cuDF apply_rows/apply_chunk tutorial

In Progress

Background

The RAPIDS cuDF library is a GPU DataFrame manipulation library based on Apache Arrow that accelerates loading, filtering, and manipulation of data for model training data preparation. It provides a pandas-like API that will be familiar to data scientists. Pandas lib provides a lot of special methods that covers most of the use cases for data scientists. However, it cannot cover all the cases, sometimes it is nice to have a way to accelerate the customized data transformation. Luckily, in cuDF, there are two special methods that serve this particular purpose: apply_rows and apply_chunk functions. They utilized the Numba library to accelerate the data transformation via GPU in parallel. In this tutorial, I am going to show a few examples of how to use it.

In this tutorial, I use dgx_p100 node in the PSG cluster after setting up conda environment, cuDF is installed by following conda command

```
conda install -c nvidia -c rapidsai -c numba -c conda-forge -c defaults
cudf=0.3.0
```

Difference between apply_rows and apply_chunk

apply_rows is a special case of apply_chunk, which processes each of the rows of the Dataframe independently in parallel. Under the hood, the apply_rows method will optimally divide the long columns into chunks, and assign chunks into different GPU blocks to compute. Here is one example, I am using apply_row to double the input array and also print out the GPU block/grid allocation information.

double array by apply_rows

```
import cudf
import numpy as np
from numba import cuda

df = cudf.dataframe.DataFrame()
df['in1'] = np.arange(1000, dtype=np.float64)

def kernel(in1, out):
    for i, x in enumerate(in1):
        print('tid:', cuda.threadIdx.x, 'bid:', cuda.blockIdx.x,
              'array size:', in1.size, 'block threads:', cuda.blockDim.x)
        out[i] = x * 2.0

outdf = df.apply_rows(kernel,
                      incols=['in1'],
                      outcols=dict(out=np.float64),
                      kwargs=dict())

print(outdf['in1'].sum()*2.0)
print(outdf['out'].sum())
```

From the output.txt, we can see that the for-loop in the kernel function is unrolled by the compiler automatically. It uses 15 CUDA blocks (0 indexed). Each CUDA block uses 64 threads to do the computation. Each of the thread in the block most of the time deals one element in the input array and sometimes deals with two elements. The order of processing row element is not defined.

We implement the same array double logic with the apply_chunks method.

double array by apply_chunks

```
import cudf
import numpy as np
from numba import cuda


df = cudf.dataframe.DataFrame()
df['in1'] = np.arange(100, dtype=np.float64)


def kernel(in1, out):
    print('tid:', cuda.threadIdx.x, 'bid:', cuda.blockIdx.x,
          'array size:', in1.size, 'block threads:', cuda.blockDim.x)
    for i in range(cuda.threadIdx.x, in1.size, cuda.blockDim.x):
        out[i] = in1[i] * 2.0

outdf = df.apply_chunks(kernel,
                        incols=['in1'],
                        outcols=dict(out=np.float64),
                        kwargs=dict(),
                        chunks=16,
                        tpb=8)

print(outdf['in1'].sum()*2.0)
print(outdf['out'].sum())
```

From the output.txt, we can see apply_chunks has more control than the apply_rows method. It can specify how to divide the long array into chunks, map each of the array chunks to different GPU blocks to process (chunks argument) and assign the number of thread in the block (tpb argument). The for-loop is no longer automatically unrolled in the kernel function as apply_rows method but stays as the for-loop for that GPU thread. Each kernel corresponds to each thread in one block and it has full access to all the elements in that chunk of the array. In this example, the chunk size is 16, and it uniformly cuts the 100 elements into 8 chunks (except the last one) and assigns them to 8 blocks (the final block's is only partially full with array/chunk-size = 4), with each block using 8 threads to process its array/chunk.

Performance benchmark compare

Here we compare the benefits of using cuDF apply_rows vs pandas apply method by the following python code:

speed benchmark

```
import cudf
import pandas as pd
import numpy as np
import time


data_length = 1e6
df = cudf.dataframe.DataFrame()
df['in1'] = np.arange(data_length, dtype=np.float64)


def kernel(in1, out):
    for i, x in enumerate(in1):
        out[i] = x * 2.0

start = time.time()
df = df.apply_rows(kernel,
                   incols=['in1'],
                   outcols=dict(out=np.float64),
                   kwargs=dict())
end = time.time()
print('cuDF time', end-start)
assert(np.isclose(df['in1'].sum()*2.0, df['out'].sum()))


df = pd.DataFrame()
df['in1'] = np.arange(data_length, dtype=np.float64)
start = time.time()
df['out'] = df.in1.apply(lambda x: x*2)
end = time.time()
print('pandas time', end-start)
assert(np.isclose(df['in1'].sum()*2.0, df['out'].sum()))
```

We change the data_length from 1e4 to 1e7, here is the computation time spent in cuDF and pandas.

data length	1e3	1e4	1e5	1e6	1e7	1e8
cuDF time	0.175	0.184	0.175	0.172	0.177	0.249
pandas time	0.0006	0.0022	0.018	0.250	2.13	21.4

As we can see, thecuDF has an overhead of launching GPU kernels (mostly the kernel compilation time), the computation time remains relatively constant in this test due to the massive number of cores in P100 card and constant kernel compilation time. While the CPU computation scales linearly with the length of the array due to the series computation nature of the "apply" function.cuDF has the advantage in computation once the array size is larger than one million.

Realistic application

In the financial service industry, data scientists usually need to compute features from time series data. The most popular method to process the time series data is to compute moving average. In this example, I am going to show how to utilize apply_chunks to speed up moving average computation for a long array.

moving average

```
import cudf
import numpy as np
import pandas as pd
from numba import cuda
import time

data_length = int(1e9)
average_window = 4
df = cudf.dataframe.DataFrame()
threads_per_block = 128
trunk_size = 10240
df['in1'] = np.arange(data_length, dtype=np.float64)


def kernel1(in1, out, average_length):
    for i in range(cuda.threadIdx.x,
                   average_length-1, cuda.blockDim.x):
        out[i] = np.inf
    for i in range(cuda.threadIdx.x + average_length - 1,
                   in1.size, cuda.blockDim.x):
        summ = 0.0
        for j in range(i - average_length + 1,
                       i + 1):
            summ += in1[j]
        out[i] = summ / np.float64(average_length)

def kernel2(in1, out, average_length):
    if in1.size - average_length + cuda.threadIdx.x - average_length + 1 <
0 :
        return
    for i in range(in1.size - average_length + cuda.threadIdx.x,
                   in1.size, cuda.blockDim.x):
        summ = 0.0
        for j in range(i - average_length + 1,
                       i + 1):
            #print(i,j, in1.size)
            summ += in1[j]
        out[i] = summ / np.float64(average_length)


start = time.time()
df = df.apply_chunks(kernel1,
                     incols=['in1'],
                     outcols=dict(out=np.float64),
                     kwargs=dict(average_length=average_window),
                     chunks=list(range(0, data_length,
                                       trunk_size))+ [data_length],
                     tpb=threads_per_block)
```

```
df = df.apply_chunks(kernel2,
                     incols=['in1', 'out'],
                     outcols=dict(),
                     kwargs=dict(average_length=average_window),
                     chunks=[0]+list(range(average_window, data_length,
                                           trunk_size))+ [data_length],
                     tpb=threads_per_block)
end = time.time()
print('cuDF time', end-start)

pdf = pd.DataFrame()
pdf['in1'] = np.arange(data_length, dtype=np.float64)
start = time.time()
pdf['out'] = pdf.rolling(average_window).mean()
end = time.time()
print('pandas time', end-start)

assert(np.isclose(pdf.out.as_matrix()[average_window:].mean(),
       df.out.to_array()[average_window:].mean()))
```

In the above code, we divide the array into subarrays of size "trunk_size", and send those subarrays to GPU blocks to compute moving average. However, there is no history for the elements at the beginning of the subarray. To fix this, we shift the chunk division by an offset of "average_window". Then we call the kernel2 to compute the moving average of those missing records only.. Note, in kernel2, we didn't define outcols as it will create a new GPU memory buffer and overwrite the old "out" values. Instead, we reuse out array as input. For an array of 1e9 length, cuDF uses 1.387s to do the computation while pandas use 7.58s.

This code is not optimized in performance. There are a few things we can do to make it faster. First, we can use shared memory to load the array and reduce the IO when doing the summation. Secondly, there is a lot of redundant summation done by the threads. We can maintain an accumulation summation array to help reduce the redundancy. This is outside the scope of this tutorial.