CPSC 524: PS 6 (GPU)

kd538 - Keyang Dong

Directory Structure

Code: task1.cu, task2.cu, task3.cu;
Make executable targets using `make task1`, `make task2`, `make task3`;
`run_task2.sh`: qsub bash script for timing task2 using different n, m, p settings.

Task 1

a) Single Precision (Multi Runs)

Run No.	**1**	**2**	**3**	**4**	**5**
GPU Time (ms)	47.6	23950.0	381.7	2915.5	3096.8
CPU Time (ms)	806.6	467020.2	7356.2	57966.1	62291

b) Double Precision

Using the same settings of block_dim, timing results are compared as following:

Single Precision (ms)	**Double Precision (ms)**
23950.0	36322.1

Double precision requires 150% amount of time as much as single precision runs.

c) Max Square Dimension

The only limit to the size of problem (square matrices) is the address capacity of NVIDIA M2070 GPU' s on-chip memory, which is 6GB according to wiki page: https://en.wikipedia.org/wiki/Nvidia_Tesla. Major memory consumption including 3 large arrays, namely a, b, and c representing the matrices A, B and C. Thus if we assume the largest dimension is x, then each array has size x * x * sizeof(float) * 3 <= 6GB, we get x <= 22.63K.

Based on estimation I made several run (on device 2) with settings and timings:

Dimension	20000	*21000*	21500	22000
Timing (s)	351.34	*460.05*	N/A	N/A

Task 2

a) Single Precision (Multi Runs)

Common Settings:

Device No.	2	
Block_dim_x, Block_dim_y	TW	Each block takes over one tile's computation.
Grid_dim_x	m / TW	
Grid_dim_y	n / TW	

Timing Results:

Tile Width (TW)	**Run 1 (ms)**	**Run 2 (ms)**	**Run 3 (ms)**	**Run 4 (ms)**	**Run 5 (ms)**
8	23.5	11525.2	183.2	1466.3	1450.7
16	12.1	6299.4	97.6	773.1	777.0

32	12.0	6298.6	97.5	772.9	777.0

TW = 64 will exceed the limit of thread number in a block (32^2=1024 is the up limit itself). As comparing to naive solver, the performance after utilizing shared memory increases by as much as 300%.

Best Settings:

1) cudaFuncSetCacheConfig(gpu_matmul_tiled, cudaFuncCachePreferShared);
2) Tile Width = 32;
3) Block_dim = 32 * 32;
4) Grid_dim = (m / Block_dim.x) * (n / Block_dim.y).

b) Double Precision

Using the same settings of block_dim, timing results are compared as following:

Single Precision (ms)	**Double Precision (ms)**
6298.6	11523.4

Double precision requires 183% amount of time as much as single precision runs.

c) Max Square Dimension

Similar to estimation in Task-1(c) section, I made several run (on device 2) with settings and timings:

Dimension	20000	21000	*21400*	21500
Timing (s)	88.169	111.853	*116.848*	N/A

The largest dimension is now 21400, with runtime 116.848s on device 2, M2070 GPU, block_dim=32. As shared memory brings some overhead, the actually dimensional limit is a little bit smaller than naive solver in Task-1(c).

Task 3

Running case 8192 x 8192 x 8192, using same block_dim = 32,
while Grid_dim = (n or m) / block_dm / TH,
so that we can keep each tile as square shaped;
still running on device 2, I obtained the timing chart below:

TH	**2**	**3**	**4**	**5**	***6***	**7**	**8**	**9**
Time (ms)	4621.4	3922.9	3659.2	3414.0	*3280.5*	N/A	N/A	N/A

It is clear that as cached consecutive tiles grows, as long as the shared memory can fit into the amount of tiles data from B, running time monotonically decreases, the efficiency raises by at most 92%.

When TH = 6, we obtained the best possible efficiency with previous optimal settings of dimensions on device 2. If we continue to increase TH, then the shared memory will be flushed, and calculation results became faulty.