奧義智慧科技

Four Ways to boost your NumPy Performance

Cheng-Lin Yang (clyang)

$whoami

- Cheng-Lin Yang : @clyang : @clyangtw
- Working for Cybersecurity company: 奧義智慧科技
  - Member of Machine Learning team
  - We are hiring: Full-time and interns

奧義官網 | 奧義粉專 | 奧義 Medium

Before we start, one quick question for you.
Which code runs faster?

A. `np.power(x, 8)`
B. `x ** 8`
C. `x * x * x * x * x * x * x * x`

Answer: C

x * x * x * x * x * x * x * x

```
np.power(x, 8)    : 2.017534017562866
x**8              : 2.033080577850342
x*x*x*x*x*x*x*x   : 0.40352702140808105
```

奧義智慧科技

Today’s Example

(and benchmark)

logsumexp (LSE) - I

- softmax function is defined as:

$$\sigma(z)_j = \frac{e^{Z_j}}{\sum_{k=1}^{K} e^{z_k}}$$ for j = 1, … , k where Z is a K-dimensional vector

- logumexp is a log-sum-trick which prevents over/underflow during softmax calculation

logsumexp (LSE) - II

- However, floating point underflow will occur during summation. For example:

$$\sigma(z)_j = \frac{e^{Z_j}}{\sum_{k=1}^{K} e^{z_k}} = \frac{e^{Z_j}}{\underbrace{e^{z_1} + e^{z_2} + \cdots + e^{z_k}}_{134217728} + \underbrace{e^{z_{k+1}}}_{\frac{1}{134217728}}}$$

```python
import numpy as np

a = np.float64(134217728)
print( a + (1/a) + (1/a) + (1/a) )
```

```
134217728.0
```

logsumexp (LSE) - III

- The problem can be solved by this simple trick

$$y = x_{max} + \log \sum_{i=1}^{n} e^{x_i - x_{max}}$$

- Applying previous example:

```
import numpy as np

x = np.log(134217728.0)
x_inv = np.log(1/134217728.0)
x_max = max(x, x_inv)
np.exp(x_max + np.exp( (x - x_max) + (x_inv - x_max) )  )
```

```
134217728.0000001
```

SciPy has it.
Why rebuild the wheel?

- Too many checks drag performance
  - For general purpose usage
- Caveats to improve performance:
  - Assuming the input data is following the conditions, so we can remove the unnecessary checks.
  - Verify what you actual need and simplify the code as per your requirements.
  - For example: only 1-D arrays will be used in my following scenario

```python
b is not None:
    a, b = np.broadcast_arrays(a, b)
    if np.any(b == 0):
        a = a + 0.  # promote to at least float
        a[b == 0] = -np.inf

a_max = np.amax(a, axis=axis, keepdims=True)

if a_max.ndim > 0:
    a_max[~np.isfinite(a_max)] = 0
elif not np.isfinite(a_max):
    a_max = 0

if b is not None:
    b = np.asarray(b)
    tmp = b * np.exp(a - a_max)
else:
    tmp = np.exp(a - a_max)

# suppress warnings about log of zero
with np.errstate(divide='ignore'):
    s = np.sum(tmp, axis=axis, keepdims=keepdims)
    if return_sign:
        sgn = np.sign(s)
        s *= sgn  # /= makes more sense but we need zero -> zero
    np.log(s)
```

logsumexp in NumPy

logsumexp in NumPy

- Based on my scenario. logsumexp can be implemented as follows:

```python
import numpy as np

def logsumexp(x):
    x_max = np.max(x)
    log_part = np.exp(x - x_max)
    return np.log(np.sum(log_part)) + x_max
```

$$y = x_{max} + \log \sum_{i=1}^{n} e^{x_i - x_{max}}$$

NumPy vs. SciPy

```python
import time
import numpy as np
from scipy.special import logsumexp

def logsumexp_numpy_v1(x):
    x_max = np.max(x)
    rest = np.exp(x - x_max)
    return x_max + np.log(np.sum(rest))

test_data =  np.random.uniform(-1, 1, (50000, 1)).astype(np.float64)
t0 = time.time()
for _ in range(5000):
    r1 = logsumexp_numpy_v1(test_data)
t1 = time.time()
for _ in range(5000):
    r2 = logsumexp(test_data)
t2 = time.time()

print('NumPy version: {}'.format(t1 - t0))
print('SciPy version: {}'.format(t2 - t1))
```

- Results:

```
clyang@snoopy:~/pycon_example$ python3 numpy_scipy.py
NumPy version: 6.252124786376953
SciPy version: 6.717078685760498
```

奧義智慧科技

Solution 1: CuPy

CuPy

- https://github.com/cupy/cupy
- Providing NumPy-compatible ND-array on CUDA
  - Utilising GPU power
  - Compatible with Existing CUDA kernel
- Providing many NumPy equivalent functions so you can minimize code refactoring effort
- Check the differences!
  - https://docs.cupy.dev/en/stable/reference/difference.html
- Moving data between CPU and GPU is expensive!

logsumexp in CuPy

```python
import cupy as cp

def logsumexp_cupy_v1(x):
    x_max = cp.max(x)
    rest = cp.exp(x - x_max)

    return x_max + cp.log(cp.sum(rest))

test_data =  np.random.uniform(-1, 1, (50000, 1)).astype(np.float64)
t0 = time.time()
for _ in range(5000):
    r1 = logsumexp_cupy_v1(cp.array(test_data))
t1 = time.time()
print('CuPy  version: {}'.format(t0 - t1))
```

- Result:

```
clyang@snoopy:~/pycon_example$ python3 1.py
CuPy  version: 1.6152799129486084
```

奧義智慧科技

Solution 2: Numba

Numba

- Two modes you need to know
  - nopython mode (equals to @njit)
    - Allows you to get rid of Python's GIL
  - object mode
- @njit + OpenMP is easy to parallelize computation without GIL limitation

```python
from numba import njit, prange

@njit(parallel=True)
def parallel_sum(A):
    sum = 0.0
    for i in prange(A.shape[0]):
        sum += A[i]

    return sum
```

logsumexp by Numba

```python
import time
import numpy as np
from numba import jit, njit

@jit
def logsumexp_numba_cpu_jit(x):
    x_max = np.max(x)
    rest = np.exp(x - x_max)
    return x_max + np.log(np.sum(rest))

@njit(fastmath=True) # eqaul to @jit(nopython=True)
def logsumexp_numba_cpu_njit(x):
    x_max = np.max(x)
    rest = np.exp(x - x_max)
    return x_max + np.log(np.sum(rest))

test_data =  np.random.uniform(-1, 1, (50000, 1)).astype(np.float64)
t0 = time.time()
for _ in range(5000):
    r1 = logsumexp_numba_cpu_jit(test_data)
t1 = time.time()
for _ in range(5000):
    r2 = logsumexp_numba_cpu_njit(test_data)
t2 = time.time()

print('CPU + Jit version            : {}'.format(t1 - t0))
print('CPU + Jit + Nopython version : {}'.format(t2 - t1))
```

- Results:

```
clyang@snoopy:~/pycon_example$ python3 nb.py
CPU + JIT version            : 6.72649884223938
CPU + JIT + Nopython version : 6.025193691253662
```

奧義智慧科技

Solution 3: Pythran

Pythran

- https://pythran.readthedocs.io/en/latest/
  - Active development and has fast growing community
- Using ahead-of-time compiling approach
  - LLVM + compiler does all the magic!
- Supporting a subset of Python and NumPy code
  - Works on Python 2.7 and 3.6/7/8
- Similar to Numba, you have to put a special decorator before the function you want to boost
  - OpenMP can also be used with Pythran

logsumexp in Pythran

- First, write the Python code as usual. (pythran_logsumexp.py)

```python
import numpy as np

#pythran export logsumexp_pythran(float64[:, :])
def logsumexp_pythran(x):
    x_max = np.max(x)
    rest = np.exp(x - x_max)

    return x_max + np.log(np.sum(rest))
```

- Compile it by using:
  - CXX=clang++ pythran -DUSE_XSIMD -march=native -O3 pythran_logsumexp.py

logsumexp in Pythran

- Import the just compiled module and run it!

```
import time
import numpy as np
from pythran_logsumexp import logsumexp_pythran

test_data =  np.random.uniform(-1, 1, (50000, 1)).astype(np.float64)
t0 = time.time()
for _ in range(5000):
    r1 = logsumexp_pythran(test_data)
t1 = time.time()

print('Pythran : {}'.format(t1 - t0))
```

- Result

```
clyang@snoopy:~/pycon_example$ python3 pythran_bench.py
Pythran : 5.456472158432007
```

奧義智慧科技

Solution 4: Cython

Cython

- Advantage
  - Utilising 3rd party C library can execute faster
  - Releasing GIL
  - Still have the run-time check for common problem provided by Python
  - Cython syntax is very similar to Python

- Disadvantage
  - You have to handle memory by yourself (if malloc is used)
  - To get ultimate performance, writing C code with low-level intrinsics CANNOT be avoided (this can be painful)

logsumexp in Cython

- First, write the Cython code. (cython_logsumexp.pyx)

```
cimport cython
import numpy as np
cimport numpy as np
from libc.math cimport exp, log

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef lse_cython(np.ndarray[np.float64_t, ndim=1] a):
    cdef unsigned int i
    cdef double result = 0.0
    cdef double max_a = a[0]
    for i in range(1, a.shape[0]):
        if (a[i] > max_a):
            max_a = a[i]

    for i in range(a.shape[0]):
        result += exp(a[i] - max_a)

    return max_a + log(result)
```

logsumexp in Cython

- Preparing a “setup.py”

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("cython_logsumexp.pyx")
)
```

- Use the compiled module

```
import cython_logsumexp

test_data32 =  np.random.uniform(-1, 1, (50000,)).astype(np.float32)
for _ in range(5000):
    r2 = cython_logsumexp.lse_cython(test_data32)
```

Cython + External C code

```
cimport cython
import numpy as np
cimport numpy as np

cdef extern from "avx_mathlib.h":
    float avx_logsumexp(const float* buf, int N) nogil

@cython.boundscheck(False)
@cython.wraparound(False)
def logsumexp(np.ndarray[dtype=np.float32_t, ndim=2] a):
    if not (a.flags['C_CONTIGUOUS'] or a.flags['F_CONTIGUOUS']):
        raise TypeError('a must be contiguous')

    return avx_logsumexp(&a[0,0], a.size)
```

Cython + External C code

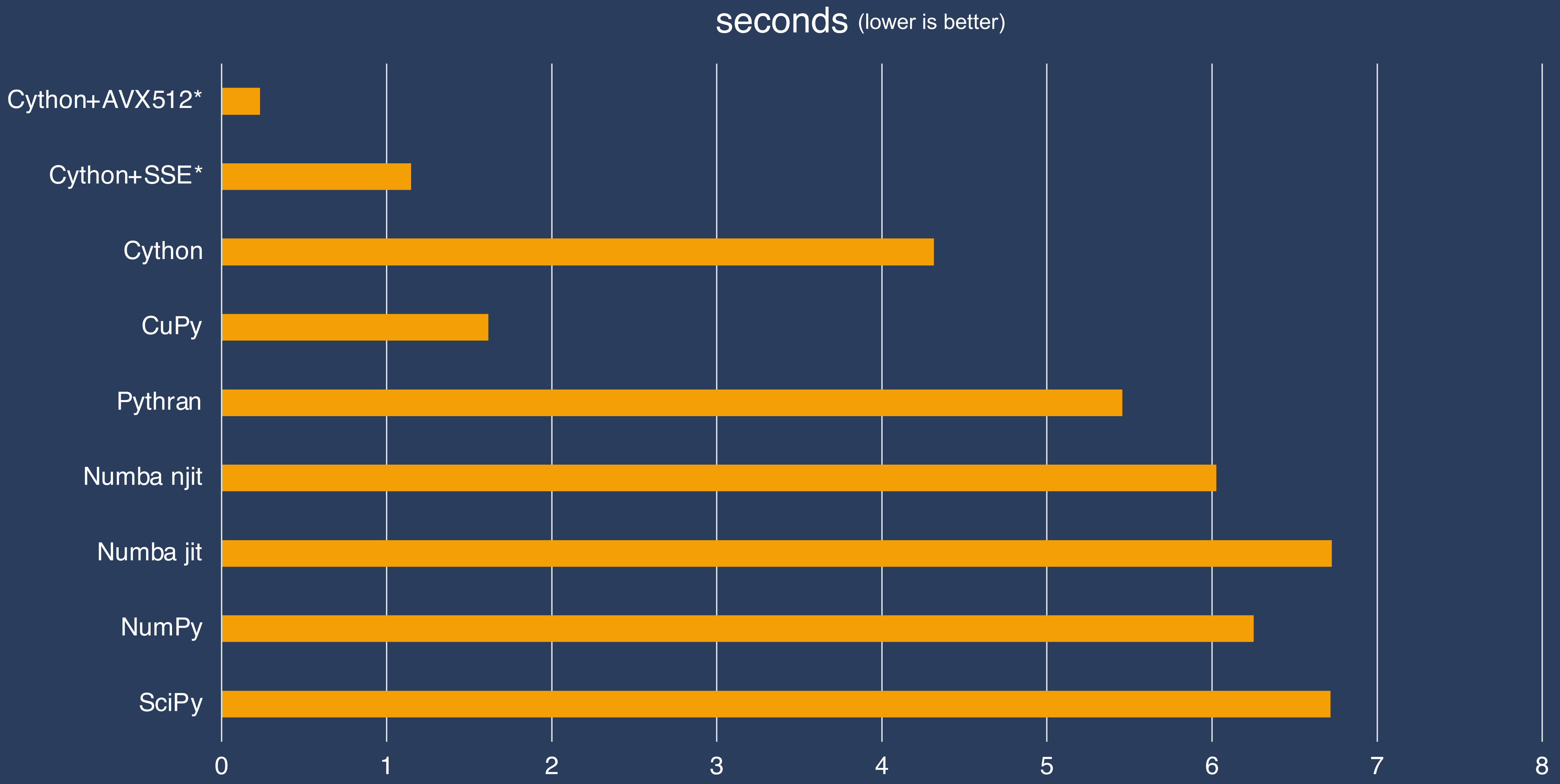
```
_r1 = _mm512_add_ps(_r1, _mm512_shuffle_f32x4(_r1, _r1, _MM_SHUFFLE(0,0,3,2)));
r = _mm512_castps512_ps128(_mm512_add_ps(_r1, _mm512_shuffle_f32x4(_r1, _r1, _MM_SHUFFLE(0,0,0,1))));
r = _mm_hadd_ps(r,r);
sum1 = _mm_cvtss_f32(_mm_hadd_ps(r,r));

_r2 = _mm512_add_ps(_r2, _mm512_shuffle_f32x4(_r2, _r2, _MM_SHUFFLE(0,0,3,2)));
r = _mm512_castps512_ps128(_mm512_add_ps(_r2, _mm512_shuffle_f32x4(_r2, _r2, _MM_SHUFFLE(0,0,0,1))));
r = _mm_hadd_ps(r,r);
sum2 = _mm_cvtss_f32(_mm_hadd_ps(r,r));
```

奧義智慧科技

So, which is better?

Benchmark

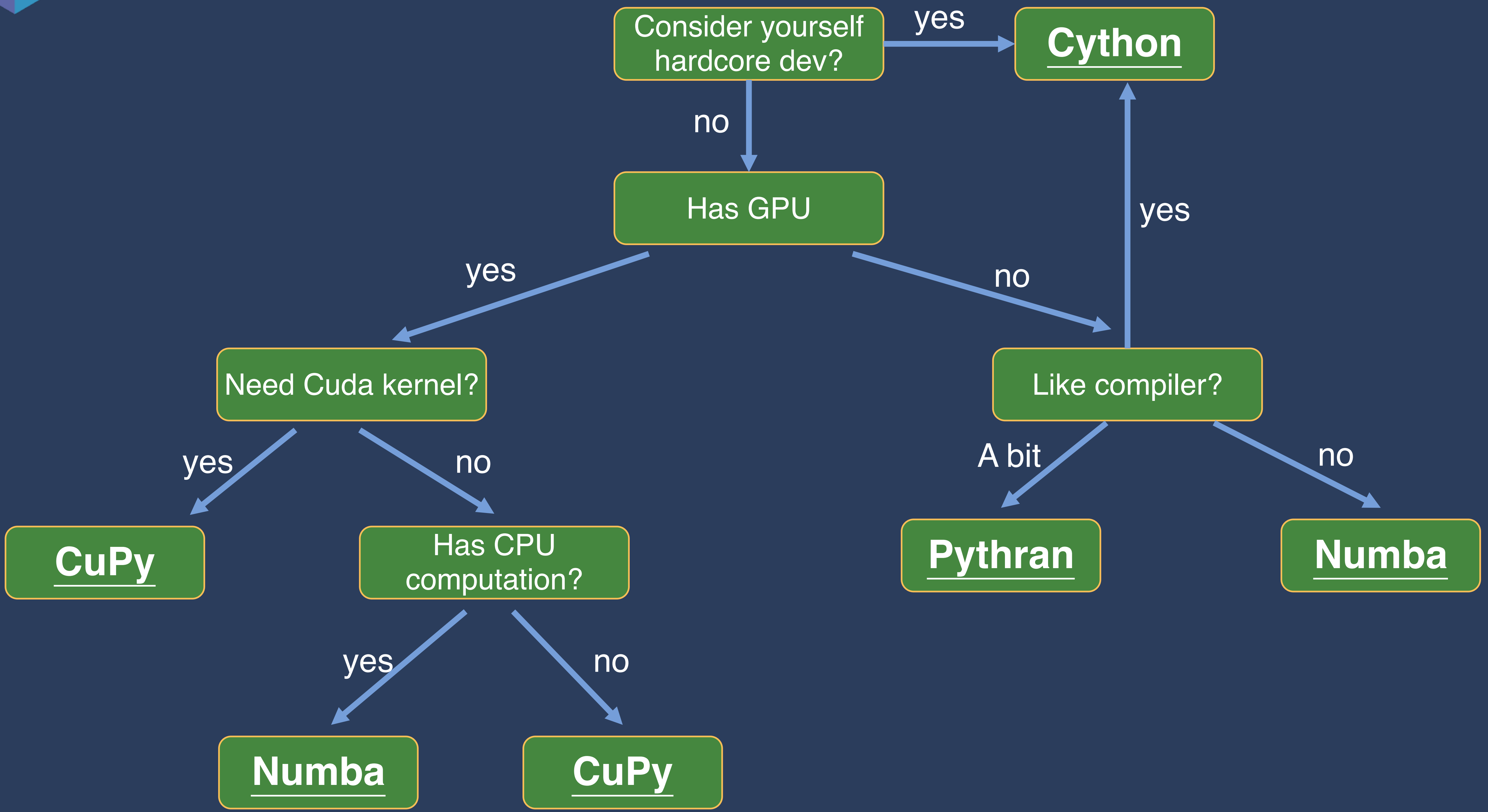
- All benchmarks were run on a bare metal machine with the following specifications:
  - CPU: Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz
  - RAM: 256GB DDR4 with ECC
  - GPU: GeForce GTX 1080 Ti
  - Python and Library information:
    - Python 3.6.9
    - Cuda 10.2
    - NumPy 1.18.1
    - CuPy   7.8.0
    - Numba 0.51.0
    - Pythran 0.9.6


seconds (lower is better)
Cython+AVX512*
Cython+SSE*
Cython
CuPy
Pythran
Numba njit
Numba jit
NumPy
SciPy
0
1
2
3
4
5
6
7
8

奧義智慧科技

My decision tree

CuPy, Numba, Pythran and Cython


Consider yourself hardcore dev?
yes
Cython
no
Has GPU
yes
yes
no
Need Cuda kernel?
Like compiler?
yes
no
A bit
no
CuPy
Has CPU computation?
Pythran
Numba
yes
no
Numba
CuPy

Four Takeaways

- If you have GPU(s), try CuPy first!
- If you only have CPU, use Numba first
  - If it works, try Pythran to get more performance
- Intel intrinsics can be joyful and painful
  - In some case, it’s even faster than GPU
  - A steep learning curve
- Each solution supports different number of NumPy functions.
  - You can easily find out which function doesn’t work (program stops :P)
  - Check its document to see which functions are provided
  - If A doesn’t work, B might work!

Thank You

奧義官網

奧義粉專

奧義 Medium