An Introduction to PyCUDA and PyOpenCL and how they are useful

David Ozog

University of Oregon

January 26, 2011
Outline

- What is PyCUDA?
- Why use PyCUDA?
- How do you use it?
  - numpy
  - SourceModule
  - GPUArray class
  - ElementWise kernels
  - other libraries
- Metaprogramming with Python
- How to use PyCUDA on Mist
Whetting your appetite

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

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

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

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

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```
Whetting your appetite

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

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Compute kernel
Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
  → complement each other
- CPU: largely restricted to control tasks (∼1000/sec)
  - Scripting fast enough
- Python + CUDA = PyCUDA
- Python + OpenCL = PyOpenCL
Scripting: Python

One example of a scripting language: Python

- Mature
- Large and active community
- Emphasizes readability
- Written in widely-portable C
- A ‘multi-paradigm’ language
- Rich ecosystem of sci-comp related software
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
Scripting: Interpreted, not Compiled

Program creation workflow:

Edit | Compile | Link | Run

Edit -> Compile -> Link -> Run
PyCUDA: Workflow

Edit → Run → SourceModule("...") → Run on GPU

Cache? → no → nvcc → .cubin → Upload to GPU → PyCUDA
How are High-Performance Codes constructed?

- “Traditional” Construction of High-Performance Codes:
  - C/C++/Fortran
  - Libraries
- “Alternative” Construction of High-Performance Codes:
  - Scripting for ‘brains’
  - GPUs for ‘inner loops’
- Play to the strengths of each programming environment.
PyCUDA Philosophy

- Provide complete access
- Automatically manage resources
- Provide abstractions
- Check for and report errors automatically
- Full documentation
- Integrate tightly with numpy
What’s this “numpy”, anyway?

Numpy: package for large, multi-dimensional arrays.

- Vectors, Matrices, ...
- A+B, sin(A), dot(A,B)
- la.solve(A, b), la.eig(A)
- cube[:,:,n-k:n+k], cube+5

All much faster than functional equivalents in Python.

“Python’s MATLAB”: Basis for SciPy, plotting, ...
**pycuda.gpuarray:**

- Meant to look and feel just like `numpy`.
  - `gpuarray.to_gpu(numpy_array)`
  - `numpy_array = gpuarray.get()`
- `+`, `-`, `*`, `/`, fill, `sin`, `exp`, `rand`, basic indexing, `norm`, `inner product`, …
- Mixed types (`int32 + float32 = float64`)
- `print gpuarray` for debugging.
- Allows access to raw bits
  - Use as kernel arguments, textures, etc.
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
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
```
Introducing... PyOpenCL

- PyOpenCL is “PyCUDA for OpenCL”
- Complete, mature API wrapper
- Has: Arrays, elementwise operations, RNG, ...
- Near feature parity with PyCUDA
- Tested on all available Implementations, OSs
- http://mathema.tician.de/software/pyopencl
Step 3: Usage

- Complex numbers
  - ...in GPUArray
  - ...in user code
    (pycuda-complex.hpp)
- If/then/else for GPUArrays
- Support for custom device pointers
- Smarter device picking/context creation
- PyFFT: FFT for PyOpenCL and PyCUDA
- scikits.cuda: CUFFT, CUBLAS, CULA
Run-Time Code Generation:

- Writing code when the most knowledge is available
- One of PyCUDA / PyOpenCL’s key selling points
- Allows for:
  - automated tuning of a program
  - dynamic data types
  - specialized code suited to a given problem
A sizable part of a CUDA programmer’s time is typically spent tuning code. This tuning answers questions like:\(^1\)

- What’s the optimal number of threads per block?
- How much data should I work on at once?
- What data should be loaded into shared memory, and how big should the corresponding blocks be?

\(^1\)Verbatim from the documentation: [http://documen.tician.de/pycuda/](http://documen.tician.de/pycuda/)
Many difficult questions

Insufficient heuristics

Answers are hardware-specific and have no lasting value
Many difficult questions

Insufficient heuristics

Answers are hardware-specific and have no lasting value

**Proposed Solution:** Tune automatically for hardware at run time, cache tuning results.

- Decrease reliance on knowledge of hardware internals
- Shift emphasis from tuning *results* to tuning *ideas*
In GPU scripting, GPU code does not need to be a compile-time constant.
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant. (Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
Machine-generated Code

Why machine-generate code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables → register pressure
- Loop Unrolling
RTCG via Templates

```python
from jinja2 import Template

tpl = Template(""
    __global__ void twice({{ type_name }} *tgt)
    {
        int idx = threadIdx.x +
        {{ thread_block_size }} * {{ block_size }} * blockIdx.x;

        {% for i in range(block_size) %}
        {{ set offset = i*thread_block_size %}
        tgt[idx + {{ offset }}] *= 2;
        {% endfor %}
    }"")

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

smod = SourceModule(rendered_tpl)
```

Andreas Klöckner

PyCUDA: Even Simpler GPU Programming with Python
Using PyCUDA on Mist is pretty easy. You need to:

- Install numpy
  - optional once installed system-wide:
  - download source code (I used version 1.5.1)
    you@mist:/home/users/ozog/usr/src/numpy-1.5.1.tar.gz
    $ python setup.py build
    $ python setup.py install
      --home=/home/users/<yourusername>/path
  - load gcc and cuda modules
    $ module load gcc cuda

- run python locally
  - $ python
  - >>> import pycuda.driver as cuda