MAP55616. GPU Programming with CUDA

Assignment 1

Luning Zhang (zhangl14@tcd.ie)

17th, Mar, 2023

Task 1 – CPU calculation

I have used a n*m = 1000*1000 matrix to test the performance of the CPU computing version of the code, and the results obtained are as follows:

```
zhangl14@cuda01:~$ ./matrix -t -n 1000 -m 1000
Matrix size: 1000x1000
Use random seed: false
Print Timing: true

Row sum: 2500577.500000
Column sum: 2500577.000000

Row sum duration: 3348.112106 microseconds
Column sum duration: 3365.993500 microseconds
Row reduce duration: 2.861023 microseconds
Column reduce duration: 3.099442 microseconds
```

The row sum function and the column sum function have similar time complexity since they both need to iterate through all the elements of the matrix once. However, the memory access patterns of each function are different. The row sum function accesses elements that are close to each other in memory, since they belong to the same row, which can take advantage of spatial locality and result in faster memory access times. On the other hand, the column sum function accesses elements that are far apart in memory, since they belong to different rows, which can result in slower memory access times due to cache misses and increased memory latency.

The reduce functions, which calculate the total sum of the row and column sums, respectively, have a time complexity that is proportional to the length of the input vector. Since the input vectors have lengths that are equal to the number of rows or columns in the matrix, the time complexity of the row reduce function and the column reduce function is proportional to n and m, respectively.

Task 2 – CPU and GPU calculation (single precision)

To organize the grid, I arrange it in one dimension with a number of blocks equal to either `num_blocks_row` or `num_blocks_col`, depending on which kernel is being launched. Additionally, I have added a new command line argument "-a" which displays both the CPU and GPU timings for each norm side by side. Use a 1000*1000 matrix for testing:

```
zhangl14@cuda01:~$ ./float -t -r -a -n 1000 -m 1000
Matrix size: 1000x1000
Use random seed: true
Print Timing: true

The results for CPU calculation:
Row sum: 2497527.250000
Column sum: 2497526.000000
The results for CPU and GPU Calculation:
Row sum: 2497527.000000
Column sum: 2497526.750000

Time Compare:
Row sum duration:
Serial Version: 5702.972412 microseconds; Parallel Version: 270.128250 microseconds
Column sum duration:
Serial Version: 5424.976349 microseconds; Parallel Version: 38.862228 microseconds
Row reduce duration:
Serial Version: 3.099442 microseconds; Parallel Version: 10.013580 microseconds
Column reduce duration:
Serial Version: 2.861023 microseconds; Parallel Version: 9.059906 microseconds
```

Task 3 – Performance improvement (single precision)

In this section, I employed various block sizes to compute matrix data of different sizes and evaluated the acceleration and error of its CPU and GPU versions. To maintain data consistency from run to run, I utilized the default random number seed "srand (123456)".

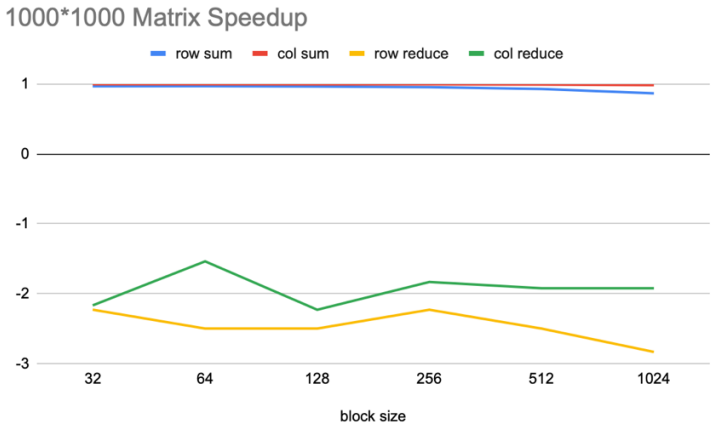
The tables and graphs presented below depict the test results of the CUDA parallel code for each matrix of different sizes. Based on the results obtained, it is evident that the row and column directions of the calculation matrix using CUDA are highly efficient and can substantially expedite the computation process, particularly for larger matrix sizes and block sizes.

(a) For a 1000x1000 matrix, the optimal block size is 64. However, in this case, the row and column reduction methods did not exhibit satisfactory performance. Given the matrix's

relatively small size, it may not be worthwhile to utilize the CUDA method for acceleration as significant time is wasted in CPU-GPU communication.

Table1 - Speedup and errors for size 1000*1000 matrix

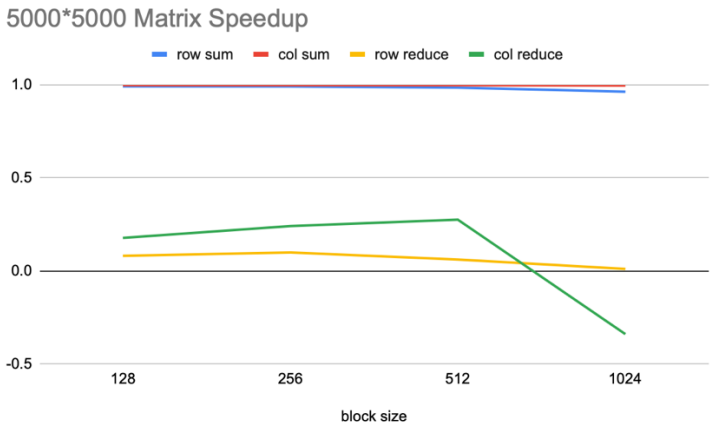
block size	row sum	col sum	row reduce	col reduce	error
32	0.966661	0.994807	-2.230769	-2.166667	3.5
64	0.968097	0.994605	-2.5	-1.538461	3.5
128	0.962791	0.993168	-2.5	-2.230769	3.5
256	0.953715	0.993057	-2.230769	-1.833333	3.5
512	0.928072	0.990243	-2.5	-1.923076	3.5
1024	0.865949	0.982652	-2.833333	-1.923076	3.5



(b) For 5000*5000 Matrix, the best block size is 128:

Table2 - Speedup and errors for size 5000*5000 matrix

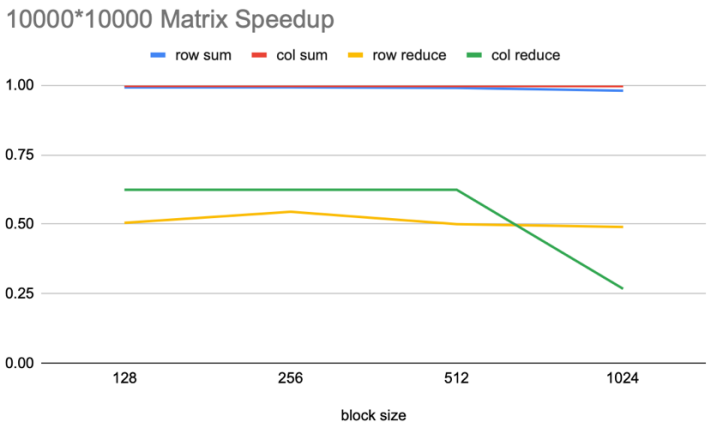
block size	row sum	col sum	row reduce	col reduce	error
128	0.991148	0.997726	0.08	0.176471	144
256	0.99061	0.997535	0.098039	0.24	148
512	0.984268	0.997248	0.06	0.27451	148
1024	0.962614	0.995604	0.009608	-0.34	140



(c) For 10000*10000 Matrix, the best block size is 256:

Table3 - Speedup and errors for size 10000*10000 matrix

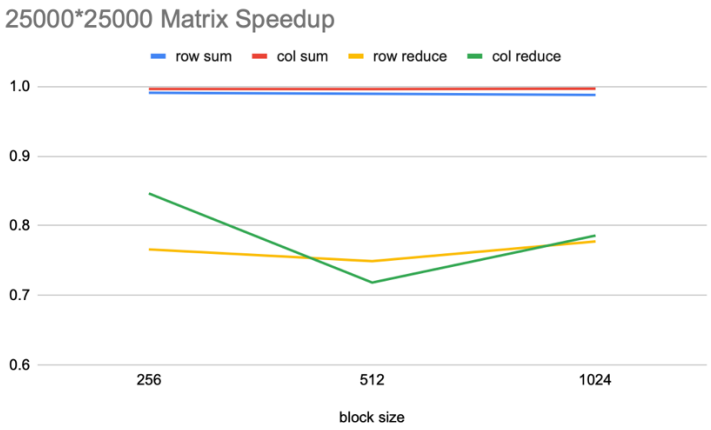
block size	row sum	col sum	row reduce	col reduce	error
128	0.992303	0.997862	0.50495	0.623762	576
256	0.99263	0.997768	0.544554	0.623762	528
512	0.990665	0.997625	0.5	0.623762	560
1024	0.980548	0.997064	0.49	0.268041	528



(d) For 25000*25000 Matrix, the best block size is 256:

Table4 - Speedup and errors for size 25000*25000 matrix

block size	row sum	col sum	row reduce	col reduce	error
256	0.99105	0.996432	0.765873	0.846154	3072
512	0.989469	0.996296	0.749004	0.718254	3584
1024	0.987906	0.996878	0.777328	0.785714	3200



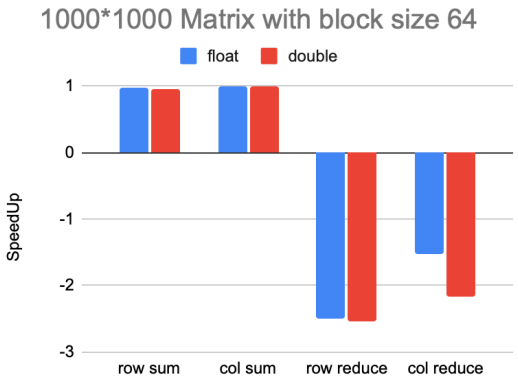
Overall, it's important to consider both the speedup and the overhead when deciding whether to use CUDA for a particular problem. For small matrices, the overhead of data transfer and initialization may be significant enough to negate the benefits of parallelization, while for larger matrices, the speedup can be significant and well worth the additional overhead.

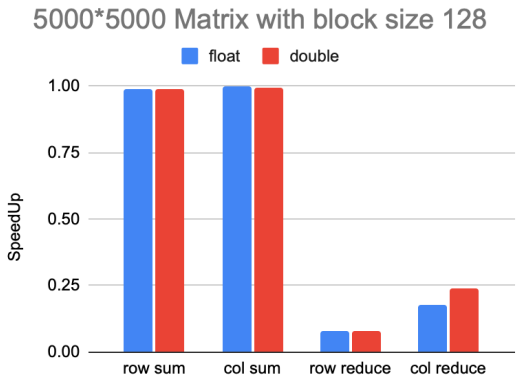
Task 4 – Double precision testing

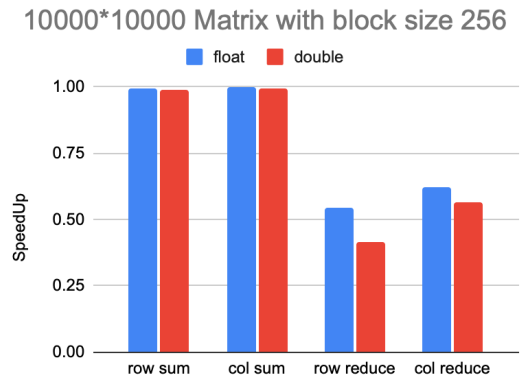
Conduct a test using the optimal block of single precision. Refer to the chart below for a comparison between the acceleration and error of double precision and single precision.

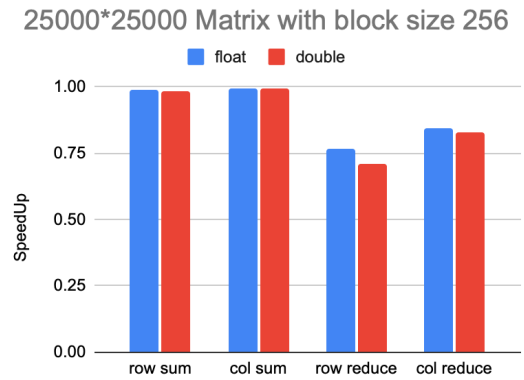
Table5 - Speedup and errors comparison for Float and Double precision

Matrix Size	Block Size	Precision	Row sum	Col sum	Row Reduce	Col reduce	Error
1000	64	float	0.968097	0.994605	-2.5	-1.538461	3.5
		double	0.959181	0.988104	-2.538461	-2.166667	0
5000	128	float	0.991148	0.997726	0.08	0.176471	144
		double	0.990588	0.995836	0.08	0.24	0
10000	256	float	0.99263	0.997768	0.544554	0.623762	528
		double	0.98765	0.996031	0.415842	0.5625	10^(-7)
25000	256	float	0.99105	0.996432	0.765873	0.846154	3072
		double	0.984888	0.994898	0.707031	0.82996	10^(-6)









The chart clearly shows that, compared to single precision, the speed improvement of double precision tests decreases slightly for the same amount of calculation. This is reasonable because double precision has a larger data storage capacity, but it still maintains good acceleration performance in parallel processing. Additionally, double precision greatly improves calculation accuracy, even in cases with large amounts of data, reaching an accuracy of 10^{-6}. Therefore, in practical applications where very high calculation accuracy is required, double precision can be used instead of single precision for parallel processing.