Prototyping and Developing GPU Accelerated Solutions with Python and CUDA

Luciano Martins

Principal Software Engineer
Oracle Corporation

Agenda

- Python introduction
- GPU programming
- Why python with GPU?
- Accelerating python
- Comparing codes with/without GPU support
- Summary

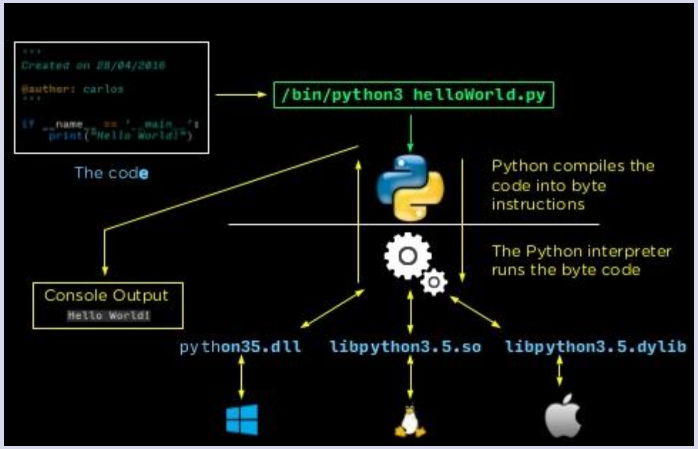
Python introduction

- created by Guido van Rossum in 1991
- "Zen of Python" which:
  - Beautiful is better than ugly
  - Explicit is better than implicit
  - Simple is better than complex
  - Complex is better than complicated
  - Readability counts
- interpreted language (CPython, JPython, ...)
- dynamically typed; based on objects

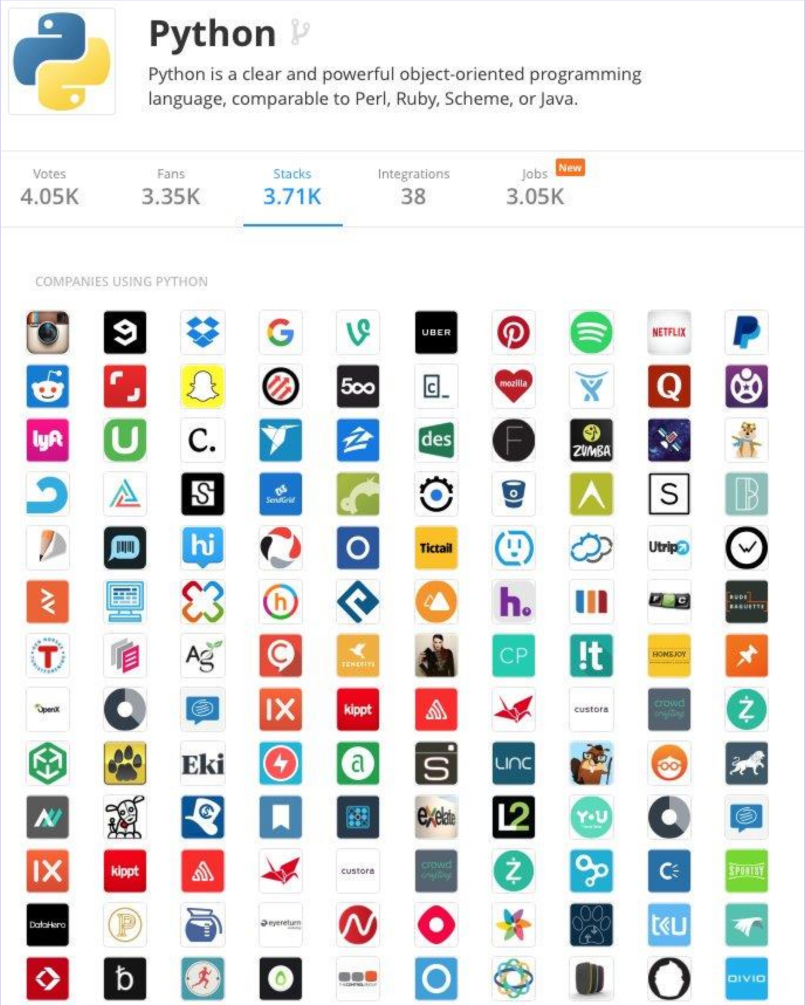
Python introduction

- small core structure:
  - ~30 keywords
  - ~80 built-in functions
- indention is a pretty serious thing
- a huge modules ecosystem
- binds to many different languages
- supports GPU acceleration via modules ←

Python introduction



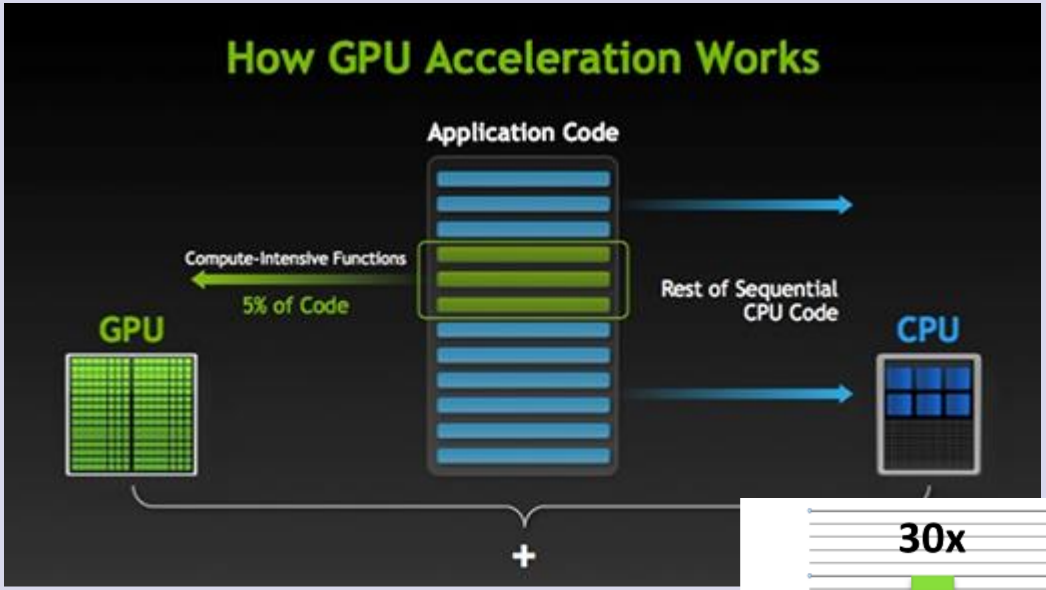
Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C		99.7
3. Java		99.5
4. C++		97.1
5. C#		87.7
6. R		87.7
7. JavaScript		85.6
8. PHP		81.2
9. Go		75.1
10. Swift		73.7

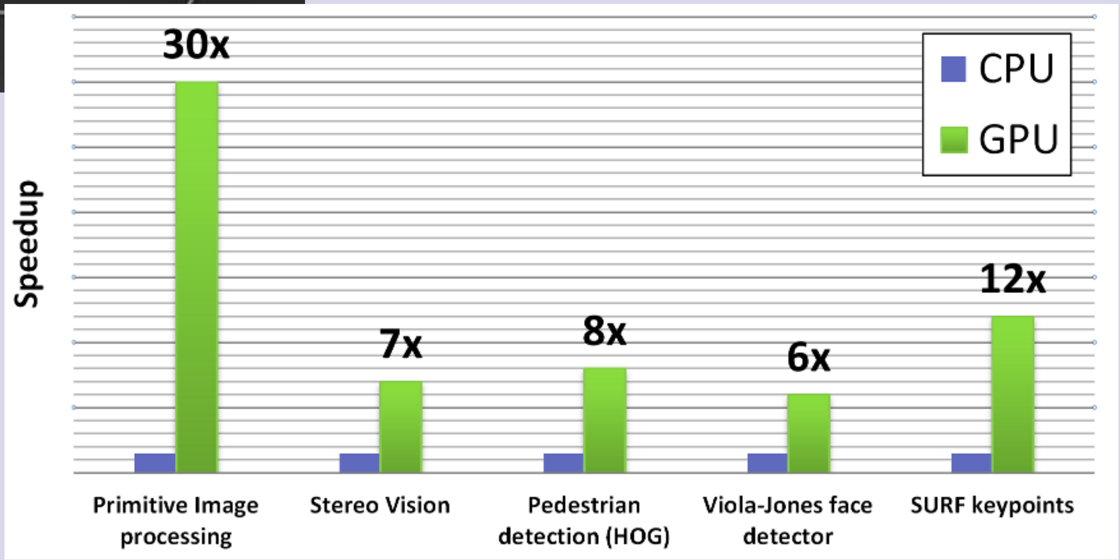


GPU programming

- “the use of a graphics processing unit (GPU) together with a CPU to accelerate deep learning, analytics, and engineering applications” (NVIDIA)
- most common GPU accelerated operations:
  - large vector/matrix operations (BLAS)
  - speech recognition
  - computer vision
  - way more

GPU programming





Why python with GPU?

- Interpreted languages has the reputation of being slow for high performance needs
- Python needs assistance for those tasks
- Keep the best of both scenarios:
  - Quick development and prototyping with python
  - Use high processing power and speed of GPU
- Can deliver quick results for complex projects
- Gives a business decision choice at the end

Accelerating python

- GPU + python projects are arising every day
- Accelerated code may be pure python or adding C code
- Focusing here on the following modules
  - PyCUDA
  - Numba
  - cudamat
  - cupy
  - scikit-cuda

Accelerating python – PyCUDA

- A python wrapper to CUDA API
- Requires C programming knowledge (kernel)
- Gives speed to python – near zero wrapping
- Compiles the CUDA code copy to GPU
- CUDA errors translated to python exceptions
- Easy installation

```
root@hell:~# pip3 install pycuda
Collecting pycuda
  Downloading pycuda-2017.1.1.tar.gz (1.6MB)
    100% |████████████████████████████████| 1.6MB 963kB/s
```

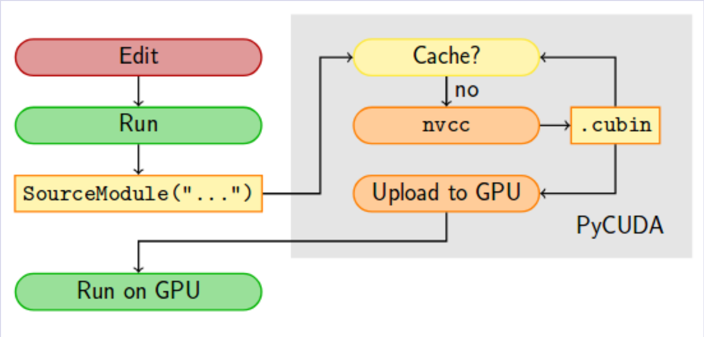
```
root@hell:~# pip3 show pycuda
Name: pycuda
Version: 2017.1.1
Summary: Python wrapper for Nvidia CUDA
Home-page: http://mathema.tician.de/software/pycuda
Author: Andreas Kloeckner
Author-email: inform@tiker.net
```

Accelerating python – PyCUDA

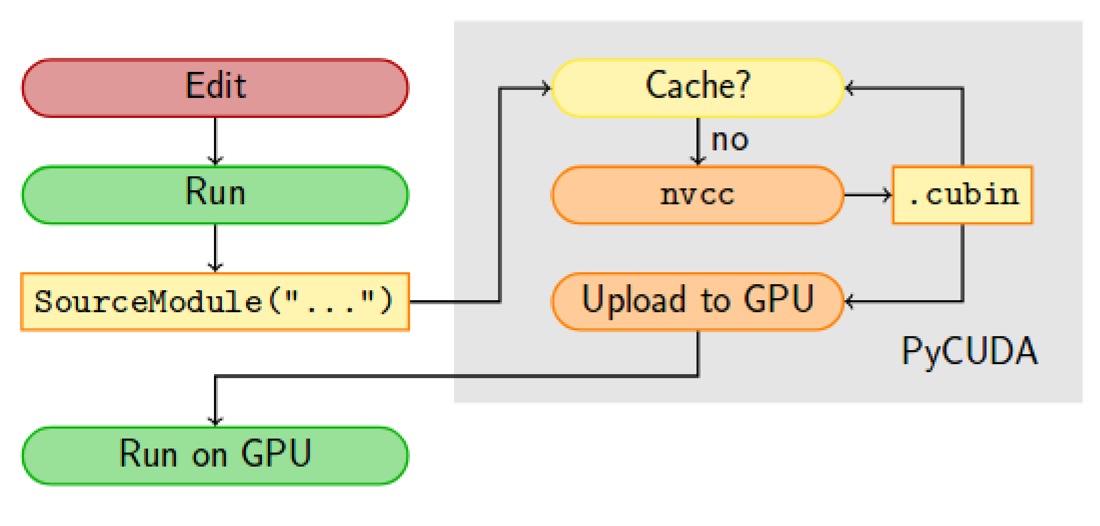
Editor - /Downloads/GTC2018/GTC_sample.py

GTC_sample.py*

```python
#!/usr/bin/env python

# importing modules
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

# creating a numpy array
a = numpy.random.randn(4,4)

# maing this array as single precision format
a = a.astype(numpy.float32)

# allocating GPU memory to store the a array
a_gpu = cuda.mem_alloc(a.nbytes)

# copy the array to GPU memory
cuda.memcpy_htod(a_gpu, a)

# declaring the kernel to be run on the GPU
# using PyCUDA as C wrapper
mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)

# importing to the python realm the 'doublify' function
# from the kernel and passing the a_gpu as argument
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

# fetching the data back from the GPU and displaying it
# also showing the original a
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print(a_doubled)
print(a)
```



Accelerating python – PyCUDA



Accelerating python – Numba

- high performance functions written in Python
- On-the-fly code generation
- Native code generation for the CPU and GPU
- Integration with the Python scientific stack
- Take advantage of Python decorators
- No need to write C code
- Code translation done using LLVM compiler

Accelerating python – Numba

Spyder (Python 3.5)

Editor - /Downloads/GTC2018/GTC_sample.py

GTC_sample.py

```python
#!/usr/bin/env python

import numpy as np
from numba import guvectorize

@guvectorize(['void(float64[:], intp[:], float64[:])'], '(n),()->(n)')
def move_mean(a, window_arr, out):
    window_width = window_arr[0]
    asum = 0.0
    count = 0
    for i in range(window_width):
        asum += a[i]
        count += 1
        out[i] = asum / count
    for i in range(window_width, len(a)):
        asum += a[i] - a[i - window_width]
        out[i] = asum / count

arr = np.arange(20, dtype=np.float64).reshape(2, 10)
print(arr)
print(move_mean(arr, 3))
```

Accelerating python – cudamat

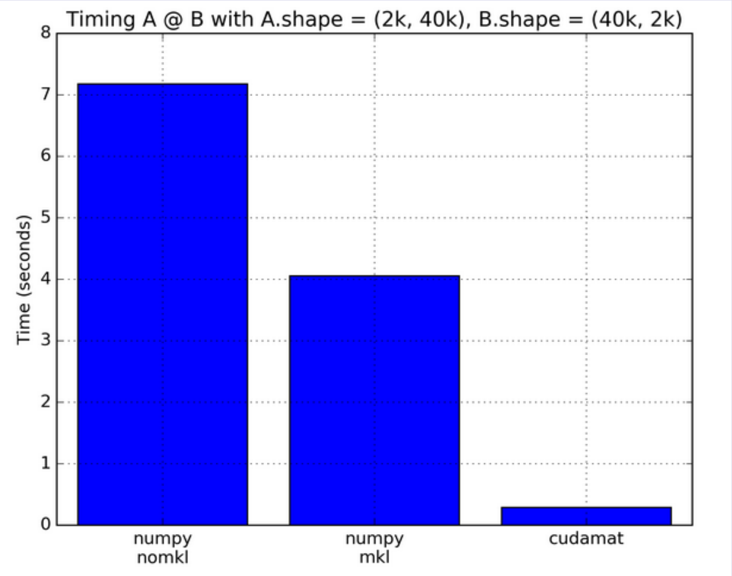
- provides a CUDA-based python matrix class
- Primary goal: easy dense matrix manipulation
- Useful to perform matrix ops on GPU
- Perform many matrix operations
  - multiplication and transpose
  - Elementwise addition, subtraction, multiplication, and division
  - Elementwise application of exp, log, pow, sqrt
  - Summation, maximum and minimum along rows or columns

Accelerating python – cudamat

Editor - /Downloads/GTC2018/GTC_sample.py

GTC_sample.py

```python
#!/usr/bin/env python

import  cudamat  as cm
import  numpy as np

cm.cuda_set_device (0)
cm.init()

# Create  some  matrices  on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
B = cm.CUDAMatrix(np.random.randn(10, 20))
T = cm.CUDAMatrix(np.random.randn(10, 20))

# Add A and B and  store  the  result  in Tx
A.add(B, target = T)

# Multiply A by 50
A.mult(50)
```



Accelerating python – cupy

- an implementation of NumPy-compatible multi-dimensional array on CUDA
- Useful to perform matrix ops on GPU
- CuPy is faster than NumPy in many ways

Table 1: Matrix operation performance

Size	NumPy [ms]	CuPy [ms]
10^4	0.03	0.58
10^5	0.20	0.97
10^6	2.00	1.84
10^7	55.55	12.48
10^8	517.17	84.73

```
>>> import time, numpy, cupy
>>>
>>> size = 10 ** 8
>>> def test(xp):
...     return xp.arange(size).reshape(1000, -1).T * 2
>>>

>>> for xp in [numpy, cupy]:
...     test(xp)                          # Avoid first call overhead
...     # Synchronize CPU and GPU for benchmark
...     cupy.cuda.runtime.deviceSynchronize()
...     t1 = time.time()
...     test(xp)
...     cupy.cuda.runtime.deviceSynchronize()
...     t2 = time.time()
...     print(xp.__name__, t2 - t1)
('numpy', 0.5105748176574707)
('cupy', 0.08418107032775879)
```