HW4 analysis

December 4, 2023

1 Homework 4

In this homework we explore how we can leverage CUDA streams and asynchronous operations to maximize the throughput of matrix-vector multiplication. We build on the matrix-vector CUDA kernel we wrote last week namely the `ShuffleSingleWarpMultiplier` which assigns a single warp to each row and uses `__shfl_down_sync` to reduce the distributed multiplication.

We explore CUDA streams by varying the number of created streams and assigning an approximately equal number of rows to each stream.

All of the relevant code can be found in `homework_4/src/benchmark.cu` of the linked repository.

Additionally, recall some notation: 1. `M` is the number of streams 2. The dimensions of the matrix is `N x K`

[1]:
```
# Initial data preparation
import matplotlib.pyplot as plt
import pandas as pd

benchmark = pd.read_csv("../data/data.csv")
```

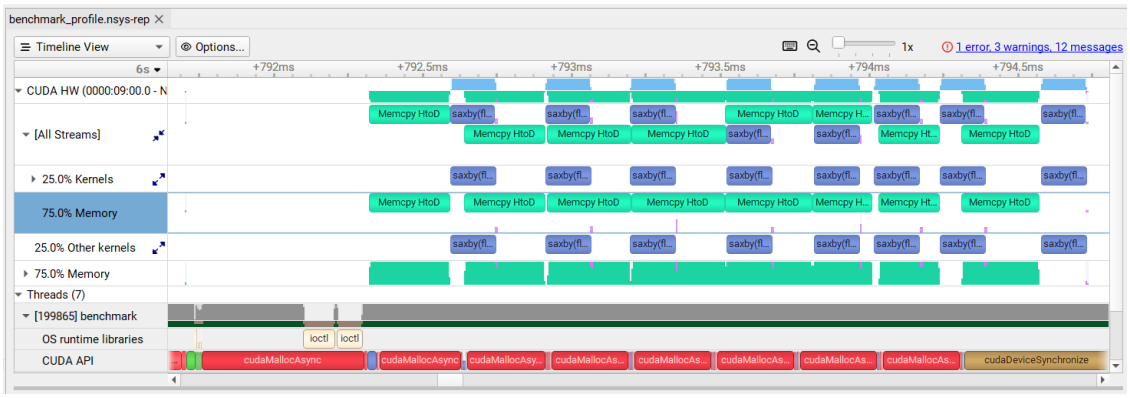
1.1 Nsight Profiling

We begin our analysis by profiling the different stream views. This gives us insight into how the streams are being parallelized, and if they are being parallelized properly. Unless otherwise specified, we set `N = K = 1e3` as our matrix dimensions.

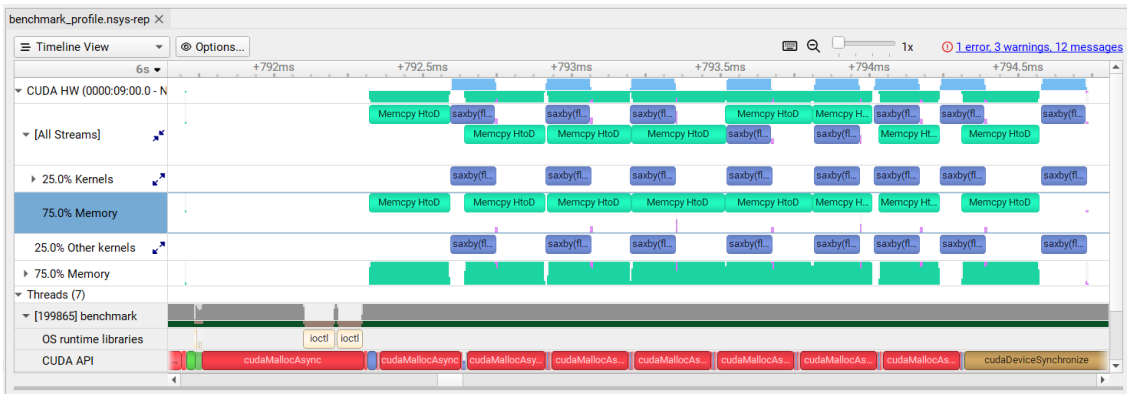
We start with `M = 1`.

1.1.1 `M = 1`

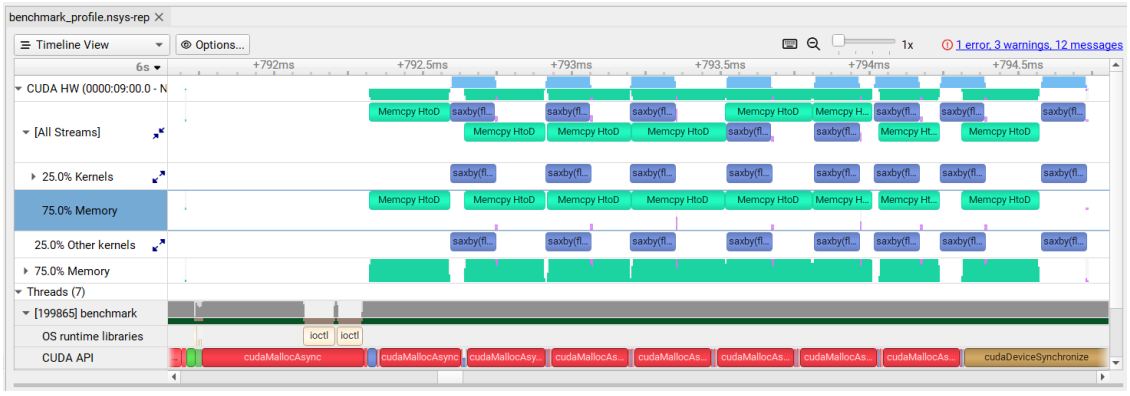
Notice, here, that we are using two kernels. This will be the case for all of our experiments. There are three calls which make up the bulk of the runtime for the matrix multiplication. The first is `cudaMallocAsync`, shown in red.



The second is `cudaStreamCreate`, shown in green (see the zoomed in image below). Since `M = 1`, we have only one stream creation as expected.



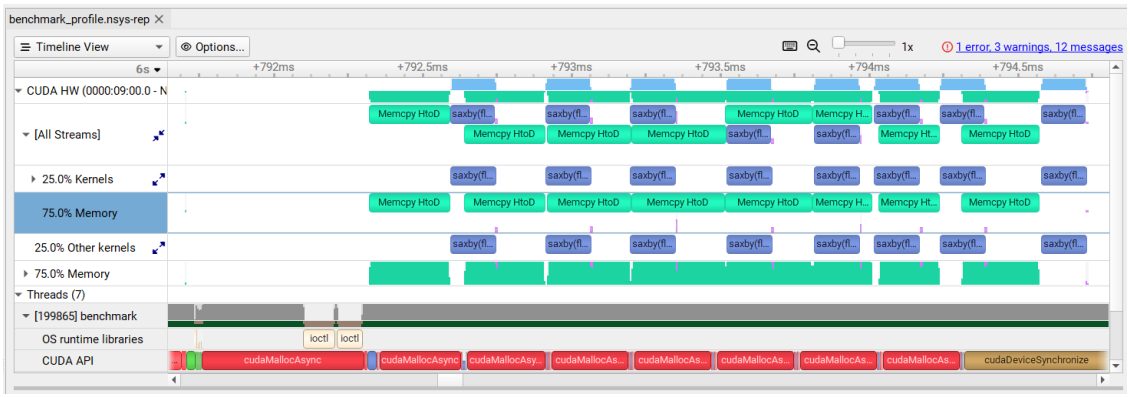
The third is `saxby`, which is the `global` matrix muliplication kernel. The fourth is `Memcpy HtoD` [CPU to GPU] (see the zoomed in image below).



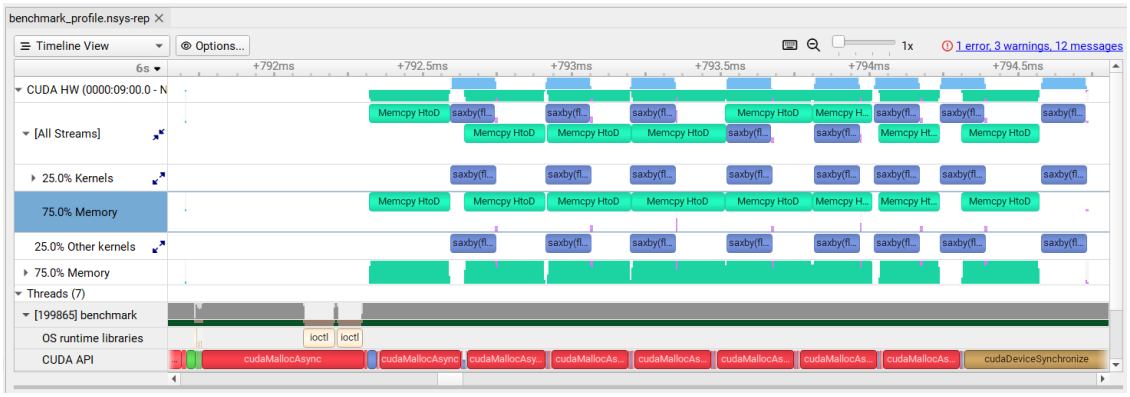
Now, we show the profile results when we have multiple streams. We choose `M = 4` arbitrarily.

1.1.2 `M = 4`

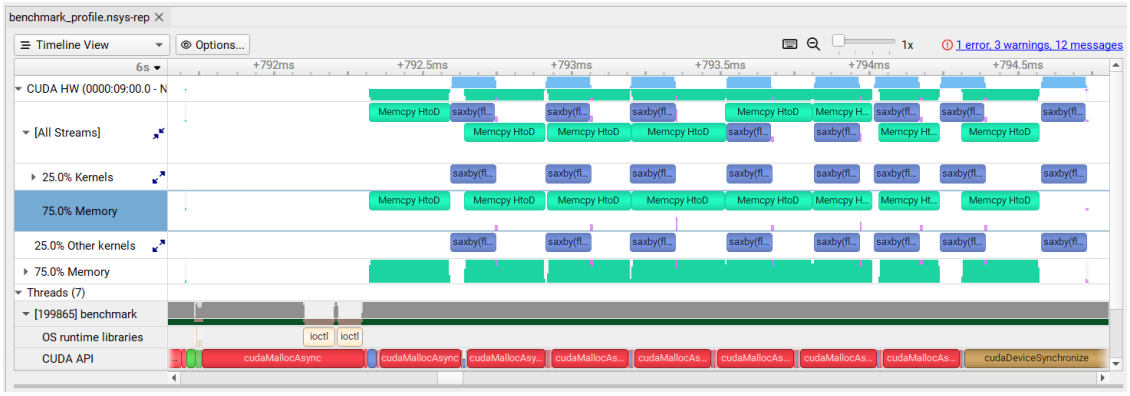
The overall profile:



A zoomed profile over the `cudaStreamCreate`:



A zoomed profile over the `saxby` and `memcpyHtoD`:



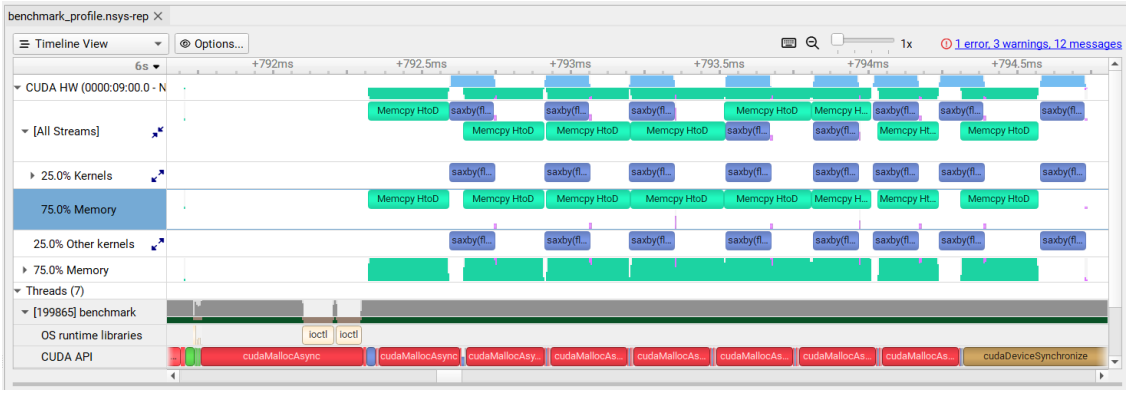
We make some interesting observations when comparing `M = 4` to `M = 1`: 1. The number of calls to `cudaStreamCreate` has increased to `M` consecutive calls. 2. `cudaMallocAsync` has broken into multiple chunks occuring concurrently with the `saxby` operations. This is expected since the call is `async`. 3. The number of calls to `memcpyHtoD` has increased to `M` consecutive calls. We did not expect calls of `memcpyHtoD` for different regions of the matrix (recall that each stream focuses on one distinct chunk of the matrix) to be consecutive, since the call is blocking and not `async`. 4. The `saxby` operations are occuring consecutive with `memcpyHtoD` calls. Specifically, each `saxby`

operation occurs concurrently when when the next region of the matrix is being copied from host to device.

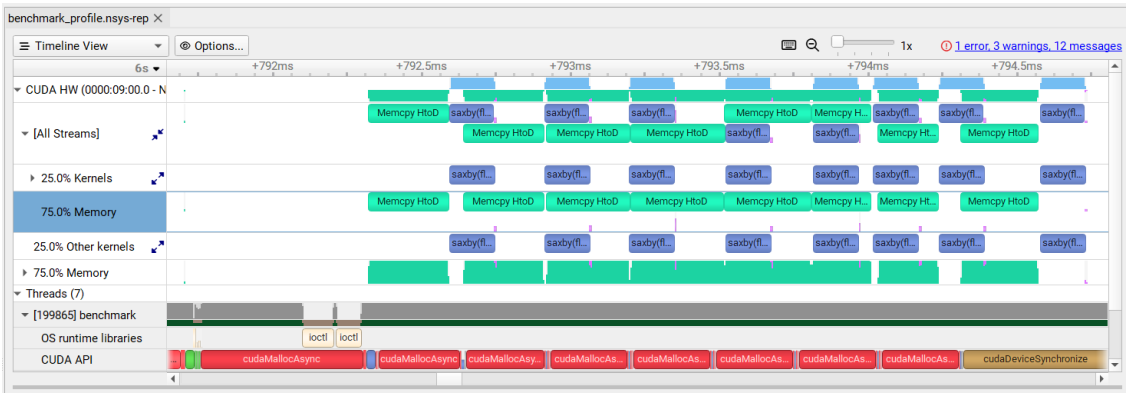
All of this is great news, since it indicates that the streams are working as intended. However, to see if we are getting noticable performance gains, we also profile `M = 8`.

1.1.3 `M = 8`

The overall profile:



A zoomed profile over `cudaStreamCreate`:



We make some *additional* observations when comparing `M = 8` to `M = 4`: 1. `cudaMallocAsync` has broken much nicer into multiple chunks occuring concurrently with the `saxby` operations. 2. `cudaStreamCreate` calls do not contribute significantly to the runtime of the overall matrix multiplication computation. 2. Each stream's `memcpyHtoD` takes much longer than the next stream's `saxby` operation.

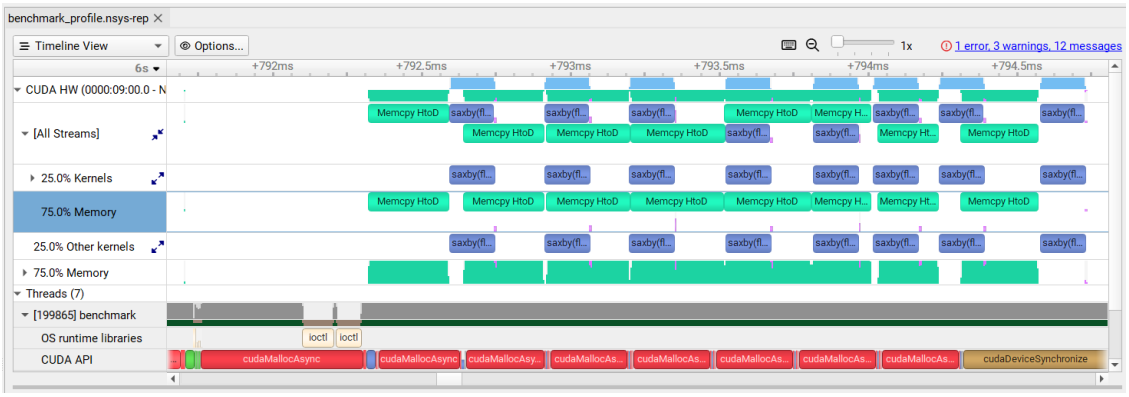
From this, we can deduce that the bottleneck in performance is the relative time for `memcpyHtoD` as opposed to `saxby`. This bottleneck outweighs the benefits of using streams since the stream's `saxby` operation must wait longer than its own computation for `memcpy` to finish before it itself runs.

To explore this further, we can make the computation more extensive by increasing the dimensions of the matrix multiplication. We choose the extreme case where `N = K = 1e4` (recall that up till now `N = K = 1e3`).

1.1.4 `M = 8` on a Larger Matrix

As expected, the stream computations are now longer than the two aforementioned *setup* calls. This would also indicate that we are getting good performance gains since we are no longer bottlenecked by setup.

The overall profile:



As expected, while `memcpyHtoD` is still the bottleneck, it has become much less so. This is because the time for `saxby` is now much larger in proportion to the time for `memcpyHtoD` compared to when the matrix was smaller (in the previous cases).

1.2 Runtime Analysis

Now that we are confident that we are getting speedups, we should quantify them. We ran 16 different experiments: 1. $M \in (1, 2, ..., 8)$ 2. $N \in (1e3, 1e4)$ 2. $K = N$

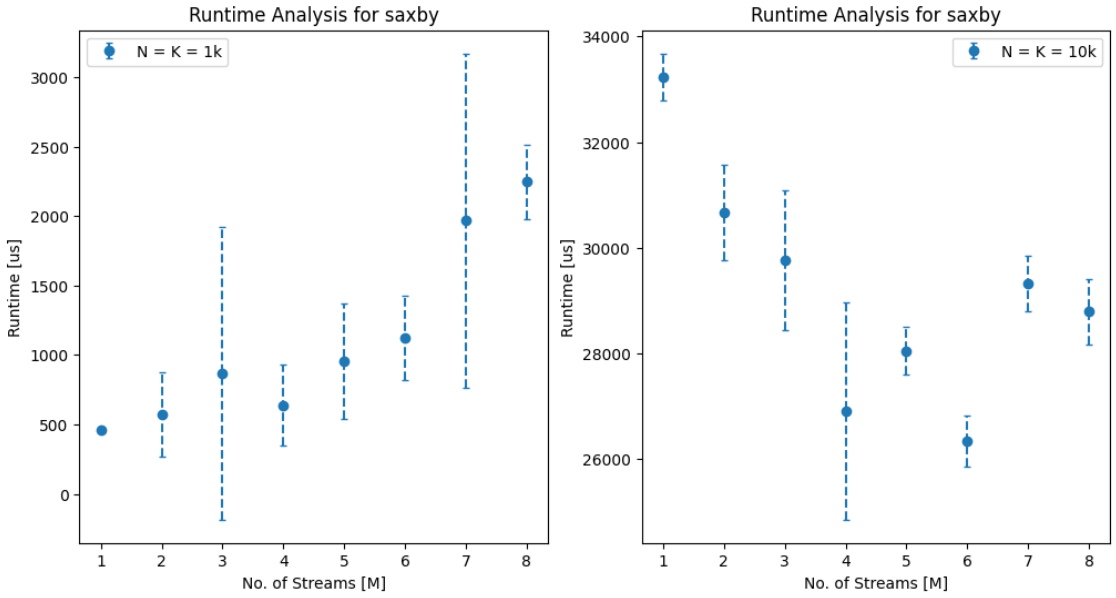
[2]:
```
rows_1k = benchmark[(benchmark['n'] == 1000) & (benchmark['k'] == 1000)]
rows_10k = benchmark[(benchmark['n'] == 10000) & (benchmark['k'] == 10000)]

fig, ax = plt.subplots(ncols=2, nrows=1, figsize=(12, 6))

eb1 = ax[0].errorbar(rows_1k['m'], rows_1k['time_us'], label='N = K = 1k',␣
 ↪yerr=rows_1k['stddev_us'], fmt='o', capsize=2)
eb1[-1][0].set_linestyle('--')

eb2 = ax[1].errorbar(rows_10k['m'], rows_10k['time_us'], label='N = K = 10k',␣
 ↪yerr=rows_10k['stddev_us'], fmt='o', capsize=2)
eb2[-1][0].set_linestyle('--')

for sub_ax in ax:
    sub_ax.set_xlabel('No. of Streams [M]')
    sub_ax.set_ylabel('Runtime [us]')
    sub_ax.set_title('Runtime Analysis for saxby')
    sub_ax.legend()
```



There are some very interesting observation here: 1. For the smaller matrix, as the number of streams increases, so does the computation. This is caused by the runting bottlenecking at the poor performance of `memcpyHtoD` in proportion to `saxby`. 2. For the larger matrix, as the number of streams increases, the computation decreases *up to a certain limit*. Beyond that limit, `memcpyHtoD` bottlenecks again. 3. In both graphs, the error bars are the standard deviations of the runtime over 10 isolated runs. The high standard deviation is caused due to, in some runs, the first `cudaStreamsCreate` takes an excessive amount of time to run. In future experiments, it would be good to discard the first runtime measurement to account for the CPU/GPU warming up.

Overall, all of our results match our expectation.