Compiling and Running the Program

The submission folder contains all the results and compiled binary file. Please find details in table 1.

Folders/Files	Comments
outputs/	results from all binary executions, the results only hold the run-time, instead of matrix value
Task1GPUsp.cu	Task 1 GPU single precision code
Task1GPUdp.cu	Task 1 GPU double precision code
Task1CPUsp.cu	Task1 CPU single precision code, I am using .cu since it is using CUDA functions for the execution time check
Task1CPUdp.cu	Task1 CPU double precision code, I am using .cu since it is using CUDA functions for the execution time check
Task2GPUsp.cu	Task 2 GPU single precision code
Task2GPUdp.cu	Task 2 GPU double precision code
Task3GPUsp.cu	Task3 GPU single precision code
Makefile	Makefile for "make"
runallGPU.sh	A bash script mimicking your execution script
makeall.sh	A bash script to compile all programs
***.o**	object files from compilation

Table 1, top-tier directory

To **compile** everything, please run:
Bash$ ./makeall.sh
To **compile only CPU code**, please run:
Bash$ make cpu
To **compile only GPU code**, please run:
Bash$ make gpu
To **compile and run** the binary file:
Bash$ ./runallGPU.sh
Note that the runallGPU.sh script mimics the execution script mentioned in the assignment description.

The following sections go through each of the tasks and its results. The timing results are in the unit of **milliseconds**. I have verified computation correctness of all the tasks using the given differential debug function. In most cases, the differences are in nano (10^-9) unit scale.

Task 1

Part a

n	1,024	2,048	4,096	8,192	1,024	8,192	8,192
m	1,024	2,048	4,096	8,192	1,024	8,192	1,024
p	1,024	2,048	4,096	8,192	8,192	1,024	8,192

block_x	16	32	32	32	16	32	32
block_y	16	32	32	32	16	32	32
grid_x	64	64	128	256	64	256	32
grid_y	64	64	128	256	64	256	256
Task1GPU sp	37.548126	303.394348	2414.713867	18488.47266	331.14209	2285.944336	2415.6604
Task1CPU sp	1075.180298	6863.082520	60403.601562	481032.343750	7206.443359	60780.449219	62120.371094

Table 2, Task 1, part A, All timing unites are in milliseconds. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds.

From table 2, I can see that in most cases, when the block dimension fits the warp size (32 threads), the calculation performs the best with the rare case of 1024 x 1024 matrices. One possible explanation is that the matrix is too small to see the apparent timing differences between 16 x 16 blocks and 32 x 32 blocks.

Also, unsurprisingly, the case with largest matrix size (n = 8192, m = 8192, and p = 8192) is the most computationally expensive, and the case with smallest matrix size (n=m=p= 1024) can finish the computation within 38 ms. Part B shows the timing comparison between double precision and single precision computation.

Part b

n	8,192
m	8,192
p	8,192
block_x	32
block_y	32
grid_x	256
grid_y	256
Task1GPUsp	18488.47266
Task1GPUdp	28620.86914
Task1CPUsp	481032.34375
Task1CPUdp	758135.25

Table 3, Task1, Part B, DP vs SP. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds.

I also included TASK1 GPU and CPU SP timing results (grey cells) in table 3 for better comparison. Since I already come to a conclusion that, with block (32 x 32) and grid (256 x 256), I can get the best timing result for SP case, I continue using the configuration for this DP case. Though the DP case is not exactly double the wall clock time as the SP case, it still takes significantly longer to complete. The data shows that DP needs about 28 seconds to finish the computation whereas the SP only needs a18.5 seconds to finish everything.

Part c

n	16384
m	16384

p	16384
bx	32
by	32
gx	512
gy	512
Task1GPUsp	156206.859375

Table 4, SP maximum square matrix. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds.

The GPU SP case can handle matrix size of up to m=n=p=16384, while the block is configured as 32 x 32, and the grid is configured as 512 x 512. Given that the M2090 has 6 GB GDDR. I would assume that the maximum SP I can allocate on shared memory is 6GB. My assumption is correct in this case since I have 3 matrices A, B, and C allocated on the shared memory. Each of them is 16384 x 16384 SP float matrix. Therefore, the total size here is 16384 x 16384 x 4(bytes per float) x 3 (matrices) = 3GB. The next size-up would be matrices of size 32768 x 32768, which leads to a total memory storage of 32768 x 32768 x 4 x 3 = 12GB, and this value is way over 6GB. The timing result shows that a 16384 x 16384 matrix needs 156.2 seconds to complete calculation.

n	8,192
m	8,192
p	8,192
block_x	32
block_y	32
grid_x	256
grid_y	256
Task1GPUdp	28620.86914

Table 5, DP maximum square matrix. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds.

Similar to the SP case, the storage limitation can be applied to the DP scenario. In the DP case, the total storage is 8192 x 8192 x 8 x 3 = 1.6 GB. The next size-up, 16384 x 16384 x 8 x 3 = 6.5 GB, is still off the 6GB GDDR limit, so the maximum size for DP is 8192 X 8192, and the calculation time is 28.6 seconds.

Task 2

Part a

n	1,024	2,048	4,096	8,192	1,024	8,192	8,192
m	1,024	2,048	4,096	8,192	1,024	8,192	1,024
p	1,024	2,048	4,096	8,192	8,192	1,024	8,192
block_x	16	32	32	32	16	32	32
block_y	16	32	32	32	16	32	32

grid_x	64	64	128	256	64	256	32
grid_y	64	64	128	256	64	256	256
Task1GPUsp	37.548126	303.394348	2414.713867	18488.47266	331.14209	2285.944336	2415.6604
Task2GPUsp	20.825567	164.923645	1318.415527	10571.77344	167.552505	1340.049805	1340.930054

Table 6, Task2, Part A, Tiled approach. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds.

Task 2 uses tiled approach to comprehend the memory bandwidth bottleneck. I do not show the tile width (TW) in table 6 as the assumption is that TW=block_x= block_y. The timing results are shown in table 6. Note that in table 6, I have also included results from Task1 GPU SP (grey cells) to show the timing differences between the two. It is apparent that the tiled approach has clear advantage over naïve implementation. The tiled implementation can almost halve the computation time in all cases.

Part b

n	8,192
m	8,192
p	8,192
bx	32
by	32
gx	256
gy	256
Task2GPUsp	10571.77344
Task2GPUdp	11834.324219

Table 7, Task 2, Part B, tiled DP. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds.

I continue using the block dimension (32 x 32) and grid dimension (256 x 256) as it has been proven to be the fastest configuration in task 1, part A.
With a matrix size n=m=p=8192. Table 7 shows the tiled DP timing results (11.8 seconds), which is not dramatically slower than SP (10.57 seconds).

Part c

n	16384
m	16384
p	16384
bx	32
by	32
gx	512
gy	512
Task1GPUsp	156206.85935
Task2GPUsp	86810.109375

Table 8, Task 2, Part C. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds.

Table 8 shows the maximum size allowed on M2090's 6GB GDDR. In both DP and SP case, the the storage constraints mentioned in Task 1 Part C applies here as well. Since the maximum size for DP is 8192 x 8192, please refer to Table 7 for the timing results.

Task 3

n	8,192	NTB Size
m	8,192	
p	8,192	
bx	32	
by	32	
gx	256	
gy	256	
Task1GPUsp	18488.47266	
Task2GPUsp	10571.77344	
Task3GPUsp	157850.2813	32
Task3GPUsp	197025.140625	16
Task3GPUsp	196667.156250	8
Task3GPUsp	198572.609375	4
Task3GPUsp	198592.25	2

Table 9, Multitile timing results. Blue cells show the optimum configurations of Blocks and Grids. The white cells show the timing results in milliseconds. Green cells show the NTB sizes tested

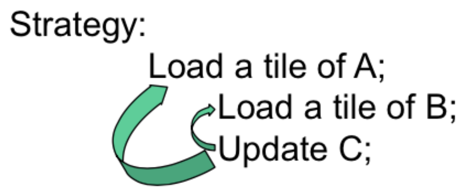

Figure 1, implementation Multitile

Figure 1 shows my strategy on implementing the multi-tile. In short, I load NTB number of B tiles for every load of A tile. I tried NTB size of 2, 4, 8, 16, 32, and 64. The program fails at NTB=64. This makes sense since a warp consists 32 threads. The timing results are also presented in table 9. Even though NTB-32 case is fastest among other NTB options. It is still slower than Tiled and naïve implementation. This anomaly behavior is probably caused by my implementation **not** assuming matrix dimensions are multiple of the tile width, which requires precise boundary check and flush 0 for all out of bound elements. My implementation basically sacrifices timing performance for program flexibility.