Investigation on How to Accelerate Python on Both Multicore CPUs and Manycore GPUs

Dingjun Chen

Calgary, Alberta

Make python fast with numba

1. Calculate PI with python

- `import random`
- `def pi(npoints):`
- `  n_in_circle = 0`
- `  for i in range(npoints):`
- `    x = random.random()`
- `    y = random.random()`
- `    if (x**2+y**2 < 1):`
- `      n_in_circle += 1`
- `  return 4*n_in_circle / npoints`
- `%timeit pi(100000000)`
- `*Run time cost: 44.2 s ± 228 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)`
- **`*Note:`** I did this test on Google Colab.

Make python fast with numba

2. Calculate PI with python numba JIT

```
from numba import njit
@njit
def fast_pi(npoints):
  n_in_circle = 0
  for i in range(npoints):
    x = random.random()
    y = random.random()
    if (x**2+y**2 < 1):
      n_in_circle += 1
  return 4*n_in_circle / npoints
%timeit fast_pi(100000000)
```

- *Run time cost: 1.25 s ± 6.99 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- Conclusion: Speedup= 44.2/1.25 =35.36, Numba can accelerate Python code about 35X via JIT compilation in terms of tests on the calculation of Pi.
- Cause: A Just-In-Time (JIT) compiler is **a feature of the run-time interpreter**, that instead of interpreting bytecode every time a method is invoked, will compile the bytecode into the machine code instructions of the running machine, and then invoke this object code instead.

What’s difference between JIT and NJIT?

- NJIT means that **the function is compiled in nopython mode**. A *dict* , *list and tuple* are python objects and therefore not supported. Not as arguments and not inside the function.

- from numba import jit
- import numpy as np
- @jit
- def fast_pi(npoints):
-   n_in_circle = 0
-   for i in range(npoints):
-     x = np.random.random()
-     y = np.random.random()
-     if (x**2+y**2 < 1):
-       n_in_circle += 1
-   return 4*n_in_circle / npoints
- %timeit fast_pi(100000000)
- *Run time cost: 1.25 s ± 17.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- Conclusion: There is no difference in run time cost in case of either JIT or NJIT in terms of tests on the calculation of Pi.
- *Note: @njit == @jit(nopython=True)
- *Note: I did this test on Google Colab.

Simultaneously using both Numba's prange and JIT compilation for accelerating Python? More investigation will need to be made soon.

- from numba import njit, prange
- @njit
- def fast_pi(npoints):
-   n_in_circle = 0
-   for i in prange(npoints):
-     x = random.random()
-     y = random.random()
-     if (x**2+y**2 < 1):
-       n_in_circle += 1
-   return 4*n_in_circle / npoints
- %timeit fast_pi(100000000)
- *Run time cost: 1.25 s ± 9.31 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- Conclusion: There is no performance improved in case of simultaneously using both JIT and prange in terms of tests on the calculation of Pi.
- *Note: I did this test on Google Colab.

what is CuPy?

- CuPy is **an open-source array library for GPU-accelerated computing with Python**. CuPy utilizes CUDA Toolkit libraries including cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN and NCCL to make full use of the GPU architecture.
- CuPy implements Numpy arrays on Nvidia GPUs by leveraging the CUDA GPU library. With that implementation, superior parallel speedup can be achieved due to the many CUDA cores GPUs have.
- Please see below performance comparison between NumPy and CuPy. Source codes are given in the next slide.
- Test result: Numpy cost: 3.906684160232544 CuPy cost: 0.7191150188446045
- CuPy Speedup: 3.906684/0.719115=5.432627
- *Note: I did this test on Kaggle notebook with its free GPU platform.

- ## Source codes for the performance comparison between Numpy and Cupy.
- import numpy as np
- import cupy as cp
- import time
- ## Numpy and CPU
- s = time.time()
- x_cpu = np.ones((1000,1000,1000))
- x_cpu *= 5
- x_cpu *= x_cpu
- x_cpu += x_cpu
- e = time.time()
- print("Numpy cost:",e - s)
- ##CuPy and GPU
- s = time.time()
- x_gpu = cp.ones((1000,1000,1000))
- x_gpu *= 5
- x_gpu *= x_gpu
- x_gpu += x_gpu
- cp.cuda.Stream.null.synchronize()
- e = time.time()
- print("CuPy cost:",e - s)
- Run Results: Numpy cost: 3.906684160232544 CuPy cost: 0.7191150188446045
- CuPy Speedup: 3.906684/0.719115=5.432627

Conclusions

- Numba can be used to accelerate Python on multicore CPU computing platform instead of Numpy.
- CuPy can be used to accelerate Python on many-core GPU computing platform instead of Numpy.
- More investigation will need to be done soon in order to further accelerate Python codes so that the Python codes can run as fast as Fortran and C/C++ codes.