6 ways to significantly speed up Pandas with a couple lines of code. Part 1

In this article I will tell you about six tools that can significantly speed up your pandas code. For most tools, just install the module and add a couple lines of code.

Magomed Aliev

Pandas has long been an indispensable tool for any developer thanks to a simple and understandable API, as well as a rich set of tools for cleaning, researching and analyzing data. And everything would be fine, but when it

comes to **data that does not fit into RAM** or **require complex calculations**, pandas performance is not enough.

In this article, I will not describe qualitatively different approaches to data analysis, such as Spark or DataFlow. Instead, I will describe six interesting tools and demonstrate the results of their use:

Chapter 1:

- Numba
- Multiprocessing
- Pandarallel

Chapter 2:

- Swifter
- Modin
- Dask

Numba

This tool directly accelerates Python itself. Numba is a **JIT compiler** that likes loops, mathematical operations and Numpy, which is a Pandas core lib. Let's check in practice what advantages it gives.

We will simulate a typical situation — you need to add a new column by applying some function to the existing one

using the `apply` method.

```
import pandas as pd
import numpy as np
import numba# create a table of 100,000 rows and
df = pd.DataFrame(np.random.randint(0,100,size=(1
def multiply(x):
    return x * 5

# optimized version of this function
@numba.vectorize
def multiply_numba(x):
    return x * 5
```

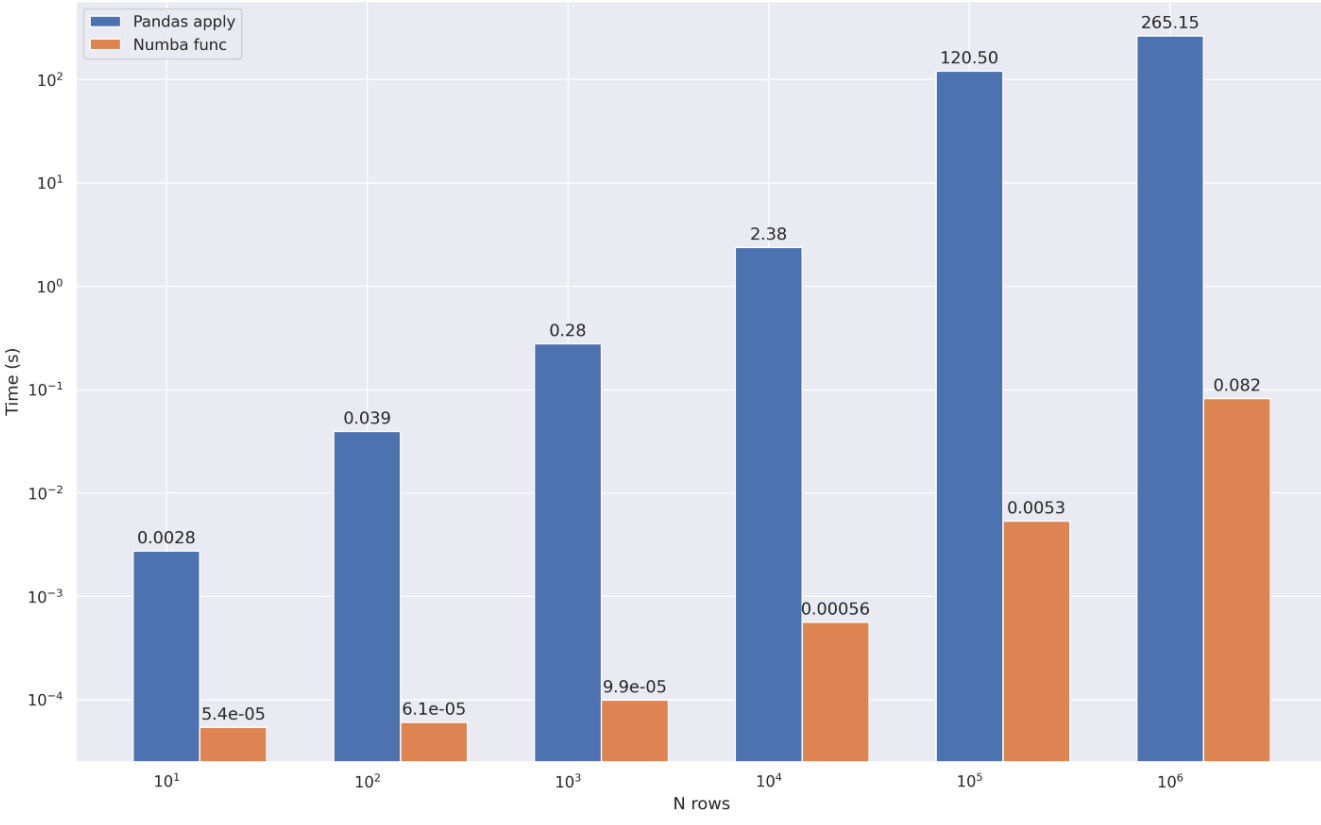
As you can see, you do not need to change anything in your code. Just add a decorator. Now look at the results:

```
# our function
In [1]: %timeit df['new_col'] = df['a'].apply(mu
23.9 ms ± 1.93 ms per loop (mean ± std. dev. of 7
In [2]: %timeit df['new_col'] = df['a'] * 5
545 µs ± 21.4 µs per loop (mean ± std. dev. of 7
# we use vector of values so that numba itself pe
In [3]: %timeit df['new_col'] = multiply_numba(df
329 µs ± 2.37 µs per loop (mean ± std. dev. of 7
```

The optimized version is ~ **70 times faster!** However, in absolute terms, Pandas implementation is slightly behind, so let's take a more complex case. Lets define new functions:

```
# square the values and take their mean
def square_mean(row):
    row = np.power(row, 2)
    return np.mean(row)
# usage:
# df['new_col'] = df.apply(square_mean, axis=1)#
# so that we use a two-dimensional numpy array
@numba.njit
def square_mean_numba(arr):
    res = np.empty(arr.shape[0])
    arr = np.power(arr, 2)
    for i in range(arr.shape[0]):
        res[i] = np.mean(arr[i])
    return res
# usage:
# df['new_col'] = square_mean_numba(df.to_numpy(
```

Let's build a graph showing the dependence of the calculation time on the number of rows in the data frame:



Summary

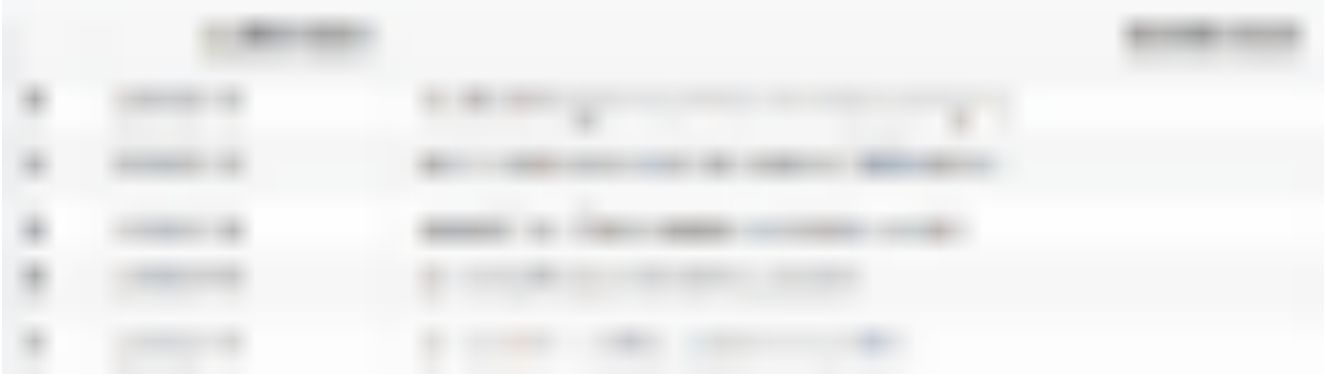
- It is possible to achieve **more than x1000 acceleration**
- Sometimes you need to rewrite the code to use Numba
- **Cannot be used everywhere**, optimizing mathematical operations is the main case
- Keep in mind that Numba does not support all features of python и numpy

Multiprocessing

The first thing that comes to mind when it comes to processing a large dataset is to parallelize all the calculations. This time we will not use any third-party libraries — only python tools.

We will use text processing as an example. Below I took dataset with news headlines. Like last time, we will try to speed up the `apply` method:

```
df = pd.read_csv('abcnews-date-text.csv', header=
# increase the dataset 10 times, adding copies to
df = pd.concat([df] * 10)
df.head()
```



```
# calculate the average word length in the title
def mean_word_len(line):
    # this cycle just complicates the task
    for i in range(6):
        words = [len(i) for i in line.split()]
        res = sum(words) / len(words)
    return resdef compute_avg_word(df):
    return df['headline_text'].apply(mean_word_le
```

Concurrency will be provided by the following code:

```
from multiprocessing import Pool# I have 4 cores
n_cores = 4
pool = Pool(n_cores)def apply_parallel(df, func):
    # split dataframe
    df_split = np.array_split(df, n_cores)
    # calculate metrics for each and concatenate
    df = pd.concat(pool.map(func, df_split))
    return df
# df['new_col'] = apply_parallel(df, compute_avg_
```

Compare speed:

Summary

- Works on the standard python library
- We got **x2–3 speedup**
- Using parallelization on small data is a bad idea, because the overhead of interprocess communication exceeds the time gain

Pandarallel

Pandarallel is a small pandas library that adds the ability to work with multiple cores. Under the hood, **it works on standard multiprocessing**, so you should not expect an increase in speed compared to the previous approach, but everything is out of the box + some sugar in the form of a beautiful progress bar ;)

```
1 df["sample-word"] = df.sample_column.parallel_apply(function_to_apply)
```

0.00% 0 / 13
0.00% 0 / 13
0.00% 0 / 13
0.00% 0 / 12
0.00% 0 / 12
0.00% 0 / 12
0.00% 0 / 12
0.00% 0 / 12

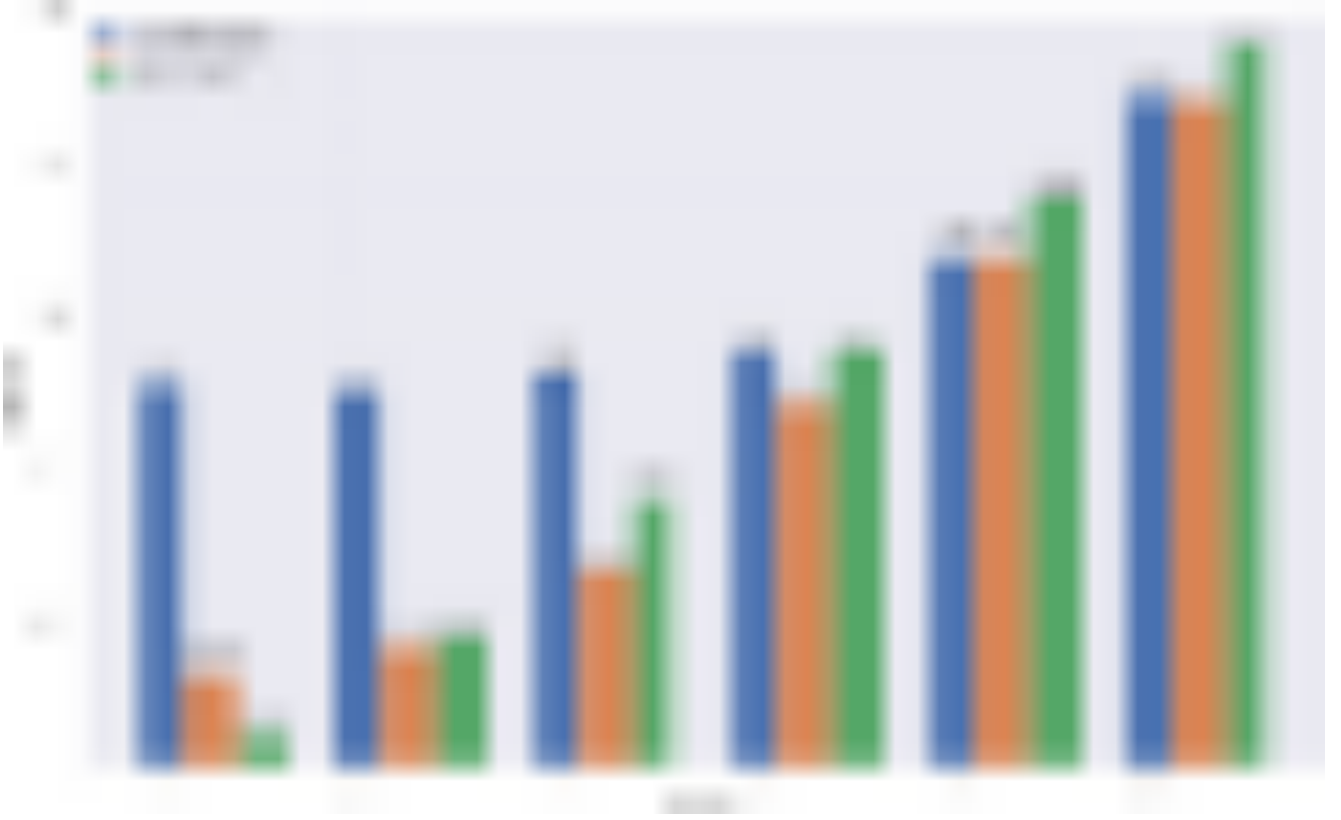
Let’s start testing. Further I will use the same data and functions to process them as in the previous part. Set up pandarallel first — it’s very simple:

```
from pandarallel import pandarallel
# pandarallel will determine how many cores you h
pandarallel.initialize()
```

Now It remains only to write an optimized version of our handler, which is also very simple — just replace `apply` with`parallel_apply`:

```
df['headline_text'].parallel_apply(mean_word_len)
```

Compare speed:



Summary

- **A rather large overhead** in about 0.5 seconds immediately catches your eye. Each time it is used, pandarallel first creates a pool of workers and then closes it. In the self-written version above, I created the pool 1 time, and then reused it, so the overhead was much lower
- If you do not take into account the costs described above, then the **acceleration is the same as in the previous version — about 2–3 times**
- Pandarallel also knows how to `parallel_apply` over grouped data (`groupby`), which is quite convenient. For a complete list of functionality and examples see here

In general, I would prefer this option to self-written,

because for medium / large data volumes there is almost no difference in speed, and we get an extremely simple API and progress bar.

To be continued

In this part, we looked at 2 fairly simple approaches to pandas optimization — **using jit compilation** and **parallelizing a task** using several cores. In the next part I will talk about more interesting and complex tools, but for now, I advise you to test the tools yourself and make sure of their efficiency.

> *Chapter 2*

> *P.s Trust, but verify — all code used in the article (benchmarks and charts drawing), I posted on github*

> *Originally posted on alievmagomed.com on may 24 2020*