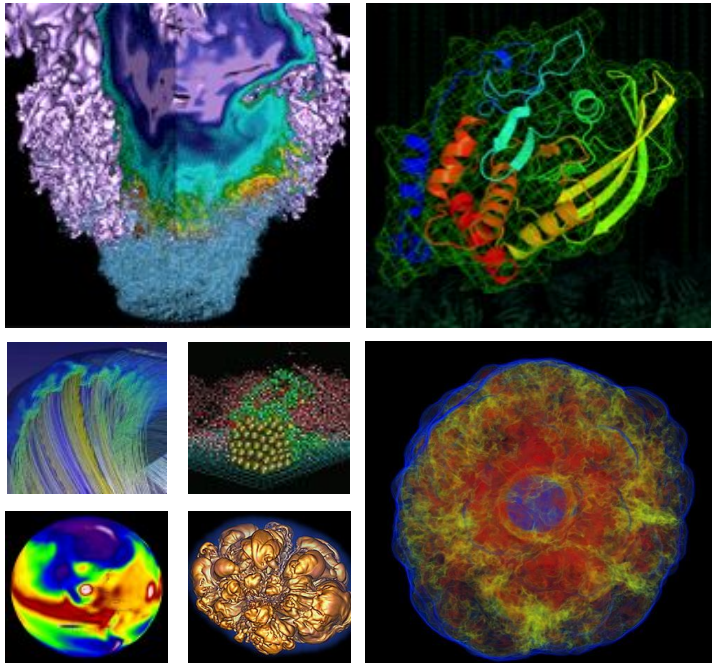
Python on GPUs (work in progress!)



Laurie Stephey

GPUs for Science Day, July 3, 2019

Rollin Thomas, NERSC

Lawrence Berkeley National Laboratory

NERSC

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Python is friendly and popular

TIOBE Index for June 2019

June Headline: Python continues to soar in the TIOBE index

This month Python has reached again an all time high in TIOBE index of 8.5%. If Python can keep this pace, it will probably replace C and Java in 3 to 4 years time, thus becoming the most popular programming language of the world. The main reason for this is that software engineering is booming. It attracts lots of newcomers to the field. Java's way of programming is too verbose for beginners. In order to fully understand and run a simple program such as "hello world" in Java you need to have knowledge of classes, static methods and packages. In C this is a bit easier, but then you will be hit in the face with explicit memory management. In Python this is just a one-liner. Enough said.

Jun 2019	Jun 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.004%	-0.36%
2	2		C	13.300%	-1.64%
3	4	▲	Python	8.530%	+2.77%
4	3	▼	C++	7.384%	-0.95%
5	6	▲	Visual Basic .NET	4.624%	+0.86%

Screenshots from: https://www.tiobe.com/tiobe-index/

So you want to run Python on a GPU?

You have some Python code you like.

Can you just run it on a GPU?

```
import numpy as np
from scipy import special
import gpu ?
```

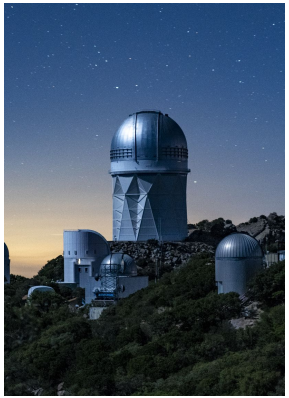
Unfortunately no.

What are your options?

Right now, there is no “right” answer

- CuPy
- Numba
- pyCUDA (https://mathema.tician.de/software/pycuda/)
- pyOpenCL (https://mathema.tician.de/software/pyopencl/)
- Rewrite kernels in C, Fortran, CUDA...

DESI: Our case study

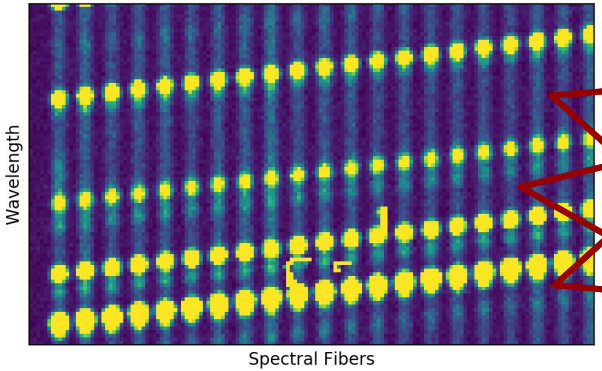


Now

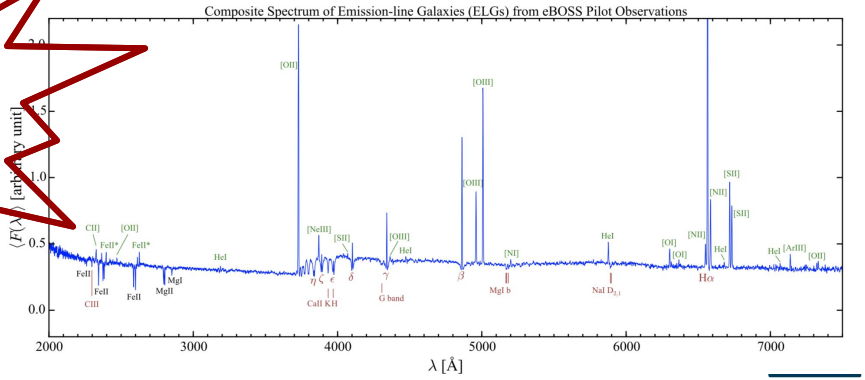
2020

Perlmutter

Goal: High quality output spectra



Spectral Extraction



CuPy (https://cupy.chainer.org/)

- Developed by Chainer, supported in RAPIDS
- Meant to be a drop-in replacement for NumPy
- Some, but not all, NumPy coverage

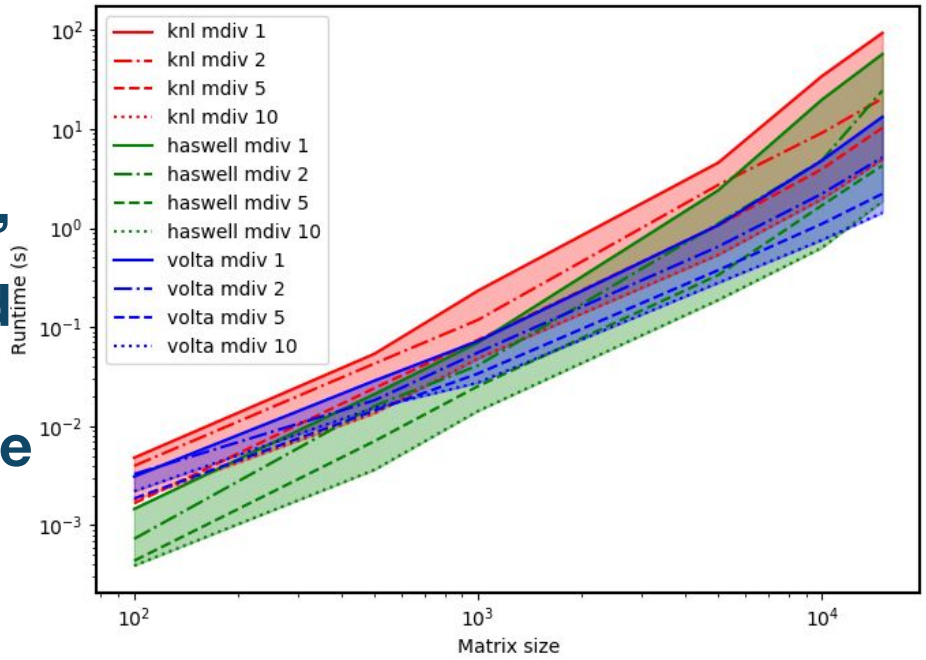
NumPy	CuPy
numpy.abs	cupy.abs
numpy.absolute	cupy.absolute
numpy.add	cupy.add
numpy.add_docstring	-
numpy.add_newdoc	-
numpy.add_newdoc_ufunc	-
numpy.alen	-
numpy.all	cupy.all
numpy.allclose	cupy.allclose

```
import numpy as np
import cupy as cp

cpu_ans = np.abs(data)

#same thing on gpu
gpu_data = cp.asarray(data)
gpu_temp = cp.abs(gpu_data)
gpu_ans = cp.asnumpy(gpu_temp)
```

Screenshot from:
https://docs-cupy.chainer.org/en/stable/reference/comparison.html

eigh in CuPy

- Important function for DESI
- Compared CuPy eigh on Cori Volta GPU to Cori Haswell and Cori KNL
- Tried “divide-and-conquer” approach on both CPU and GPU (1, 2, 5, 10 divisions)
- Volta wins only at very large matrix sizes
- Major pro: eigh really easy to use in CuPy!



legval in CuPy

```python
import cupy as cp
#expects cupy arrays as input
def legval_cupy(x, c):

    #x and c are cupy arrays

    ndd = c.shape[0] #will be an int
    #and now change nd into a cupy array
    nd = cp.array(ndd)

    xlen = x.shape[0] #will be an int

    c0=c[-2]*cp.ones(xlen) #cupy
    c1=c[-1]*cp.ones(xlen) #cupy

    for i in range(3, ndd, + 1):
        tmp = c0
        nd = nd - 1
        nd_inv = 1/nd
        c0 = c[-i] - (c1*(nd - 1))*nd_inv
        c1 = tmp + (c1*x*(2*nd - 1))*nd_inv
        cupy_result = c0 + c1*x
    return cupy_result
```

- Easy to convert from NumPy arrays to CuPy arrays
- This function is ~150x slower than the cpu version!

```
10 iterations of legval_numba ran in 0.0012524127960205078 s
10 iterations of legval_cupy ran in 0.1504673957824707 s
```

- This implies there is probably some undesirable data movement between the cpu and gpu
- Maybe I'm just doing it wrong

Numba (https://numba.pydata.org/)

- We like Numba because it has worked well for JIT-compiling code for a CPU
- Unlike CPU Numba, CUDA Numba doesn't allow most NumPy (which is a problem for scientific users like DESI)
- Numba for GPUs doesn't look very much like "normal" Python, looks a lot more like CUDA

legval in Numba for GPU (in progress!)

Step 1: Invoke kernel with thread information

```
threadsperblock = 32
blockspergrid = (xx.size + (threadsperblock - 1)) // threadsperblock
for i in range(niter):
    legval_numba_gpu[blockspergrid, threadsperblock](xx, cc, c0, c1, results)
```

Step 2: Numba gpu function

```
from numba import cuda
@cuda.jit
def legval_numba_gpu(x, c, c0, c1, results):

    nd = len(c)
    ndd = nd
    xlen = x.size
    #c0=c[-2]*np.ones(xlen)
    #c1=c[-1]*np.ones(xlen)
    #need a way to do this without numpy

    for i in range(3, ndd + 1):
    #instead use i from cuda.grid
    #i = cuda.grid(1)
        tmp = c0
        nd = nd - 1
        nd_inv = 1/nd
        c0 = c[-i] - (c1*(nd - 1))*nd_inv
        c1 = tmp + (c1*x*(2*nd - 1))*nd_inv
        #gpu kernels have no return
        results = c0 + c1*x
```

Step 3: Troubleshoot Numba type errors! (Common)

```
This error is usually caused by passing an argument of a type that is unsupported by the named f
unction.
[1] During: typing of intrinsic-call at /global/cscratch1/sd/stephey/git_repo/specter/py/specter
/util/util.py (299)

File "../git_repo/specter/py/specter/util/util.py", line 299:
def legval_numba_gpu(x, c, c0, c1, results):
    <source elided>
        nd_inv = 1/nd
        c0 = c[-i] - (c1*(nd - 1))*nd_inv
        ^
```

Step 4: Profit!

A working cuda.jit example

- Screenshot from: https://numba.pydata.org/numba-doc/dev/cuda/examples.html

```
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication of C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

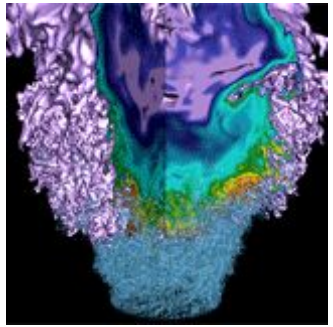
Some words of caution

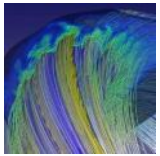
- Code that runs well on a CPU might not be good for a GPU
- More than just porting some kernels/functions, it could require a substantial rewrite
- How to avoid doing this every few years? How to be able to run on many architectures?
- Unfortunately there are no easy answers

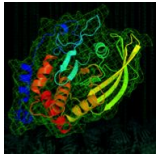
What have we learned?

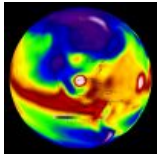
- Python on GPUs is still evolving
- We have tried:
  - CuPy → difficulty easy, but not every NumPy/SciPy function
  - Numba → difficulty hard, looks less like Python, but more flexible
- Our job is to help DESI and our users figure out the best strategy (performance + maintainability + portability)
- Stay tuned!

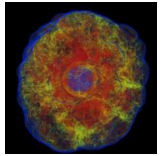
Thank you!

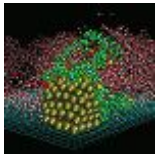












NERSC

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB