Parallel and Fast Algorithms for Finding Top-K Frequent Elements

Mostafa Hosseini
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
mostafahosseini@cmail.carleton.ca

December 7, 2022

Abstract

Estimating frequency of elements over data stream when we have large amount of data with high-rate stream is challenging, especially when we want to estimate the high frequency items precisely considering that we have small working space, and the stream processing should be fast and accurate. Using Count-Min Sketch (CMS) algorithm as a famous approximate algorithm can be beneficial for this issue. However, we are looking for an improved method that can increase overall throughput and the accuracy of frequency estimation in the real world. In this article a new method will be explained that is based on CMS and benefits from a pre-filtering stage for saving high frequency items and data can be exchanged between the filter and the sketch. Also, using local heavy hitter in each cell of sketch helps us not to store the frequency of all items and it will lead to higher speed. We will show that by using SIMD intrinsic in CPU and multi-core parallelism, we can improve the performance of the proposed algorithm significantly and this method can overcome other state of the art methods.

1 Introduction

Since dealing with large amount of data stream has become one of the hot topics recently, using parallel algorithms is very helpful to process a massive data stream, specially when we are facing a time limit. One of the important issue in data stream processing is finding the most frequent items. This topic has many applications in the real world such as Internet traffic analysis for anomaly detection to find heaviest users and high-traffic IPs [4].

Finding proper algorithms for solving this issue is very important considering two main limitations. First is processing time when our algorithm must be processed very quickly in a real time aspect, second is the space limit when we just have a short memory space for working with the algorithm. Therefore, we should consider a tradeoff between space, accuracy, and efficiency.

The main problems that we seek to improve are speed of frequent items estimation with respect to the working space limitation and accuracy of algorithm. For addressing this issue, Count-Min Sketch (CMS) [6] has been selected as a famous approximate algorithm and recently, several researches have been done in order to improve the speed and accuracy of this algorithm. A sketch is a 2-dimensional array that has d hash functions and w cells. Each hash function maps an element uniformly at random to a cell. Also, we need a memory

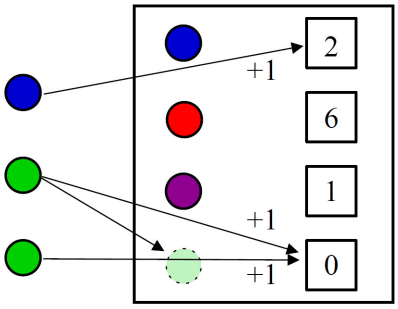

Figure 1: An example of counter-based method

space for storing top-K frequent items. Therefore, there are several ways for implementing parallel processing to improve the performance of the algorithm such as running each sketch in different thread or parallelizing counting the elements and finding the top frequent items at the same time.

This project works on state-of-the-art algorithms based on CMS and uses parallel processing methods for boosting processing speed. First, we review the main related works as a literature review. Next, we explained proposed algorithm in detail, and we will show how parallel processing is used in this project to increase the speed. After that, we evaluate the performance and accuracy of proposed algorithm, and we compare suggested algorithm with two other methods.

2 Literature Review

There are several methods for data stream summarization, but the most famous are sketch-based and counter-based methods. Counter-based data structures such as Frequent [10], Lossy Counting [13] and Space Saving [3] are designed for finding the top-K frequent items and they maintain a set of counters associated with a subset of words from the data stream it has traversed. Counter-based algorithms track a subset of items from the inputs, and monitor counts associated with these items. For each new arrival, the algorithm decides whether to store this item or not, and if so, what count to associate with it. A simple counter-based algorithm has been shown in Figure 1 . When one blue item comes to the algorithm, because the blue one already was in the counter list, we just increment the counter by 1. But when green item comes to the algorithm, we check the counter list to find a counter equal to zero. If so, we add that item to the related counter. In other scenario, if all counters are full, we will never add this item to the counter and we decrement all counters by one.

Sketches can keep approximate counts for all items , while counter-based approaches maintain approximate counts only for the frequent items. Sketches can support top-K queries with an additional heap or a hierarchical data structure [9]. However, counter-based approaches have slower update time than sketches for the top-K estimation [5], as they do not count the frequency of all items.

Sketches have been widely applied to item frequency estimation in data streams. The most widely used sketch is the Count-Min Sketch [6]. CMS will be discussed in this article as a baseline model. Other famous sketch-based algorithms include Count-sketch [11] and CM-CU sketch [1]. In contrast to the CMS, the Count-Sketch does not require the strict turnstile

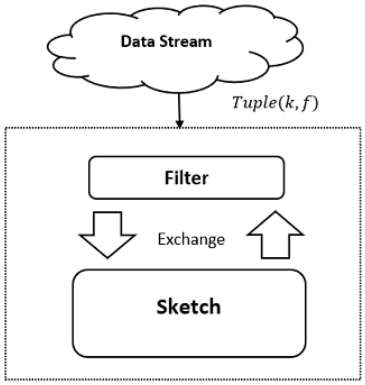

Figure 2: The main structure of Asketch

model, where the final histogram may have positive or negative entries. Also, Count-Sketch suffers from both overestimation and underestimation, Whereas CMS and CM-CU sketch suffer from overestimation only. The CM-CU sketch achieves higher accuracy than CMS and the only difference is that CM-CU only increments the smallest one(s) among all hashed counters. CM-CU does not support deletions, and consequently it has not received as wide acceptance in practice as the CMS.

There are several methods for improving throughput of sketches such as Holistic UDAFs [8] and Frequency-Aware Counting [7] and gSketch [15]. However, they usually use additional data structures and increase the overall storage requirement in compare with CMS, which is often significant with respect to small-space sketch data structures.

Augmented sketch framework (Askech) [14] improves both accuracy and throughput considering it has same space as CMS. As it is shown in Figure 2, Asketch is structurally similar to CMS, but it has a filter for saving the most frequent items (e.g., 32 hot items) and data can be exchanged between the sketch and the filter. The filter size is very smaller than sketch size, so it has higher speed than sketch for saving and updating data. The filter has two variables named new count and old count. New count keeps last frequency of each item and old count keeps the frequency of each item when it was in sketch. First of all, the algorithm checks if the item is in the filter or not. If it is in the filter, it just updates new count and if it is not in filter and filter is not full, it adds the item into the filter and set old count to zero and new count to its item frequency. If filter is full, it updates the sketch and related buckets for this item and send an item to the filter if the sketch has an item with higher frequency than items in the filter. The Filter can run in one core and the sketch can run in another core to improve the speed of finding hot items.

Topkapi [2] uses the advantage of both counter-based and sketch-based methods. CMS has an excellent update time, but it suffers from reducibility which is needed for exploiting available massive data parallelism. On the other side, the popular Frequent algorithm (FA) leads to reducible summaries, but its update costs are significant. Topkapi proposes a fast and parallel algorithm for finding top-K frequent items, which gives the best of both methods. It is reducible and has fast update time similar to CMS. As it is shown in Figure 3, this algorithm inserts a counter to each cell of the sketch in order to store the frequency of the most frequent item. Therefore, it keeps a local heavy hitter for each cell. It has high accuracy, but because of having additional structure for saving the local heavy hitter, it needs some improvenemts for this part. For speed up, the authors used multi-core processing

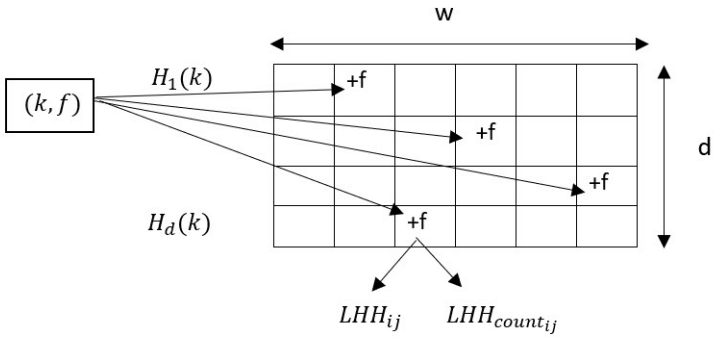

Figure 3: The main structure of Topkapi

and the result of each thread will be merged, after all threads finished their process.

Cold Filter [16] proposes a novel meta-framework by filtering small flows at the first stage and running additional algorithms separately on the residual large flows. Cold Filter uses a two-layer filter to prevent mouse flows from entering some data structures (e.g., Space-Saving and the CMS). Although Cold Filter shares some similarity with Asketch it uses entirely different algorithms, and it has complex data structures and cannot be easily implemented in the data-plane.

A recent state-of-the-art method presented the Diamond Sketch [17], which dynamically assigns to each flow a proper number of atom sketches in order to reduce memory footprint. Though it greatly reduces the relative error, it is designed specifically for flows with non-uniform distribution and this algorithm is incapable of working with arbitrary distribution.

All mentioned methods have fixed counter size for recording the frequency of items and maybe in very large data stream, some hot items have more counts than the size of counter. Pyramid sketch [18] framework can automatically enlarge the size of counters according to the current frequency of the incoming item, and has good accuracy and speed compared with other methods. However, hot items require quite a few memory accesses, and thus insertion speed of the Pyramid sketch in the worst case is poor.

3 Problem Statement

Due to the advancements on new technologies such as Internet of things, cloud computing and social networking, we are dealing with a huge amount of data. As a consequence, many algorithms are being designed for solving the problems related to sorting and processing massive data stream.

One important problem in this feild is finding the most frequent elements in a data stream. Because it has many applications in practice. For example, one of the most popular passwords attacks is dictionary attack. The attacker attempts to guess password of users by trying all the words in some kind of a list. This kind of attack is especially dangerous if the attacker has information about the most popular passwords used in the system. Finding the most popular passwords will helps us to prevent users to choose these kinds of passwords and protect the security of our users. Another usage is in an online shopping market when we want to know what products are best seller in order to let the shoppers know for making

better decision. There are many other applications related to this problem in the real world. Therefore, finding efficient algorithm and new data structure that can work on high performance hardware as well as small IoT devices with limited resources is very critical considering limited memory space and processing time as two main issues that we should address.

One solution can be using an exact algorithm. For example, in this solution because we are dealing with a text-based data, we can compute all the frequencies of items using a standard wordcount. Then sort the words based on their frequencies. But this algorithm has high computation time. If we consider M as the number of all elements in a data stream, the computation time will be O(MlogM) and we require O(M) memory space. Parallelizing this algorithm is easy, but it is still not a proper solution.

Based on description given, approximate algorithm [11] can be a good choice. Because the goal of the approximate algorithm is to come as close as possible to the optimal solution in polynomial time. Therefore, in approximate algorithm the main focus is to have high performance method that can reach to result close to the actual result with high accuracy. It is very important that the run time complexity and required memory space in the approximate algorithm is less than exact algorithm. Nowadays, skethced-based algorithms are very famous for addresing this problem and many researches have been done based on this approach [4, 6]. In the next part we will see how an state of the art method based on this approach can improve the performance of finding top-K frequent items.

4 Proposed Solution

The base structure of this solution is based on sketch. The most famous sketch-based algorithm for finding frequency of each item is Count Min Sketch(CMS) [6]. Assume we have a data stream which has N tuples. Each tuple has a data item key for hashing named k and a value for each item named f. Therefore, the i-th tuple will be (k_i, f_i).

The actual sketch data structure is a two-dimensional array of w columns and d rows. The parameters d and w are fixed when the sketch is created and are related to the computation time and space and the probability of error when the sketch is queried for a frequency estimation. Associated with each of the d rows is a separate hash function and the hash functions must be pairwise independent.

In CMS, we use sketch structure. Therefore, we have d hash functions and w buckets. For each row, there is a hash function, and each hash function maps the elements uniformly at random to one of the buckets in columns. CMS structure has been shown in Figure 4. Because the size of buckets is much less than the size of all unique elements in data stream, we have collision in each cell and the frequency of more than one items will be stored in each bucket. Therefore, for finding frequency of an item, we find the related bucket of each hash function based on the item key and after that we compute minimum value among all buckets related to the item key. If we name the true value c_i , the estimated value will be at least equal to c_i and we have overestimation. Considering $w = \frac{e}{\epsilon}$, the estimated value will be at most $(c_i + \frac{e}{w}N)$ with probability at least $(1 - e^{-d})$. In other words, with the probability of at most e^{-d} , the expected error is at most $\frac{e}{w}N$. As a result, the number of buckets and the number of hash functions are two main parameters that determine the accuracy of CMS.

CMS can estimate the frequency of all items, whereas we need to estimate the frequency of top-K frequent items in adition to frequency estimation for all items. Therefore, we need

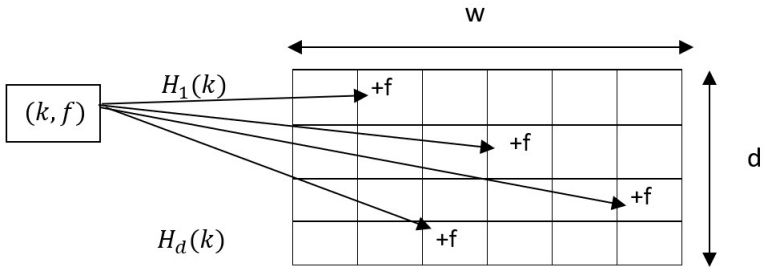

Figure 4: The main structure of CMS

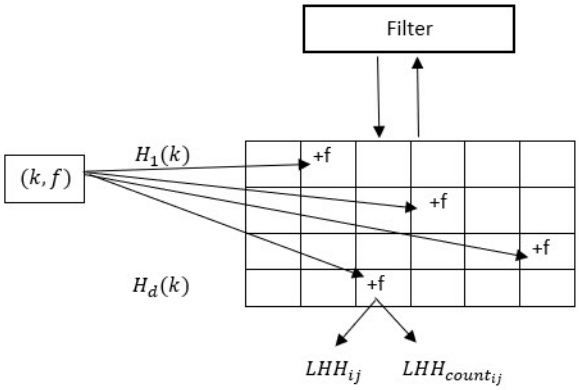

Figure 5: The main structure of proposed method

an additional data structure or new method for addressing this issue. As you can see in Figure 5 we benefit from the idea of Asketch [14] and Topkapi [2] as the two state of the art methods. Therefore, the sketch part for proposed method is based on Topkapi and the filter part comes from Asketch.

4.1 Algorithm

For the filter part, based on algorithm 1, we have 2 variables named new_count and old_count in the filter. New_count keeps last frequency of item k and old_ count keeps the frequency of item k when it was in sketch. When item k comes, first of all, we check if item k is in the filter or not. If it is in the filter, we just update new_count. If it is not in filter and filter is not full, we add it to the filter and set old_count to zero and new_count to f. If filter is full, we update the sketch and related buckets for this item. After that, we calculate estimated frequency of this item in the sketch and check if it is bigger than the minimum frequency value in the filter or not. If it is bigger, we exchange this item with the least frequent item in the filter. We set new_count and old_count of this item to estimated frequency value of related item from the sketch. Also, we update the related buckets of that least frequent item in the sketch by adding difference between new_count and old_count to the count of related buckets.

Because we keep high frequency items in the filter and each cell of filter just has one item, the frequency estimation of high frequency items has more accuracy. On the other

hand, we reduce the processing cost of applying multiple hash functions in the sketch for high frequency items and this leads to improving the throughput. It should be mentioned that the algorithm just has one exchange in each insertion phase.

Algorithm 1 Stream processing algorithm in the filter

Ensure: *insert tuple*(k, f) *into the algorithm*
lookup k *in the Filter*
if *item found* **then**
    $new\_count[k] \leftarrow new\_count[k] + f$
else if *filter not full* **then**
    $new\_count[k] \leftarrow f$
    $old\_count[k] \leftarrow 0$
else
    update sketch with (k, f)
    if $(estimated\ freq[k] > min\ freq[filter])$ **then**
        find minimmum freq item k_i *in Filter*
        $\Delta = new\_count[k_i] - old\_count[k_i]$
        update sketch with (k_i, Δ)
    end if
    add k *to Filter*
    $new\_count[k] \leftarrow estimated\ freq[k]$
    $old\_count[k] \leftarrow estimated\ freq[k]$
end if

For the sketch part, the structure is like CMS, but in each cell we have 2 parameters. One is for saving the local heavy hitter. Thus, LHH keep the item in the backet that has highest frequent item among all items in that bucket. Another parameter LHHcount is for saving the frequency count of related local heavy hitter. As you can see in algorithm 2 whenever an item k with frequenct f comes to the sketch, it will be hashed to the related bucket. If this item is the local heavy hitter, the LHHcount will be increased by f. If not, the count of related LHH will be reduced by f. After that, If the bucket has the count equal to zero, the LHH will be replaced by k and the LHHcount will be 1. This help us to reduce the collision of items in the same bucket.

4.2 Impact of filter on throughput and accuracy

If N_1 frequency counts are processed in the filter and N is total counts of data stream, then $N_2 = N - N_1$ counts will be processed by sketch. Note that sketch keeps the items that are in the filter too, but it does not update them until they are in the filter. Therefore, the update time of proposed method will be $t_f + \frac{N_2}{N} t_s + t_{ex}$, where t_f is update time of filter part, t_{ex} is the exchange time between filter and sketch and t_s is update time of sketch part in CMS. We can ignore t_{ex} due to very low exchange rate in compared with total data stream size.

Manerikar et al. [12] proved that skewness will affect finding high frequent items. Streams with a high skew have a few items which occur very frequently and streams with low skew have a more uniform distribution of items, and it is more difficult for the algorithm to distinguish the frequent items. Hence, if data stream has high skew the exchange rate will dramatically decrease. As it is mentioned before, size of filter is very smaller than sketch

Algorithm 2 Stream processing algorithm in the sketch

Ensure: *insert tuple*(k, f) *into the algorithm*
for $i \in 1, 2, ..., d$ **do**
    calculate $h_i(k)$
    if $(C[i][h_i(k)].LHH == k)$ **then**
        $C[i][h_i(k)].LHHcount \leftarrow C[i][h_i(k)].LHHcount + f$
    else
        $C[i][h_i(k)].LHHcount \leftarrow C[i][h_i(k)].LHHcount - f$
        if $(C[i][h_i(k)].LHHcount == 0)$ **then**
            $C[i][h_i(k)].LHH \leftarrow k$
            $C[i][h_i(k)].LHHcoun \leftarrow f$
        end if
    end if
end for

and by considering good hardware implementation such as using heap structure, it leads to $t_f << t_s$. $\frac{N_2}{N}$ is selectivity factor and it calculates what proportion of all counts will be implemented in sketch. This factor is directly related to popularity of top-K frequent items. It means if we have data skewness and proportion of top-K frequent items is significant in compared with other items, by considering that filter has enough space for saving K items, the selectivity factor for sketch will be low. So, in the most cases processing time for proposed method is lower than CMS and throughput of this method is higher than CMS and other methods.

On the other hand, Because we store high frequency items in the filter and we count just the frequency of related item instead of using hash functions and adding count for several items, proposed method increases accuracy of estimated frequency of high frequent items and it will decrease the collision of stored items in the sketch with high frequent items. As a result, the misclassifying of low frequency items as top frequent items will reduce. Based on previous information, the expected error for frequency estimation query in CMS is $(\frac{e}{w})N$. Hence, the expected error in our algorithm if we just have sketch is $\frac{e}{(w-\frac{s_f}{d})}N$. But we have frequency estimation query in the filter and sketch so expected error for sketch will be $\frac{e}{(w-\frac{s_f}{d})}N_2$. By assuming that most of queries will happen for high frequency items, selectivity factor determines when frequency count happens in the sketch. As a result, the expected error is $\frac{e}{(w-\frac{s_f}{d})}N_2(\frac{N_2}{N})$. Due to the fact that N_2 is very smaller than N in skewed data stream, $N_2(\frac{N_2}{N}) << N$ and we can say frequency estimation expected error for this method is smaller than CMS in most cases.

4.3 Parallel processing

In the filter, we have two issues that increase the computation time. First one is finding an item in order to update items and the second one is finding minimum item for exchange process. For addressing these issues, we used SIMD intrinsic which boost the speed of these two processes. Unlike scalar processors, which process data individually, modern vector processors process one-dimensional arrays of data. Therefore, as it is shown in the Figure 6 , we can store some items in a vector and do the same operations at the same time on different elements in vectors, which means we can implement SIMD via vectorization in

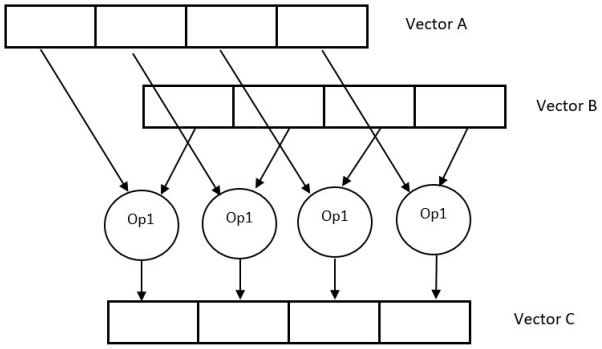

Figure 6: SIMD Vectorization

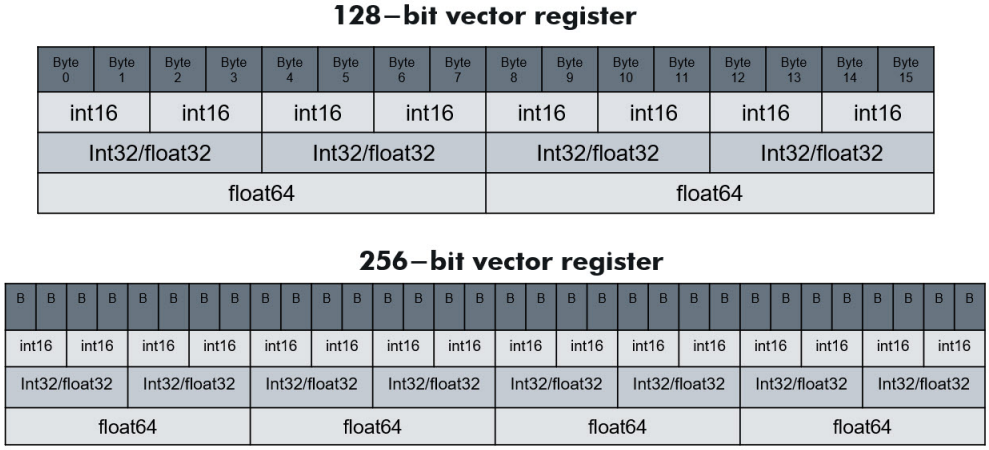

Figure 7: 128-bit vector and 256-bit vector

CPU. This implementation can be done by intrinsincs which are some instruction sets that many new CPUs support it. There are different instruction sets for implementing SIMD in CPU and we should check which one the CPU supports. The most important of them are SSE which works with 128-bit vector, AVX which works with 256-bit vector and AVX-512 that works with 512-bit vectors. Each CPU supports one or some of them. For example, as you can see in Figure 7 in 128-bit vector, in each vector we can store sixteen 8-bit integer values, or eight 16-bit integer values, or four 32-bit integer values, or even two 64-bit float values. Also, in the 256-bit vector we can store thirty two 8-bit integer values or sixteen 16-bit integer values and so on.

The implementation code for finding an item in the filter with 128-bit vector has been illustrated in Figure 8. In this project we have text-based data stream and it is not possible to store the text into the vector. Therefore, we use a hash function to generate an index for each item and for decreasing the collision the hash function can generate different hash values much more than the size of the filter. Because we use 16-bit integer value, we can store 8 items in each vector. First of all, we broadcast the item that we want to search to all 8 elements in a vector named item. After that in each iteration we compare item vector with 4 vectors including 32 elements. If there is an element in one vector that is matched to that item, we return the index of that element as the result.

```
int bucket_num = (*_filter)._filtersize / 32 ;
for (int i = 0; i < bucket_num; i++)
{
     const __m128i item = _mm_set1_epi16((int)key);

     __m128i *keys = (__m128i *)((*_filter).fhash+ (i << 5));

     __m128i a_comp = _mm_cmpeq_epi16(item, keys[0]);
     __m128i b_comp = _mm_cmpeq_epi16(item, keys[1]);
     __m128i c_comp = _mm_cmpeq_epi16(item, keys[2]);
     __m128i d_comp = _mm_cmpeq_epi16(item, keys[3]);

     a_comp = _mm_packs_epi16(a_comp, b_comp);
     c_comp = _mm_packs_epi16(c_comp, d_comp);

     uint16_t matched1 = _mm_movemask_epi8(a_comp);
     uint16_t matched2 = _mm_movemask_epi8(c_comp);
     uint32_t matched =  (static_cast<uint32_t>(matched2) << 16)
                             + static_cast<uint32_t>(matched1);

     if(matched != 0)
     {
         int matched_index = __builtin_ctz(matched)+ (i << 5);
         (*_filter).filter_newcount[matched_index] += 1 ;
         return;
     }
 }
```

Figure 8: Searching an item in the filter based on SIMD 128-bit vector

For finding the item with minimum frequency as you can see in figure 9, we use 32-bit integer value to cover the frequency count of all items. Therefore, each 128-bit vector can keep 4 elements. In each iteration we compare 2 vectors including 8 elements and we store the items with minimum frequency to a min vector. After meeting all items in the filter, we can find the min frequent item among the 4 elements in the final min vector.

We impelement multi-core parallelism in order to improve the speed of data processing. As it is shown in Figure 10, each thread has a sketch with its related filter. Each filter works on a chunk of data and has its own items, but we know that in distributed data because we have data skewness, the top frequent items in each chunk of data will be almost the same. According to this assumption, after all threads finish their work, we merge all filters and we sum the frequency count of all similar items. Finally, we can report the top-K frequent items by sorting all elements.

5 Experimental Evaluation

This project has run in a system with AMD Ryzen 9 CPU with 8 cores and 16 threads. The system has 16 GB RAM. The language is C++ and and OpenMP API has been used for parallelism. The dataset is Gutenberg project. Gutenberg consists of about 60,000 eBooks. We just downloaded 1.3 GB text file that has around 300,000,000 words. The number of buckets is 1024 and the number of hash functions is 4. The experimental results are based on averaging 10 runs. For comparing the proposed method with other methods, we implemented Asketch and Topkapi based on the information of related papers. We consider the same sketch size and filter size for all methods. Figure 11 and Figure 12 show the sequential

```
int find_min(Filter* _filter)
{
    const __m128i increment = _mm_set1_epi32(4);
    __m128i indices         = _mm_setr_epi32(0, 1, 2, 3);
    __m128i minindices      = indices;
    __m128i minvalues       = _mm_loadu_si128((__m128i*)((*_filter).filter_newcount));
    int index;
    int min_value ;

    for (size_t i=4; i < (*_filter)._filtersize; i += 4)
    {
    indices = _mm_add_epi32(indices, increment);
    const __m128i values        = _mm_loadu_si128((__m128i*)((*_filter).filter_newcount + i));
    const __m128i lt            = _mm_cmpgt_epi32(minvalues,values);
    minindices = _mm_blendv_epi8(minindices, indices, lt);
    minvalues  = _mm_blendv_epi8(minvalues,values,lt);
    }

    int32_t values_array[4];
    uint32_t indices_array[4];

    _mm_storeu_si128((__m128i*)values_array, minvalues);
    _mm_storeu_si128((__m128i*)indices_array, minindices);

    index = indices_array[0];
    min_value = values_array[0];
    for (int i=1; i < 4; i++) {
        if (values_array[i] < min_value) {
            min_value = values_array[i];
            index = indices_array[i];
        }
    }
    return index;
}
```

Figure 9: Finding the minimmum item in the filter with SIMD 128-bit vector

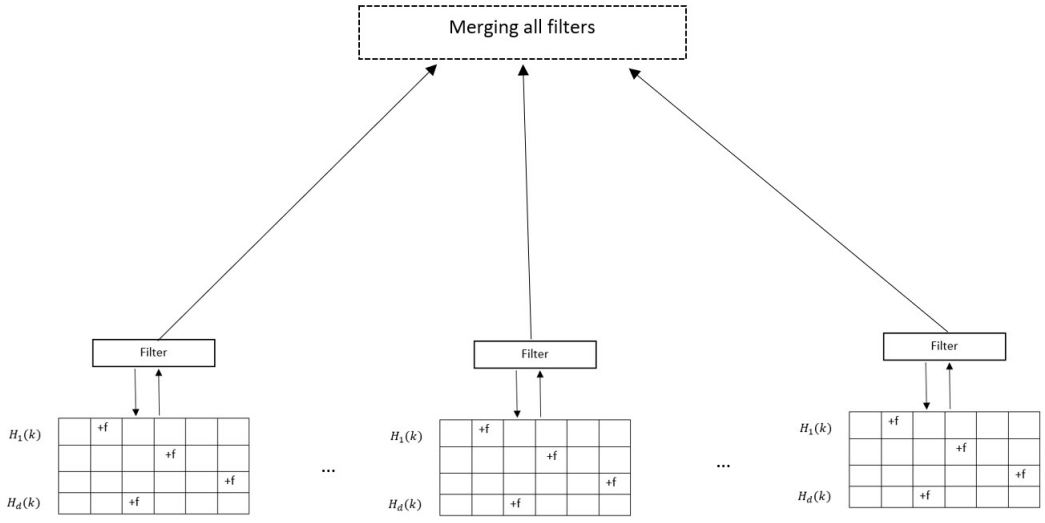

Figure 10: Multi-thread parallelism of proposed method

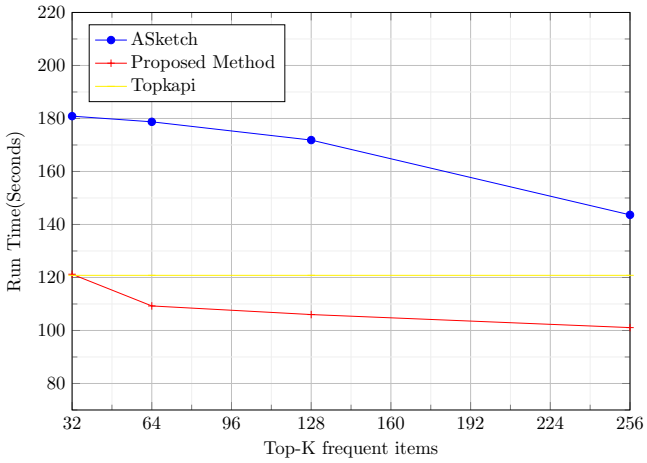

Figure 11: Sequential Implementation

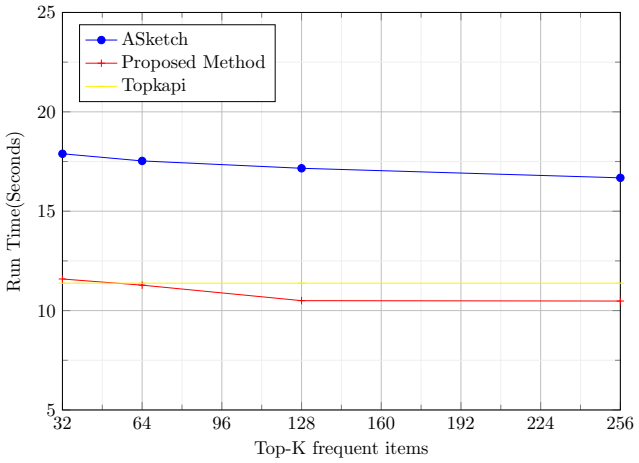

Figure 12: Parallel Implementation

and parallel implementation results of Asketch, Topkapi and the proposed method. As it is shown, the proposed method has lower computation time in both sequential and parallel implementation. Topkapi has the second place for computation time and Asketch is the worst one. We implemented Asketch and proposed method with different filter sizes equal to 32, 64, 128 and 256. For example, when K is 256 the proposed method is about 20 seconds faster than Topkapi in sequential implementation and about 1 second faster in parallel implementation. It should be mentioned that due to not having filter part in Topkapi, there is no noticeable difference among different K in Topkapi and the computation time for merging all threads is insignificant.

Figure 13 shows the speedup of proposed method for filter size equal to 128 with different number of threads. It is obvious that by increasing the number of threads, the difference between speedup and number of threads will increase and in other words the efficiency decreases. The reason is having more threads running causes more overhead on using shared IO and shared memory. Also, when we shrink data stream into some chunks, we have to start and terminate threads alternately and this causes to more overhead on several starting and terminating all threads. Ofcourse, when we use all threads of a CPU, some

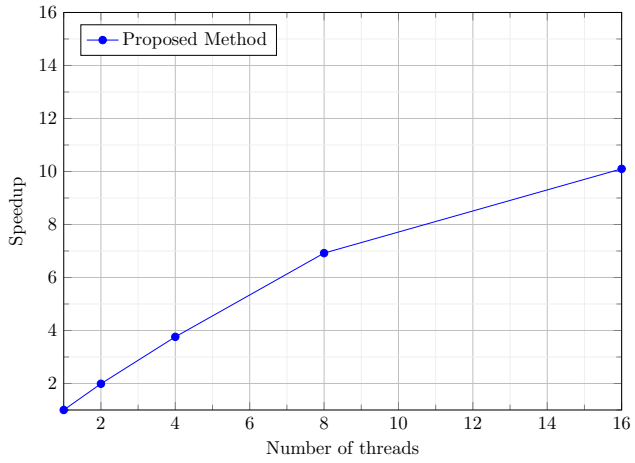

Figure 13: Proposed Method Speedup

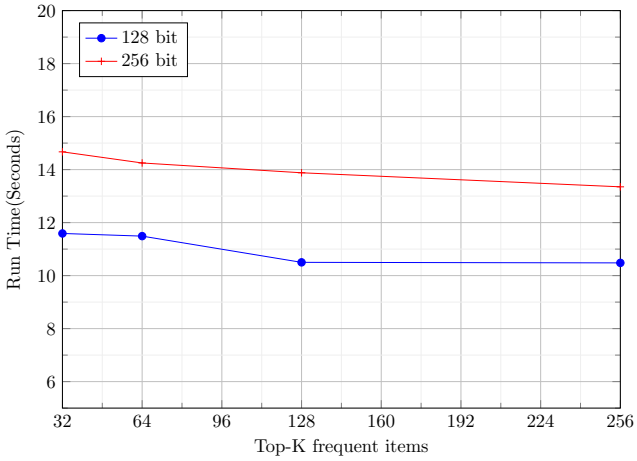

Figure 14: SIMD vectorization: 128-bit vector vs 256-bit vector

threads are dealing with other applications and we have CPU bound in that area.

For SIMD vectorization, we use SSE for 128-bit vector and AVX for 256-bit vector. As it is shown in Figure 14, the 128-bit vector has better performance than 256-bit vector. It is true that 256-bit vector computes more elements in each instruction, but this result may because some instructions in AVX have more latency and coputation time than SSE. On the other side, the CPU may have more overhead for fetching larger size of data from shared memory in each iteration.

6 Conclusions

The goal of this project was finding top-K frequent elements in a massive data stream. Because approximate algorithm is one optimal choice for solving this problem, we proposed an approximate algorithm based on CMS and compared this method with two state of the art methods. This algorithm benefits from both sketch-based and counter-based approachs. For parallelism, by using SIMD Intrinsic in the filter part for searching and finding an item, and implementing multi-thread parallelism we improved the execution speed of all three

algorithms. The results show that the proposed method has better performance among two state of the art methods. The main reason is using a pre-filtering statge over the Topkapi sketch which benefits from SIMD intrinsic. For the future work, we can implement these algorithms on multi processors to find out how fast they can be. Although we will face overhead for switching between processors ,but it can be faster because of having processors with local IO and memory. Also, it seems we can improve the SIMD intrinsic part by working on different instructions with lower latency especially in AVX instruction set that we expected better performance based on some other researches.

References

[1] H. Daume A. Goyal and G. Cormode. Sketch algorithms for estimating point queries in nlp. *Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, page 1093–1103, 2012.

[2] A. Shrivastava A. Mandal, H. Jiang and V. Sarkar. Topkapi: Parallel and fast sketches for finding top-k frequent elements. *32nd Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[3] D. Agrawal A. Metwally and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. *International Conference on Database Theory*, 3363:398–412, 2005.

[4] A. Anbarasu B. Sigurleifsson and K. Kangur. An overview of count-min sketch and its applications. *EasyChair Preprint*, (879), 2019.

[5] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2):1530–1541, 2008.

[6] G. Cormode and S. Muthukrishnan. An improved data-stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[7] R. Bordawekar D. Thomas, C. Aggarwal, and P. S. Yu. On efficient query processing of stream counts on the cell processor. *IEEE 25th International Conference on Data Engineering*, 2009.

[8] F. Korn G. Cormode, T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic udafs at streaming speeds. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, page 35–46, 2004.

[9] M. Garofalakis G. Cormode, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *IFoundations and Trends in Databases*, 4(1-3):1–294, 2012.

[10] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[11] K. Chen M. Charikar and M. Farach-Colton. Finding frequent items in data streams. *International Colloquium on Automata, Languages, and Programming*, 2380:693–703, 2002.

[12] N. Manerikar and T. Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data Knowl. Eng.*, page 415–430, 2009.

[13] G. Singh Manku and R. Motwani. Approximate frequency counts over data streams. *In Proceedings of the 28th International Conference on Very Large Data Bases*, pages 346–357, 2002.

[14] A. Khan P. Roy and G. Alonso. Augmented sketch: Faster and more accurate stream processing. *SIGMOD '16: Proceedings of the 2016 International Conference on Management of Data*, page 1449–1463, 2016.

[15] C. C. Aggarwal P. Zhao and M. Wang. gsketch: On query estimation in graph streams. *Proceedings of the VLDB Endowment (PVLDB)*, 5(3):193–204, 2012.

[16] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li. Cold filter: A meta-framework for faster and more accurate stream processing. *Proceedings of the 2018 International Conference on Management of Data*, page 741–756, 2018.

[17] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li. Diamond sketch: Accurate per-flow measurement for big streaming data. *IEEE Transaction Parallel Distribution Systym*, 30(12):2650–2662, 2019.

[18] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.