CSC612M Project 1

Analysis of CUDA Cooperative groups simulation

By:
Hannah Chen
Tro Jan Ong
Ronber Prado

I. Results and Discussion

A.) C++ Program

Vector Size	Execution time (us)
2^20	3458.300000
2^22	13750.333333
2^24	73930.700000

Table 1

The Dot Product C++ program was run thirty times, and the execution time increased by a factor of 3.976 to 5.3767 when the vector size increased by a factor of 4. The rate of increase was calculated using the formula $final\ execution\ time\ /\ initial\ execution\ time$.

C++ vector sizes	Rate of increase in Execution time
2^20 to 2^22	3.976
2^22 to 2^24	5.3767

Table 2: C++ Dot Program - Rate of increase in ET vs Increase in vector size

B.) Cuda + Grid Stride Loop + Prefetching + Memory Advise

Vector Size	256 Threads Execution time (us)	512 Threads Execution time (us)	1024 Threads Execution time (us)
2^20	11297.410000	17521.450000	33368.820000

2^22	33013.380000	56198.670000	118693.950000
2^24	110543.900000	211421.500000	458755.500000

Table 3

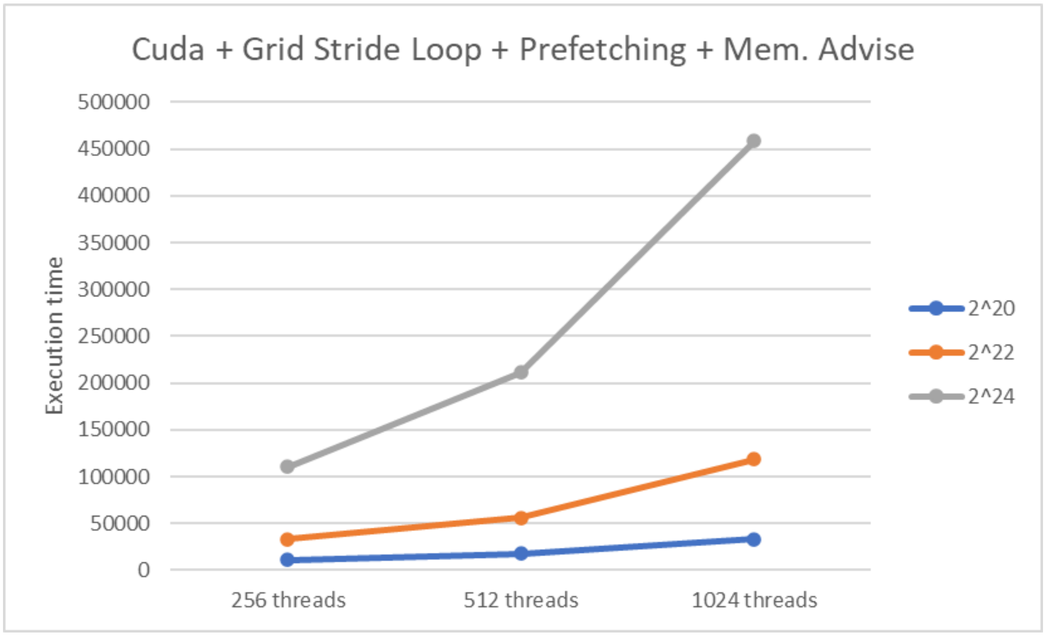

Figure 1: ET vs Threads for Cuda + Grid Stride Loop + Prefetching + Memory Advise

For this version of the Dot product program, cuda was used to execute the algorithm, along with other techniques which includes Grid Stride loop, Prefetching and Memory advise.The initial outlook for this method was that it would lead to a faster execution time compared to the C++ version, but interestingly, the results showed that this cuda version was much slower.

There are several reasons why the Cuda program may have been slower. One of the reasons could be because the addition section of the Cuda program was implemented in a serial and consecutive manner, which does not utilize the parallel execution capabilities of the threads of the GPU. Another reason could be because of the usage of additional malloc variables with large sizes, which may slow down the execution. Lastly, because of the simpler and more efficient implementation of the C++ program.

Comparing the execution time of the Cuda program shows that a smaller thread size would be more optimal for this algorithm as shown in Figure 1.

C.) Cuda + Grid Stride Loop + Prefetching + Memory Advise + Sum Reduction

Vector Size	256 Threads Execution time (us)	512 Threads Execution time (us)	1024 Threads Execution time (us)
2^20			595.982000
2^22			2303.570000
2^24			9180.599999

Table 4

D.) Cuda + Cooperative Group + Grid Stride Loop + Prefetching + Memory Advise

Vector Size	256 Threads Execution time (us)	512 Threads Execution time (us)	1024 Threads Execution time (us)
2^20			610.572000
2^22			2360.399999
2^24			9372.200000

Table 5

Unfortunately, for the Dot product program implemented using CUDA with cooperative group, grid-stride loop, prefetching, and memory advise, the group was only able to get correct outputs for the program with 1024 threads. If the number of threads is set to 256 or 512, the actual computed dot product is slightly different from the expected dot product. Possible reason for this will be errors in the implementation of the “sumBlock” function causing it to fail to handle a larger number of blocks correctly.

Based on the results gathered, when comparing with a program that only uses grid-stride loop, prefetching, and memory advise, adding a cooperative group to the implementation of the program is able to bring a significant improvement in terms of kernel execution time. This is due to their ability of being able to optimize parallel executions and minimize synchronization overheads through proper thread collaborations.

II. Conclusion

Through analyzing the Dot product program, the group can conclude that in these types of programs, it is best to implement it using cooperative groups together with grid-stride loop, prefetching, and memory advise as this type of implementation performs the best in terms of kernel execution time. On the other hand, for the program that is not using a cooperative group, it is best to apply sum reduction to further improve its execution time, especially when the array size is large.