GPU ACCELERATED COMPUTING WITH PYTHON

Sayak Bhowmick | Solutions Architect | December 2019

WHY PYTHON?

- Very popular in many fields
  - Most preferred language for Deep Learning and Data Science
  - Scientific coding
- Features:
  - High-level, interactive, and interpreted
  - Garbage collection (reclaim memory from deleted variables)
  - Dynamically typed
  - Good data types and structures (intrinsic complex data type and Boolean)
  - Lots of libraries (modules) available
  - Easy to extend

WHY PYTHON?

And GPUs?

- You don’t have to learn CUDA!
- High-level scripting languages are in many ways' polar-opposite to GPUs
  - GPUs are highly parallel, designed for maximum throughput, and they offer a tremendous advance in performance
  - Scripting languages such as Python favor ease of use over computational speed and do not generally emphasize parallelism
- Python coding for GPUs is getting better
  - Island for misfit Python/GPU code:
    - Lots of unsupported code hanging around
    - Not very Pythonic
  - No standards -> some standards (RAPIDS)
  - Interoperability is KEY

TL;DR - OVERALL PYTHON-GPU CODING RECOMMENDATIONS

- These apply to any language coding for GPU, but are especially important for Python
- Avoid data movement to/from GPU and CPU
  - Do as much as you can on the GPU
- “What happens on the GPU, stays on the GPU”
- Look for loops, arrays
- When “porting” to GPU, intermediate code can be slower than CPUs
  - Don’t be surprised, don’t give up (profile)

AGENDA

- Generalities
  - Quick, high-level recommendations
- Python Coding for GPUs
  - Numba (JIT compiler)
  - CuPy
  - Custom Kernels
    - CuPy
    - PyCuda

GENERALITIES

Scientific/Engineering Applications

- Good: Python has a huge number of libraries
- Bad: Python has a huge number of libraries
- Many scientific/engineering/data science codes are built using:
  - Numpy
  - SciPy
  - Scikit-learn
  - Pandas
- Much focus has been on Numpy for GPUs

NUMBA

DIRECTIVES AND THE JIT

JIT Compiler

- Language "Directives" tell compiler about code and how to build for target architecture (descriptive)
  - OpenACC for Fortran and C/C++ (compiled languages)
- Python is designed to be an interpreted language
  - No compilation or static typing
  - "Interactive"
- JIT = "Just in Time" Compiler
  - Compiles or creates object code "on the fly"
  - Combines interactivity and compilation
- Allows for computationally intensive sections of code to be run on GPU
  - Doesn't have to be a function in NVIDIA libraries

NUMBA

Introduction

- Numba is a just-in-time (JIT), type-specializing, function compiler for accelerating numerically-focused Python
  - Not every Python function can be compiled (subset of Python and Numba)
  - Look for functions with high arithmetic intensity (loops of computations)
- Typically enabled by applying a *decorator* to a Python function
- Numba runs inside the standard Python interpreter
  - Can compile for CPU or GPU

NUMBA

Introduction - 2

- Uses LLVM to compile Python functions (comes with Numba)
- The first time the function is called, the compiler creates a machine code implementation for float inputs
  - Saves the original python implementation as .py_func
  - Can test compiled function against original Python function
- Subsequent calls to function use machine code (much faster)
  - You can create compiled code prior to running
- Data Types (dtypes):
  - bool_, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64
  - numpy.complex64, numpy.complex128

NUMBA EXAMPLE

```
import numba
import math

@cuda.jit
def hypot(x, y):
    # From https://en.wikipedia.org/wiki/Hypot
    x = abs(x);
    y = abs(y);
    t = min(x, y);
    x = max(x, y);
    t = t / x;
    return x * math.sqrt(1+t*t)
```

Function Decorator

Function name/definition

Function to be compiled

NUMBA EXAMPLE

```
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32)'], target='cuda')
def Add(a, b):
  return a + b

N = 100000

# Initialize arrays
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on GPU
C = Add(A, B)
```

@vectorize turns a scalar function to an elementwise array functions

Support multiple targets: 'cpu', 'parallel', 'gpu'

List of function type signatures

A scalar function

Numba takes care of data movement to/from CPU/GPU

EXPLANATION

What happened?

- Compiled a CUDA kernel to execute the `ufunc` operation in parallel over all the input elements
- Allocated GPU memory for the input(s) and the output
- Copied the input data to the GPU
- Executed the CUDA kernel with the correct kernel dimensions given the input sizes
- Copied the result back from the GPU to the CPU
- Returned the result as a NumPy object on the host

ALLOWED NUMBA FUNCTIONS

(WARNING: It's not everything)

- Allowed statements/functions:
  - `if/elif/else`
  - `while` and `for` loops
  - Basic math operators
  - Selected functions from the math and cmath modules
  - Tuples

  http://numba.pydata.org/numba-doc/latest/cuda/cudapysupported.html

NUMBA EXAMPLE 2

Matrix Operations

- Compare Numpy, to Numba on CPU, to Numba on the GPU
- Compute sin(x)*cos(x)
  - Effectively loop over 1,000,000 values of x
  - Numpy: Vector notation
  - Numba+CPU: vectorize function
  - Numba+GPU: vectorize function
    - Includes H -> D and D -> H

NUMBA EXAMPLE 2

Define Numba functions

```
# Get all the imports we need
import numba
import numpy as np
import math

# CPU version
@numba.vectorize(['float32(float32, float32)',
                  'float64(float64, float64)'], target='cpu')
def cpu_sincos(x, y):
    return math.sin(x) * math.cos(y)

# CUDA version
@numba.vectorize(['float32(float32, float32)',
                  'float64(float64, float64)'], target='cuda')
def gpu_sincos(x, y):
    return math.sin(x) * math.cos(y)
```

- Each function has two prototypes:
  - Float32
  - Float64

NUMBA EXAMPLE 2

Generate input data

```
# Generate data
n = 1000000
x = np.linspace(0, np.pi, n)
y = np.linspace(0, np.pi, n)

# Check result
np_ans = np.sin(x) * np.cos(y)
nb_cpu_ans = cpu_sincos(x, y)
nb_gpu_ans = gpu_sincos(x, y)

print("CPU vectorize correct: ", np.allclose(nb_cpu_ans, np_ans))
print("GPU vectorize correct: ", np.allclose(nb_gpu_ans, np_ans))
```

- Data is created on Host (CPU)
- Answers are generated on host as well (creates compiled functions)
- Answers are checked (functions work as expected)

NUMBA EXAMPLE 2

Run benchmark (timeit)

```
print("NumPy")
%timeit np.sin(x) * np.cos(y)

print("CPU vectorize")
%timeit cpu_sincos(x, y)

print("GPU vectorize")
%timeit gpu_sincos(x, y)

# Optional cleanup
del x, y

NumPy
10 loops, best of 3: 32 ms per loop
CPU vectorize
10 loops, best of 3: 26.4 ms per loop
GPU vectorize
10 loops, best of 3: 15.1 ms per loop
```

JIT DECORATORS FOR GPU

- What is the difference between @cuda.jit and @vectorize(target='gpu')?
  - @vectorize will create ufuncs to take what were scalar inputs and allow vectors to be used
  - @cuda.jit creates PTX code and compiles it. It has CUDA related functions and variables.
- @vectorize is good for CPUs or learning how to port code to GPUs
- @cuda.jit is the best way to write functions that can use CUDA functions to tune compiling

CUPY

CUPY

https://github.com/cupy/cupy

- NumPy-like API accelerated with CUDA
  - Implements a subset of NumPy, but it’s very close to being complete
- Used by Chainer (DL framework popular in Japan)
- Calling sequence is like NumPy

```
>> import numpy as np
>> import cupy as cp
>> x_cpu = np.array([1, 2, 3])
>> l2_cpu = np.linalg.norm(x_cpu)
>> x_gpu = cp.array([1 ,2 ,3])
>> l2_gpu = cp.linalg.norm(x_gpu)
```

- Very Pythonic and very easy to use!

CUPY

Installation Notes

- Only available for Linux
- "`pip install cupy`"
    - Very old version
- "`conda install cupy`"

CUPY FEATURES

- Patterned after Numpy
  - Doesn't include all Numpy functions, but a very high percentage
- Data Types (dtypes):
  - bool_, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64
  - numpy.complex64, numpy.complex128
    - Test these since not every function may have it
  - Allows you to easily experiment with precision
- Can do custom kernels
- Documentation is pretty good

CUPY EXAMPLE - 1

Matrix Multiplication on GPU

```
import math
import cupy as cp

A = cp.random.uniform(low=-1., high=1., size=(64,64)).astype(cp.float32)
B = cp.random.uniform(low=-1., high=1., size=(64,64)).astype(cp.float32)

C = cp.matmul(A,B)
```

CUPY EXAMPLE - 2

SVD

```
import cupy as cp

A = cp.random.uniform(low=-1., high=1., size=(64, 64)).astype(cp.float32)

u, s, v = cp.linalg.svd(A)
```

u, s, v are still on GPU

CUPY EXAMPLE - 3

SVD – 2 (copy data back to CPU)

```
import cupy as cp
import numpy as np

A_cpu = np.random.uniform(low=-1., high=1., size=(64, 64)).astype(np.float32)
A_gpu = cp.asarray(A_cpu)

u_gpu, s_gpu, v_gpu = cp.linalg.svd(A_gpu)
print ("type(u_gpu) = ",type(u_gpu))

u_cpu = cp.asnumpy(u_gpu)
print ("type(u_cpu) = ",type(u_cpu))
```

Copy A_cpu to GPU.
Becomes CuPy object

Copy u_gpu to Host.
Becomes Numpy object

```
[sayakb@hsw225 cupy]$ python3 svd2.py
type(u_gpu) =  <type 'cupy.core.core.ndarray'>
type(u_cpu) =  <type 'numpy.ndarray’>
```

CUSTOM KERNELS

CUSTOM KERNELS

- What do you do if your needs are not met by any of the libraries/tools?
- Perhaps it's time to learn _some_ CUDA and write your own custom kernel
    - CuPy
    - PyCUDA

CUPY CUSTOM KERNELS

- Four types of kernels:
  - cupy.ElementwiseKernel
  - cupy.ReductionKernel
  - cupy.RawKernel
  - cupy.fuse
- Most custom kernels come with some (many?) CUDA functions

CUPY ELEMENTWISE KERNEL

- Focuses on kernels that operate on an element-wise basis
- Consist of 4 components:
  - input argument list (comma-separated)
  - output argument list (comma-separated)
  - loop body code
  - kernel name

```
import cupy as cp

kernel = cp.ElementwiseKernel(
    'float32 x, float32 y', 'float32 z',
    '''if (x - 2 > y) {
            z = x * y;
        } else {
            z = x + y;
    }''', 'my_kernel')
```

CUPY REDUCTION KERNEL

- Has four parts:
  - Identity value: Initial value of the reduction
  - Mapping expression: Preprocesses each element to be reduced
  - Reduction expression: An operator to reduce the multiple mapped values. Two special variables, `a` and `b`, are used for this operand
  - Post-mapping expression: Transforms the reduced values. The special variable `a` is used as input. The output should be written to the output variable.
- This function uses type placeholders for both the input and output

```
import cupy as cp

l2norm_kernel = cp.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)
```

CUPY RAW KERNEL

- Define using “raw” CUDA code
- Lacks helpful variables or CuPy functions or elementwise or reduction custom functions

```
import cupy as cp

add_kernel = cp.RawKernel(r'''
    extern "C" __global__
    void my_add(const float* x1, const float* x2, float* y) {
        int tid = blockDim.x * blockIdx.x + threadIdx.x;
        y[tid] = x1[tid] + x2[tid];
    }
    ''', 'my_add')

x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
y  = cp.zeros((5, 5), dtype=cp.float32)
add_kernel((5,), (5,), (x1, x2, y))  # grid, block and arguments

print(y)

array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

CUPY FUSE

- This decorator can be used to define an elementwise or reduction kernel more easily than ElementwiseKernel or ReductionKernel

```
@cupy.fuse()
def squared_diff(x, y):
    return (x - y) * (x - y)

x = cupy.arange(10)
y = cupy.arange(10)[::-1]

squared_diff(x, y)

array([81, 49, 25,  9,  1,  1,  9, 25,
49, 81])
```

PYCUDA – CUSTOM KERNLS

- Earliest integration of Python and CUDA but requires a knowledge of CUDA and Python
- Really a tool to compile and interface CUDA code with Python
  - It does have some Python variables that correspond to CUDA variables
  - Functions to make GPU coding easy (https://documen.tician.de/pycuda/util.html)
    - Example: Copying data to/from GPU
- PyCUDA's base layer is written in C++

PYCUDA SIMPLE EXAMPLE

- Generate random numbers on CPU
- Copy to GPU
- Double the values in the array in parallel
- Copy the data back to CPU

SIMPLE PYCUDA

- `pycuda.autoinit` is used for automatic initialization, context creation, and cleanup
- Numpy is used to generate a random 4x4 array
- Allocate space (`mem_alloc`) on GPU
- Copy CPU data to GPU
  - Use built-in PyCUDA function `memcpy_htod()`
- Define "C" code using SourceModule
- Compile function (calls nvcc)
- Call function with "execution configuration" (grid properties)
- Copy data back to CPU

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule

import numpy as np

a = np.random.randn(4, 4)
a = a.astype(np.float32)   # convert to FP32

a_gpu = cuda.mem_alloc(a.nbytes)   # Malloc on GPU
cuda.memcpy_htod(a_gpu, a)    # Copy data to GPU

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

func = mod.get_function("doublify")   # compile function
func(a_gpu, block = (4, 4, 1))   # "call" function, 4x4 grid

a_doubled = np.empty_like(a) # Create data for results
cuda.memcpy_dtoh(a_doubled, a_gpu)
```

SUMMARY

SUMMARY

- Coding GPUs for Python can be very easy –or- more complex, but with more control
    - Compile Python code using Numba (almost 100% pure python) – don't need to know CUDA!
    - Use Numpy like functions on GPU with CuPy – you don't necessarily need to know CUDA!!
    - Custom kernels with CuPy and PyCUDA - need to know CUDA!!!
- Many of these tools are interoperable
    - The start of making GPU's first-class citizens with Python

GET STARTED TODAY

You might already have a CUDA-capable GPU in your laptop or desktop PC!

NVIDIA Developer Zone
https://developer.nvidia.com/

CUPY
https://cupy.chainer.org/

NUMBA
http://numba.pydata.org/numba-doc/latest/cuda/index.html

PYCUDA
https://documen.tician.de/pycuda/index.html

sayakb@nvidia.com

Q & A

NVIDIA