# Adding the rolling and ewm functions to the cuDF

In Progress


- Objective
- Overview of Pandas Rolling function
- Implementation idea
- An example window function
- All the other implemented window functions
- The Rolling class
- The EWM class
- Test the API
- Limitations


## Objective

In my previous tutorial, I talked about how to use Numba CUDA to accelerate cuDF data transformation. The example I used in that tutorial is to compute the moving average. While it is working fine to get the job done, it is not convenient to use in your project. In Pandas, there is a rolling fu nction that I really like a lot. It abstracts the operations of doing the transformation in moving window fashion into a single method. While RAPIDS team is actively working on the official rolling function in the cuDF library, I am going to show in this tutorial how to implement your own rolling function for those who are impatient or curious. This tutorial is built on top of the previous tutorial. Please review it if you have any questions about how to use Numba CUDA in the cuDF library. The complete code for this tutorial can be obtained at this yagr repo.

## Overview of Pandas Rolling function

In Pandas, if you want to compute the moving average for a certain array, you can use the rolling function as:

**moving average in pandas**

```
import pandas as pd
import numpy as np
random_array = np.random.rand(1e4)
average_window = 30
pdf = pd.DataFrame()
pdf['in'] = random_array
pdf['mean'] = pdf['in'].rolling(average_window).mean()
```

To do the max/min/sum/std/var, just change the different aggregation functions like this:

**aggregation**

```
pdf['max'] = pdf['in'].rolling(average_window).max()
pdf['min'] = pdf['in'].rolling(average_window).min()
pdf['sum'] = pdf['in'].rolling(average_window).sum()
pdf['std'] = pdf['in'].rolling(average_window).std()
pdf['var'] = pdf['in'].rolling(average_window).var()
```

For people who want to do customized window computations, Pandas provides apply method. For example, someone just wants the difference between window beginning and window ending:

**customized window function**

```
pdf['var'] = pdf['in'].rolling(average_window).apply(lambda x: x[0] - x
[-1])
```

## Implementation idea

As the rolling API is so easy to use in Pandas, we want to build something similar for the cuDF library.  Starting with the code from the last tutorial, it is shown that the best performance for computation moving average is achieved if shared memory and thread tile are used. The window computations can be abstracted into two parts: the first part is to copy the global memory into shared memory and divide the shared memory into thread tiles. It also divides the output array for the corresponding thread tile. These steps are encapsulated into a Rolling class discussed later. The second part deals with the specific window computations. The threads management is abstracted away. The user just needs to deal with one single CUDA thread and focus on how to do window computation for "thread_tile" number of elements. Following is the window function signature:

**window function signature**

```
@cuda.jit(device=True)
def window_kernel(shared, histroy_len, out_arr, window_size, arr_len,
offset):
    """
    This function is to do window computation. Only one thread is assumed
    to do the computation. This thread is responsible for the `arr_len` of
    elements in the input array, which is cached in the `shared` array.
    Due to the limitation of numba, shared array cannot be sliced
properly. As
    a work around, the passed-in `offset` position specify the beginning of
    the input array.

    Arguments:
        shared: a chunk of the array stored in shared memory. The first
                element of the data starts at `offset`. It has
`history_len`
                of historical records for the first elements computation.
        history_len: the history length, which is mostly `window_size -
1`. For
                the early elements of the array, the history can be
shorter for
                the beginning elements of the original array.
        out_arr: the output array of size `arr_len`.
        window_size: the window size for the window function
        arr_len: the length of the output array
        offset: the starting position of the input shared array for the
                current thread
    """
    pass
```

It gets "history_len" information as input, so the user has the chance to handle the case that the history information is not enough (history_len < window_size - 1). The inputs can be loaded from the input array stored in the shared cache starting at the position "offset".

## An example window function

As an example, here is a window function that is used to compute the "min" of the moving window elements

**min window function**

```
@cuda.jit(device=True)
def min_window(shared, histroy_len, out_arr, window_size, arr_len, offset):
    """
    This function is to compute the min for the window
    See `window_kernel` for detailed arguments
    """
    for i in range(arr_len):
        if i + histroy_len < window_size-1:
            out_arr[i] = np.inf
        else:
            s = np.inf  # minimum
            for j in range(0, window_size):
                # bigger than the max
                if shared[i + offset - j] < s:
                    s = shared[i + offset - j]
            out_arr[i] = s
```

As shown, the implementation is very straightforward. It loops through the "arr_len" of elements in the cached input array "shared" and computes the minimum for "window_size" of historical elements.

## All the other implemented window functions

**all the window functions**

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

@cuda.jit(device=True)
def mean_window(shared, histroy_len, out_arr, window_size, arr_len,
offset):
    """
    This function is to compute the moving average for the window
    See `window_kernel` for detailed arguments
    """
    s = 0.0
    first = False
    for i in range(arr_len):
        if i + histroy_len < window_size-1:
            out_arr[i] = np.inf
        else:
            if not first:
                for j in range(0, window_size):
                    s += shared[i + offset - j]
                s = s / np.float64(window_size)
                out_arr[i] = s
                first = True
            else:
                s += (shared[i + offset]
```

```
                        - shared[i + offset - window_size])  / np.float64
(window_size)
                out_arr[i] = s

@cuda.jit(device=True)
def var_window(shared, histroy_len, out_arr, window_size, arr_len, offset):
    """
    This function is to compute the var for the window
    See `window_kernel` for detailed arguments
    """
    s = 0.0 # this is mean
    var = 0.0 # this is variance
    first = False
    for i in range(arr_len):
        if i < arr_len:
            if i + histroy_len < window_size-1:
                out_arr[i] = np.inf
            else:
                if not first:
                    for j in range(0, window_size):
                        s += shared[i + offset - j]
                        var += shared[i + offset - j] * shared[i + offset
- j]
                    s = s / np.float64(window_size)
                    var = var / np.float64(window_size)
                    out_arr[i] = (var - s * s) * np.float64(window_size) /
np.float64(window_size - 1.0)
                    first = True
                else:
                    s += (shared[i + offset]
                             - shared[i + offset - window_size])  / np.
float64(window_size)
                    var += (shared[i + offset] * shared[i + offset]
                             - shared[i + offset - window_size] * shared[i
+

offset - window_size])  / np.float64(window_size)
                    out_arr[i] = (var - s * s) * np.float64(window_size) /
np.float64(window_size - 1.0)

@cuda.jit(device=True)
def std_window(shared, histroy_len, out_arr, window_size, arr_len, offset):
    """
    This function is to compute the std for the window
    See `window_kernel` for detailed arguments
    """
    s = 0.0 # this is mean
    var = 0.0 # this is variance
    first = False
    for i in range(arr_len):
        if i + histroy_len < window_size-1:
            out_arr[i] = np.inf
        else:
```

```
            if not first:
                for j in range(0, window_size):
                    s += shared[i + offset - j]
                    var += shared[i + offset - j] * shared[i + offset - j]
                s = s / np.float64(window_size)
                var = var / np.float64(window_size)
                out_arr[i] = math.sqrt((var - s * s) * np.float64
(window_size) / np.float64(window_size - 1.0))
                first = True
            else:
                s += (shared[i + offset]
                         - shared[i + offset - window_size])  / np.float64
(window_size)
                var += (shared[i + offset] * shared[i + offset]
                         - shared[i + offset - window_size] * shared[i +

offset - window_size])  / np.float64(window_size)
                out_arr[i] = math.sqrt((var - s * s) * np.float64
(window_size) / np.float64(window_size - 1.0))


@cuda.jit(device=True)
def sum_window(shared, histroy_len, out_arr, window_size, arr_len, offset):
    """
    This function is to compute the sum for the window
    See `window_kernel` for detailed arguments
    """
    s = 0.0
    first = False
    for i in range(arr_len):
        if i + histroy_len < window_size-1:
            out_arr[i] = np.inf
        else:
            if not first:
                for j in range(0, window_size):
                    s += shared[i + offset - j]
                s = s
                out_arr[i] = s
                first = True
            else:
                s += (shared[i + offset]
                      - shared[i + offset - window_size])
                out_arr[i] = s

@cuda.jit(device=True)
def max_window(shared, histroy_len, out_arr, window_size, arr_len, offset):
    """
    This function is to compute the max for the window
    See `window_kernel` for detailed arguments
    """
    for i in range(arr_len):
        if i + histroy_len < window_size-1:
            out_arr[i] = np.inf
```

```
        else:
            s = -np.inf  # maximum
            for j in range(0, window_size):
                # bigger than the max
                if shared[i + offset - j] > s:
                    s = shared[i + offset - j]
            out_arr[i] = s
```

## The Rolling class

To mimic the Pandas rolling function behavior, a Rolling class is defined to handle the part one work as mentioned above

**Rolling class**

```
from numba import cuda
import numba
from windows import (mean_window, std_window, var_window, max_window,
                     min_window, sum_window)


kernel_cache = {}


def get_kernel(method):

    if method in kernel_cache:
        return kernel_cache[method]

    @cuda.jit
    def kernel(in_arr, out_arr, average_length, arr_len, thread_tile):
        shared = cuda.shared.array(shape=0,
                                  dtype=numba.float64)
        block_size = cuda.blockDim.x
        tx = cuda.threadIdx.x
        # Block id in a 1D grid
        bid = cuda.blockIdx.x
        starting_id = bid * block_size * thread_tile

        # copy the thread_tile * number_of_thread_per_block into the shared
        for j in range(thread_tile):
            offset = tx + j * block_size
            if (starting_id + offset) < arr_len:
                shared[offset + average_length - 1] = in_arr[
                    starting_id + offset]
            cuda.syncthreads()
        # copy the average_length - 1 into the shared
        for j in range(0, average_length - 1, block_size):
            if (((tx + j) <
                 average_length - 1) and (
                     starting_id - average_length + 1 + tx + j >= 0)):
                shared[tx + j] = in_arr[starting_id
```

```
                                            - average_length + 1 + tx + j]
        cuda.syncthreads()
        # slice the shared memory for each threads
        start_shared = tx * thread_tile
        his_len = min(average_length - 1,
                      starting_id + tx * thread_tile)

        # slice the global memory for each threads
        start = starting_id + tx * thread_tile
        end = min(starting_id + (tx + 1) * thread_tile, arr_len)
        sub_outarr = out_arr[start:end]
        sub_len = end - start
        method(shared, his_len, sub_outarr,
               average_length, sub_len, average_length - 1 + start_shared
               )
    kernel_cache[method] = kernel
    return kernel

class Rolling(object):

    def __init__(self, window, input_arr, thread_tile=48,
                 number_of_threads=64):
        self.gpu_in = input_arr.to_gpu_array()
        self.window = window
        self.number_of_threads = number_of_threads
        self.array_len = len(self.gpu_in)
        self.gpu_out = numba.cuda.device_array_like(self.gpu_in)
        self.thread_tile = thread_tile
        self.number_of_blocks = (self.array_len +
                                 (number_of_threads * thread_tile - 1)) //
(
                                     number_of_threads * thread_tile)

        self.shared_buffer_size = (self.number_of_threads * self.
thread_tile
                                   + self.window - 1)

    def apply(self, method):
        gpu_out = numba.cuda.device_array_like(self.gpu_in)
        kernel = get_kernel(method)
        kernel[(self.number_of_blocks,),
               (self.number_of_threads,),
               0,
               self.shared_buffer_size * 8](self.gpu_in,
                                            gpu_out,
                                            self.window,
                                            self.array_len, self.
thread_tile)
        return gpu_out

    def mean(self):
        return self.apply(mean_window)
```

```
    def std(self):
        return self.apply(std_window)

    def var(self):
        return self.apply(var_window)

    def max(self):
        return self.apply(max_window)

    def min(self):
        return self.apply(min_window)

    def sum(self):
        return self.apply(sum_window)
```

As shown in the code, get_kernel function handles the kernel compilation and cache. Note, the shared memory is dynamically allocated. It is not officially documented in the Numba library yet. The only example I can find is from the Stack Overflow page. The apply function does all the heavy lifting work. It first copies the global memory to the shared memory and makes sure all the history information is available for the window function computation. Each thread is responsible for "arr_len" number of elements in the shared memory. It computes the starting position in the shared array for the current thread. Then it slices the output GPU memory properly for the window functions to use. Those mean/std/var/max/min /sum methods just call apply function with predefined window functions. Once Rolling class is instantiated, it can use different methods to do different aggregations. Here is a test code.

## The EWM class

The implementation of the exponentially weighted moving class is very similar to the Rolling class. The only difference is that it needs a much larger window than the span size to compute the average accurately.

**EWM class**

```
from numba import cuda
import numba
from windows import (ewma_mean_window)


kernel_cache = {}


def get_ewm_kernel(method):

    if method in kernel_cache:
        return kernel_cache[method]

    @cuda.jit
    def kernel(in_arr, out_arr, average_length, span, arr_len,
thread_tile):
        shared = cuda.shared.array(shape=0,
                                  dtype=numba.float64)
        block_size = cuda.blockDim.x
        tx = cuda.threadIdx.x
        # Block id in a 1D grid
        bid = cuda.blockIdx.x
        starting_id = bid * block_size * thread_tile
```

```
        # copy the thread_tile * number_of_thread_per_block into the shared
        for j in range(thread_tile):
            offset = tx + j * block_size
            if (starting_id + offset) < arr_len:
                shared[offset + average_length - 1] = in_arr[
                    starting_id + offset]
            cuda.syncthreads()
        # copy the average_length - 1 into the shared
        for j in range(0, average_length - 1, block_size):
            if (((tx + j) <
                 average_length - 1) and (
                     starting_id - average_length + 1 + tx + j >= 0)):
                shared[tx + j] = in_arr[starting_id
                                        - average_length + 1 + tx + j]
        cuda.syncthreads()
        # slice the shared memory for each threads
        start_shared = tx * thread_tile
        his_len = min(average_length - 1,
                      starting_id + tx * thread_tile)

        # slice the global memory for each threads
        start = starting_id + tx * thread_tile
        end = min(starting_id + (tx + 1) * thread_tile, arr_len)
        sub_outarr = out_arr[start:end]
        sub_len = end - start
        method(shared, his_len, sub_outarr,
               average_length, span, sub_len, average_length - 1 +
start_shared
               )
    kernel_cache[method] = kernel
    return kernel


class Ewm(object):

    def __init__(self, span, input_arr, thread_tile=48,
                 number_of_threads=64, backward=10):
        self.gpu_in = input_arr.to_gpu_array()
        self.span = span
        self.window = span * backward
        self.number_of_threads = number_of_threads
        self.array_len = len(self.gpu_in)
        self.gpu_out = numba.cuda.device_array_like(self.gpu_in)
        self.thread_tile = thread_tile
        self.number_of_blocks = (self.array_len +
                                 (number_of_threads * thread_tile - 1)) //
(
                                     number_of_threads * thread_tile)

        self.shared_buffer_size = (self.number_of_threads * self.
thread_tile
                                   + self.window - 1)
```

```
    def apply(self, method):
        gpu_out = numba.cuda.device_array_like(self.gpu_in)
        kernel = get_ewm_kernel(method)
        kernel[(self.number_of_blocks,),
               (self.number_of_threads,),
               0,
               self.shared_buffer_size * 8](self.gpu_in,
                                            gpu_out,
                                            self.window,
                                            self.span,
                                            self.array_len, self.
thread_tile)
        return gpu_out

    def mean(self):
        return self.apply(ewma_mean_window)
```

Following is the window function for the exponential weighted moving average:

**ewma**

```
@cuda.jit(device=True)
def ewma_mean_window(shared, histroy_len, out_arr, window_size, span,
                     arr_len, offset):
    """
    This function is to compute the exponetially-weighted moving average
for
    the window. See `window_kernel` for detailed arguments
    """
    s = 0.0
    first = False
    alpha = 2 / (span + 1)
    lam = 1
    total_weight = 0
    counter = 0
    for i in range(arr_len):
        if i + histroy_len < span - 1:
            out_arr[i] = np.inf
        else:
            if not first:
                # print(i, i + histroy_len + 1, window_size, histroy_len)
                for j in range(0, min(i + histroy_len + 1,
                                      window_size)):
                    s += shared[i + offset - j] * lam
                    counter += 1
                    total_weight += lam
                    lam *= (1 - alpha)
                out_arr[i] = s / total_weight
                first = True
            else:
                if counter >= window_size:
                    s -= shared[i + offset - window_size] * lam / (1 -
alpha)
                else:
                    counter += 1
                    total_weight += lam
                    lam *= (1 - alpha)
                s *= (1 - alpha)
                s += shared[i + offset]
                out_arr[i] = s / total_weight
```

## Test the API

**test API**

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

from rolling import Rolling
from ewm import Ewm

array_len = int(1e4)
average_window = 300
number_type = np.float64
random_array = np.random.rand(array_len)

df = cudf.dataframe.DataFrame()
df['in'] = random_array
df['mean'] = Rolling(average_window, df['in']).mean()
df['max'] = Rolling(average_window, df['in']).max()
df['min'] = Rolling(average_window, df['in']).min()
df['sum'] = Rolling(average_window, df['in']).sum()
df['std'] = Rolling(average_window, df['in']).std()
df['var'] = Rolling(average_window, df['in']).var()
df['ewma'] = Ewm(average_window, df['in']).mean()

pdf = pd.DataFrame()
pdf['in'] = random_array
pdf['mean'] = pdf['in'].rolling(average_window).mean()
pdf['max'] = pdf['in'].rolling(average_window).max()
pdf['min'] = pdf['in'].rolling(average_window).min()
pdf['sum'] = pdf['in'].rolling(average_window).sum()
pdf['std'] = pdf['in'].rolling(average_window).std()
pdf['var'] = pdf['in'].rolling(average_window).var()
pdf['ewma'] = pdf['in'].ewm(span=average_window,
                            min_periods=average_window).mean()

assert(np.isclose(pdf['mean'].as_matrix()[average_window:].mean(),
       df['mean'].to_array()[average_window:].mean(), atol=1e-6))
assert(np.isclose(pdf['max'].as_matrix()[average_window:].mean(),
       df['max'].to_array()[average_window:].mean(), atol=1e-6))
assert(np.isclose(pdf['min'].as_matrix()[average_window:].mean(),
       df['min'].to_array()[average_window:].mean(), atol=1e-6))
assert(np.isclose(pdf['sum'].as_matrix()[average_window:].mean(),
       df['sum'].to_array()[average_window:].mean(), atol=1e-6))
assert(np.isclose(pdf['std'].as_matrix()[average_window:].mean(),
       df['std'].to_array()[average_window:].mean(), atol=1e-6))
assert(np.isclose(pdf['var'].as_matrix()[average_window:].mean(),
       df['var'].to_array()[average_window:].mean(), atol=1e-6))
assert(np.isclose(pdf['ewma'].as_matrix()[average_window:].mean(),
       df['ewma'].to_array()[average_window:].mean(), atol=1e-6))
```

As shown above, the Rolling API is very similar to the Pandas rolling method and ewm method. And they all produce the same results.

## Limitations

We use Numba to compile the kernel. It adds some overhead into the computation due to the compilation. As the compiled kernel is cached, calling the same window function for the second time is very fast.