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
- indentation is a pretty serious thing
- a huge modules ecosystem
- binds to many different languages
- supports GPU acceleration via modules
Python introduction

Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby, Scheme, or Java.

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Python</td>
<td>🐍igator</td>
<td>100.0</td>
</tr>
<tr>
<td>2. C</td>
<td>🐍igator</td>
<td>99.7</td>
</tr>
<tr>
<td>3. Java</td>
<td>🐍igator</td>
<td>99.5</td>
</tr>
<tr>
<td>4. C++</td>
<td>🐍igator</td>
<td>97.1</td>
</tr>
<tr>
<td>5. C#</td>
<td>🐍igator</td>
<td>87.7</td>
</tr>
<tr>
<td>6. R</td>
<td>🐍igator</td>
<td>87.7</td>
</tr>
<tr>
<td>7. JavaScript</td>
<td>🐍igator</td>
<td>85.6</td>
</tr>
<tr>
<td>8. PHP</td>
<td>🐍igator</td>
<td>81.2</td>
</tr>
<tr>
<td>9. Go</td>
<td>🐍igator</td>
<td>75.1</td>
</tr>
<tr>
<td>10. Swift</td>
<td>🐍igator</td>
<td>73.7</td>
</tr>
</tbody>
</table>
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

How GPU Acceleration Works

- Application Code
  - Compute-intensive Functions: 5% of Code
  - Rest of Sequential CPU Code
- GPU
- CPU

Graph showing speedup:
- Primitive Image processing: 30x
- Stereo Vision: 7x
- Pedestrian detection (HOG): 8x
- Viola-Jones face detector: 6x
- SURF keypoints: 12x
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.tifan.de/software/PyCUDA
Author: Andreas Kloeckner
Author-email: inform@tiker.net
```
Accelerating python – PyCUDA

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

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)

# ensuring 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
mod = SourceModule(''
    __global__ void doubleify(float *a)
    {
        int idx = threadIdx.x + threadIdx.y*4;
        a[idx] *= 2;
    }
''
)

# importing to the python realm the 'doubleify' function
func = mod.get_function('doubleify')
func(a_gpu, blocks=(4, 4, 1))

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

PyCUDA workflow:
- **Edit**: Code modification
- **Run**: Code execution
- **Cache**: Check for existing compiled module
- **nvcc**: Compile the module
- **.cubin**: Compiled module
- **Upload to GPU**: Transfer to GPU
- **Run on GPU**: Execute on GPU
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

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

import numpy as np
from numba import guvectorize

@guvectorize(['void(float64[:], intp[:], float64[:])'], '(n),(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, 4))
```
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

```python
#!/usr/bin/env python
import cudamat as cm
import numpy as np

cm.cuaddons.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 T
A.add(B, target=T)

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

Timing A @ B with A.shape = (2k, 40k), B.shape = (40k, 2k)
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

<table>
<thead>
<tr>
<th>Size</th>
<th>NumPy [ms]</th>
<th>CuPy [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^1$</td>
<td>0.03</td>
<td>0.58</td>
</tr>
<tr>
<td>$10^5$</td>
<td>0.20</td>
<td>0.97</td>
</tr>
<tr>
<td>$10^6$</td>
<td>2.00</td>
<td>1.84</td>
</tr>
<tr>
<td>$10^7$</td>
<td>55.55</td>
<td>12.48</td>
</tr>
<tr>
<td>$10^8$</td>
<td>517.17</td>
<td>84.73</td>
</tr>
</tbody>
</table>

```python
>>> 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.cudart.runtime.deviceSynchronize()
...     t2 = time.time()
...     print(xp.__name__, t2 - t1)
('numpy', 0.5105748176574707)
('cupy', 0.08418107032775879)
```