PyCUDA and PyUblas: Hybrid HPC in Python made easy

Andreas Klöckner

Applied Mathematics, Brown University

March 5, 2009
Thanks

- Jan Hesthaven (Brown)
- Tim Warburton (Rice)
- Lucas Wilcox (UT Austin)
- Akil Narayan (Brown)
- PyCUDA contributors
- Nvidia Corporation
Outline

1. PyCUDA: What and Why?
2. PyCUDA: Overview
3. PyUblas
4. Conclusions
Outline

1. PyCUDA: What and Why?
   - Scripting for GPUs
   - (Very) brief GPU 101

2. PyCUDA: Overview

3. PyUblas

4. Conclusions
PyCUDA: Why do Scripting for GPUs?

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

→ complement each other
PyCUDA: Why do Scripting for GPUs?

- GPUs are everything that Python is not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
  → complement each other

- CPU: largely restricted to control tasks (~1000/sec)
  - Python fast enough
PyCUDA: Why do Scripting for GPUs?

- GPUs are everything that Python is not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
→ complement each other
- CPU: largely restricted to control tasks (∼1000/sec)
  - Python fast enough
- Realize a promise: Use Python...
  - from first prototype
  - to full-scale production code.
Usual answer to the “Speed Question”: Hybrid ("mixed") Code.
Usual answer to the “Speed Question”: Hybrid ("mixed") Code.

Plays to the strengths of each language.
Python: Speed

- Usual answer to the “Speed Question”: Hybrid (“mixed”) Code.
- Plays to the strengths of each language.
- But: Introduces (some) complexity.
Python: Speed

- Usual answer to the "Speed Question": Hybrid ("mixed") Code.
- Plays to the strengths of each language.
- But: Introduces (some) complexity.

**Observation:** GPU code is already hybrid.
Python: Speed

- Usual answer to the “Speed Question”: Hybrid (“mixed”) Code.
- Plays to the strengths of each language.
- But: Introduces (some) complexity.

Observation: GPU code is already hybrid.

Consequence: No added complexity through hybrid code.
Outline

1. **PyCUDA: What and Why?**
   - Scripting for GPUs
   - *(Very) brief GPU 101*

2. **PyCUDA: Overview**

3. **PyUblas**

4. **Conclusions**
GPUs: What?

- Design target for CPUs:
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate
GPUs: What?

- **Design target for CPUs:**
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate

- **Design target for GPUs:**
  - Throughput matters—single threads do not
  - Hide latency through massive parallelism
  - Let programmer deal with “raw” storage hierarchy
What is CUDA?

- CUDA is Nvidia’s proprietary compute abstraction.
What is CUDA?

- CUDA is Nvidia’s proprietary compute abstraction.
- Main merit: A well-balanced model of GPU computing.
  - Abstract enough to not be hardware-specific.
  - Concrete enough to expose most hardware features.
What is CUDA?

- CUDA is Nvidia’s proprietary compute abstraction.
- Main merit: A well-balanced model of GPU computing.
  - Abstract enough to not be hardware-specific.
  - Concrete enough to expose most hardware features.
- (Very) close semantic relative of OpenCL.
What is CUDA?

- CUDA is Nvidia’s proprietary compute abstraction.
- Main merit: A well-balanced model of GPU computing.
  - Abstract enough to not be hardware-specific.
  - Concrete enough to expose most hardware features.
- (Very) close semantic relative of OpenCL.
- Python + CUDA = PyCUDA.
GPUs: Execution Model

- Multi-tiered Parallelism
  - Block
  - Grid

Image Credit: Johan S. Seland, Sintef
GPUs: Execution Model

- Multi-tiered Parallelism
  - Block
  - Grid
- Only threads within a block can communicate

Image Credit: Johan S. Seland, Sintef
GPUs: Execution Model

- Multi-tiered Parallelism
  - Block
  - Grid
- Only threads within a block can communicate
  - Each Block is assigned to physical execution unit.

Image Credit: Johan S. Seland, Sintef
GPUs: Execution Model

- Multi-tiered Parallelism
  - Block
  - Grid

- Only threads within a block can communicate
  - Each Block is assigned to physical execution unit.

- Grids and Blocks replace outer loops in an algorithm.

Image Credit: Johan S. Seland, Sintef
Outline

1 PyCUDA: What and Why?

2 PyCUDA: Overview
   - Whetting your Appetite
   - Working with PyCUDA

3 PyUblas

4 Conclusions
Whetting your appetite

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

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

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

Andreas Klöckner

PyCUDA and PyUblas: Hybrid HPC in Python made easy
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

```
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_gpus, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpus)
print a_doubled
print a
```
Did somebody say “Abstraction is good”? 
1 import numpy
2 import pycuda.autoinit
3 import pycuda.gpuarray as gpuarray

4 a_gpu = gpuarray.to_gpu(
5     numpy.random.randn(4,4).astype(numpy.float32))
6 a_doubled = (2*a_gpu).get()
7 print a_doubled
8 print a_gpu
9
Outline

1. PyCUDA: What and Why?
2. PyCUDA: Overview
   - Whetting your Appetite
   - Working with PyCUDA
3. PyUblas
4. Conclusions
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
- (GL Interop)
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 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.gpudata`, `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.autodriver
import pycuda.driver as drv
import numpy

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 = numpy.random.randn(1000).astype(numpy.float32)
b = numpy.random.randn(1000).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(a_gpu, b_gpu, dest)
```

PyCUDA and PyUblas: Hybrid HPC in Python made easy
PyCUDA: Workflow

- Edit
- Run
PyCUDA: Workflow

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

Edit

Run

SourceModule("...")

PyCUDA
PyCUDA: Workflow

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

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

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

PyCUDA
PyCUDA: Workflow

Edit

Run

SourceModule("...")

Cache!

nvcc

.cubin

PyCUDA

PyCUDA and PyUblas: Hybrid HPC in Python made easy
PyCUDA: Workflow

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

Andreas Klöckner
Applied Math, Brown

PyCUDA and PyUblas: Hybrid HPC in Python made easy
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.
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())
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.
pycuda.gpuarray:

- Meant to look and feel just like numpy.
gpuarray: Simple Linear Algebra

pycuda.gpuarray:

- Meant to look and feel just like numpy.
  - gpuarray.to_gpu(numpy_array)
  - numpy_array = gpuarray.get()
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)
gpusarray: Simple Linear Algebra

pycuda.gpusarray:
- Meant to look and feel just like numpy.
  - gpusarray.to_gpu(numpy_array)
  - numpy_array = gpusarray.get()
- No: indexing, slicing, etc. (yet)
- Yes: +, -, *, /, fill, sin, exp, rand, take, ...

Andreas Klöckner

PyCUDA and PyUblas: Hybrid HPC in Python made easy
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, rand, take, ...

- Mixed types (int32 + float32 = float64)
**gpuarray: Simple Linear Algebra**

`pycuda.gpuarray`:

- Meant to look and feel just like `numpy`.
  - `gparray.to_gpu(numpy_array)`
  - `numpy_array = gparray.get()`
- No: indexing, slicing, etc. (yet)
- Yes: +, -, *, /, fill, sin, exp, rand, take, ...
- Mixed types (int32 + float32 = float64)
- `print gparray` 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, rand, take, ...

- Mixed types (int32 + float32 = float64)

- Print `gpuarray` for debugging.

- Memory behind `gpuarray` available as `.gpudata` attribute.
  - 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
```
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.
In PyCUDA, GPU code does not need to be a compile-time constant.
In PyCUDA, GPU code does not need to be a compile-time constant.

(unlike native C interface)
Shameless Plug

- How would you use this stuff to write a Discontinuous Galerkin solver?
- How well does that work?

MS134: Large-Scale Computing with High-Order Mth.
Tomorrow (Friday), 10:00–10:25, Concerto A
Outline

1 PyCUDA: What and Why?
2 PyCUDA: Overview
3 PyUblas
   - A Toolset for building Hybrid Codes
4 Conclusions
Making C++ and Python talk to each other

Making C++ and Python a workable combination:

- Python
- C++
Making C++ and Python talk to each other

Making C++ and Python a workable combination:

![Diagram showing integration of Python and C++ through Boost.Python](image-url)
Making C++ and Python talk to each other

Making C++ and Python a workable combination:

Python

Boost.Python

Numpy

C++
Making C++ and Python talk to each other

Making C++ and Python a workable combination:

Python

Boost.Python

Numpy

C++

Boost.Ublas
Making C++ and Python talk to each other

Making C++ and Python a workable combination:

- Python
  - Boost.Python
  - Numpy
- C++
  - Boost.Ublas
  - PyUblas
PyUblas: Features

- Exploits pluggable storage backends in Ublas: `numpy_vector<T>, numpy_matrix<T>`
  - Normal citizens of the Ublas universe (take part in ET etc.)
- Type-safe, zero-copy language transition
- OK anywhere: argument, return value, data member
- Strided storage: `numpy_strided_vector<T>`
  - Not OK: non-contiguous storage
C++:

```cpp
#include <pyublas/numpy.hpp>

numpy_vector<double> triple(numpy_vector<double> x)
{
    return 3*x;
}

BOOST_PYTHON_MODULE(sample_ext)
{
    boost::python::def(" triple ", triple);
}
```

Python:

```python
import sample_ext
import pyublas

vec = numpy.ones((5,), dtype=float)
print sample_ext. triple (vec)
```
Conclusions

- GPUs and Python: good combination
GPUs and Python: good combination

Been considering playing with GPUs? PyCUDA makes that easier.
GPUs and Python: good combination

Been considering playing with GPUs? PyCUDA makes that easier.

Allows full access to low-level bits, if needed.
GPUs and Python: good combination

Been considering playing with GPUs? PyCUDA makes that easier.

Allows full access to low-level bits, if needed.

Provides many nice abstractions to make life easier.
Conclusions

- GPUs and Python: good combination
- Been considering playing with GPUs? PyCUDA makes that easier.
- Allows full access to low-level bits, if needed.
- Provides many nice abstractions to make life easier.
- Enables just-in-time GPU code generation.
  → See tomorrow’s talk
Questions?

? 

Thank you for your attention!

http://mathema.tician.de/software/pycuda
MS134: Tomorrow (Friday), 10:00–10:25, Concerto A
Image Credits

- C870 GPU: Nvidia Corp.
- Snail: flickr.com/hadi_fooladi (CC)
- GT200 die shot: Nvidia Corp.
- Old Books: flickr.com/ppdigital (CC)
- Adding Machine: flickr.com/thomashawk (CC)
- Floppy disk: flickr.com/ethanhein (CC)