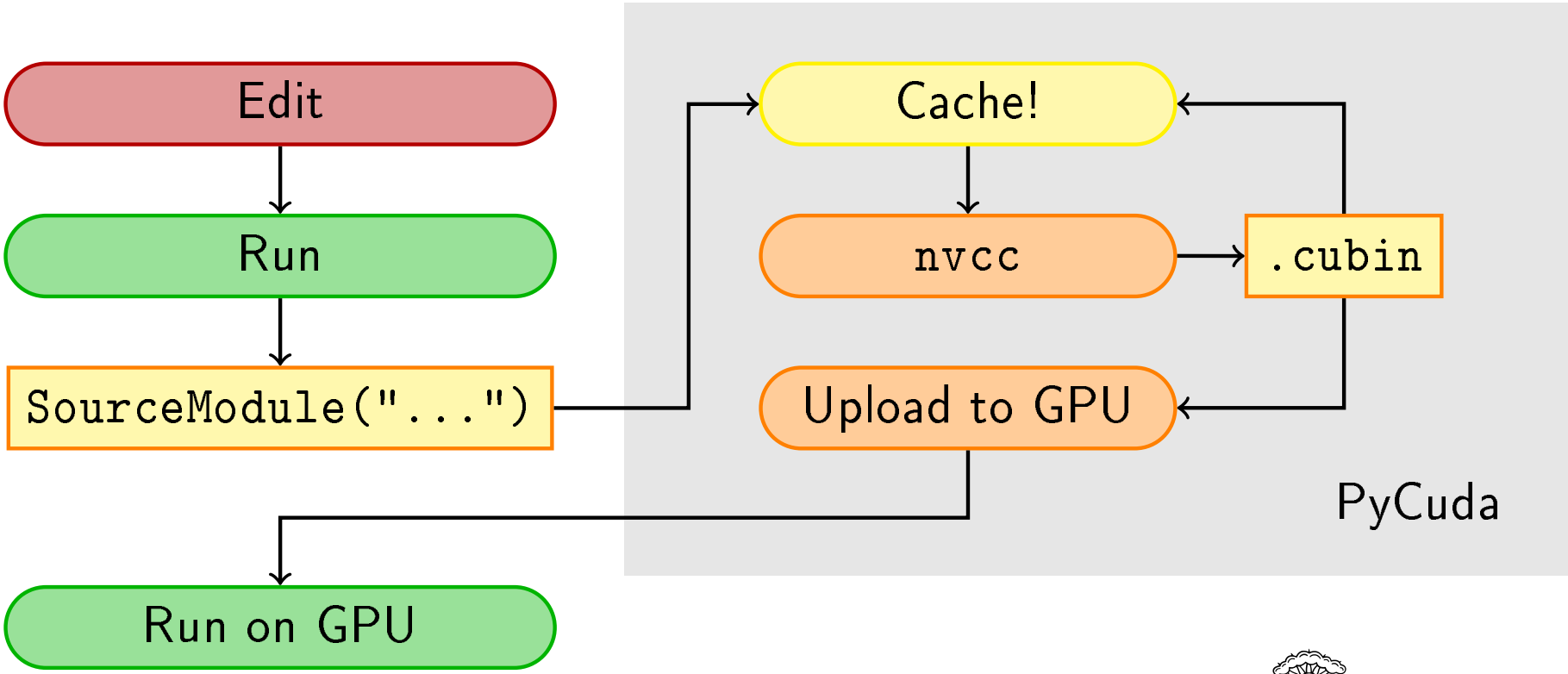
PyCuda: Workflow



PyCuda: Workflow



PyCuda: Workflow



Kernel Invocation

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

Kernel Invocation

```
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:

Kernel Invocation

```
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))
```

Kernel Invocation

```
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: pydoc struct
func.prepared_call((bx,by), 17, mem)
```

Kernel Invocation

```
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: Fast, Safe

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

Kernel Invocation

```
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: Convenient :-)

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

- Prepared:

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

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)
```

BROWN

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.

BROWN

Kernel Invocation: Automatic Copies

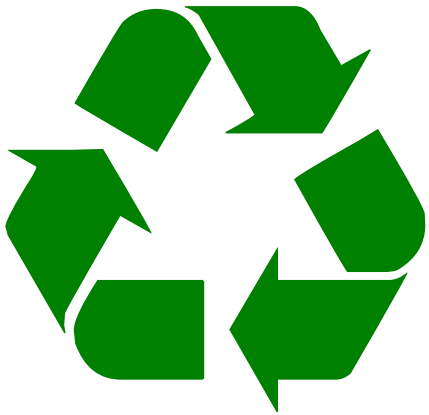
```
mod = pycuda.driver.SourceModule(
    "__global__ my_func(float *out, float *in){...}")
func = mod.get_function("my_func")

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.
- Only for immediate invocation style.

Automatic Cleanup

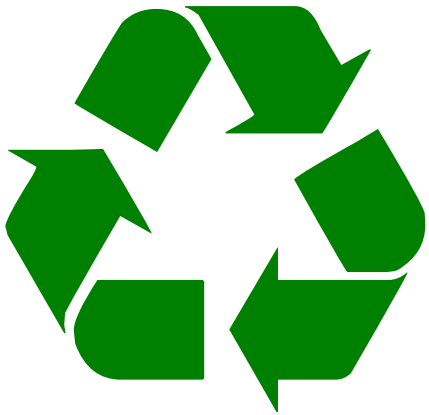
- Reachable objects (memory, streams, ...) are never destroyed.



BROWN

Automatic Cleanup

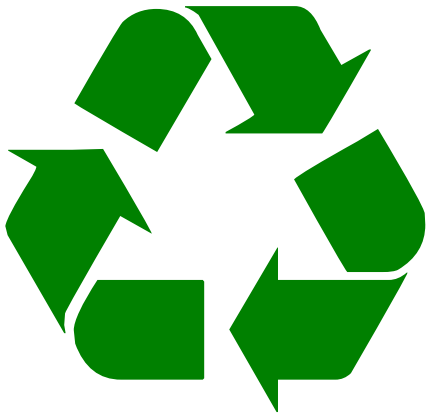
- Reachable objects (memory, streams, ...) are never destroyed.
- Once unreachable, released at an unspecified future time.



BROWN

Automatic Cleanup

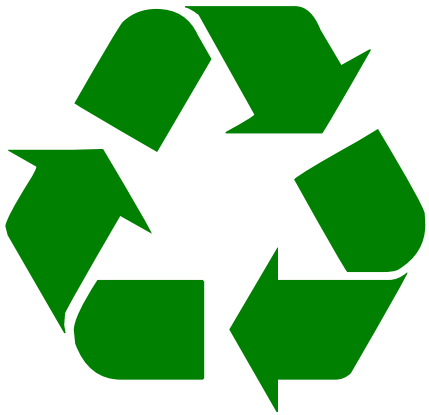
- Reachable objects (memory, streams, ...) are never destroyed.
- Once unreachable, released at an unspecified future time.
- Scarce resources (memory) can be explicitly freed. (`obj.free()`) (partially true now, in VC and next release)



BROWN

Automatic Cleanup

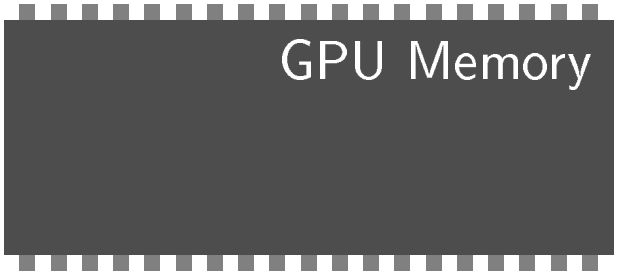
- Reachable objects (memory, streams, ...) are never destroyed.
- Once unreachable, released at an unspecified future time.
- Scarce resources (memory) can be explicitly freed. (`obj.free()`) (partially true now, in VC and next release)
- Correctly deals with multiple contexts and dependencies.



BROWN

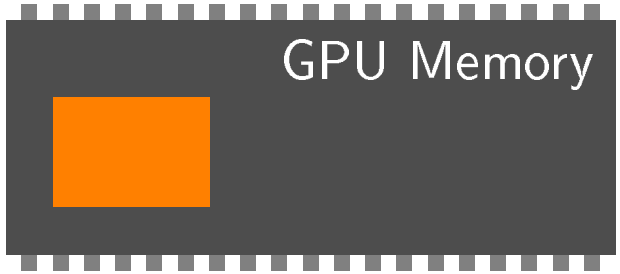
Working with Textures

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



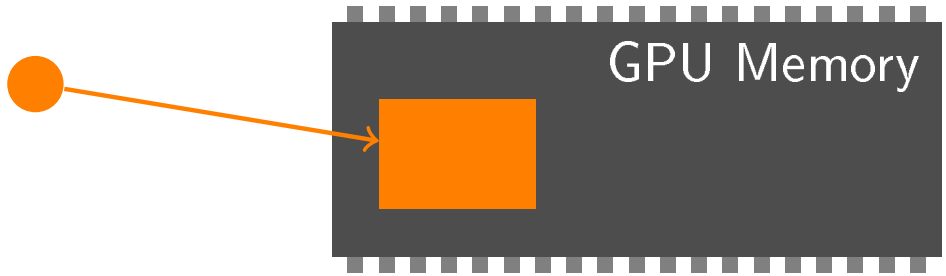
Working with Textures

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



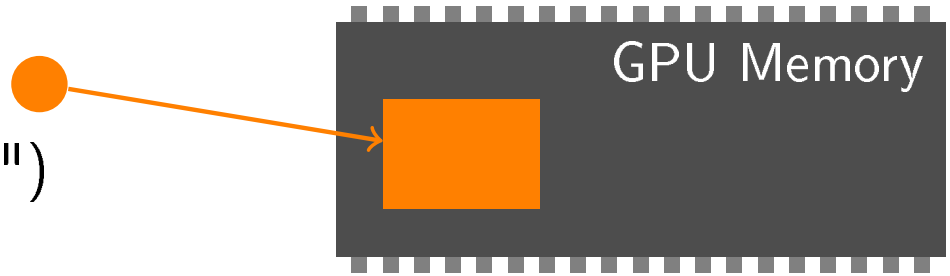
Working with Textures

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



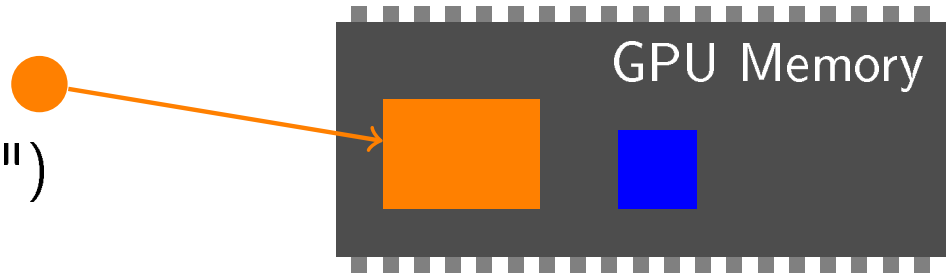
Working with Textures

```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
```



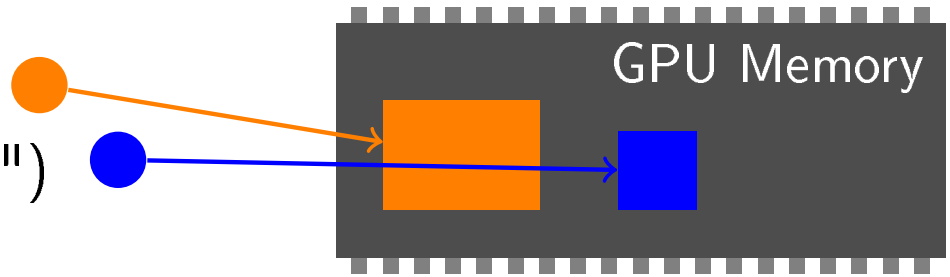
Working with Textures

```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
```



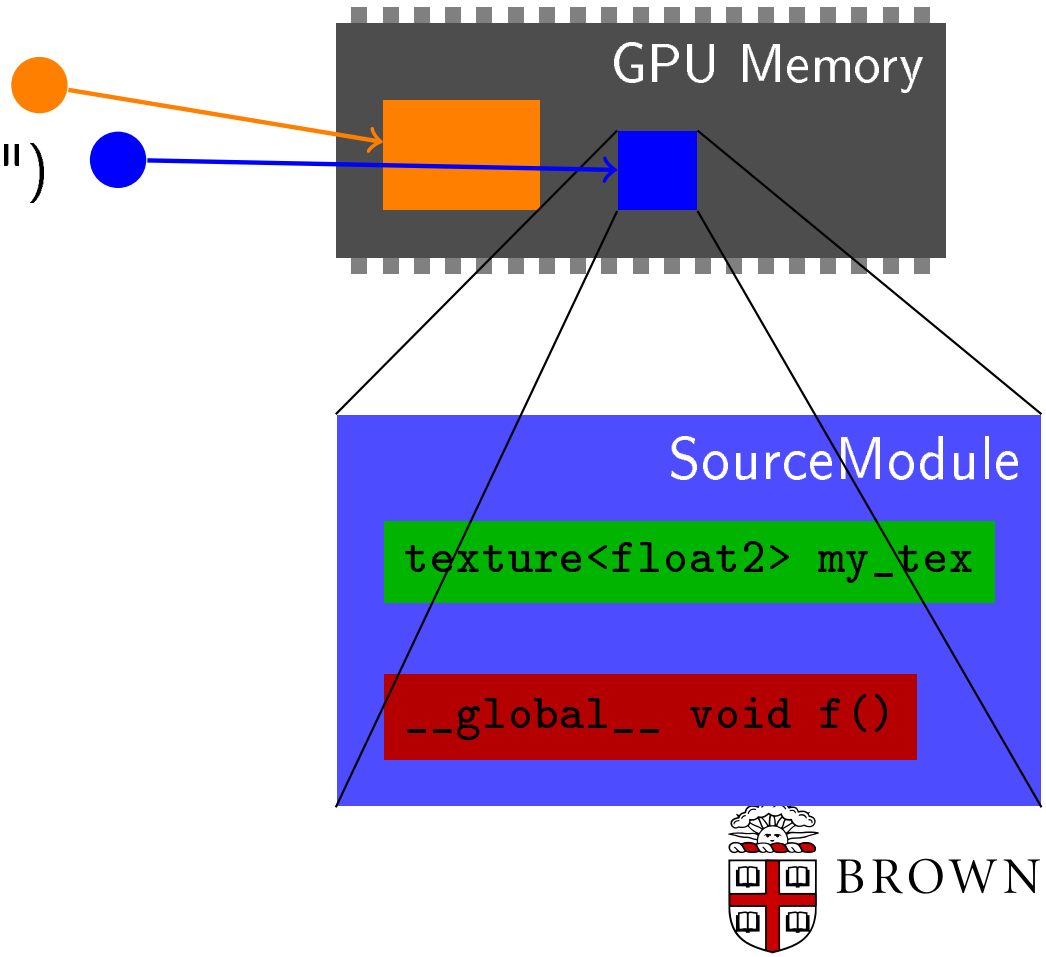
Working with Textures

```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
```



Working with Textures

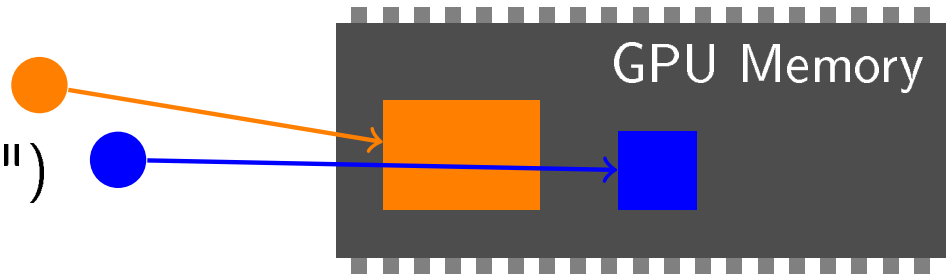
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
```



BROWN

Working with Textures

```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
```



SourceModule

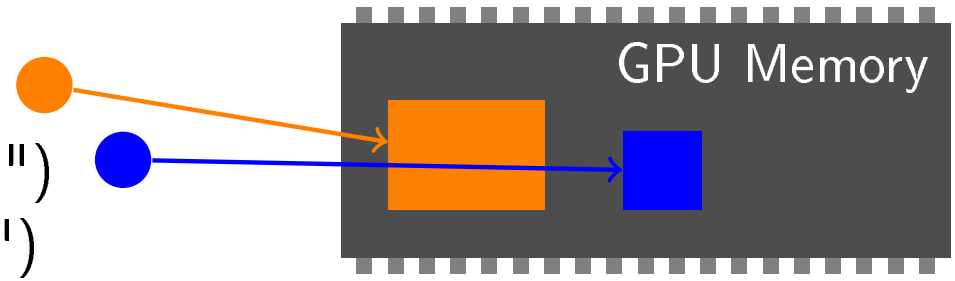
```
texture<float2> my_tex
```

```
__global__ void f()
```

BROWN

Working with Textures

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



SourceModule

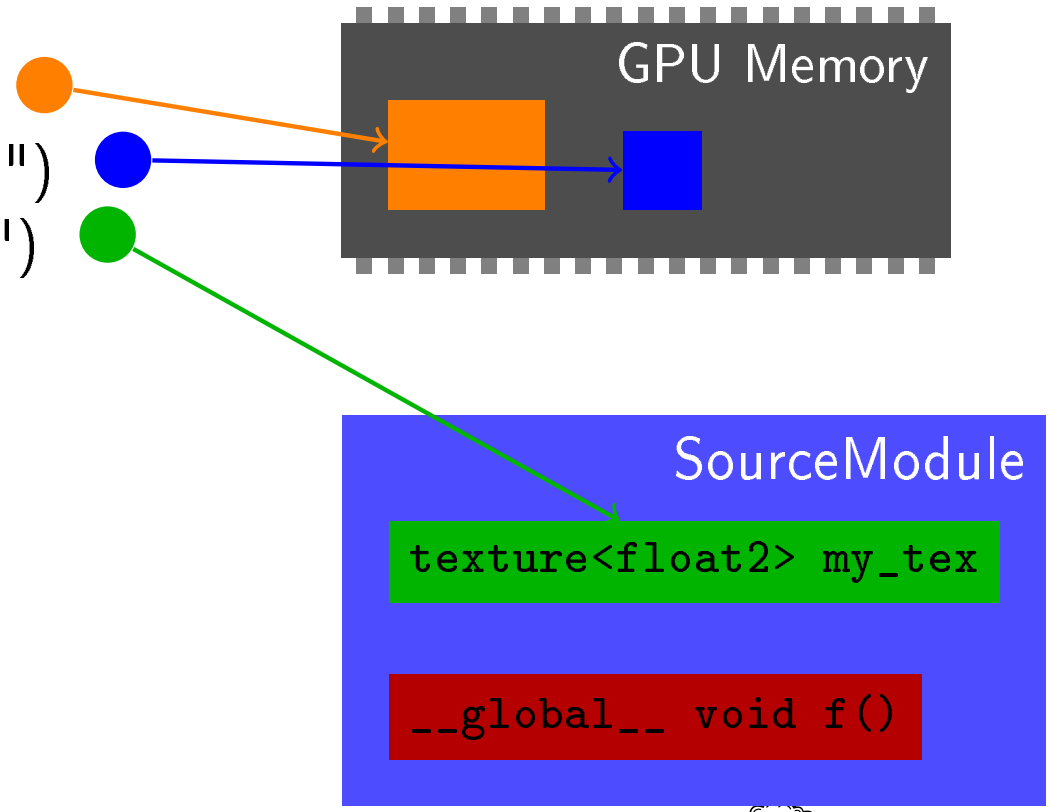
```
texture<float2> my_tex
```

```
__global__ void f()
```

BROWN

Working with Textures

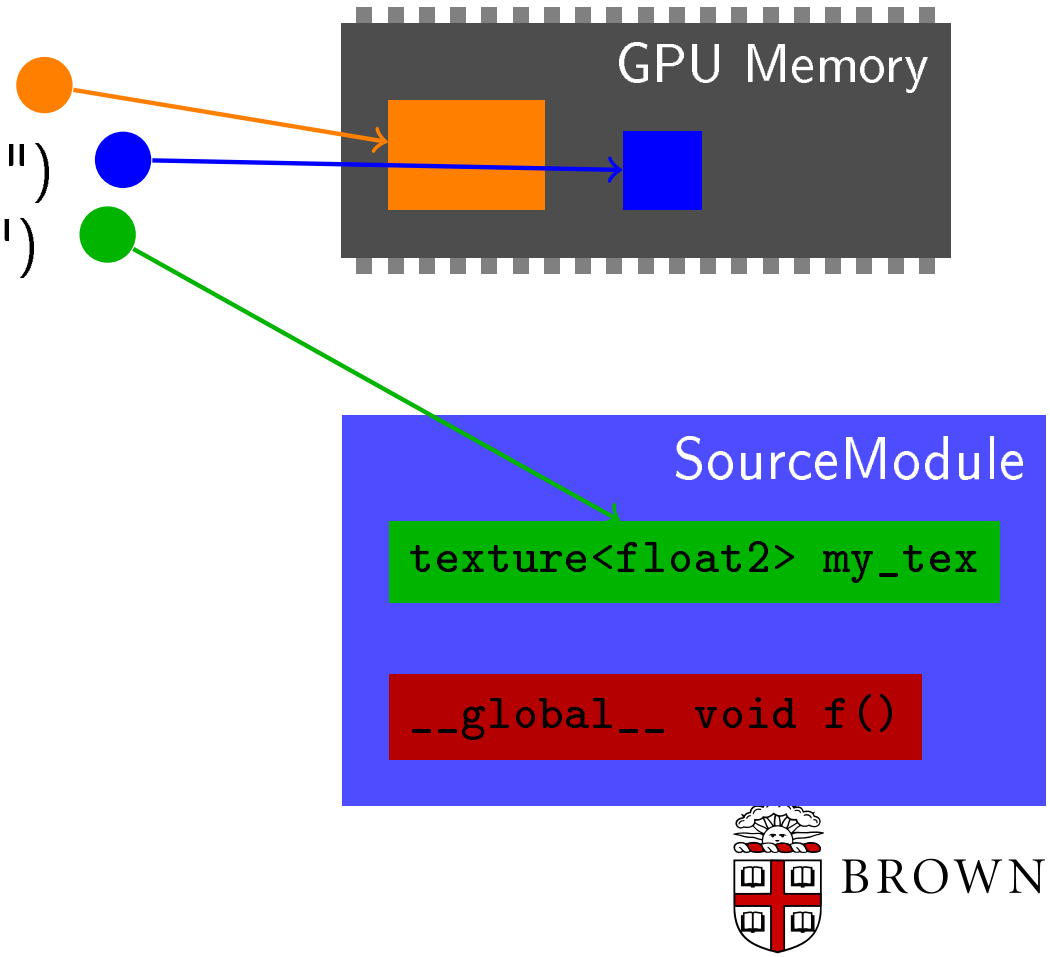
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
```



BROWN

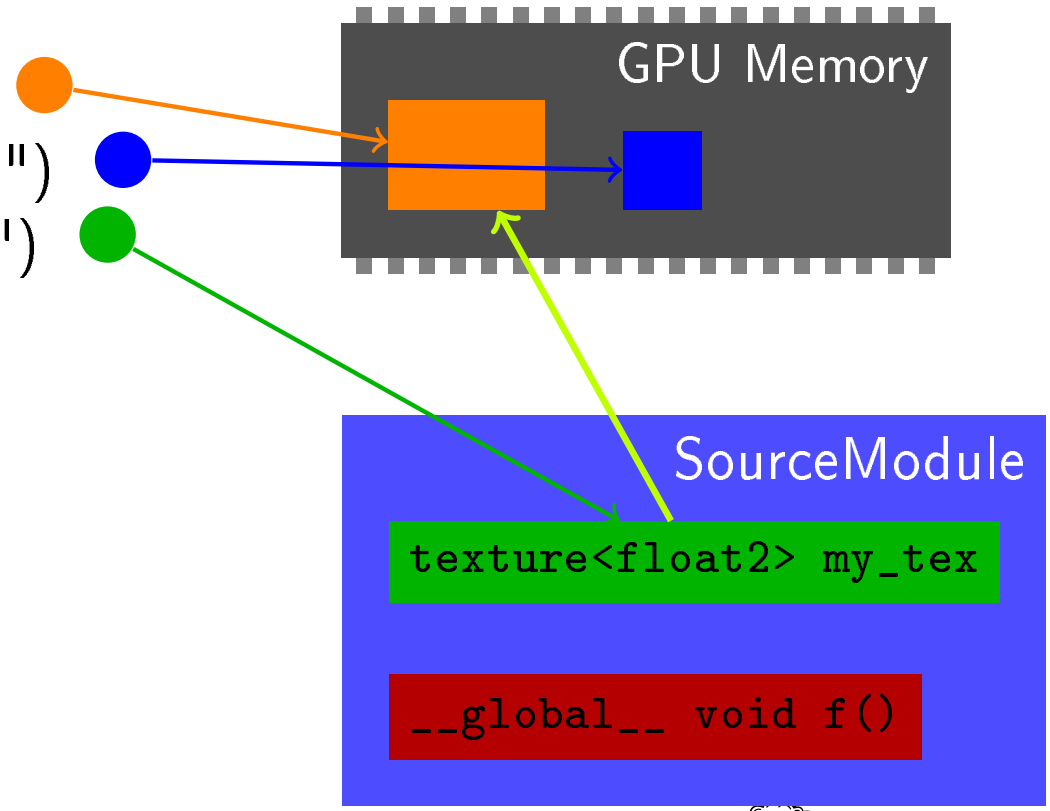
Working with Textures

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



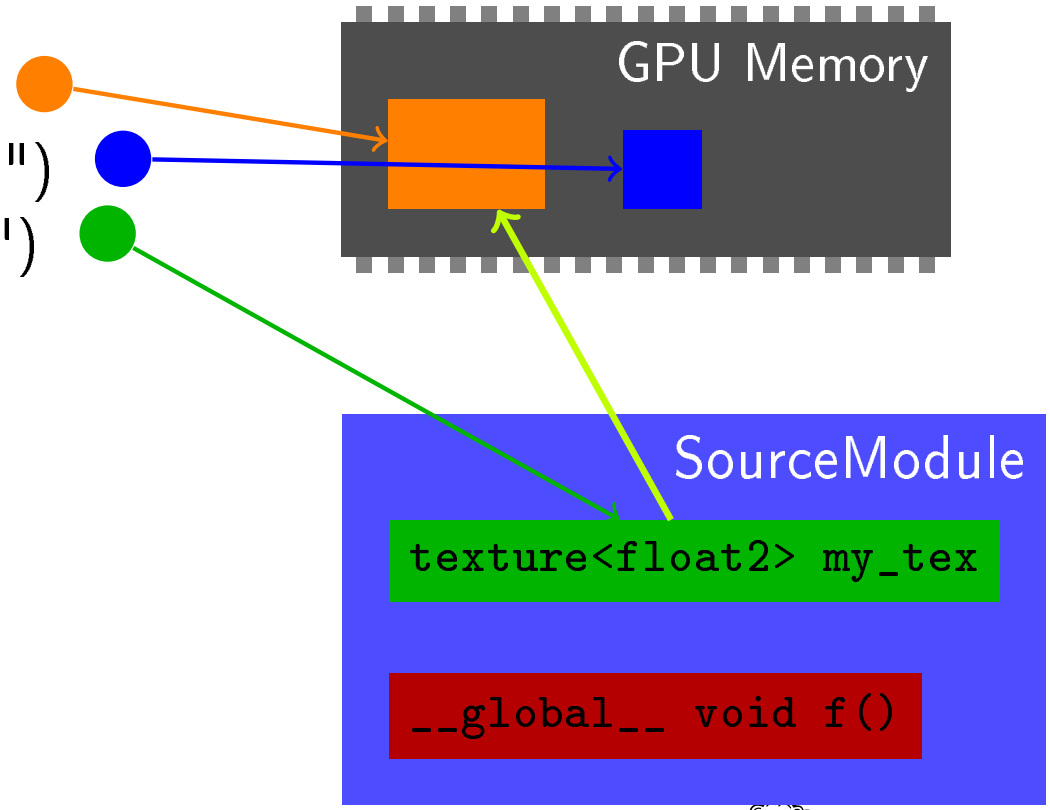
Working with Textures

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



Working with Textures

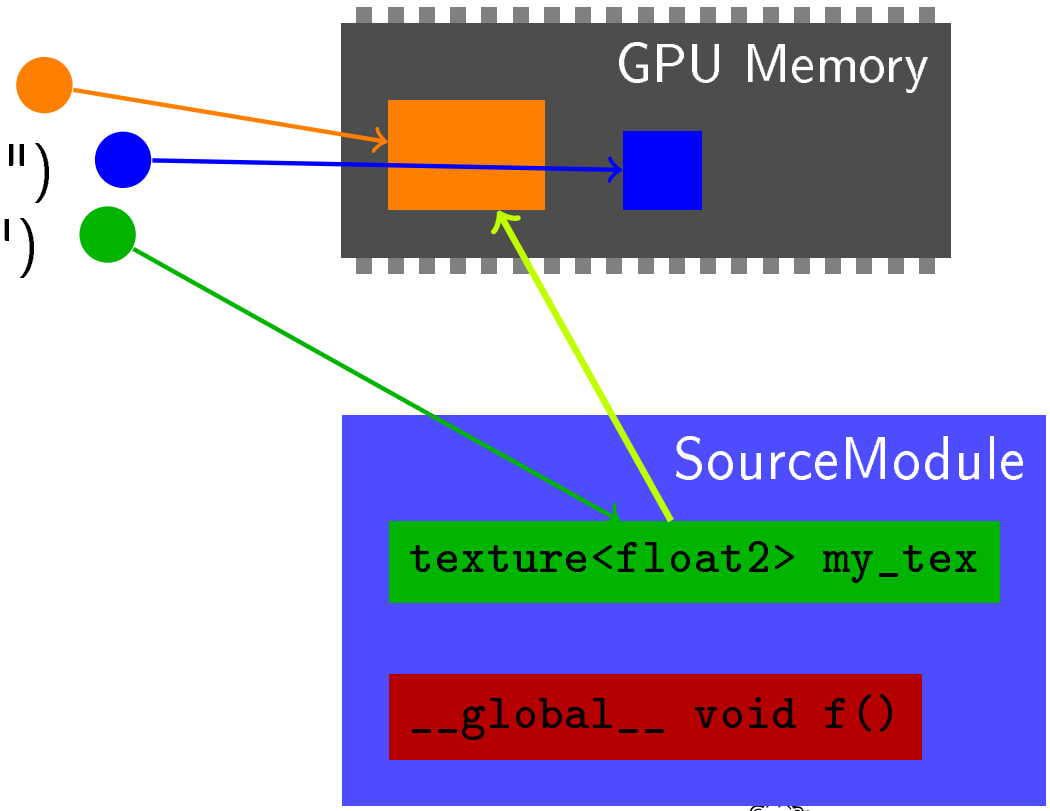
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)
```



BROWN

Working with Textures

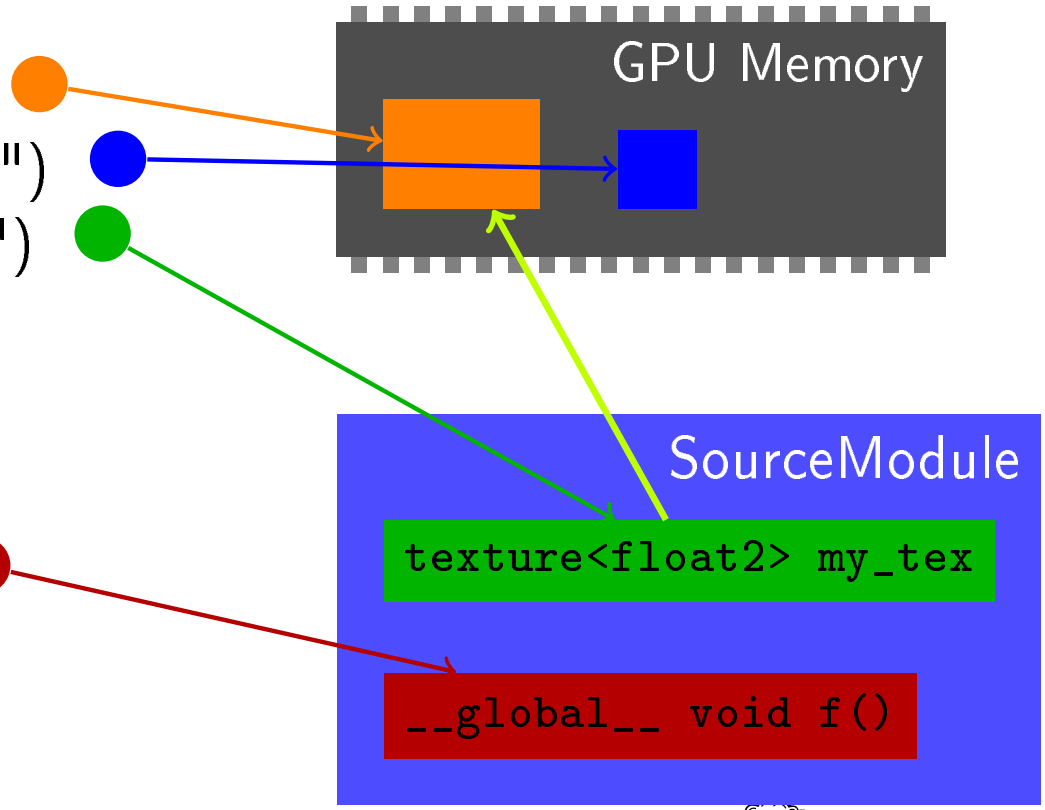
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)

f = mod.get_function("f")
```



BROWN

Working with Textures

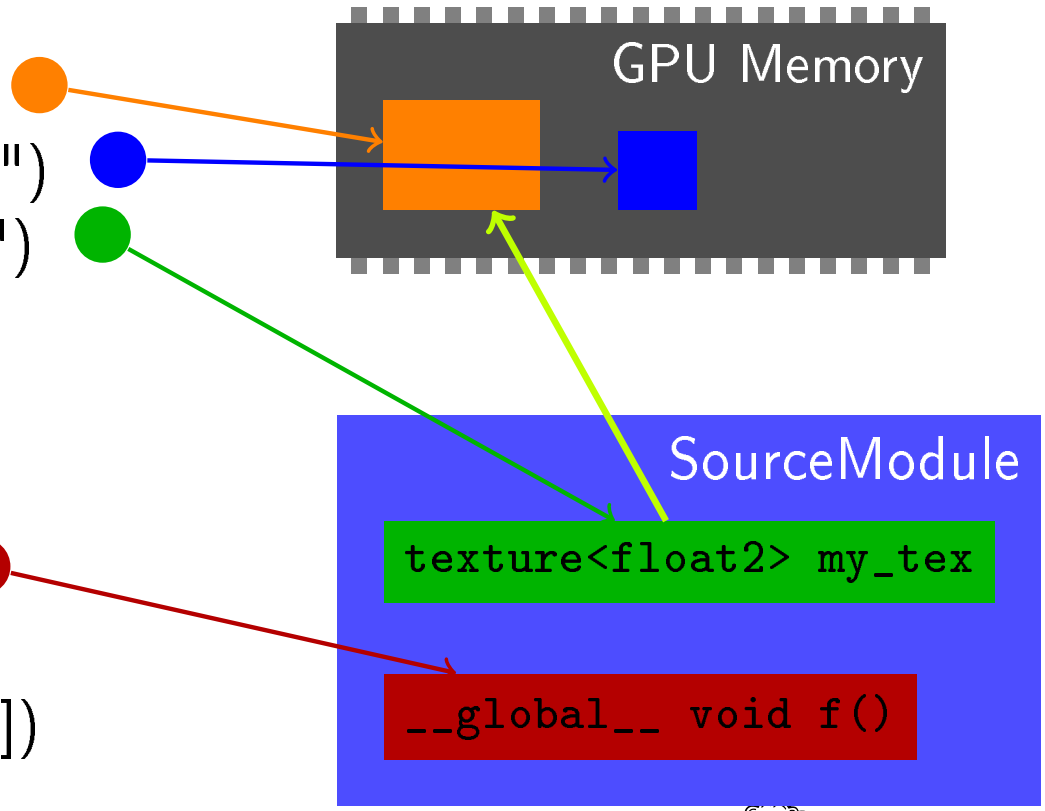
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)

f = mod.get_function("f")
```



BROWN

Working with Textures

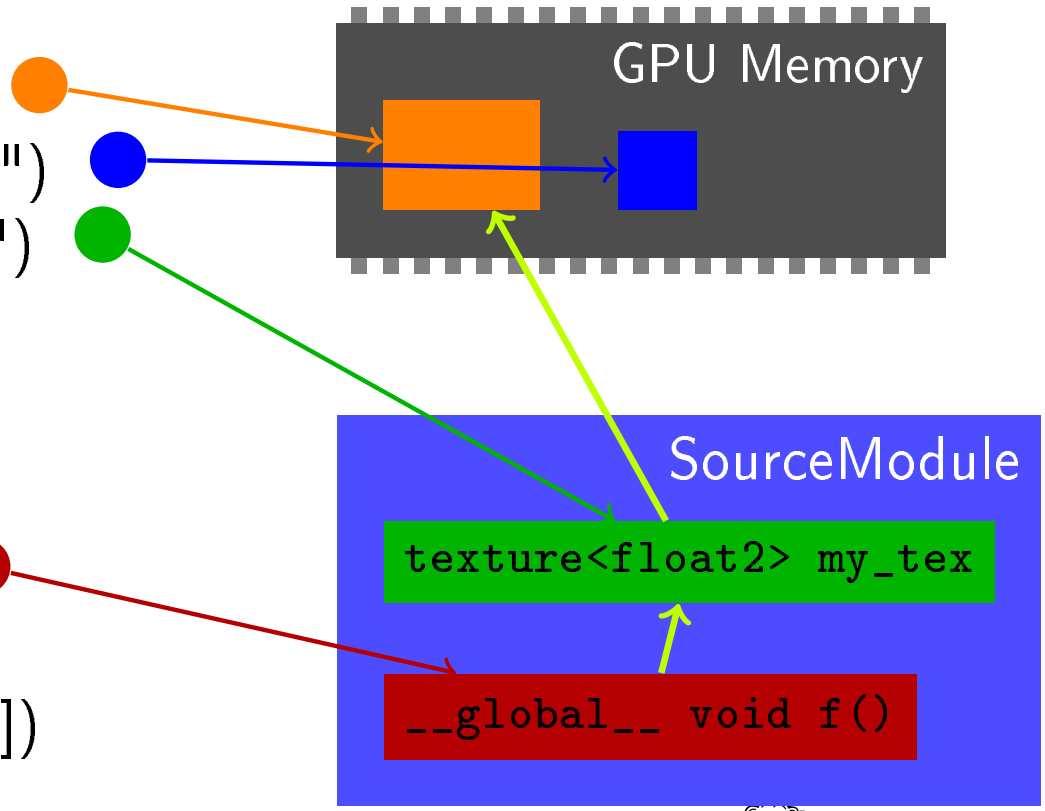
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)

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



BROWN

Working with Textures

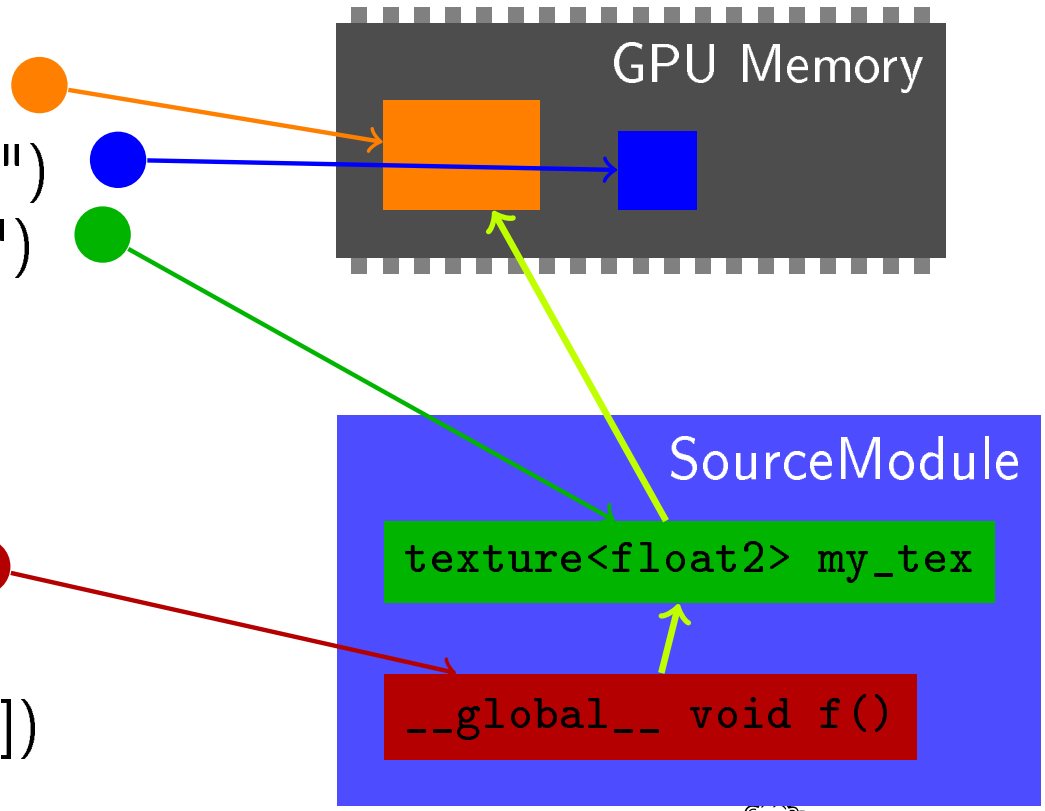
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)

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



BROWN

Working with Textures

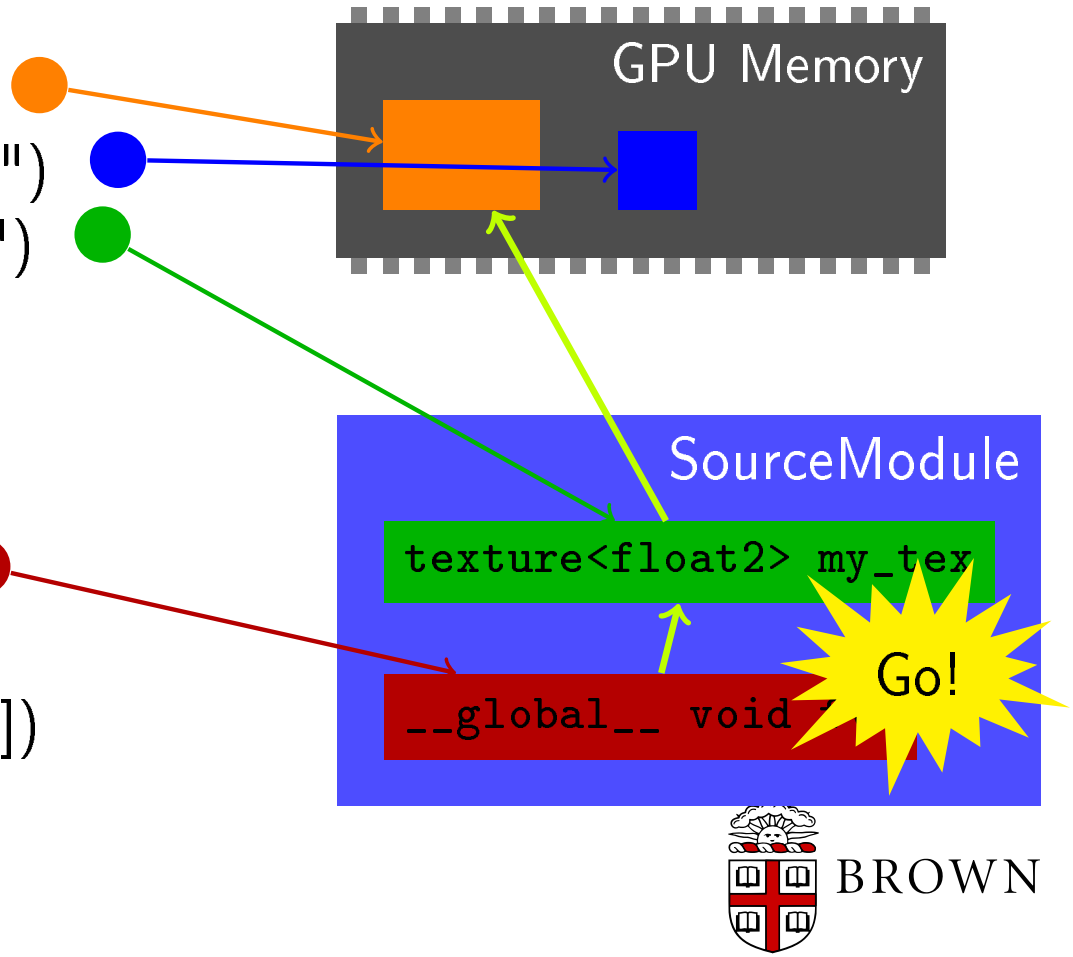
```
mem = cuda.mem_alloc(size)
mod = cuda.SourceModule("...")
tr = mod.get_texref("my_tex")
tr.set_address(mem, size)
tr.set_format(...float, 2)
tr.set_flags(...)

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



BROWN

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])
f()
```

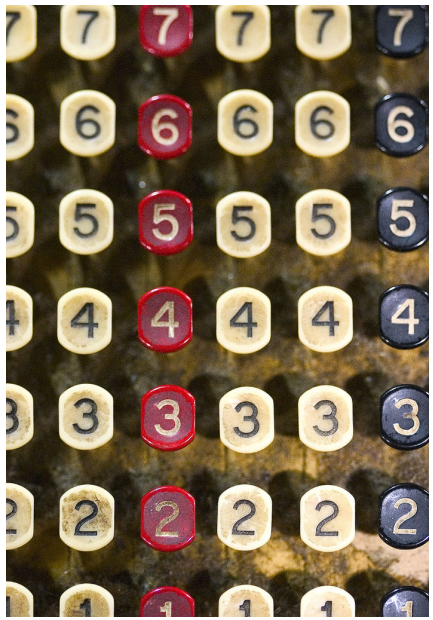


BROWN

gpuarray: Simple Linear Algebra

`pycuda.gpuarray`:

- Meant to look and feel just like `numpy`.

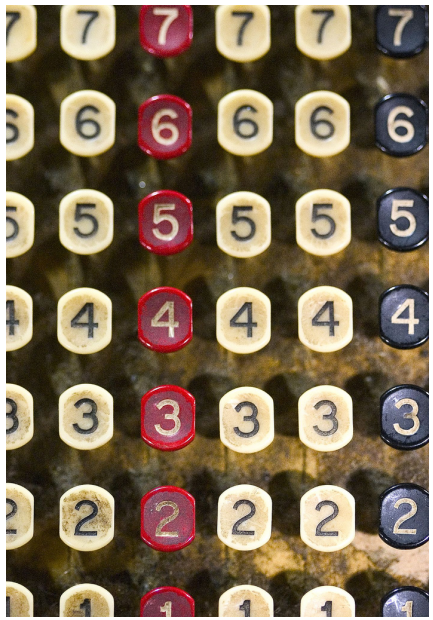


BROWN

gpuarray: Simple Linear Algebra

`pycuda.gpuarray`:

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

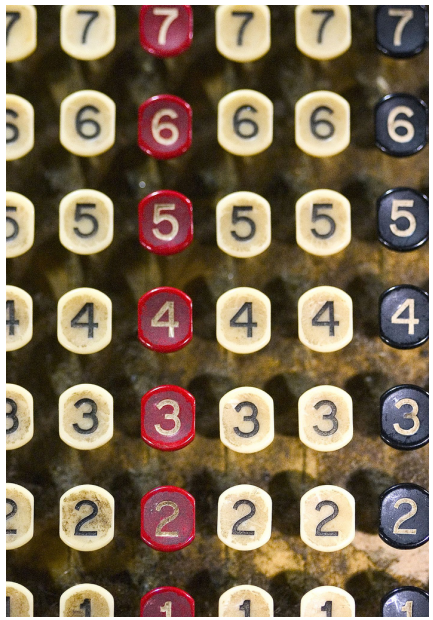


BROWN

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)

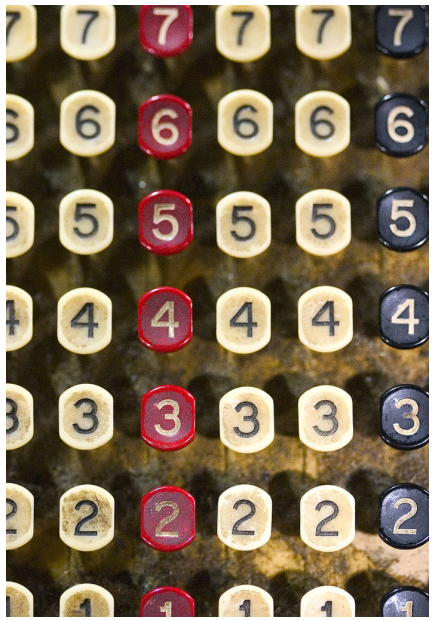


BROWN

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, ...

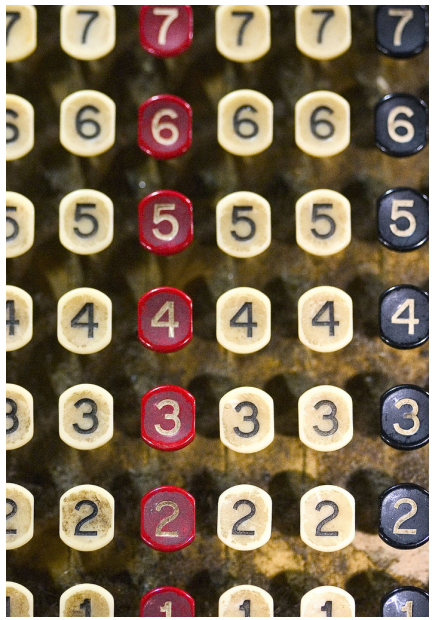


BROWN

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.

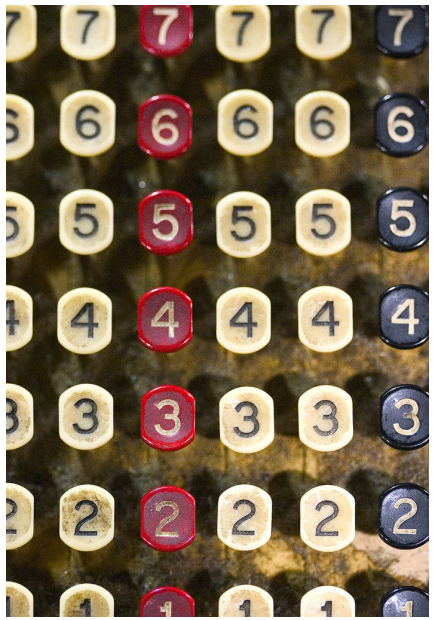


BROWN

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.

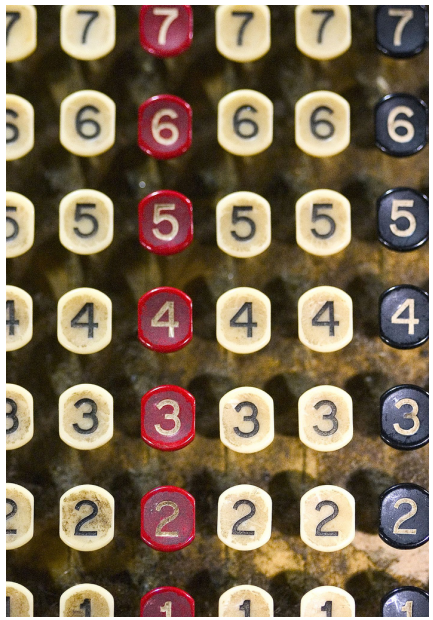


BROWN

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.
- Control concurrency through streams.



BROWN

PyCuda: Vital Information

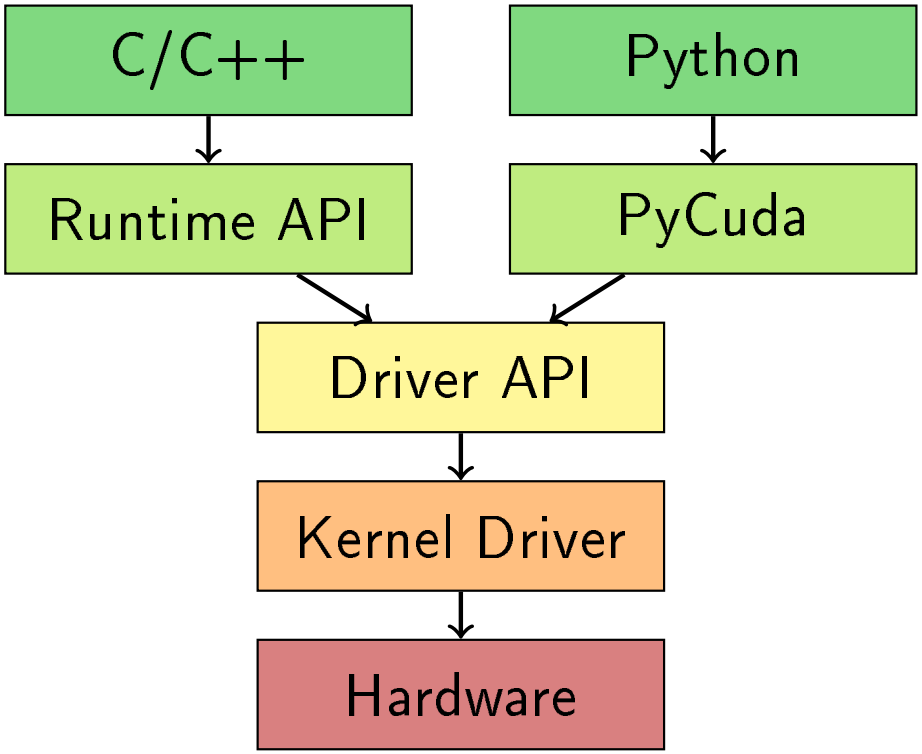
- `http://mathema.tician.de/software/pycuda`
- X Consortium License
  (no warranty, free for all use)
- Requires: numpy, Boost C++, Python 2.4+.
- Support via mailing list.

BROWN

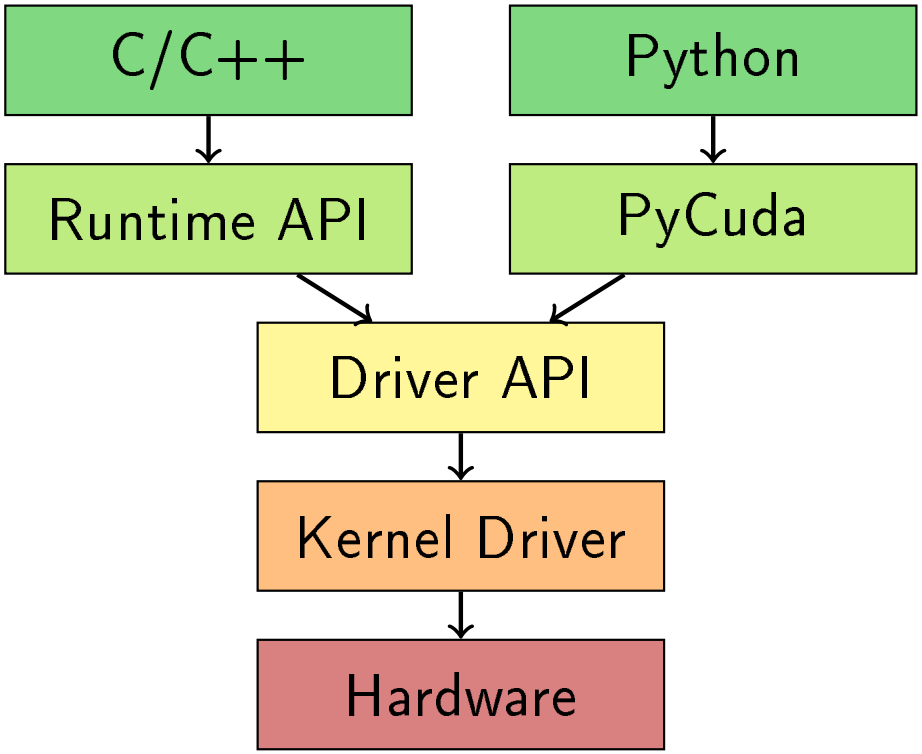
Outline

BROWN

CUDA APIs


C/C++
Python
Runtime API
PyCuda
Driver API
Kernel Driver
Hardware

CUDA APIs

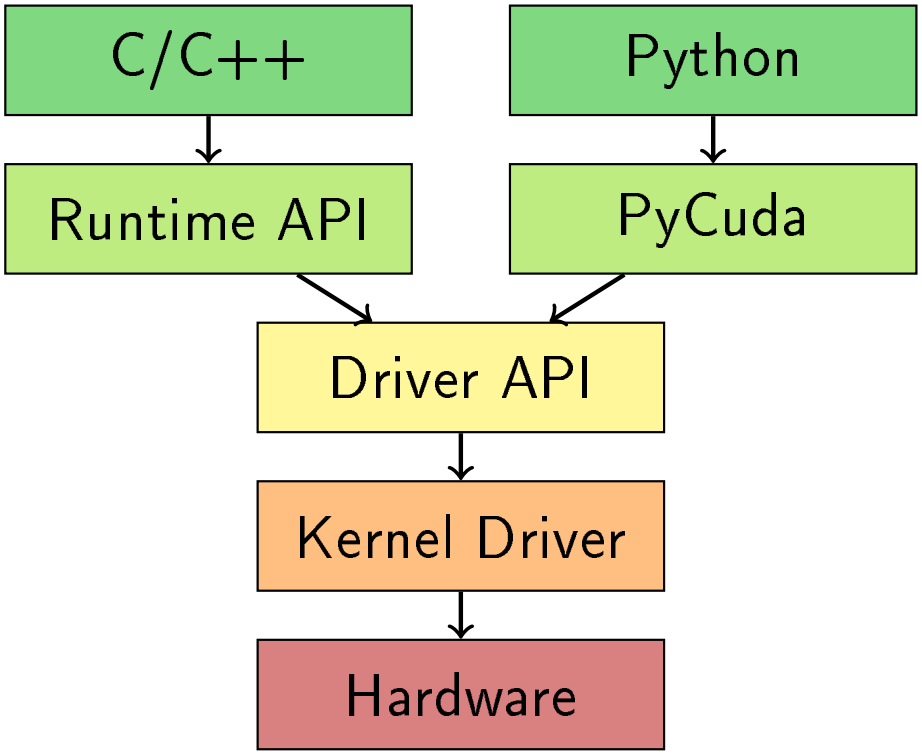


CUDA has two Programming Interfaces:

- "Runtime"
- "Driver"


BROWN

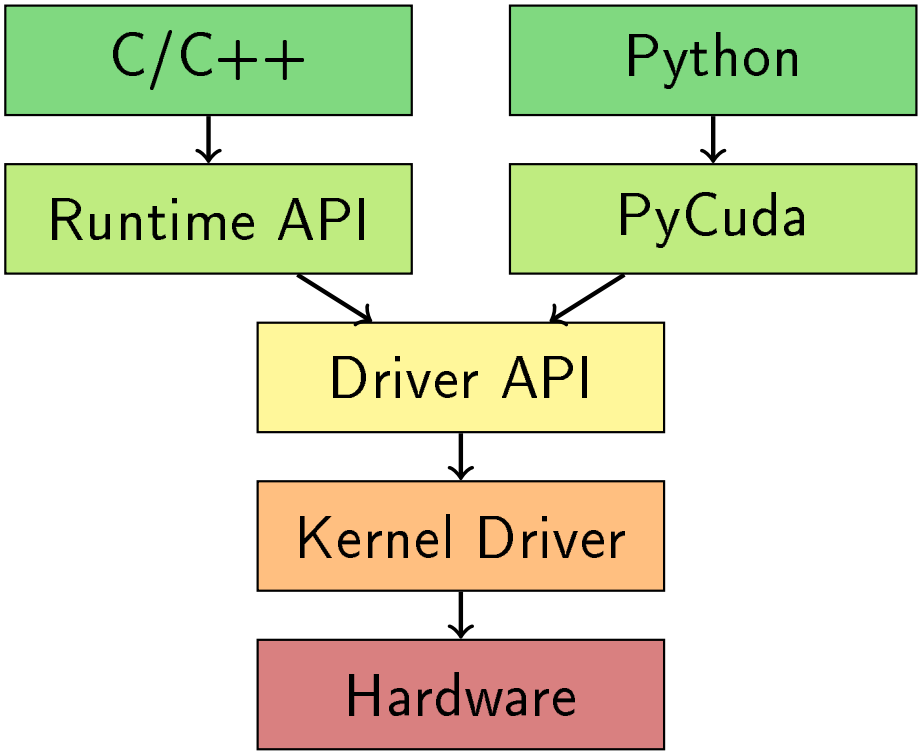
CUDA APIs



CUDA has two Programming Interfaces:

- "Runtime" high-level
- "Driver" low-level

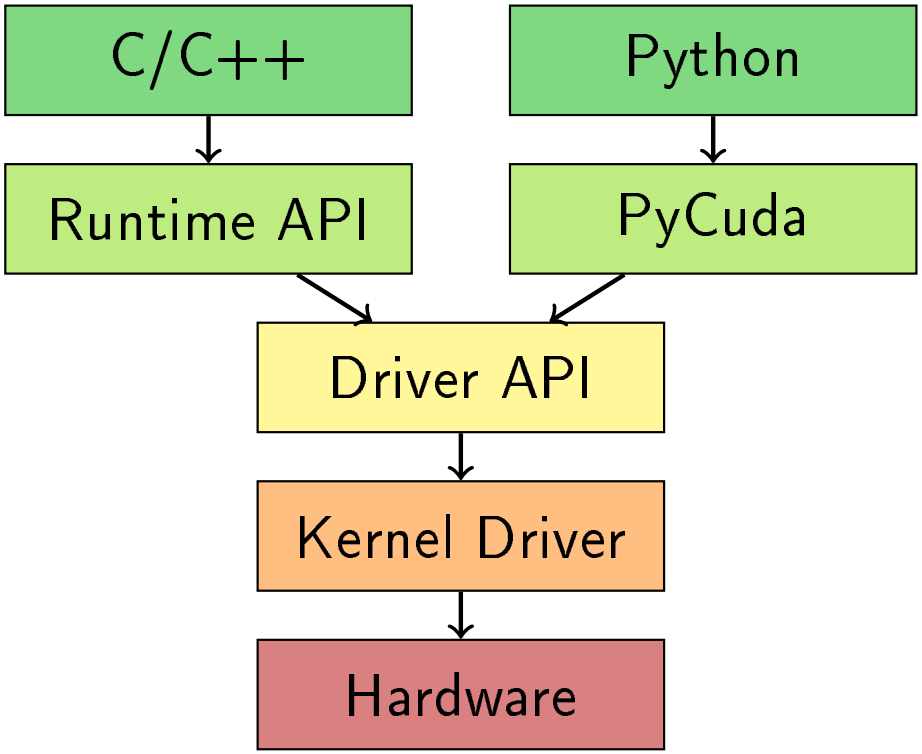
CUDA APIs



CUDA has two Programming Interfaces:

- "Runtime" high-level (`libcudart.so`, in the "toolkit")
- "Driver" low-level

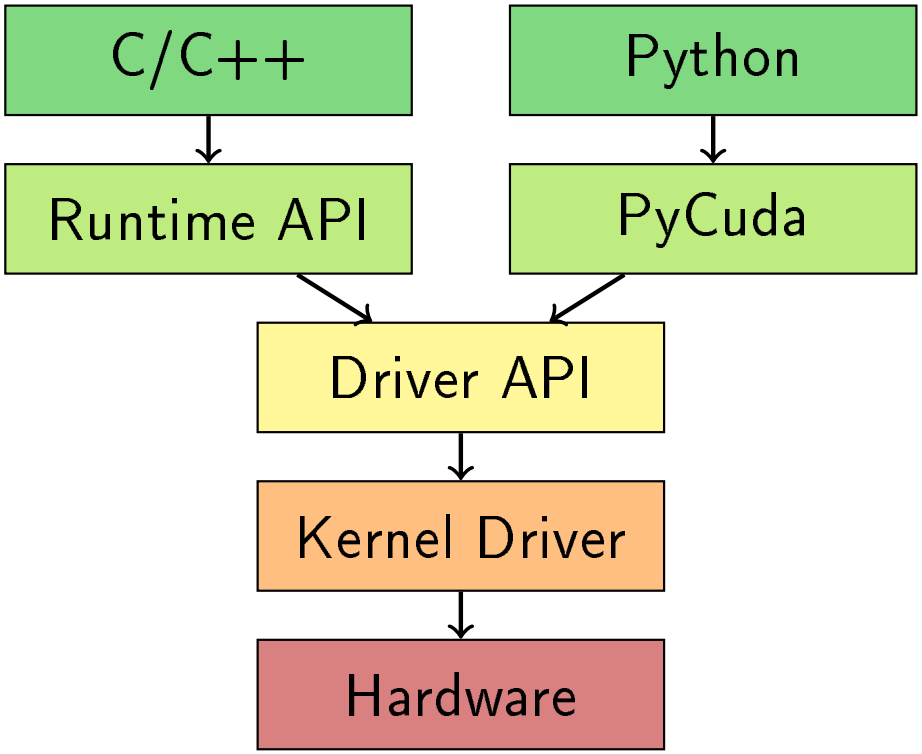
CUDA APIs



CUDA has two Programming Interfaces:

- "Runtime" high-level (`libcudart.so`, in the "toolkit")
- "Driver" low-level (`libcuda.so`, comes with GPU driver)

CUDA APIs



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 $\leftrightarrow$ Driver differences:

- Explicit initialization.

Runtime vs. Driver API

Runtime $\leftrightarrow$ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.

BROWN

Runtime vs. Driver API

Runtime $\leftrightarrow$ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.
- Texture handling requires slightly more work.

BROWN

Runtime vs. Driver API

Runtime $\leftrightarrow$ Driver differences:

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

BROWN

Runtime vs. Driver API

Runtime $\leftrightarrow$ 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`:

```
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
```

```
cuInit
cuDeviceGetCount
cuDeviceGet
cuCtxCreate
cuMemAlloc
cuMemcpyHtoD
cuCtxGetDevice
cuDeviceComputeCapability
cuModuleLoadData
cuModuleGetFunction
cuFuncSetBlockShape
cuParamSetv
cuParamSetSize
cuLaunchGrid
cuMemcpyDtoH
cuCtxPopCurrent
cuCtxPushCurrent
cuMemFree
cuCtxPopCurrent
cuCtxPushCurrent
cuModuleUnload
cuCtxPopCurrent
cuCtxDestroy
```

BROWN

Questions?

?

BROWN

Outline

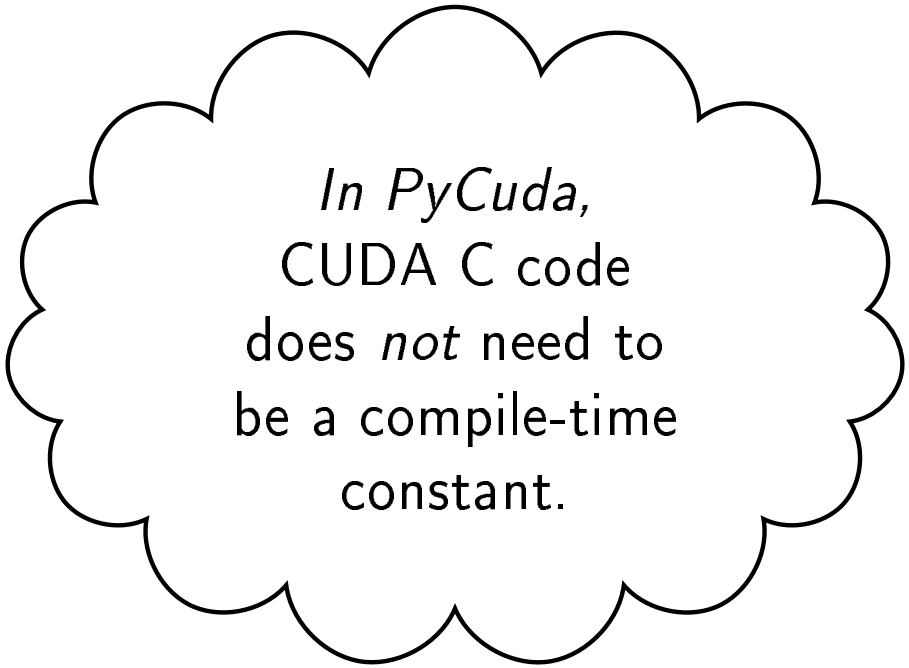
BROWN

Metaprogramming

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

BROWN

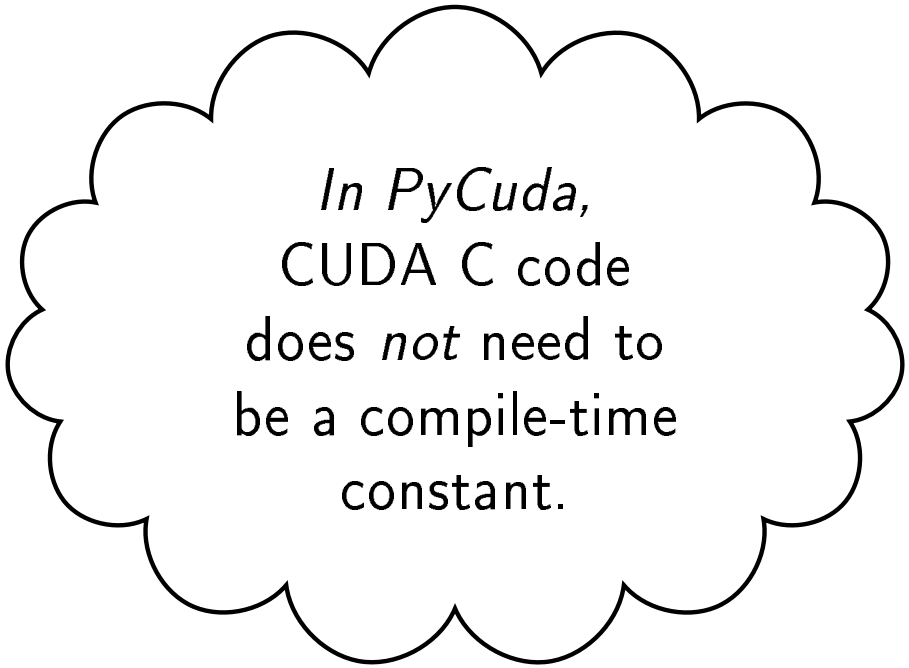
Metaprogramming



(unlike the CUDA Runtime API)

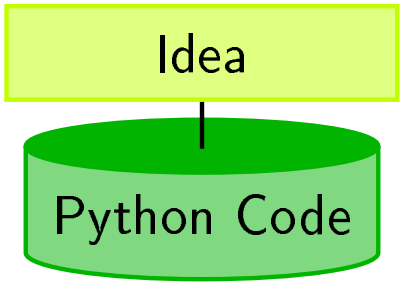
Metaprogramming

Idea



(unlike the CUDA Runtime API)

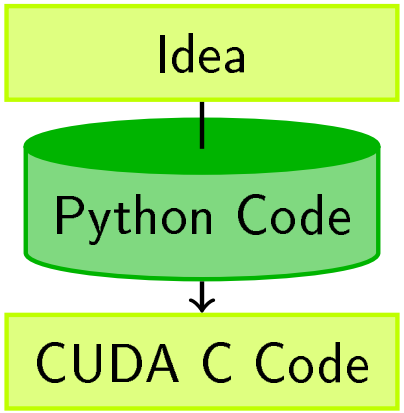
Metaprogramming



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

(unlike the CUDA Runtime API)

Metaprogramming

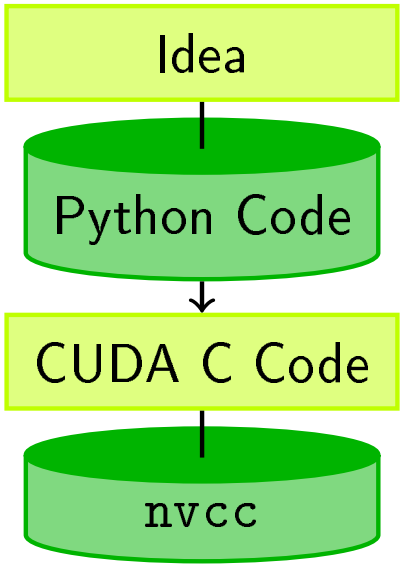


In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)


BROWN

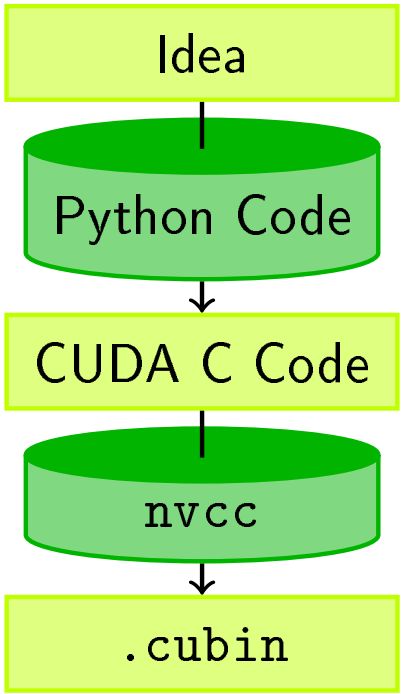
Metaprogramming



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

(unlike the CUDA Runtime API)

Metaprogramming

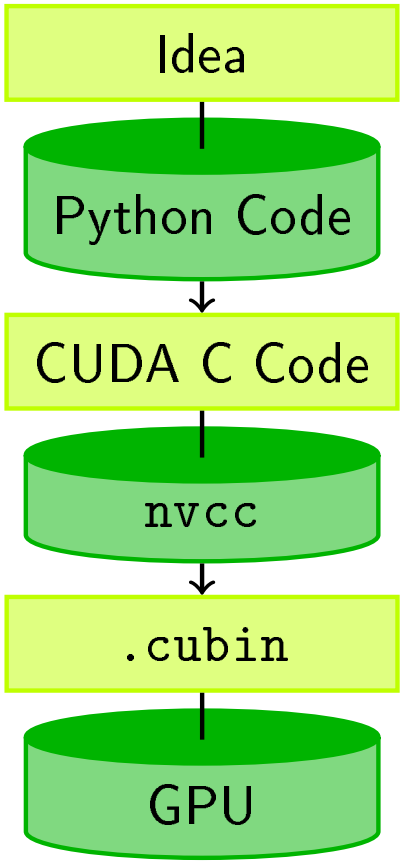


In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)


BROWN

Metaprogramming

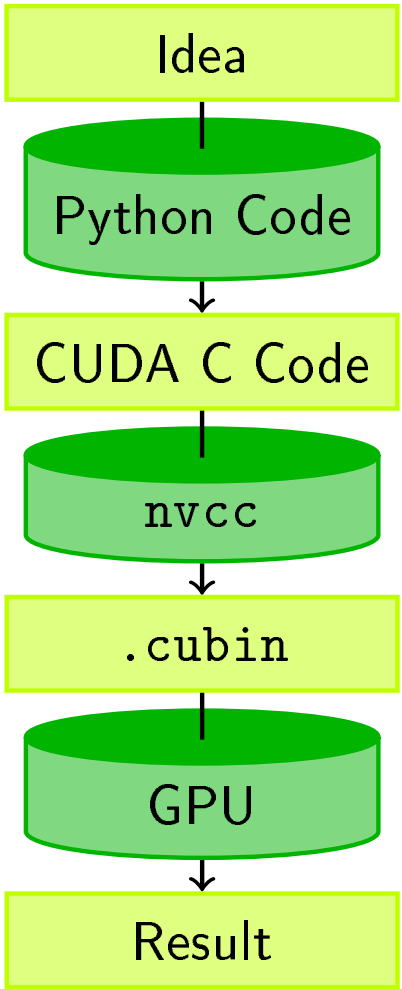


In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)


BROWN

Metaprogramming

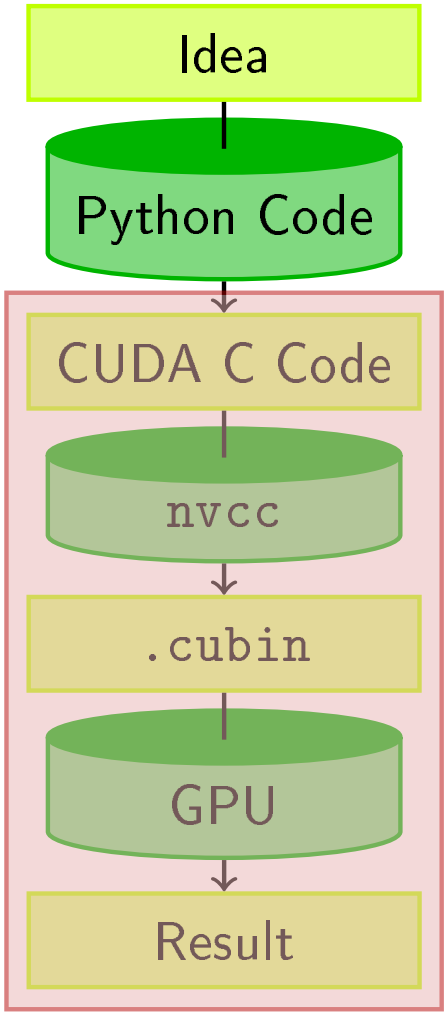


In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)


BROWN

Metaprogramming



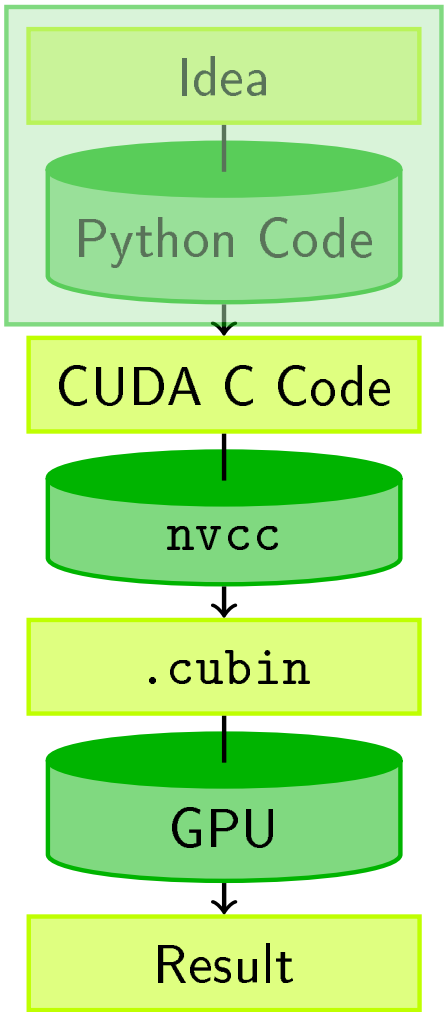
Machine

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

(unlike the CUDA Runtime API)

BROWN

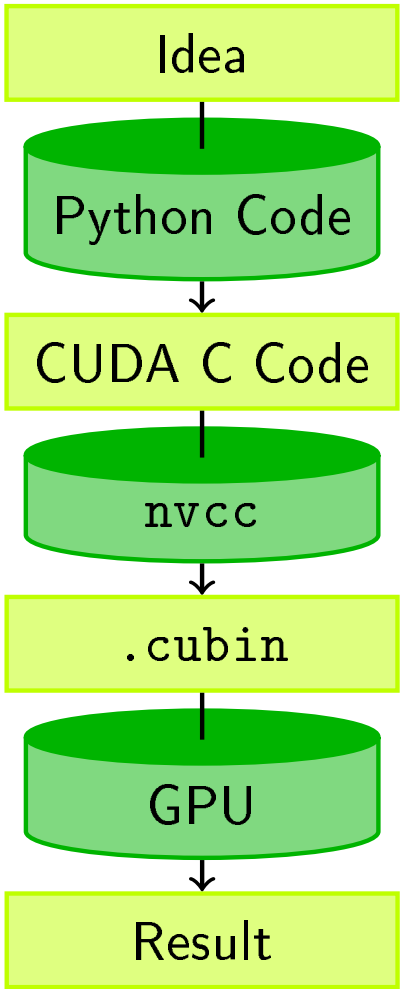
Metaprogramming



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

(unlike the CUDA Runtime API)

Metaprogramming



Easy to write

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

(unlike the CUDA Runtime API)

BROWN

Machine-generated Code

Why machine-generate code?

- Automated Tuning
  (cf. ATLAS, FFTW)

BROWN

Machine-generated Code

Why machine-generate code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types

BROWN

Machine-generated Code

Why machine-generate code?

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

BROWN

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)

BROWN

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

BROWN

PyCuda: Support for Metaprogramming

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

PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  `func.{registers,lmem,smem}`
- Exact GPU timing via events

BROWN

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:
  `func.{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?

?

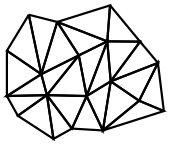
BROWN

Outline

BROWN

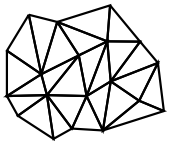
Discontinuous Galerkin Method

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



Discontinuous Galerkin Method

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

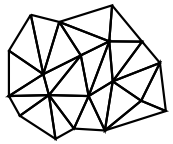


Goal

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

Discontinuous Galerkin Method

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



Goal

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

Example

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

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

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

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

$$0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx$$

BROWN

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

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

BROWN

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

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

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

BROWN

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

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

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

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

BROWN

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

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

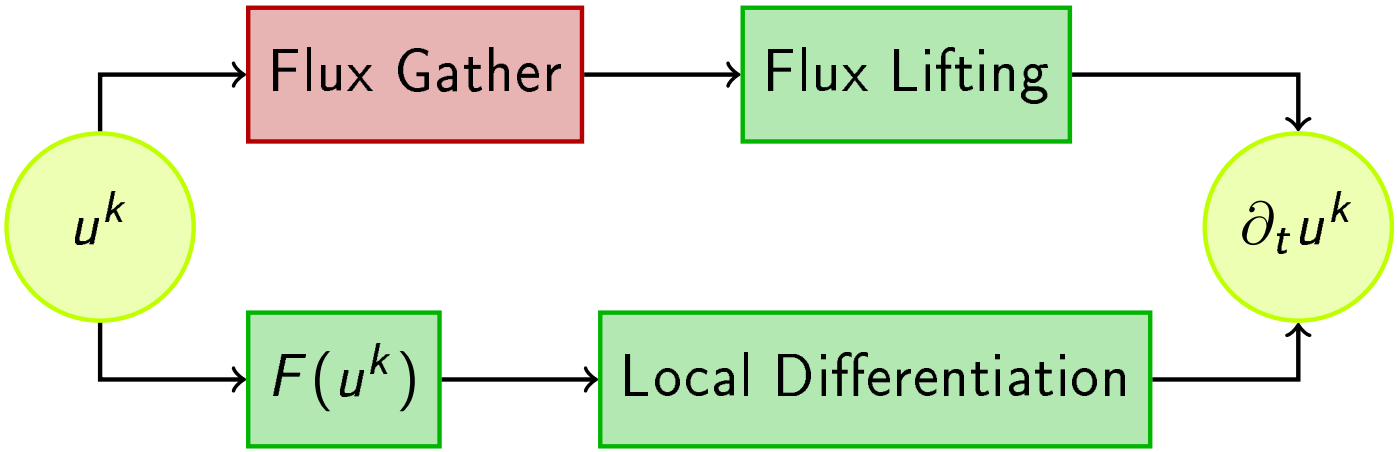
Integrate by parts again, subsitute in basis functions, introduce elementwise differentiation and "lifting" matrices D, L:

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

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

Decomposition of a DG operator into Subtasks

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



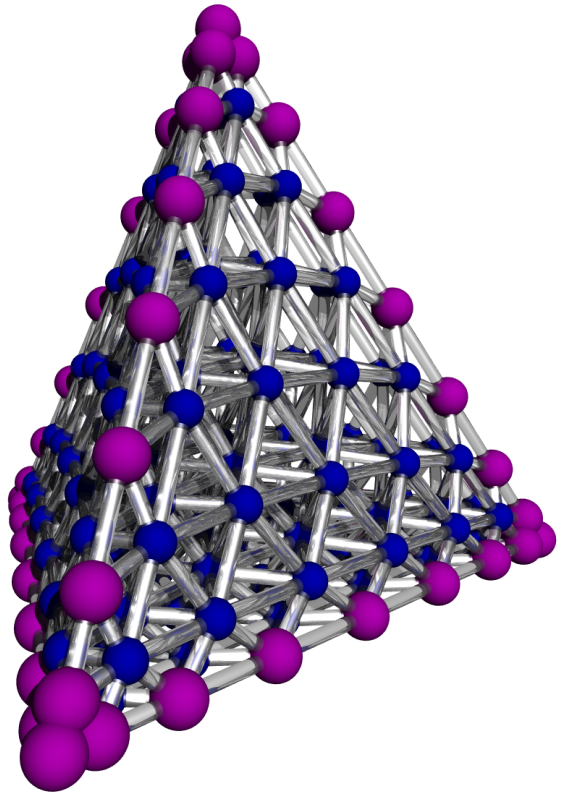
Green: Element-local parts of the DG operator.

Note: Explicit timestepping.

DG: Properties

Flexible:

- Variable order of accuracy

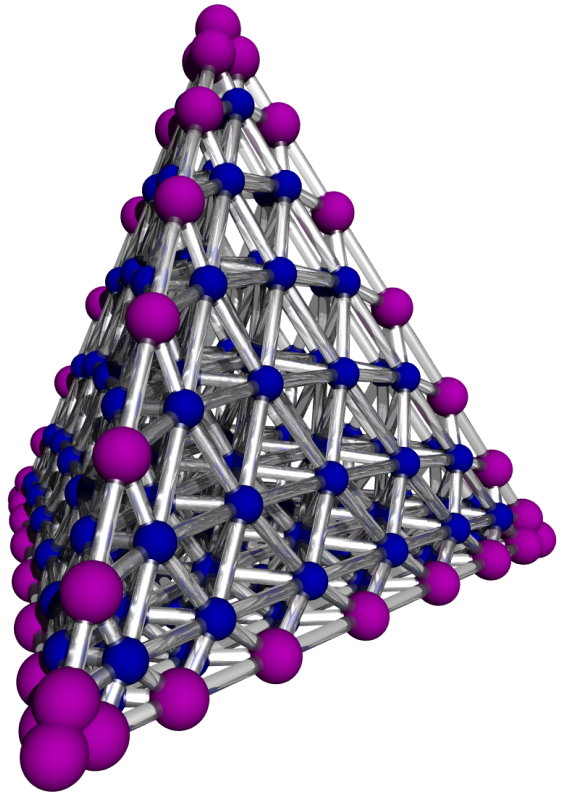


BROWN

DG: Properties

Flexible:

- Variable order of accuracy
- Unstructured discretizations

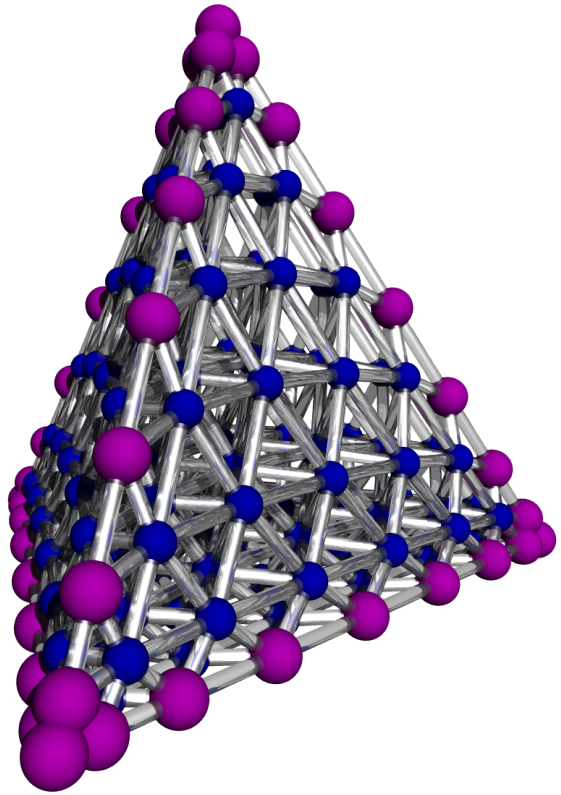


BROWN

DG: Properties

Flexible:

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



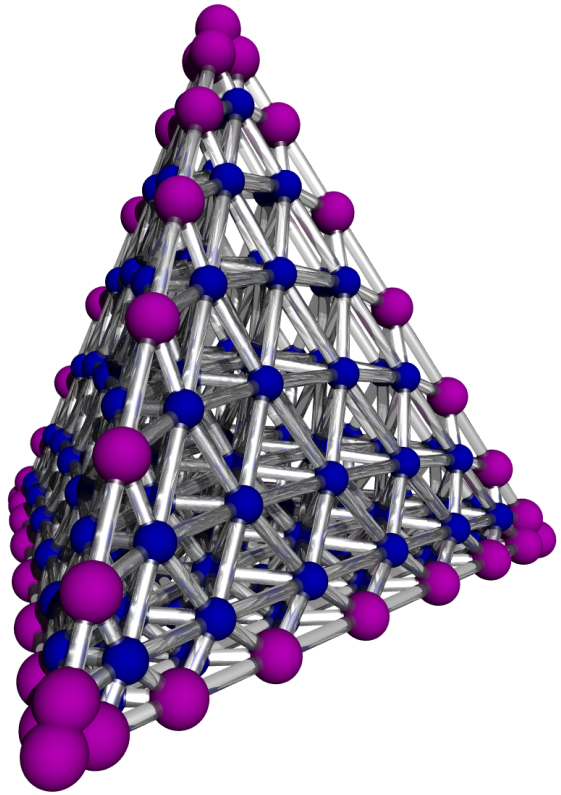
BROWN

DG: Properties

Flexible:

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

Implementation-friendly:



BROWN

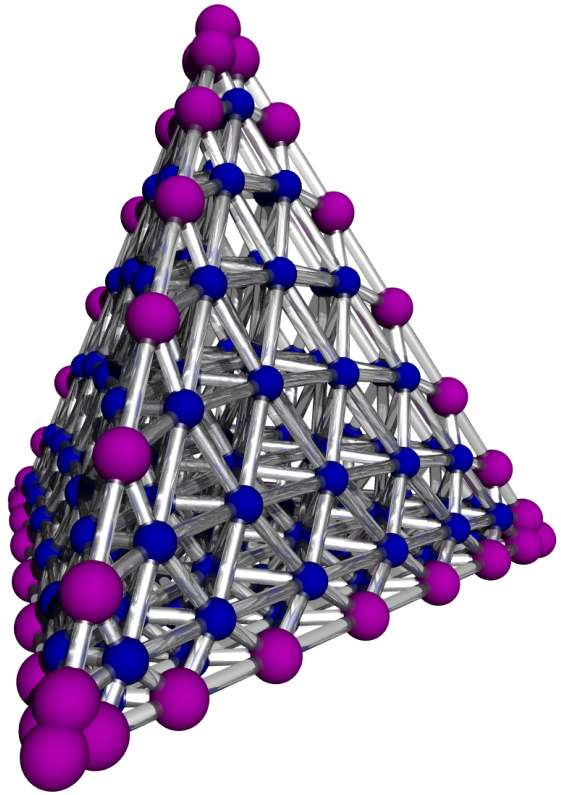
DG: Properties

Flexible:

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

Implementation-friendly:

- Good stability properties



BROWN

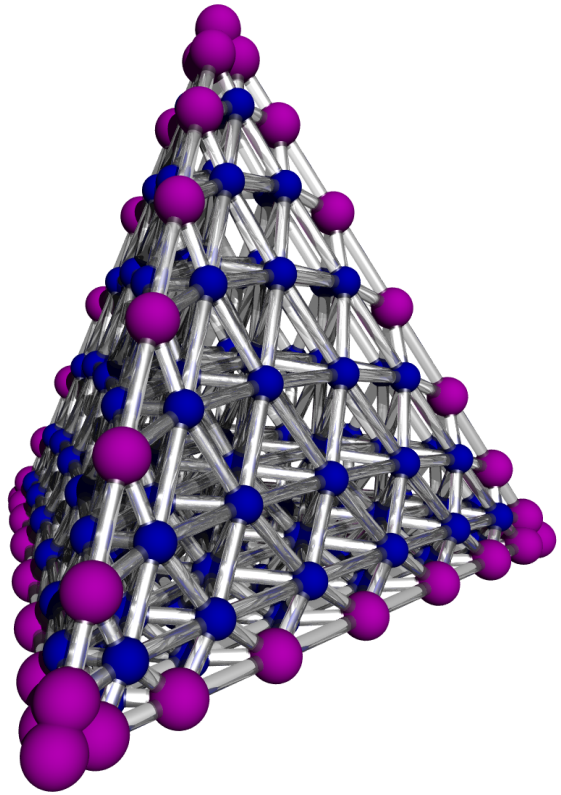
DG: Properties

Flexible:

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

Implementation-friendly:

- Good stability properties
- Parallelizes well



BROWN

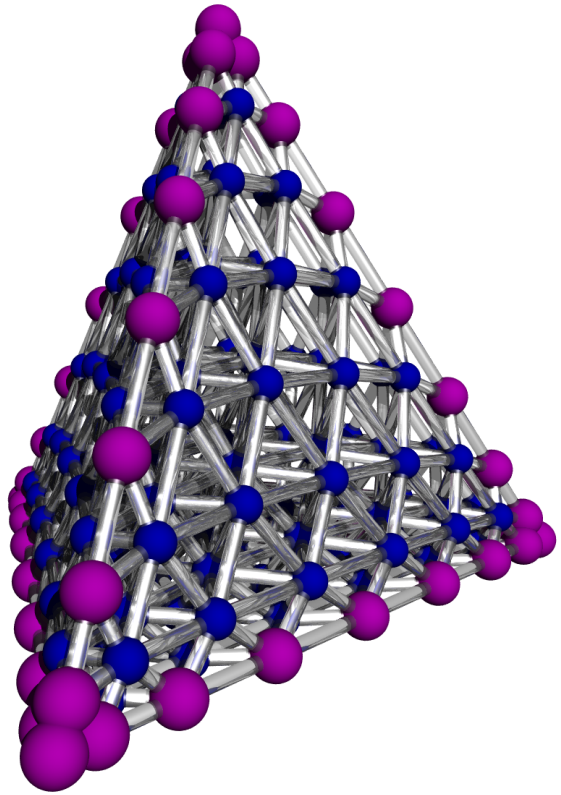
DG: Properties

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)



BROWN

Why do DG on Graphics Cards?

DG on GPUs: Why?

BROWN

Why do DG on Graphics Cards?

DG on GPUs: Why?

- GPUs have deep Memory Hierarchy
  - The majority of DG is local.

BROWN

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.

BROWN

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.

BROWN

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.

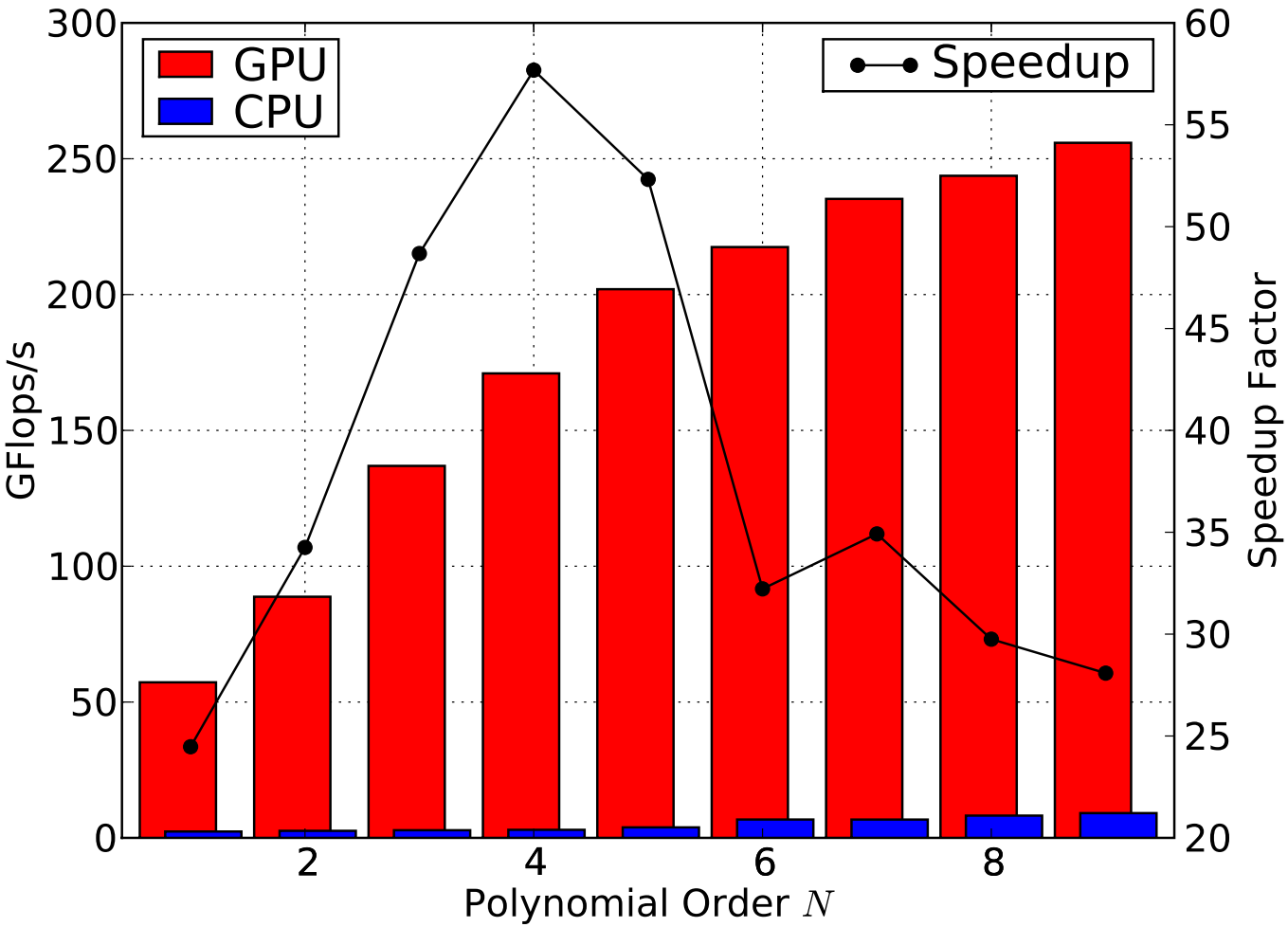
"A match made in heaven?"

BROWN

Outline

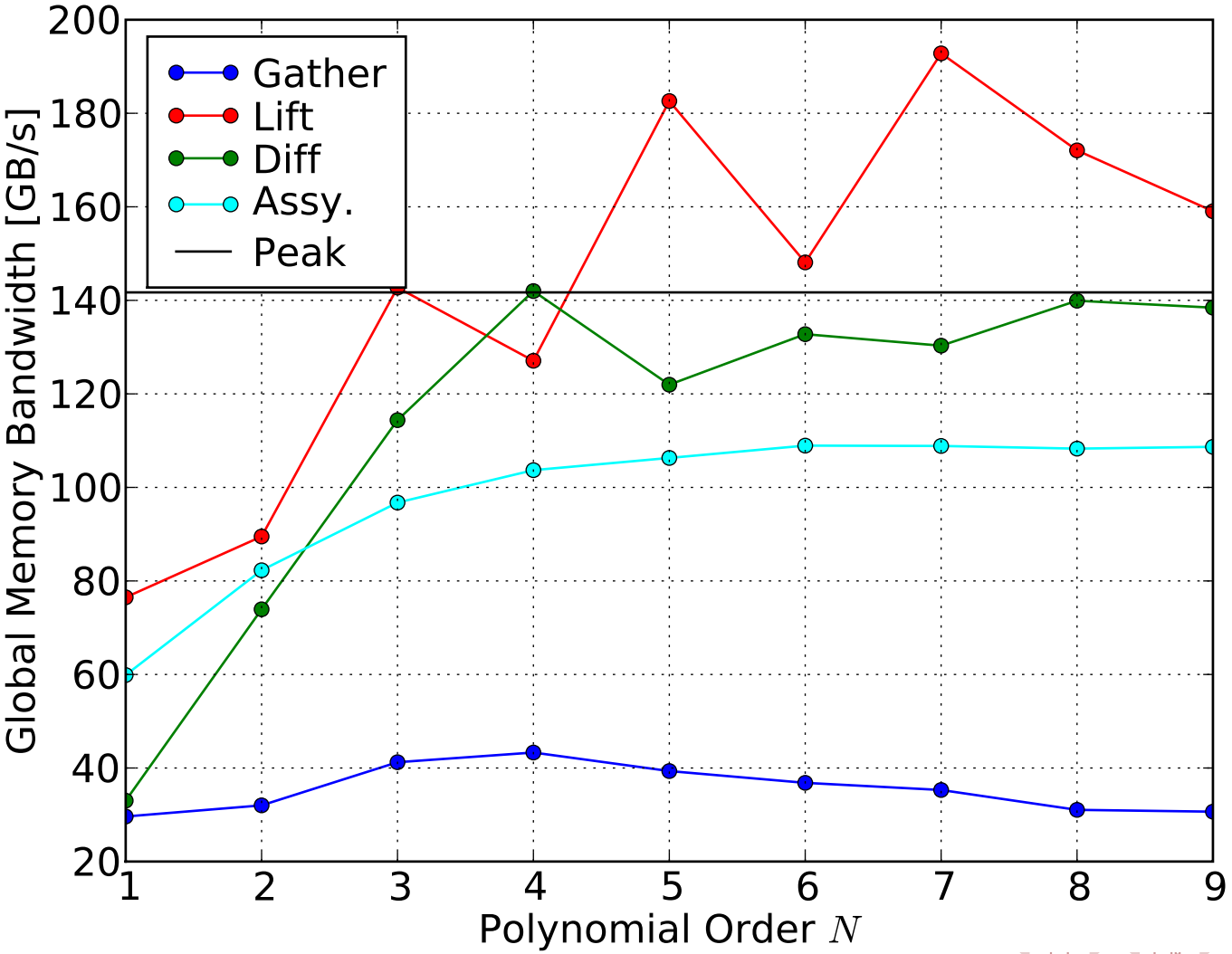
BROWN

GTX280 vs. single core of Intel Core 2 Duo E8400



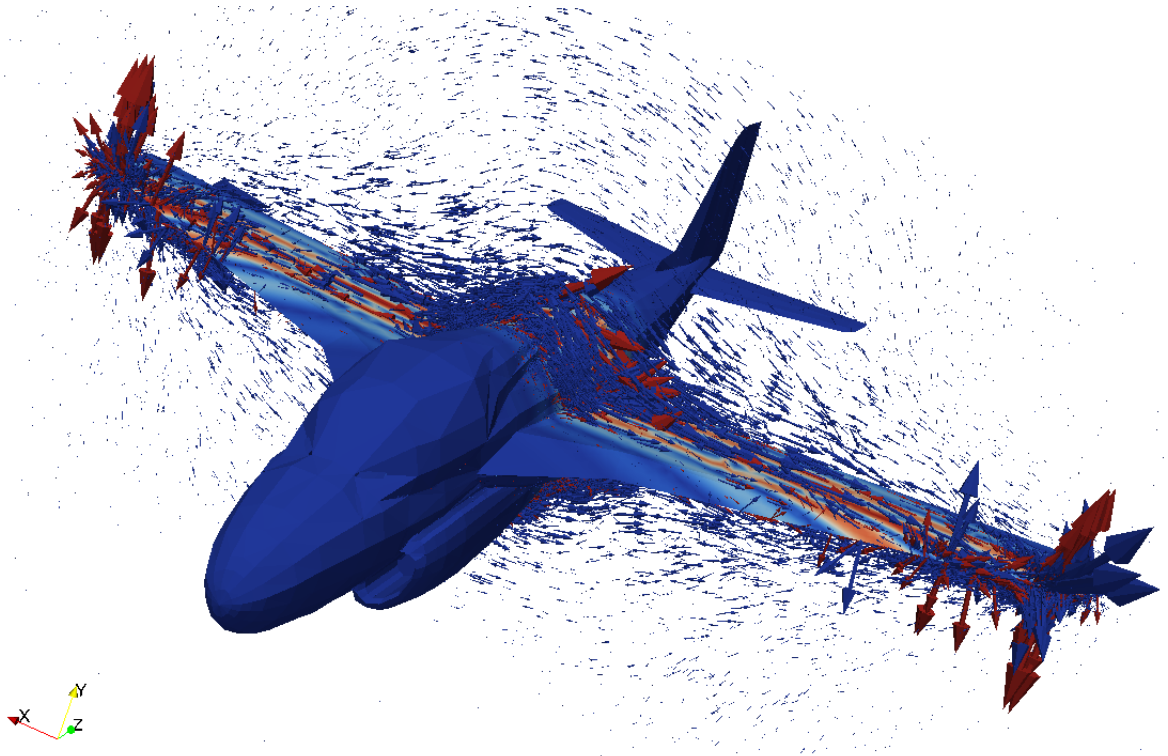
BROWN

Memory Bandwidth on a GTX 280



BROWN

"Real-World" Scattering Calculation



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

BROWN

Outline

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 :-)

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

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

→ `http://arxiv.org/abs/0901.1024`

BROWN

Questions?

?

Thank you for your attention!

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

http://arxiv.org/abs/0901.1024

BROWN

Image Credits I

- Batteries: flickr.com/thebmag CC
- Python logo: python.org
- Snail: flickr.com/hadi_fooladi CC
- Old Books: flickr.com/ppdigital CC
- Adding Machine: flickr.com/thomashawk CC
- Floppy disk: flickr.com/ethanhein CC
- Machine: flickr.com/13521837@N00 CC