# A gentle tutorial on programming scientific applications on NVIDIA GPU's with Python and CUDA

Daniel Wysocki

Rochester Institute of Technology

RIT Scientific Computing Group
October 18th, 2018

R·I·T

# Scientific computing on a *graphics* processor?

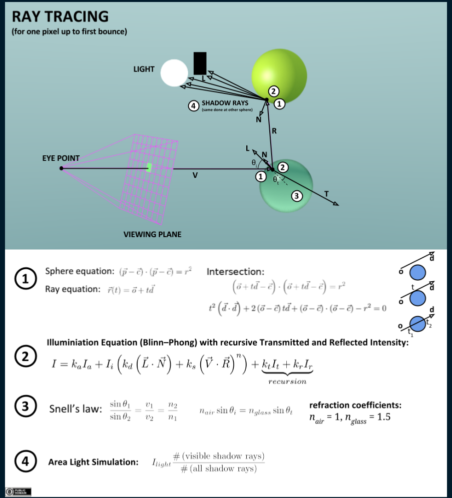

© Rob Boudon – https://www.flickr.com/people/88562024@N00






- Scientific computing on a graphics card?
- Isn't that just for computer graphics and video games?
- Is this a talk about data visualizations?

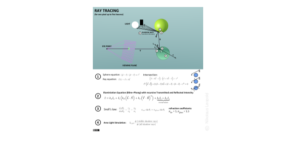
# Graphics requires lots of linear algebra and trig – really fast









- No! We're going to re-use things originally just meant for graphics.
- Computer graphics is mostly linear algebra and trigonometry.
- And it needs to be really fast to do things like high-framerate videogames, or to render CGI in movies in an acceptable timeline.
- Linear algebra is the foundation of scientific computing, so we're going to be very happy!

# Single Instruction, Multiple Data (SIMD)

Ford assembly line – public domain





Single Instruction, Multiple Data (SIMD)

- GPU's are specially designed for the SIMD paradigm – single instruction, multiple data
- SIMD is like an assembly line, you want to perform the same operation over and over again
- key difference: in SIMD, the things coming down the assembly line might be different each time
- Example: one worker on the assembly line's job is to "multiply by two", next worker's job is to "multiply by the matrix $A$", then the data gets merged with another assembly line, where the next worker's job is to "add together everything from the two input assembly lines".
- Modern consumer CPU's typically have 2 or 4 cores running in parallel, but GPU's of the same grade have hundreds of cores, albeit each core is typically less powerful than a CPU core

# NVIDIA CUDA

- Currently the best performing GPUs in the world are made by NVIDIA, and are most efficiently programmed using their proprietary (freeware) language `CUDA`
- `CUDA` is basically an extension to `C/C++`
- Very low-level, requiring understanding of how GPU's operate to get the most efficient code possible

R·I·T

# CuPy – CUDA programming in NumPy style

CuPy

- CuPy is a free and open source Python library, meant as a replacement for NumPy, but using CUDA under the hood
- It includes a large subset of NumPy's features, along with additional tools for low-level GPU stuff, and for converting data between CuPy and NumPy
- Available at https://cupy.chainer.org/

R·I·T

## CuPy – Basic example

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([  3.,  12.], dtype=float32)
```





- Here's a basic CuPy usage example from their documentation
- First line imports `CuPy` as "`cp`", similar to the common convention of importing `NumPy` as "`np`".
- Then they create an array using `arange`, just like you would in `NumPy`, and then they perform some manipulations also available in `NumPy`, `reshape` and `astype`.
- Finally they sum the array, using `axis=1` to specify that it's along the "column" direction, as you can do in `NumPy`.

# CuPy installation

Assuming you already have CUDA installed, you can install `CuPy` with the standard Python package manager `pip`.

```
# (For CUDA 8.0)
$ pip install cupy-cuda80

# (For CUDA 9.0)
$ pip install cupy-cuda90

# (For CUDA 9.1)
$ pip install cupy-cuda91

# (Install CuPy from source)
$ pip install cupy
```

# CuPy Arrays

- The core data structure in CuPy is the $N$-dimensional array, `cupy.ndarray`
- Looks and behaves almost exactly like the $N$-dimensional array in NumPy, except it is allocated in the GPU's memory instead

```
>>> import numpy, cupy

>>> cupy.arange(5)
array([0, 1, 2, 3, 4])
>>> numpy.arange(5)
array([0, 1, 2, 3, 4])

>>> cupy.ones((2,2), dtype=float)
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> numpy.ones((2,2), dtype=float)
array([[ 1.,  1.],
       [ 1.,  1.]])
```





- First read bullet points
- See CuPy has the same `ndarray` constructor methods as NumPy, e.g., `arange` for making sequences of increasing numbers, or `ones` for making arrays of just ones.

# Basic arithmetic in CuPy

- CuPy arrays support all basic arithmetic NumPy arrays do

```
>>> x = cupy.arange(4)
>>> x
array([0, 1, 2, 3])
```

```
>>> x + x
array([0, 2, 4, 6])
>>> x * x
array([0, 1, 4, 9])
>>> x / x
array([0, 1, 1, 1])
>>> x - x
array([0, 0, 0, 0])
>>> 2 * x
array([0, 2, 4, 6])
>>> x**2
array([0, 1, 4, 9])
>>> 3*(x**2 + 4*x + 1)
array([ 3, 18, 39, 66])
```

R·I·T

# More math functions in CuPy

```
>>> cupy.sin(x)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001])
>>> cupy.cos(x)
array([ 1.        ,  0.54030231, -0.41614684, -0.9899925 ])
>>> cupy.exp(x)
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692])
>>> cupy.sqrt(x)
array([ 0.        ,  1.        ,  1.41421356,  1.73205081])
>>> cupy.outer(x, x)
array([[0, 0, 0, 0],
       [0, 1, 2, 3],
       [0, 2, 4, 6],
       [0, 3, 6, 9]])
```

# Mixing CuPy and NumPy arrays (wrong way)

```
>>> cupy.arange(0, 5) + cupy.arange(3, 8)
array([ 3,  5,  7,  9, 11])

>>> cupy.arange(0, 5) + numpy.arange(3, 8)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-9951f4a4a370> in <module>()
----> 1 cupy.arange(0, 5) + numpy.arange(3, 8)

cupy/core/core.pyx in cupy.core.core.ndarray.__add__()

cupy/core/elementwise.pxi in cupy.core.core.ufunc.__call__()

cupy/core/elementwise.pxi in cupy.core.core._preprocess_args()

TypeError: Unsupported type <type 'numpy.ndarray'>
```





└Mixing CuPy and NumPy arrays (wrong way)

# Mixing `CuPy` and `NumPy` arrays (right way)

- must convert between `CuPy` and `NumPy` arrays to mix
  - `cupy.asnumpy()`: CuPy→NumPy
  - `cupy.asarray()`: NuMPy→CuPy

```
>>> cupy.asnumpy(cupy.arange(0, 5)) + numpy.arange(3, 8)
array([ 3,  5,  7,  9, 11])

>>> type(cupy.asnumpy(cupy.arange(0, 5)) + numpy.arange(3, 8))
numpy.ndarray

>>> type(cupy.arange(0, 5) + cupy.asarray(numpy.arange(3, 8)))
cupy.core.core.ndarray
```

- Beware: slow process, should avoid everywhere possible

R·I·T

# Writing CuPy/NumPy agnostic code

- Can write functions that work on both CuPy and NumPy

```
>>> def euler_formula(x):
...     "exp(i*x) = cos(x) + i*sin(x)"
...     # Can be either `numpy` or `cupy`.
...     xpy = cupy.get_array_module(x)
...     # Compute the result with the right library.
...     return xpy.cos(x) + 1j*xpy.sin(x)

>>> type(euler_formula(cupy.arange(10)))
cupy.core.core.ndarray

>>> type(euler_formula(numpy.arange(10)))
numpy.ndarray
```





- If you want to write code that works on both CuPy and NumPy arrays (e.g., generic-enough functions that you might use it on both types of arrays at some point, or perhaps you plan on having both a CPU and GPU version of your code – no need to duplicate effort!

# Writing CuPy/NumPy agnostic code (optimized)

- For extra speed, you can save the call to `cupy.get_array_module`

```
>>> def euler_formula(x, xpy=cupy):
...     "exp(i*x) = cos(x) + i*sin(x)"
...     # Compute the result with the right library.
...     return xpy.cos(x) + 1j*xpy.sin(x)

>>> type(euler_formula(cupy.arange(10), xpy=cupy))
cupy.core.core.ndarray

>>> type(euler_formula(numpy.arange(10), xpy=numpy))
numpy.ndarray

>>> type(euler_formula(numpy.arange(10), xpy=cupy))
TypeError: Unsupported type <type 'numpy.ndarray'>
```





- `cupy.get_array_module` makes it convenient and safe to write functions that work on both `CuPy` and `NumPy` arrays
- However, it's an extra operation, and if you call this function millions/billions/trillions of times, you're going to lose a sizable amount of time to it
- Can simply add `xpy` as an argument to the function
- This does mean it can crash your code if you make a mistake in which module you pass in

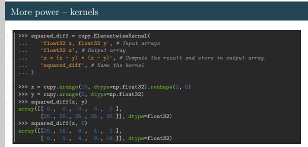
# More power – kernels

```
>>> squared_diff = cupy.ElementwiseKernel(
...     'float32 x, float32 y', # Input arrays
...     'float32 z', # Output array
...     'z = (x - y) * (x - y)', # Compute the result and store in output array.
...     'squared_diff', # Name the kernel
... )

>>> x = cupy.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cupy.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
>>> squared_diff(x, 5)
array([[25., 16.,  9.,  4.,  1.],
       [ 0.,  1.,  4.,  9., 16.]], dtype=float32)
```







- Sometimes you want more power, and need to write your own CUDA kernel.
- CuPy can help you write these kernels in a micro-language it provides.
- Basic example is element-wise kernel, which takes in two arrays of the same shape, performs an operation on each corresponding pair of elements, and returns the result in a new array of the same shape.

# Type-generic kernels

```
>>> squared_diff_generic = cupy.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_generic',
...     )
```





Type-generic kernels

```
>>> squared_diff_generic = cupy.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_generic',
...     )
```

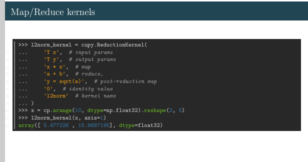
# Map/Reduce kernels

```
>>> l2norm_kernel = cupy.ReductionKernel(
...     'T x',  # input params
...     'T y',  # output params
...     'x * x',  # map
...     'a + b',  # reduce,
...     'y = sqrt(a)',  # post-reduction map
...     '0',  # identity value
...     'l2norm'  # kernel name
... )
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([ 5.477226 , 15.9687195], dtype=float32)
```

R·I·T







- map step increases dimensionality of the array, in this case it's an outer product
- reduce step reduces (duh) dimensionality of the array
- `a` and `b` are special variables
  - `a` denotes the result accumulated thus far
  - `b` denotes the next element being operated on
- post-reduction step doesn't change shape – it's just elementwise
- identity value is the initial value of `a`
- may need to draw a grid on the whiteboard to explain map/reduce parts

# Profiling – cProfile

- cProfile records function-level timing statistics
- Simply run python command as usual, but with -m cProfile
  - e.g., python my_script.py becomes python -m cProfile my_script.py




└Profiling – cProfile

# Test program – naive version

```python
import numpy

seed = 1; random = numpy.random.RandomState(seed)

x = random.uniform(size=(5000,5000)); y = numpy.zeros(5000)

def sum_2d():
    for i in range(5000):
        for j in range(5000):
            y[i] += x[i,j]
def sum_1d():
    global z
    z = 0
    for i in range(5000):
        z += y[i]
sum_2d()
sum_1d()
```

# Profiling – naive version

```
$ python -m cProfile big_calculation_naive.py
12502566.4643
         17098 function calls (16977 primitive calls) in 9.976 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
   ..........................................................................
        1    0.001    0.001    9.976    9.976 big_calculation_naive.py:1(<module>)
        1    0.001    0.001    0.001    0.001 big_calculation_naive.py:11(sum_1d)
        1    9.378    9.378    9.482    9.482 big_calculation_naive.py:7(sum_2d)
   ..........................................................................
```

R·I·T



2018-10-18
GPU Programming with Python and CUDA
└Profiling – naive version

# Test program – efficiently with NumPy

```
import numpy

seed = 1; random = numpy.random.RandomState(seed)

x = random.uniform(size=(5000,5000)); y = numpy.zeros(5000)

def sum_2d():
    x.sum(axis=1, out=y)
def sum_1d():
    global z
    z = y.sum()
sum_2d()
sum_1d()

print(z)
```

# Profiling – efficiently with NumPy

```
$ python -m cProfile big_calculation_numpy.py
12502566.4643
         12102 function calls (11981 primitive calls) in 0.508 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
   ................................................................
        1    0.001    0.001    0.508    0.508 big_calculation_numpy.py:1(<module>)
        1    0.000    0.000    0.015    0.015 big_calculation_numpy.py:7(sum_2d)
        1    0.000    0.000    0.000    0.000 big_calculation_numpy.py:9(sum_1d)
   ................................................................
```

R·I·T



GPU Programming with Python and CUDA
2018-10-18
Profiling – efficiently with NumPy

# Test program – efficiently with CuPy

```
import cupy

seed = 1; random = cupy.random.RandomState(seed)

x = random.uniform(size=(5000,5000)); y = cupy.zeros(5000)

def sum_2d():
    x.sum(axis=1, out=y)
def sum_1d():
    global z
    z = y.sum()
sum_2d()
sum_1d()

print(z)
```





Test program – efficiently with CuPy

import cupy

seed = 1; random = cupy.random.RandomState(seed)

x = random.uniform(size=(5000,5000)); y = cupy.zeros(5000)

def sum_2d():
    x.sum(axis=1, out=y)
def sum_1d():
    global z
    z = y.sum()
sum_2d()
sum_1d()

print(z)

# Profiling – efficiently with CuPy

```
$ python -m cProfile big_calculation_cupy.py
12499843.3008
         247065 function calls (242929 primitive calls) in 1.514 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
   ...................................................................
        1    0.001    0.001    1.516    1.516 big_calculation_cupy.py:1(<module>)
        1    0.000    0.000    0.020    0.020 big_calculation_cupy.py:7(sum_2d)
        1    0.000    0.000    0.033    0.033 big_calculation_cupy.py:9(sum_1d)
   ...................................................................
```






└Profiling – efficiently with CuPy

Profiling – efficiently with CuPy

## Profiling – kernprof

- `kernprof` records line-level timing statistics
- Need to `pip install line_profiler`
- Add `@profile` before functions you want profiled
- Run script with `kernprof -l` instead of `python`
  - e.g., `python my_script.py` becomes `kernprof -l my_script.py`
- Then read profiling summary with `python -m line_profiler my_script.py.lprof`

# Test program – naive version

```python
import numpy

seed = 1; random = numpy.random.RandomState(seed)

@profile
def main():
    x = random.uniform(size=(1000,1000))
    z = numpy.empty((1000,1000))
    for i in range(1000):
        for j in range(1000):
            z[i,j] = x[i,j] + x[j,i]
    y = z.sum()
    print(y)

main()
```





└Test program – naive version

Test program – naive version

# Profiling – naive version (I)

```
$ kernprof -l kernprof_demo_slow.py
999896.508257
Wrote profile results to kernprof_demo_slow.py.lprof
$ python -m line_profiler kernprof_demo_slow.py.lprof
...
```

R·I·T



2018-10-18
GPU Programming with Python and CUDA
└Profiling – naive version (I)

Profiling – naive version (I)

$ kernprof -l kernprof_demo_slow.py
999896.508257
Wrote profile results to kernprof_demo_slow.py.lprof
$ python -m line_profiler kernprof_demo_slow.py.lprof

# Profiling – naive version (II)

```
Timer unit: 1e-06 s

Total time: 3.19258 s
File: kernprof_demo_slow.py
Function: main at line 5

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def main():
     7         1      16749.0  16749.0      0.5      x = random.uniform(size=(1000,1000))
     8         1         13.0     13.0      0.0      z = numpy.empty((1000,1000))
     9      1001       1238.0      1.2      0.0      for i in range(1000):
    10   1001000    1235929.0      1.2     38.7          for j in range(1000):
    11   1000000    1937855.0      1.9     60.7              z[i,j] = x[i,j] + x[j,i]
    12         1        728.0    728.0      0.0      y = z.sum()
    13         1         64.0     64.0      0.0      print(y)
```



GPU Programming with Python and CUDA

2018-10-18

└Profiling – naive version (II)

# Test program – fast version

```python
import numpy

seed = 1; random = numpy.random.RandomState(seed)

@profile
def main():
    x = random.uniform(size=(1000,1000))
    z = x + x.T
    y = z.sum()
    print(y)

main()
```





Test program – fast version

# Profiling – fast version (I)

```
$ kernprof -l kernprof_demo_fast.py
999896.508257
Wrote profile results to kernprof_demo_fast.py.lprof
$ python -m line_profiler kernprof_demo_fast.py.lprof
...
```

R·I·T





Profiling – fast version (I)

$ kernprof -l kernprof_demo_fast.py
999896.508257
Wrote profile results to kernprof_demo_fast.py.lprof
$ python -m line_profiler kernprof_demo_fast.py.lprof

# Profiling – fast version (II)

```
Timer unit: 1e-06 s

Total time: 0.022384 s
File: kernprof_demo_fast.py
Function: main at line 5

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def main():
     7         1      16635.0  16635.0     74.3      x = random.uniform(size=(1000,1000))
     8         1       5027.0   5027.0     22.5      z = x + x.T
     9         1        645.0    645.0      2.9      y = z.sum()
    10         1         77.0     77.0      0.3      print(y)
```

R·I·T

# Profiling GPU code with NVIDIA Profiler (nvprof)

```
$ nvprof python big_calculation_cupy.py
==3127650== NVPROF is profiling process 3127650, command: python big_calculation_cupy.py
12499843.3008
==3127650== Profiling application: python big_calculation_cupy.py
==3127650== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   33.59%  52.765ms         1  52.765ms  52.765ms  52.765ms  generate_seed_pse
                   21.08%  33.114ms         2  16.557ms  12.287us  33.102ms  cupy_sum
                   13.30%  20.890ms         1  20.890ms  20.890ms  20.890ms  cupy_multiply
                   13.29%  20.882ms         1  20.882ms  20.882ms  20.882ms  cupy_add
                   11.90%  18.695ms         1  18.695ms  18.695ms  18.695ms  cupy_random_1_mi
                    6.84%  10.752ms         1  10.752ms  10.752ms  10.752ms  void gen_sequence
                    0.00%  6.3350us         1  6.3350us  6.3350us  6.3350us  [CUDA memset]
                    0.00%  1.4400us         1  1.4400us  1.4400us  1.4400us  [CUDA memcpy DtoH
      API calls:   52.39%  169.80ms         6  28.299ms  1.1880us  116.57ms  cudaFree
.................................................................................
```

R·I·T



2018-10-18
GPU Programming with Python and CUDA
└Profiling GPU code with NVIDIA Profiler (nvprof)

# Profiling GPU code with NVIDIA Visual Profiler (nvvp)

```
$ nvvc python big_calculation_cupy.py
```

**Will add this if I can get it to work in time.**





Profiling GPU code with NVIDIA Visual Profiler (nvvp)
$ nvvc python big_calculation_cupy.py
Will add this if I can get it to work in time.