Adam Alsegard, A0153097B

Assignment 1 - Report

Question 1:

There are two attached cuda files; mm-cuda.cu and mm-cuda-firstTry.cu. These are two different optimizations but mm-cuda.cu is the main one.

The first optimization (mm-cuda-firstTry.cu, called $\mathbf{T}_{\mathbf{par\_opt1}}$ in *Table 1*) is basically one change. The i and j values have been swapped in mm_kernel. That's because the matrices are allocated in row-major so therefore you want to link the threaded rows to the rows in the matrix to better utilize the cache memory. The block size is also kept to 32 to best utilize the blocks. The maximum threads per block is 1024 which means that the warp size is 32 (32*32 = 1024). By just doing this a speedup of ~13 could be achieved (see *Table 1*).

Another method is shown in mm-cuda.cu (called $\mathbf{T}_{\mathbf{par\_opt2}}$ in *Table 1*), this is the main optimization. Here the block size is kept at 32 for the same reasoning as above but the kernel differs quite a lot. Instead of keeping everything in the global memory, which is slow, sub-matrices of size 32x32 is copied into the shared memory, which is much faster. This way every thread in the block handles one element in the resulting matrix, and all memory is shared within the block. Originally, each element in matrix A gets loaded by each thread that needs it but now it will only load once for every block on that row in the resulting matrix. So instead of reading elements in matrices A & B *size* times from global memory, now they are only read *size/block_size* times. Because memory access is one of the bottlenecks in GPU programming this is a big improvement.

The total amount of shared memory per block on the Jetson is 49152 bytes. Every pair of shared matrices requires 2x32x32x8 bytes memory (64 bit system gives 8 byte word size) which means that there is room for 49152/16384 = 3 pairs of matrices in the same block at one point in time. The threads are synchronized before and after the main calculation to make sure that the correct memory is loaded and up to speed.

For the calculations the sub-matrices moves horizontally on matrix A and vertically on matrix B. A couple of statements are added to make sure that the algorithm works for sizes that cannot be equally divided by the block size. This slows down the program a bit but makes it more dynamic. With these optimizations a speedup of around 69 can be derived for a matrix of size 512.

Table 1 – execution time

	Time (seconds)					
Matrix size:	$\mathbf{T}_{\mathbf{seq}}$	$\mathbf{T}_{\mathbf{par\_original}}$	$\mathbf{T}_{\mathbf{par\_opt1}}$	**Speedup**	$\mathbf{T}_{\mathbf{par\_opt2}}$	**Speedup**
512	0.65	12.4	0.97	12.4/0.97 = 12.8	0.18	12.4/0.18 = 68.9
600	1.45	20.1	1.54	20.1/1.54 = 13.1	0.30	20.1/0.30 = 67.0
1024	32.1	17.9	1.42	17.9/1.42 = 12.7	0.28	17.9/0.28 = 63.9

Note: all timings are an average of at least 10 runnings, after execution time has "stabilized".

Adam Alsegard, A0153097B

Question 2

a) Run mm-seq.c ($\mathbf{T_{1\_seq}}$) and mm-omp.c ($\mathbf{T_{n\_par}}$) with matrix size 4096 on Intel i7 tower system and complete the table:

Table 2 – execution time

Number of Cores	Execution time (sec), 64 threads	Speedup(T_{1_seq})	Speedup(T_{1_par})
1	$\mathbf{T_{1\_seq}}$**:** 197 $\mathbf{T_{1\_par}}$**:** 378	- -	- -
2	$\mathbf{T_2}$**:** 302	197/302 = 0.65	378/302 = 1.25
4	$\mathbf{T_4}$**:** 160	197/160 = 1.23	378/160 = 2.36
6	$\mathbf{T_6}$**:** 164	197/164 = 1.20	378/164 = 2.30
8	$\mathbf{T_8}$**:** 166	197/166 = 1.19	378/166 = 2.28

Note: time is the average of at least 3 runnings

b) Is there any difference between $\mathbf{T_{1\_seq}}$ and $\mathbf{T_{1\_par}}$ regarding **execution time**?

Yes! The parallel version takes way longer because there are too many threads on one core. They block each other and therefore takes longer time. If we use 2 or 1 thread we get around 205 seconds instead which is more in par with the sequential algorithm.

c) Comment on the **speedup** using $\mathbf{T_{1\_seq}}$ and $\mathbf{T_{1\_par.}}$

Regarding the speedup from the sequential version there isn't any for 1 or 2 cores. The execution time is actually worse because of the management of the threads. The speedup is optimal for 4 cores and then gets gradually worse again.

Almost the same can be said regarding the speedup from the parallel version, but here 2 cores is better than 1. However 4 cores is still the best utilization which leads to the assumption that the parallel portion of the algorithm can only be divided into 4 sub-portions to be best utilized.

d) For shared-memory, what should be an appropriate **number of threads** if we use 8 cores? Device an experiment to determine the number of threads.

If we run *lscpu* on Ubuntu we see that Intel i7 has 2 threads per core, 4 cores per socket and only one socket. This means 4 hyper-threaded physical cores which gives 8 logical cores. Logically the best would therefore be to run 8 threads on 8 cores.

However, it also depends on the code and how big the parallel portion is. The running time will never exceed the sequential portion so to use more threads when that limit approaches will only slow down the program.

Adam Alsegard, A0153097B

The experiment would be to determine the sequential portion of the program by using Amdahl's Law and then adapt the number of threads for optimal performance. The upper bound should be 8 threads, and that is best utilized when you can follow Gustafson's Law.

Amdahl's law: Speedup = p / (1 + (p – 1)*f)
Because of earlier assumption we will try 8 cores with 4 and 8 threads.

8 cores, 4 threads gives T_par: 94.4 and speedup: 2.08→ f = 0.48
8 cores, 8 threads gives T_par: 103 and speedup: 1.91 → f = 0.52

4 threads gives the best speedup. The sequential portion is probably somewhere around 40-45%. In theory the best number of threads for this size would be 3, but there is always some bottlenecks so in practice the best utilization is given by 4 threads.

e) Devise an experiment to determine the **average threading overhead** in terms of overhead time per thread!

OpenMP has a fixed startup time and then a per-thread overhead. To find the average threading overhead you could make a program that only create threads and then destroys them. Then run the program, first with only one thread and then with lots, let's say 1000. Subtract the first execution time from the second and divide the resulting time by 999, that is the average overhead per thread.

(Or you could use a tool, such as OpenMP EPCC Michrobenchmarks.)

To just look at the execution time of a program like mm-omp when adding more threads wouldn't do any good because the blocking of threads is a larger time factor than the overhead.