# COMP3231

## Term Project Report

Wong Ngai Sum (3035380875) - 23/11/2018

## A. Execution time

```
I1124 16:42:20.860296  5088 solver.cpp:239] Iteration 9900 (15.1521 iter/s, 6.59977s/100 iters), loss = 0.0090371
I1124 16:42:20.860360  5088 solver.cpp:258]     Train net output #0: loss = 0.00903683 (* 1 = 0.00903683 loss)
I1124 16:42:20.860373  5088 sgd_solver.cpp:112] Iteration 9900, lr = 0.00596843
```

Instead of using three kernels to replace the original **cublasSgemm**, I combined **matrix_multiplication** and **matrix_addition** into one. Before I combined two kernels, it performs ~14.5 iterations/s. After combination, it performs ~15 iterations/s. It is because there is no need to write the multiplication results to global memory and copy back to do addition. In fact, global memory reads/writes may take 300-800 clock cycles, which are likely to be the performance bottleneck. Thus, this avoids costly data transfers and saves lots of time.

It is much slower than the original version. It performs ~45 iterations/s, which is 3x faster than my version. It does not mean my implementation is not good, but cuBLAS library is internally optimised.

## B. Variable declaration consideration

I tried to reuse the computation results by putting them into variables, such as row and column number of a thread in the grid. In the matrix multiplication and addition kernel, I declared a variable to store the partial computation results, rather than keep reading and writing to global memory. This saves lots of time. I defined some preprocessor macros (#define) instead of using variables because they can be used in different functions and is more efficient. Also, I declared variables with meaningful names and comments.

## C. Block size

Matrix transpose: 32x8x1

Matrix multiplication and addition: 16x16x1

Each block is divided into warps (32 threads) and warps are scheduled to the functional units for execution. To obtain optimal efficiency, the number of threads per block should be in multiples of 32. Otherwise, lots of computing power will be wasted, as the hardware will not coalesce threads from different warps.

If a warp is waiting for memory, the warp scheduler stalls its execution and select the next "ready to go" warp for execution, in order to hide the long memory access latency and keep the hardware busy. Therefore, the block size should be large enough to hide the latency. In both kernels, the block size is 256.

2D indexing is used in both kernels. Using 1D indexing or 2D indexing does not make a noticeable difference in execution time, but 2D indexing intuitively matches
up with the matrix and is much easier to understand.

Execution resources are assigned to threads on a block-by-block basis. In each thread, a small number of variables are declared. The resource usage of blocks in each SM should not be too high.

## D. Shared memory reuse

I used shared memory to reduce non-coalesced and global memory accesses. Shared memory performs 100x faster than the uncached global memory in accessing. An uncached global memory latency can take 300-800 cycles.

In the **matrix multiplication**, threads in the same warp collaborate to reduce global memory traffic. There is no multiple access to global memory for fetching the same data. There are two shared memory tiles of 16x16 elements, which are equal to the block dimension. Max amount of shared memory per thread block is very limited (mostly 48KB), therefore it is unlikely to store a very large array. The dot product of each thread is calculated in multiple phrases. In each phrase, each thread loads two elements into the shared memory. The number of phrases required to calculate an element is equal to **K/tile width**. Shared memory tiles are reused in each phrase. Each value in shared memory is read multiple times (<= **tile width**). Therefore, the use of shared memory can at most reduce number of global memory accesses by a factor of **tile width**.

## E. Shared memory conflict

Shared memory is arranged into banks for concurrent SIMT access and there are 32 banks. In the **matrix transpose** kernel, the shared memory tile is of 32x33 elements. There is no coalescing problem when writing data into the tile, as adjacent threads access adjacent elements, but there are non-coalesced global memory accesses. To achieve coalesced global memory accesses, we have to change the access pattern from column major to row major. Then all elements in a column of data would map to the same bank, resulting in a 32-way memory bank conflicts. To avoid the conflicts, I padded the width in the declaration of the shared memory tile, making the tile 33 elements wide rather than 32. Therefore, adjacent threads read shared memory from consecutive banks with stride 33.

In the **matrix multiplication and addition** kernel, there is no coalescing problem. It is because each thread accesses different banks. There is no need to pad the width of the shared memory tiles since the access pattern is not row-major.

## F. Global memory coalescing

In the matrix, elements in each row are arranged in a linear manner, followed by the next row. Row access cannot be coalesced but column access can be coalesced as they are adjacent. Grouping of threads into warps is relevant to global memory accesses. In the two kernels, sequential memory accesses are adjacent, consecutive threads access consecutive memory locations and there is no stride access. Then, all the accesses are combined into one single request by the hardware. To sum up, coalescing minimises the number of bus transactions and greatly improves throughput.

## G. Other aspects

I provided extensive comments to explain how the code works and the use of variables, so that the code can be understandable and maintainable. CUDA errors are checked by function **CUDA_CHECK**. Device global memory would be freed after the computation finished. The code style is good as it follows the original indentation.