CS610 - Programming for Performance
2020-2021-I: Assignment 5

Name: Sarthak Singhal Roll No.: 170635

Note: All the codes were run on gpu2.cse.iitk.ac.in. Before compiling the codes run: export PATH=$PATH:/usr/local/cuda-10.0/bin

1. Problem 1

Compilation and execution instructions:

- nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p1.cu -o assignment5-p1
- ./assignment5-p1

Explanation and results:

After figuring out the dependences in the problem, only j loop doesn't carry a dependence and hence only it can be parallelized. So, I permuted the loops to jki from kij to parallelize the j loop. The serial version takes 26.3 seconds. I kept 1024 threads per block for kernel1 and it gave a speedup of 26.3/2.16 = 12.17. For kernel2, I tried changing threads per block to 1024,512,256,128,64,32 and the best performance was at 32. The speedup was 26.3/1.51 = 17.41. The difference is because in the first case only 8 blocks will be created for kernel 1 which will be 9 blocks for kernel 2 if threads per block is 1024. But as the number of SMs is 28, all the resources are not exploited, hence small number of threads per block perform better as more blocks are created in this case. Also, in kernel2 size is not power of 2 and hence, if we keep more number of threads per block then most of the threads in the last block will be created but they will be of no use.
I also tried 2-way unrolling in the kernel2 and the speedup was 26.3/1.25 = 21.04. One of the reason for increase in speedup can be less number of loop checks and increments.
Also, there was no serial version for SIZE = 8200 and hence to check the result, I compared the result obtained from kernel2 and kernel3. The kernel1, kernel2 and unrolled kernel2 are implemented in the functions *kernel1*, *kernel2* and *kernel3* respectively.

2. Problem 2

Compilation and execution instructions:

- nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p2.cu -o assignment5-p2
- ./assignment5-p2

Explanation and results:

In the first kernel, there are two kernel functions: *kernel_excl_prefix_sum_ver1_1* and *kernel_excl_prefix_sum_ver1_2*. The first kernel computes exclusive sum for each chunk will some CHUNK_SIZE. The second kernel does the actual work of computing the exclusive sum for each element. In the first iteration launch of second kernel it computes the exclusive sum for CHUNK_SIZE blocks of array. Alternating blocks are computed, like first ignored, second computed, etc. Then in the second iteration, CHUNK_SIZE is doubled and same thing happens. This way exclusive sum for whole array is computed. I tried various values of CHUNK_SIZE and threads per block and the best performance was achieved at CHUNK_SIZE = 2048 and threads per block = 256. The serial version takes 43.63 ms. The speedup was 43.63/39.21 = 1.11.

In the second kernel also, there are two kernel functions: *kernel_excl_prefix_sum_ver2_1* and *kernel_excl_prefix_sum_ver2_2*. The difference is just that *ver2_2* has code with 4-way unrolling. The speedup was 43.63/38.38 = 1.13. One of the reason for increase in speedup can be less number of loop checks and increments.

3. Problem 3

Compilation and execution instructions:

- `nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p3.cu -o assignment5-p3`
- `./assignment5-p3`

Explanation and results:

In the first kernel each thread computes each element of the matrix. For matrix size 2048 the speedup was 29.119/0.575 = 50.6.

In the second kernel, I tried tiling where each thread block computes the result for a block of the matrix. I tried different block sizes of 8,16 and 32. The best performance was obtained at block size = 16. The speedup was 29.119/0.505 = 57.6.

I also tried 4-way unrolling in the kernel2 and the speedup was 29.119/0.410 = 71.02. One of the reason for increase in speedup can be less number of loop checks and increments.

The kernel1, kernel2 and unrolled kernel2 are implemented in the functions *ATAkernel1*, *ATAkernel2* and *ATAkernel3* respectively.

4. Problem 4

Compilation and execution instructions:

- nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p4.cu -o assignment5-p4
- ./assignment5-p4

Explanation and results:

In the first kernel each thread computes each element of the matrix. For matrix size 4096 the speedup was 1.03329e+06/40345 = 25.6.

In the second kernel, I tried tiling where each thread block computes the result for a block of the matrix. I tried different block sizes of 8,16 and 32. The best performance was obtained at block size = 32. The speedup was 1.03329e+06/15003.3 = 68.8.

The speedups for matrix size 1024 without tiling and with tiling with 32 block size were: 6201.93/665.806 = 9.31 and 6201.93/247.688 = 25.03 respectively.

The speedups for matrix size 2048 without tiling and with tiling with 32 block size were: 114176/10141.1 = 11.25 and 114176/3950.65 = 28.9 respectively.

The speedups are more for bigger matrices as there is more data and computations for parallelization.

The speedups for matrix size 4096 with block size = 8 and 16 were: 1.03329e+06/18855.5 = 54.8 and 1.03329e+06/16770.4 = 61.61 respectively.

The speeups are more for larger block sizes as accesses to memory are reduced by using tiling and accesses to memory decrease on increasing block size but just we have to keep in mind that all the blocks at a given time fit in the shared memory alloted to a block.

I also tried 4-way unrolling in the kernel2 and the speedup was 1.03329e+06/14691.5 = 70.33.

One of the reason for increase in speedup can be less number of loop checks and increments.

The kernel1, kernel2 and unrolled kernel2 are implemented in the functions *kernel1*, *kernel2* and *kernel3* respectively.

5. Problem 5

Compilation and execution instructions:

- nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p5.cu -o assignment5-p5
- ./assignment5-p5

Explanation and results:

In the first and second kernel each thread computes each element of the matrix. For matrix size 512 the speedups were 1058.25/510.678 = 2.072 and 1058.25/246.877 = 4.286 respectively. In both of them thread block dimensions are kept (32,32,1). The difference is, in kernel1 i,j,k are y,x,z respectively but in kernel2 i,j,k are z,y,x respectively. The benefit is because in a warp the thread with id (0,0) will be adjacent to thread with id (0,1). So, it would be best to keep k along the x-axis for better memory coalescing. Similarly, for y and z.

In the third kernel, I tried tiling. The block size is 32 X 8 X 4. Using blocking the memory accesses are reduced but my implementation doesn't get the much benefit from tiling as each thread loads its corresponding element from the matrix and all the elements on the surface of the 3D grid have to load some elements from global memory. Also, many if conditions are present due to which this will also increase thread divergence. The speedup was 1058.25/240.712 = 4.39.

The kernel1, kernel2 and kernel3 are implemented in the functions *kernel1*, *kernel2* and *kernel3* respectively.