CS3210 – AY 2016/17 – Semester I
Comments for Assignment 1

Question 1

Common approaches:

1. Optimizations – the simplest way to improve the performance was to perform some basic optimizations on the code, such as:
    - Use a local variable to accumulate the result of multiplications inside the CUDA kernel. This is done to avoid accessing the slow global memory.
    - Change matrix from a 2-dimensional data structure to a 1-dimensional data structure (array) to avoid extra time on allocating and transferring the data structure.
    - Transpose matrix B in order to access it row by row. In this way you can take advantage of coalesced memory access. This coalesced access means that the hardware performs less data transfers (the size of one transfer can be 128B or more, depending on the GPU) when the requested data is closely groped in the global memory (data locality).

2. Tiling – to further improve the performance, you could use the "tiling method" for multiplying two matrices. This method divides the matrices into sub-matrices or "tiles". Given square matrices of size N, we divide them into square sub-matrices of size M (M << N). Let A and B be the input matrices and C the result matrix. For each sub-matrix of size M in the result matrix C, $\lceil N/M \rceil$ sub-matrices from A and B are used, as in Fig. 1 (where BLOCK_SIZE is M).

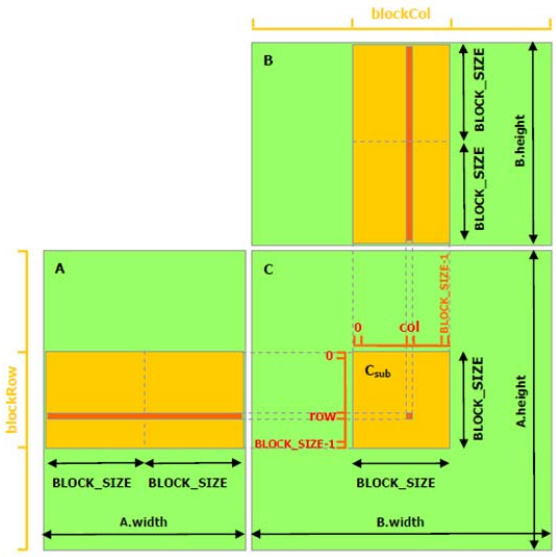

Figure 1. Matrix multiplication using tiling

(source: *http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/matrix-multiplication-with-shared-memory.png*)

This method is suitable for GPUs because it can use the fast, but small, shared memory instead of slow, but large, global memory. At each iteration, a tile from A and a tile from B can be

loaded from the global memory into the shared memory. These tiles contribute to M*M values from the result matrix C. In the provided code, there are $2N^3$ global memory accesses because each element from matrices A and B contributes to N elements of the result matrix C. But using the tiling method, each element is loaded ⌈N/M⌉ times, thus, reducing the global memory accesses by a factor of M.

To implement this method on the Jetson TK1 system, you should first notice that the GPU supports 1024 threads per block, thus, you can use a square thread block of 32 to compute a tile from the result matrix (thus, M=32). In the CUDA kernel, there is a loop of size ⌈N/M⌉ to iterate through all tiles from matrices A and B needed to compute one tile from matrix C. In this loop, the threads have to load the tile in shared memory. Because they may do this loading at different pace, you need to synchronize them using a *__syncthreads()* call. Then you compute the partial result by iterating through M values.

Special attention need to be paid when N is not multiple of M (or of 32 in this case). One way to overcome this is to use padding by adding 0 values in the matrices. Another way is to check the boundary inside the kernel.

Grading scheme:
- As a large number of submissions performed very well and are closed together, I have decided to reward more submissions with bonus mark to recognize your hardwork ☺.
- So, instead of the "top 2 submission get 2 bonus marks", I gave out bonus mark in the range of [+0.5 to +2.0] on increment of 0.5. You will get bonus mark as long as your submission shows 20+ times speedup at least.
- However, still true to our words, only 2 student receive the full bonus (one with a speedup of >80, then other ~70).

Question 2

a. Two values for speedup should have been computed. The time for sequential program *T1(seq)* should be smaller than the time for parallel program run on a single core *T1(par)*. A number of submissions have high values for the execution time which may have happened because the code was not compiled with –O3 flag as specified in the question.
   Grading scheme:
   - If T1(seq) is larger than T1(par) = -0.5p.

b. Explain that *T1(seq)* is smaller than *T1(par)* because of the overhead due to using 64 threads on a single core which includes thread creation and context switch.
   Grading scheme:
   - Explanation is not complete / not clear = -0.5p to -1p

c. Comment that the speedup computed with *T1(seq)* is smaller because *T1(seq)* is a smaller execution time. Also, explain the **meaning of using T1(seq)** vs **T1(par)**: The former give a fair comparison between sequential and parallel approach, with using the latter gives a better idea on how more CPU resource help to improve the parallel performance as thread overheads are present in T1(par).

   Explain that the speedup degrades or it is even lower when using 6 and 8 cores because the machine has only 4 physical cores and uses HyperThreading to create the illusion of having 8 cores. But the ALU/FPU resources are limited, only the frontend being duplicated, thus, a compute-intensive program may not benefit from it. Moreover, running 64 threads also creates many context-switches and cache misses which affect the execution time.

   Grading scheme:
   - Incomplete explanations – 0.5p

d. The experiment should have been conducted on 8 cores by varying the number of threads from 1, 4, 8 to **64 or more** (using multiples of 8). The best execution time is obtained for 8 threads. You could use a smaller input size such as 2048 to reduce the waiting time for experiments.
   Grading scheme:
   - Focus on three ingredients: The experiment setup, the data collected and the explanation / conclusion.
   - Each ingredient can range from (not clear, ok, good).
   - Every "not clear" = -0.5p

e. The experiment could start from the data gather at point *d.*, take the time for 8 threads, T_8 as baseline, and run the program with even more number of threads, p (up to 1024 or 2048). The execution time, Tp, should increase and by plotting it, one can compute the slope of the ascendant line to get the overhead of treading. It is similar to computing the overhead as:

$$T_o = \frac{T_p - T_8}{p - 8}$$

   Any experiment with the same core idea is acceptable. The value is around $0.1s$.
   Grading scheme:
   - Similar to part (d).