2020

SOFT354 Coursework

CUDA IMPLEMENTATION OF MERGE SORT

EDWARD GAVIN

Introduction

For this project I have decided to implement a parallel version of the merge sort algorithm. Merge sorting is a “divide and conquer” sorting method which involves splitting up the unsorted array into “sorted” arrays of one element and then merging these sorted arrays back together in order to create sorted arrays of two elements. This process is then repeated, merging sorted arrays of increasing size until the whole array has been sorted. It currently has an average (and worst) time complexity of O(n log n), which is the best time complexity achievable by a comparison based sorting algorithm (Cormen, et al., 2009).

Merge sorting is a good candidate for parallelisation; this is because as a divide and conquer algorithm, it splits up the sorting procedure into smaller problems which can be individually worked on by another processing element, without effecting any other part of the algorithm. However, parallelising this algorithm with CUDA (Compute Unified Device Architecture) also entails breaking it down even further. This is to take advantage of features such as shared memory so read times can be minimised, as well as exposing a lot of parallelism to make as much use of the available resources as possible. If done correctly though, CUDA parallel programming will significantly reduce the time taken to sort larger arrays.

Implementation

For the serial implementation of the algorithm, a bottom up merge sort was used. Rather than recursively navigating through the array in a tree-like structure and splitting up the arrays as they are reached, a bottom up merge sort is iterative and involves merging all adjacent pairs of one element array before starting to merge the now sorted arrays of two, then four, then eight elements and so on, until all elements have been sorted.

This implementation can be found in the project MergeSort-Serial. The function mergeSort takes in the array of elements to be sorted and then allocates space for a temporary array to be used while sorting. The first loop will set the size of the arrays to be sorted (starting at 1, and increasing by size * 2 for each iteration), the second loop will then iterate over all of the array pairs to be merged and calculate the start, end and middle of the pair (where the end of the first array is mid and the start of the next is mid + 1). Using these values it will call the merge function for these two array pairs, which will iterate over the elements in each pair and add the lowest element of the two into the temp array and then move the start or mid pointer to the next element. This will continue until both pointers, point to the end of their arrays and all elements have been merged. The elements will then be copied back into the main array.

The initial attempt to parallelise this algorithm involved invoking the kernel multiple times; once for each array width to be sorted and giving each thread on the GPU (Graphics Processing Unit) one pair of arrays to merge. Nonetheless, despite the initial few kernel invocations being very fast, this implementation ended up being much slower than the serial implementation of the algorithm, especially for larger arrays, which can be seen in figure 1.

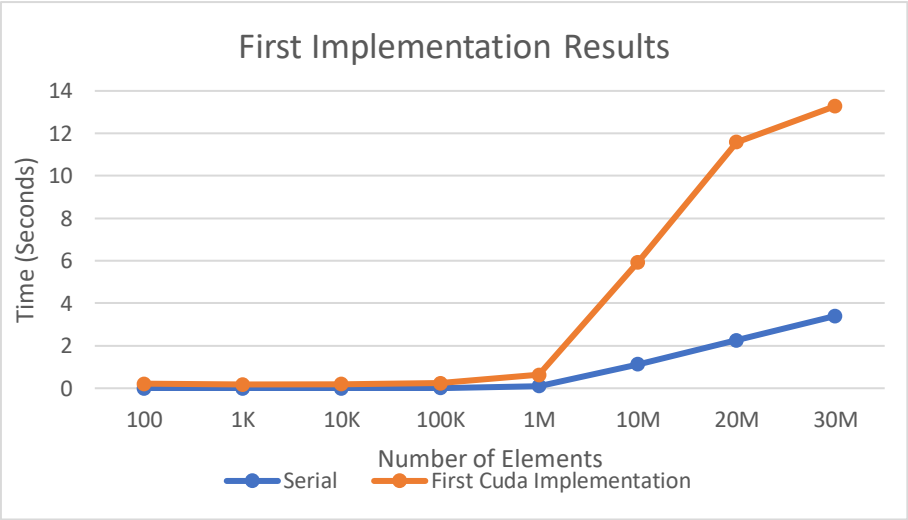

Figure 1: Graph showing the execution time of the initial CUDA implementation (Project MergeSort-Cuda)

The problems with this implementation stemmed from the parallelisation being far too coarse-grained, to the point where the final invocation of the kernel was effectively serial as it was only using one thread to do the merging. Even for small array sizes this implementation was slower due to the overhead that comes from allocating memory and copying data to and from the device. Consequently, this meant for the CUDA version of the algorithm to operate faster than the serial, it would need to expose much more fine-grained parallelism to make up for the lower speed of individual CUDA threads, when compared to the speed of a single CPU thread.

In order to achieve a better level of parallelism, Satish, Harris and Garland (2009) describe the process of using binary search to determine the “co-ranks” of a value in each array. Using this process, each thread can take an index or a group of elements in the array and use the co-ranking function to determine where the start and end of its output should be, allowing for the array to be split up into non data-dependant sections which can then be merged by the individual threads. I chose to use this co-ranking method because being able to isolate the parts of the array that can be merged independently means more threads can be assigned to do the array merging process, thereby removing one of the main issues with the initial implementation.

The implementation of this algorithm can be seen in the function ‘coRank’ in the project MergeSortCoRank and it is based on the algorithm outlined by Siebert and Traff (2013). This function takes in the integer ‘i’, which in this case represents the beginning or end of the threads output, as well as the arrays A and B and their respective lengths. It will then search in the two arrays to find the points in A and B which the thread can start or end its merging from.

The CUDA implementations which use co-ranking were adapted from Schmidt's (2017) code for a parallel merge, being modified to perform a full merge sort rather than just a merge operation. In this version, each pair of arrays is assigned a block of 1024 threads to merge them together into a sorted array. The output array will then be divided up among each of the threads, each thread will take the start and end index of its output and use the co-ranking function to determine where in each of the input arrays it should begin and end it's merge. This implementation proved to be much faster than the initial version.

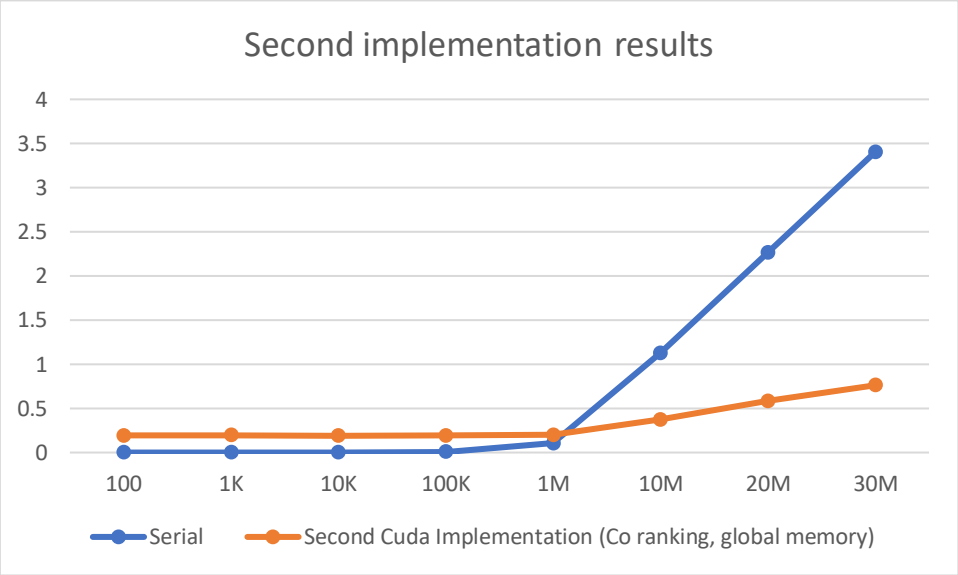

Figure 2: Graph showing the execution time of the second CUDA implementation (Project: MergeSortCoRank, function: mergeSortCuda)

This implementation can be seen within the project MergeSortCoRank and is called from the function mergeSortCuda. This function will allocate all the required memory on the device and copy the data to/from it, as well as deciding how many threads and blocks would be launched. Once launched, the kernel (mergeCuda) will work out the start, middle and end of the array using the index of the thread block to decide which thread it should take. It will then call the device function coRankMerge, which will split up the output array by the number of threads in the block and use the co ranking function to determine the start and end points for each thread block in each of the two input arrays, prior to calling the merge function so each thread can perform its merging operation on its array section.

There were still problems with this implementation for instance, it was limited by the slow speed of global memory access in CUDA with Nsight reporting that in one kernel launch, 74% of the reasons for stalling were due to warps waiting to get data from global memory. As each thread had to perform two binary searches in the array before they could start merging, further speedup could easily be achieved by removing this apparent limitation.

The final implementation, located in the MergeSortCoRank project and called through the cudaMergeSortTiled function, achieved further speedup by adding in the

use of shared memory to the implementation. For this, the output array was split up into blocks of 4096 elements, which were small enough to fit into shared memory on the GPU. Each of these blocks was then assigned a block of 1024 threads to do the merging operation for them.

The process would start with the output array being split up among the blocks within the global kernel function mergeCudaTiled using the blocks index and the previously calculated number of blocks per array. This would then call the device function coRankTiledMerge which would then calculate the blocks output section and its block level co ranks for the arrays A and B to determine where it is merging from in each of the input arrays. Once this had been calculated all the threads would then collaborate to load the relevant elements in to shared memory. After this, it would then be split up among the threads in the block, which would compute their co ranks and merge in the same way as the global memory version.

Overall, I believe this implementation makes good use of the available features of CUDA. It splits up the merging procedure among all the threads within a block and uses their indexes to decide which parts of the array that they should be merging. It makes use of shared memory to reduce the time needed to read the data, for example, Nsight now reports for one of the kernel launches, memory dependencies are only 4% of the reasons for stalling and thread synchronisation is now the largest reason for stalling at 74%. It also takes advantage of coalesced memory access, meaning that global memory accesses from different threads could be combined, leading to lower overall read times when getting the data to shared memory.

Evaluation

Time measurements for this application will take all the overheads into account, the timer measures the time taken for the function mergeSortCudaTiled (same applies for the previous two implementations that were timed and their relevant functions) to complete. This function includes all the CUDA functionality which would influence the runtime of the application, such as memory allocations, copying data to the device, calculating grid and block sizes and launching the kernel. To ensure all results are consistent and that anomalies do not affect the final statistics, the application will be run ten times for each number of values in the array and the average of those times will be plotted on the graph, a table featuring all of the results will be available in Appendix A. The values within each array will also all be generated by the same function and code. This is to make sure that all implementations sort arrays with values that are roughly the same distribution.

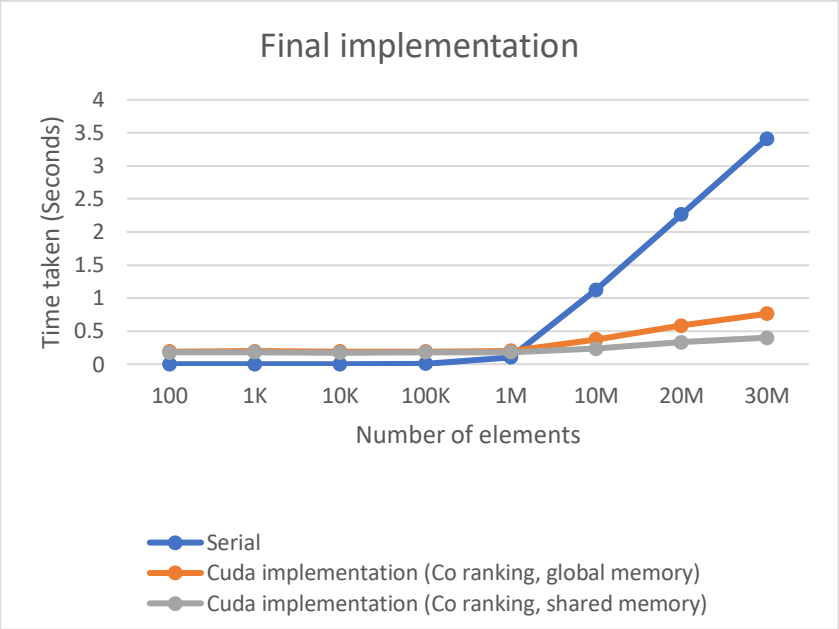

Figure 3: Graph showing the execution time of the tiled version of the algorithm, compared to the serial and global co ranked version. (Project: MergeSortCoRank, function: mergeSortCudaTiled)

Figure 3 indicates the CUDA implementation of merge sort is now much faster than the serial one for large array sizes however, due to all the overheads which are involved in running a CUDA application, it is still slower than the serial implementation for smaller array sizes. Despite this, it is worth considering that sorting is usually a smaller part of a much larger problem, so it may still be worth it to do the sorting on the GPU for smaller array sizes if that data is being used on the GPU for another purpose once it has been sorted.

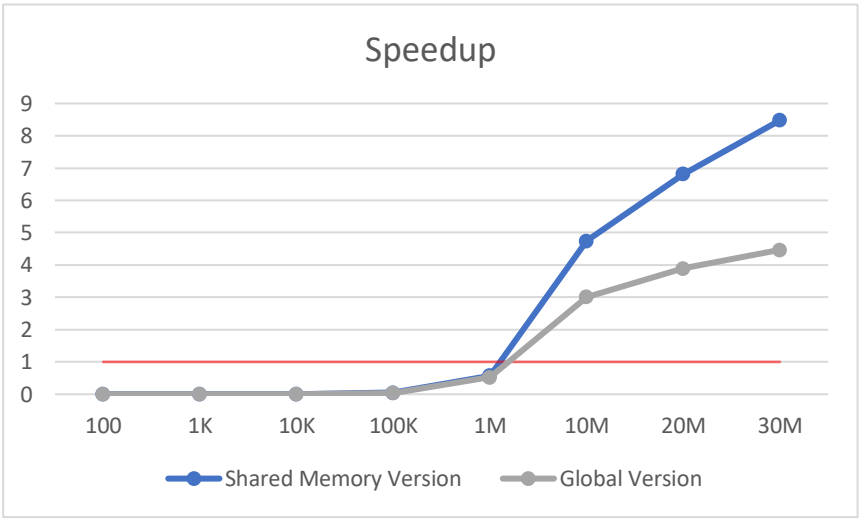

Figure 4: Graph showing the speedup of the tiled and global implementations compared over the serial one

As can be seen in Figure 4, once the CUDA implementation becomes faster than the serial version, the amount of speedup gained through parallelism increases

alongside the number of elements. This is because the number of processing elements increases alongside the number of elements as more tiles can be added to process each part of the array, meaning the amount of data to be processed per thread stays roughly the same. This also shows that the CUDA version is very scalable and should continue to provide better sorting speeds than the serial version as the array sizes get even larger.

The efficiency of the solution is very low however, sorting an array of ten million elements uses an average processing element count of 5916506 (due to the number of processing elements changing each time the kernel is launched, an average count has been used). This means there is an efficiency of 0.0000008 and a cost of 1405598 for ten million elements and the efficiency for processing thirty million elements is 0.00000002 and it has a cost of 170133473. As a comparison, the efficiency of the co ranked global memory solution for 10 million elements is 0.0000006 with a cost of 1718789. Showing that although the shared memory version uses far more processing elements, the speedup gained from this makes it more efficient than using the previous global memory version and overall it still achieves great speedup over the serial implementation.

From this it can be seen while there is a big increase in speedup as the size of the problem grows, the overall efficiency of the solution does go down as more tiles are needed for each array, though the efficiency is still greater than in other versions of the CUDA implementation. Especially as the amount of speedup gained in the global memory version plateaus much faster due to the increased amount of work per thread as the array sizes increase. This increase in speedup as the array sizes increase occurs because the work for the single thread in the serial implementation increases with the number of elements, which does not occur with the tiled parallel implementation, though hardware limitations, such as the fixed number of SM's will still mean that the amount of speedup will eventually taper off.

Though taking advantage of the thousands of threads the CUDA architecture offers will generally lead to lower efficiencies than a parallel implementation on another architectures, this low efficiency could be fixed by reducing the number of threads within each block and having each thread do more of the work. This could also reduce the speed of the application as well due to hardware limitations of CUDA. This is because currently there is 96KB of shared memory per Streaming Multiprocessor (SM), with a limit of 48KB per thread block. Consequently, if the size of the thread block were to be reduced to 512 from 1024 there would be only 48 active warps per SM instead of 64 as the amount of shared memory in use per block is 32KB, which further indicates, although one more block could fit on the SM, there would be less warps active at a time and less data would be able to be processed each cycle as a result.

Increasing the size of the section which each thread block is responsible for is another way the efficiency could be improved. If each thread block were to instead process 16384 elements and it was loaded into and sorted in shared memory in a few phases, this could increase the efficiency of the algorithm by reducing the number of processing elements in use down to a quarter of what it was.

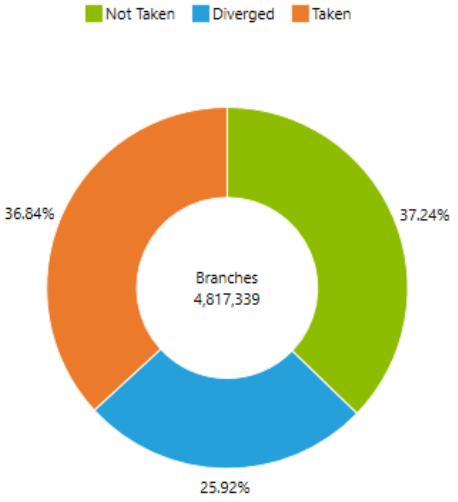

Figure 5: Nsight graph showing the branch divergence statistics.

Another issue influencing the processing time of this implementation are the problems with warp divergence in the application. As can be seen in Figure 5., Nsight currently reports 25.92% of the branch instructions resulted in a different outcome for threads within the same warp. This is a problem as it means not all the instructions can be executed at once and the GPU is now going to take more cycles to process the datasets although; due to the comparative nature of merge sort, warp divergence cannot be completely avoided, though there may be a way to reduce it. As found by Satish, et al. (2009) a non-comparative sorting algorithm, such as radix sort, would be able to alleviate some of these issues and provide better performance than even parallelised CPU implementations of sorting algorithms.

The serial implementation has an additional space requirement of O(n) as a temporary array is needed to store the elements while they are being sorted. The first two CUDA implementations also had this additional space requirement, however, the final implementation has a slightly higher additional space requirement due to the need to allocate space in shared memory for each tile that needs to be merged. However, with some modifications, this solution could use much less additional memory, the use of shared memory as a temporary storage area for the current block should allow for the removal of the temporary array. Doing this would lower the additional space requirement for the solution to below O(n) as only a limited number of blocks can be processed at once, meaning the allocated shared memory would effectively be re used during the sorting process and no additional space would be required in global memory.

Conclusion

In conclusion, despite the low efficiency of the solution the CUDA implementation of merge sort has performed very well. It has achieved a large amount of speedup

over the serial implementation and the processing time goes up much slower as well, meaning that the amount of speedup should keep increasing as the number of elements increases. Much of this speedup has come from exposing lots of fine-grained parallelism to take advantage of CUDA's manycore architecture, particularly breaking down the merging operation so that it can be split up across multiple threads and done in parallel. Overall, GPU sorting using CUDA is a good choice for large array sizes, and even could be useful for smaller arrays if the data isn't just being copied to the GPU to be sorted and will have further processing after it.

Successes notwithstanding, there are still changeable elements which could possibly increase the speed or efficiency for instance; as mentioned above, the tiling system could be modified so that each thread block is given more of the array to merge which is then broken up into smaller chunks and loaded in to shared memory by the thread block, this may not increase speed however, it could increase the efficiency of the system.

Another element which could be included to further improve the speed and efficiency of the application, would be to make use of the extern keyword for shared memory, so that the size of shared memory can be set when the kernel is launched rather than at compile time. This could help because it would mean blocks are not allocating any more shared memory than they require, it could allow for more blocks per SM in earlier phases when the block contains less than 1024 threads which could result in better occupancy overall. Another approach to improved shared memory use and have fewer shared memory allocations overall would be to alter the logic within the kernel and how blocks are allocated, so that when the array widths are smaller than the size of the shared memory tile, multiple array pairs can be put into shared memory so that the allocated space can be used much more efficiently.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C., 2009. *Introduction to Algorithms.* 3rd ed. Cambridge, MA: The MIT Press.

Satish, N., Harris, M. & Garland, M., 2009. Designing Efficient Sorting Algorithms For Manycore GPUs. *2009 IEEE International Symposium on Parallel & Distributed Processing,* pp. 1-10.

Schmidt, T., 2017. *CSE 599 I Accelerated Computing - Programming GPUs. Parallel Patterns: Merge.* PowerPoint Presentation
[Online]
Available at: https://tschmidt23.github.io/cse599i/
[Accessed 18 December 2019].

Siebert, C. & Träff, J. L., 2013. Perfectly load-balanced, optimal, stable, parallel merge. Pre-print. arXiv:1303.4312, 2013

Appendices.

Appendix A – Full results of timing

Serial Times

Elements	1	2	3	4	5	6	7	8	9	10	Average
100	0	0	0	0	0	0	0	0	0	0	0
1K	0	0	0	0	0	0	0	0	0	0	0
10K	0	0	0	0	0	0	0	0	0	0	0
100K	0.009974	0.009973	0	0.015677	0	0.015568	0	0	0.015621	0.015621	0.0082434
1M	0.109346	0.10935	0.109402	0.109298	0.093778	0.10935	0.093779	0.09378	0.10841	0.109403	0.1045896
10M	1.143934	1.155972	1.097749	1.12756	1.109109	1.122476	1.124732	1.168062	1.09785	1.127484	1.1274928
20M	2.319791	2.279551	2.281805	2.257532	2.249519	2.26509	2.273102	2.249417	2.261142	2.217769	2.2654718
30M	3.50462	3.426947	3.378035	3.405447	3.405443	3.401054	3.372329	3.401996	3.389879	3.358581	3.4044331

First Implementation Times

Elements	1	2	3	4	5	6	7	8	9	10	Average
100	0.203128	0.157273	0.171826	0.213317	0.22309	0.179542	0.176173	0.231588	0.218753	0.234319	0.2009009
1K	0.218696	0.171886	0.184253	0.218713	0.174762	0.154181	0.156212	0.156262	0.156262	0.18746	0.1778687
10K	0.218393	0.19449	0.182939	0.163746	0.173179	0.183136	0.191672	0.180648	0.192426	0.190653	0.1871282
100K	0.27701	0.246523	0.234318	0.231019	0.218748	0.222005	0.238586	0.219597	0.218749	0.25101	0.2357565
1M	0.657686	0.633203	0.637423	0.624901	0.625795	0.634743	0.624907	0.609229	0.609281	0.60789	0.6265058
10M	5.980914	5.920472	5.904856	5.873207	5.931548	5.920018	5.935688	5.96265	6.049885	5.861756	5.9340994
20M	11.7943	11.58334	11.54498	11.53836	11.56384	11.62439	11.58779	11.55944	11.573864	11.53724	11.590754
30M	13.31632	13.33945	13.22369	13.22695	13.23969	13.25262	13.30969	13.31775	13.400027	13.21704	13.284324

Second implementation times (Co ranked – Global)

Elements	1	2	3	4	5	6	7	8	9	10	Average
100	0.255317	0.18586	0.18886	0.183406	0.194206	0.181459	0.186844	0.189987	0.17879	0.190005	0.1934734
1K	0.260148	0.183418	0.203212	0.183754	0.193871	0.182985	0.18044	0.203459	0.181386	0.183376	0.1956049
10K	0.266275	0.196239	0.182827	0.177283	0.17794	0.176689	0.174749	0.176841	0.177035	0.187508	0.1893386
100K	0.174106	0.177973	0.263166	0.182366	0.184576	0.190031	0.179511	0.186048	0.183502	0.187463	0.1908742
1M	0.264744	0.185498	0.203452	0.194327	0.193046	0.198582	0.189244	0.191319	0.192704	0.198466	0.2011382
10M	0.437957	0.375083	0.374064	0.366079	0.37112	0.369305	0.362272	0.353737	0.364366	0.375814	0.3749797
20M	0.641696	0.577401	0.567891	0.575884	0.59452	0.570636	0.564348	0.575858	0.58108	0.575421	0.5824735
30M	0.75255	0.757897	0.752031	0.748356	0.76303	0.758735	0.754075	0.754017	0.76387	0.8152	0.7619761

Final implementation times (Co ranked – Shared)

Elements	1	2	3	4	5	6	7	8	9	10	Average
100	0.252567	0.160257	0.156754	0.166707	0.185172	0.170924	0.173104	0.165952	0.168023	0.182787	0.1782247
1K	0.261256	0.172579	0.186521	0.179758	0.181537	0.180618	0.159982	0.184416	0.158151	0.178666	0.1843484
10K	0.178177	0.184079	0.177766	0.191227	0.173184	0.172113	0.1623	0.15569	0.163185	0.1692	0.1726921
100K	0.259641	0.182873	0.16651	0.158953	0.161674	0.181827	0.178065	0.159653	0.175309	0.19691	0.1821415
1M	0.18995	0.180521	0.190446	0.195079	0.184263	0.184318	0.186478	0.165865	0.178162	0.162176	0.1817258
10M	0.254544	0.246518	0.234074	0.227147	0.228019	0.257929	0.232067	0.235401	0.233026	0.226999	0.2375724
20M	0.330883	0.324431	0.32236	0.328459	0.354805	0.341664	0.335806	0.331296	0.321585	0.333578	0.3324867
30M	0.423072	0.387031	0.399297	0.393207	0.404616	0.40004	0.407648	0.404141	0.40643	0.390372	0.4015854