# Going deep into the cuDF

● In Progress

## Objective

In my previous tutorial, I showed how to use apply_rows and apply_chunks methods in cuDF to implement customized data transformations. Under the hood, they are all using Numba library to compile the normal python code into GPU kernels. Numba is an excellent python library that accelerates the numerical computations. Most importantly, Numba has direct CUDA programming support. For detailed information, please check out this Numba CUDA document. As we know, the underlying data structure of cuDF is a GPU version of Apache Arrow. We can directly pass the GPU array around without the copying operation. Once we have the nice Numba library and standard GPU array, the sky is the limit. In this tutorial, I will show how to use Numba CUDA to accelerate cuDF data transformation and how to step by step accelerate it using CUDA programming tricks.

Same as my last 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
```

## A simple example

As usual, I am going to start with a simple example of doubling the numbers in an array:

**double an array**

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

array_len = 1000
number_of_threads = 128
number_of_blocks = (array_len + (number_of_threads - 1)) //
number_of_threads
df = cudf.dataframe.DataFrame()
df['in'] = np.arange(array_len, dtype=np.float64)


@cuda.jit
def double_kernel(result, array_len):
    """
    double each element of the array
    """
    i = cuda.grid(1)
    if i < array_len:
        result[i] = result[i] * 2.0


before = df['in'].sum()
gpu_array = df['in'].to_gpu_array()
print(type(gpu_array))
double_kernel[(number_of_blocks,), (number_of_threads,)](gpu_array,
array_len)
after = df['in'].sum()
assert(np.isclose(before * 2.0, after))
```

From the output of this code, it shows the underlying GPU array is of type "numba.cuda.cudadrv.devicearray.DeviceNDArray". We can directly pass it to the kernel function that is compiled by the "cuda.jit". Because we passed in the reference, the effect of number transformation will automatically show up in the original cuDF Dataframe. Note we have to manually enter the block size and grid size, which gives us the maximum of GPU programming control. The "cuda.grid" is a convenient method to compute the absolute position for the threads. It is equivalent to the normal "block_id * block_dim + thread_id" formula.

## Practical example

### Baseline

We will work on the moving average problem as the last time. Because we have the full control of the grid and block size allocation, the vanilla moving average implementation code is much simpler compared to the apply_chunks implementation.

**moving average**

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

```
array_len = int(1e9)
average_window = 3000
number_of_threads = 128
number_of_blocks = (array_len + (number_of_threads - 1)) //
number_of_threads
df = cudf.dataframe.DataFrame()
df['in'] = np.arange(array_len, dtype=np.float64)
df['out'] = np.arange(array_len, dtype=np.float64)


@cuda.jit
def kernel1(in_arr, out_arr, average_length, arr_len):
    s = numba.cuda.local.array(1, numba.float64)
    s[0] = 0.0
    i = cuda.grid(1)
    if i < arr_len:
        if i < average_length-1:
            out_arr[i] = np.inf
        else:
            for j in range(0, average_length):
                s[0] += in_arr[i-j]
            out_arr[i] = s[0] / np.float64(average_length)


gpu_in = df['in'].to_gpu_array()
gpu_out = df['out'].to_gpu_array()
start = time.time()
kernel1[(number_of_blocks,), (number_of_threads,)](gpu_in, gpu_out,
                                                   average_window,
array_len)
cuda.synchronize()
end = time.time()
print('Numba with comipile time', end-start)

start = time.time()
kernel1[(number_of_blocks,), (number_of_threads,)](gpu_in, gpu_out,
                                                   average_window,
array_len)
cuda.synchronize()
end = time.time()
print('Numba without comipile time', end-start)

pdf = pd.DataFrame()
pdf['in'] = np.arange(array_len, 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()))
```

Note, in order to compare the computation time accurately, I launch the kernel twice. The first time kernel launching will include the kernel compilation time. In this example, it takes 17.31s for the kernel to run without compilation.

To compare it with apply_chunks implementation, I also set "array_len" to be 1e9 and 4 as "average_window", which is the same as the parameters I used in apply_chunks code in my last tutorial, the cuDF achieves 0.16s with kernel compilation time and 0.048s without. This is already a great improvement vs apply_chunk method, which takes 1.38s.

## Use shared memory

In the baseline code, each thread is reading the numbers from the global memory. When doing the moving average, the same number is read multiple times by different threads. GPU global memory IO, in this case, is the speed bottleneck. To mitigate it, we load the data into shared memory for each of the computation blocks. Then the threads are doing summation from the numbers in the cache. To do the moving average for the elements at the beginning of the array, we make sure to load the "average_window" more data in the shared_memory.

**add shared memory**

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

array_len = int(1e9)
average_window = 3000
number_of_threads = 128
number_of_blocks = (array_len + (number_of_threads - 1)) //
number_of_threads
shared_buffer_size = number_of_threads + average_window - 1
df = cudf.dataframe.DataFrame()
df['in'] = np.arange(array_len, dtype=np.float64)
df['out'] = np.arange(array_len, dtype=np.float64)


@cuda.jit
def kernel1(in_arr, out_arr, average_length, arr_len):
    block_size = cuda.blockDim.x
    shared = cuda.shared.array(shape=(shared_buffer_size),
                               dtype=numba.float64)
    i = cuda.grid(1)
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    bid = cuda.blockIdx.x
    starting_id = bid * block_size

    shared[tx + average_length - 1] = in_arr[i]
    cuda.syncthreads()
    for j in range(0, average_length - 1, block_size):
        if (tx + j) < average_length - 1:
            shared[tx + j] = in_arr[starting_id -
                                    average_length + 1 +
                                    tx + j]
    cuda.syncthreads()

    s = numba.cuda.local.array(1, numba.float64)
```

```
        s[0] = 0.0
        if i < arr_len:
            if i < average_length-1:
                out_arr[i] = np.inf
            else:
                for j in range(0, average_length):
                    s[0] += shared[tx + average_length - 1 - j]
                out_arr[i] = s[0] / np.float64(average_length)


    gpu_in = df['in'].to_gpu_array()
    gpu_out = df['out'].to_gpu_array()
    start = time.time()
    kernel1[(number_of_blocks,), (number_of_threads,)](gpu_in, gpu_out,
                                                        average_window,
    array_len)
    cuda.synchronize()
    end = time.time()

    print('Numba with comipile time', end-start)

    start = time.time()
    kernel1[(number_of_blocks,), (number_of_threads,)](gpu_in, gpu_out,
                                                        average_window,
    array_len)
    cuda.synchronize()
    end = time.time()
    print('Numba without comipile time', end-start)

    pdf = pd.DataFrame()
    pdf['in'] = np.arange(array_len, 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()))
```

Running this, the computation time is reduced to 5.22s without kernel compilation time.

### Reduced redundant summations.

Each thread in the above code is doing one moving average in a for-loop. It is easy to see that there are a lot of redundant summation operations done by different threads. To reduce the redundancy, the following code is changed to let each thread to compute a consecutive number of moving averages. The later moving average step is able to reuse the sum of the previous steps. This eliminated "thread_tile" number of for-loops.

**reduce redundancy**

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

```
from numba import cuda
import numba
import time

array_len = int(1e9)
average_window = 3000
number_of_threads = 64
thread_tile = 48
number_of_blocks = (array_len + (number_of_threads * thread_tile - 1)) //
(number_of_threads * thread_tile)
shared_buffer_size = number_of_threads * thread_tile + average_window - 1
df = cudf.dataframe.DataFrame()
df['in'] = np.arange(array_len, dtype=np.float64)
df['out'] = np.arange(array_len, dtype=np.float64)


@cuda.jit
def kernel1(in_arr, out_arr, average_length, arr_len):
    block_size = cuda.blockDim.x
    shared = cuda.shared.array(shape=(shared_buffer_size),
                               dtype=numba.float64)
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    bid = cuda.blockIdx.x
    starting_id = bid * block_size * thread_tile

    for j in range(thread_tile):
        shared[tx + j * block_size + average_length - 1] = in_arr
[starting_id
                                                                   + tx +
                                                                   j *
block_size]
        cuda.syncthreads()
    for j in range(0, average_length - 1, block_size):
        if (tx + j) < average_length - 1:
            shared[tx + j] = in_arr[starting_id -
                                                 average_length + 1 +
                                                 tx + j]
    cuda.syncthreads()

    s = numba.cuda.local.array(1, numba.float64)
    first = False
    s[0] = 0.0
    for k in range(thread_tile):
        i = starting_id + tx * thread_tile + k
        if i < arr_len:
            if i < average_length-1:
                out_arr[i] = np.inf
            else:
                if not first:
                    for j in range(0, average_length):
                        s[0] += shared[tx * thread_tile + k +
average_length - 1 - j]
```

```
                    s[0] = s[0] / np.float64(average_length)
                    out_arr[i] = s[0]
                    first = True
                else:
                    s[0] = s[0] + (shared[tx * thread_tile + k +
average_length - 1]
                                   - shared[tx * thread_tile + k +
average_length - 1 - average_length])  / np.float64(average_length)

                    out_arr[i] = s[0]


gpu_in = df['in'].to_gpu_array()
gpu_out = df['out'].to_gpu_array()
start = time.time()
kernel1[(number_of_blocks,), (number_of_threads,)](gpu_in, gpu_out,
                                                   average_window,
array_len)
cuda.synchronize()
end = time.time()
print('Numba with comipile time', end-start)

start = time.time()
kernel1[(number_of_blocks,), (number_of_threads,)](gpu_in, gpu_out,
                                                   average_window,
array_len)
cuda.synchronize()
end = time.time()
print('Numba without comipile time', end-start)

pdf = pd.DataFrame()
pdf['in'] = np.arange(array_len, 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()))
```

After this change, the computation time is reduced to 1.12s without kernel compilation time, we achieved a total of 15.4x speedup compared with the baseline.

## Conclusion

In this tutorial, we take advantage of CUDA programming model in the Numba library to do moving average computation. We show by using a few CUDA programming tricks, we can achieve 15.4x speed up in moving average computations for long arrays.

cuDF is a powerful tool for data scientists to use. It provides the high-level API that covers most of the use cases. However, it also exposes its low-level components. Those components including gpu_array and Numba integration make the cuDF library to be very flexible to process data in a customized way.