ENABLING PARALLELISM AND OPTIMIZATIONS IN DATA MINING ALGORITHMS FOR POWER-LAW DATA

A Dissertation
Presented to
The Academic Faculty

By

Ankush Mandal

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2020

ENABLING PARALLELISM AND OPTIMIZATIONS IN DATA MINING ALGORITHMS FOR POWER-LAW DATA

Approved by:

Dr. Vivek Sarkar, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Anshumali Shrivastava,
Co-advisor
Department of Computer Science
Rice University

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: June 30, 2020

ACKNOWLEDGEMENTS

First and foremost, I would like to convey my deepest gratitude to my advisor Dr. Vivek Sarkar for his mentorship, encouragement, and support throughout my graduate study at Georgia Institute of Technology. His guidance and enthusiasm in research were invaluable to my development as a graduate student. He has been my source of inspiration whenever I faced difficulties. I owe him so much for giving me the opportunity to join Habanero Extreme Scale Software Research Group.

I would like to express the greatest degree of appreciation to my other mentor Dr. Anshumali Shrivastava for giving me the opportunity to work with him. His expertise and insights were indispensable to the success of my works. I would like to thank the rest of my committee members - Dr. Hyesoon Kim, Dr. Santosh Pande, and Dr. Richard Vuduc for their valuable feedback on my work.

I am grateful to all the members of the Habanero Extreme Scale Software Research group for their help and feedback in my research work. My study at Georgia Tech would not have been as entertaining and educational without them. Additionally, I have been very fortunate to have great friends in my graduate life. I am very thankful for all their support and encouragement. Special thanks to Abhishek, Ananya, Arijit, Arkabandhu, Arpita, Arunim, Debarshi, Hamim, Himadri, Nilanjan, Pranabendu, Prasanth, Pushan, Puspita, Rabimba, Rohan, Sagnik, Saikat, Sandip, Shams, Shantanu, Suvadip, Siam, Sourav, Sriraj, Trijit.

Finally, I would like to conclude by acknowledging that I am indebted to my parents Kalpana Mandal and Durgapada Mandal, and my sister Ankita Mandal for their unconditional love. They deserve the maximum credit for this work. They continuously supported and encouraged me to pursue my dreams. Without them standing by my side, I could not have come this far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Today's data mining tasks aim to extract meaningful information from a large amount of data in a reasonable time mainly via means of — a) algorithmic advances, such as fast approximate algorithms and efficient learning algorithms, and b) architectural advances, such as machines with massive compute capacity involving distributed multi-core processors and high throughput accelerators. For current and future generation processors, parallel algorithms are critical for fully utilizing computing resources. Furthermore, exploiting data properties for performance gain becomes crucial for data mining applications. In this work, we focus our attention on power-law behavior —- a common property found in a large class of data, such as text data, internet traffic, and click-stream data. Specifically, we address the following questions in the context of power-law data: How well do the critical data mining algorithms of current interest fit with today's parallel architectures? Which algorithmic and mapping opportunities can be leveraged to further improve performance?, and What are the relative challenges and gains for such approaches?

Specifically, we first investigate the suitability of the "frequency estimation" problem for GPU-scale parallelism. Sketching algorithms are a popular choice for this task due to their desirable trade-off between estimation accuracy and space-time efficiency. However, most of the past work on sketch-based frequency estimation focused on CPU implementations. In our work, we propose a novel approach for sketches, which exploits the natural skewness in the power-law data to efficiently utilize the massive amounts of parallelism in modern GPUs.

Next, we explore the problem of "identifying top-K frequent elements" for distributed data streams on modern distributed settings with both multi-core and multi-node CPU parallelism. Sketch-based approaches, such as Count-Min Sketch (CMS) with top-K heap, have an excellent update time but lacks the important property of reducibility, which is needed for exploiting data parallelism. On the other end, the popular Frequent Algorithm

(FA) leads to reducible summaries, but its update costs are high. Our approach Topkapi, gives the best of both worlds, i.e., it is reducible like FA and has an efficient update time similar to CMS. For power-law data, Topkapi possesses strong theoretical guarantees and leads to significant performance gains, relative to past work.

Finally, we study Word2Vec, a popular word embedding method widely used in Machine learning and Natural Language Processing applications, such as machine translation, sentiment analysis, and query answering. This time, we target Single Instruction Multiple Data (SIMD) parallelism. With the increasing vector lengths in commodity CPUs, such as AVX-512 with a vector length of 512 bits, efficient vector processing unit utilization becomes a major performance game-changer. By employing a static multi-version code generation strategy coupled with an algorithmic approximation based on the power-law frequency distribution of words, we achieve significant reductions in training time relative to the state-of-the-art.

CHAPTER 1

INTRODUCTION

Large scale data processing is ubiquitous in today's computational environments given the ever-increasing sources of structured and unstructured data. We can see "big data" scenarios in a wide range of fields, such as internet search, social media, e-commerce, genomics, weather prediction, complex physics simulation, meteorology, and so on. As a consequence, there has been a surge in designing methods for extracting valuable statistical information from a large amount of data, which we can see from Deep Learning (DL) or in general Machine Learning (ML) and data mining communities. Interestingly, most of these methods require non-trivial computing power and are made practical in part through advances in processor technology. Another means to address the intractability of traditional algorithms on big data, is to develop approximate solutions, which give approximately correct results with some bound on the error metric while reducing the computation significantly.

There has been tremendous growth in the processing power over the last two decades, and we are now at a stage where the current generation supercomputers deliver peak performance in the range of hundreds of petaflops. However, this advancement has not followed a simple linear path during the last 15 years. Before 2004, architects used Dennard scaling to dramatically increase the frequency of processors to improve performance. However, as leakage power became significant, Dennard scaling was no longer feasible. As a practical solution, architects started using multiple cores in a single chip to improve processor performance without increasing clock frequency. As we reach the limit of Moore's law, the computer architecture community significantly increased investments in the design of custom accelerators. One widely adopted accelerator in the ML community is the General Purpose Graphics Processing Unit (GPGPU).

A consistent trend during the last decade for both homogeneous and heterogeneous

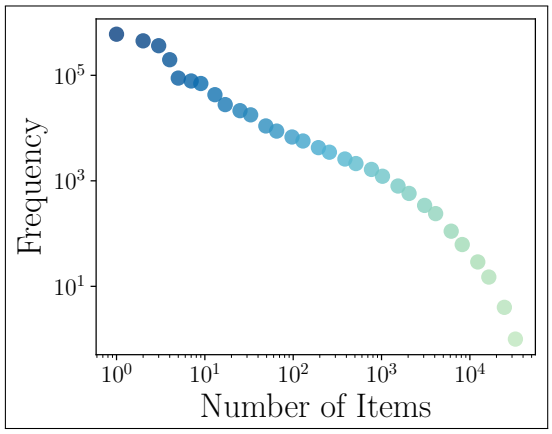
processors has been an increasing level of parallelism over time. For example, we have:

- Distributed nodes with multi-core CPUs in current supercomputers, e.g., the recent AMD® EPYC [1] CPU has up to 64 cores per chip
- Fine-grain single instruction multiple data (SIMD) parallelism in vector processing units where the vector length can be as high as 512 bits (AVX-512) in the x86 ISA
- Fine-grain single instruction multiple thread (SIMT) parallelism in GPGPUs, e.g., the NVIDIA® V100 [2] GPU has 84 streaming multi-processors, each of which contains 64 FP32 cores, 32 FP64 cores, 64 INT32 cores, and 8 Tensor cores that can be used simultaneously by 32-thread warps.
- Low power multi-core CPUs and GPUs for mobile devices, e.g., the Qualcomm® 630 mobile platform [3] has 8 Arm® Cortex®-A53 cores in its CPU and 96 ALUs in its Adreno™ 508 GPU

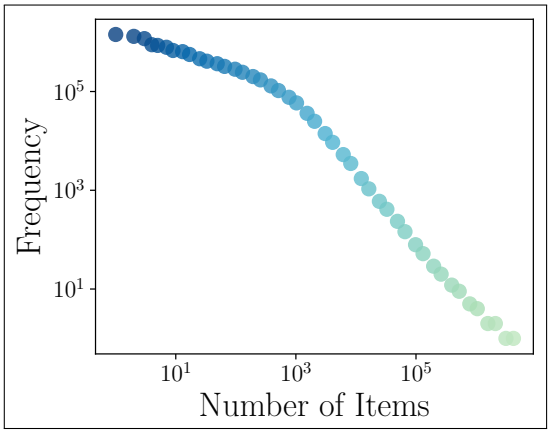
Needless to say, developing parallel algorithms is a necessity to exploit the computing power of modern hardware efficiently.

On the other frontier of handling big data by approximate solutions, we see a tremendous amount of progress over the last decade, to the point where we can enable data mining technology in edge or IoT devices too. For example, Stochastic Gradient Descent [4, 5], one of the most common computations in Deep Learning (DL), approximates error propagation in the feedback loop of Neural Networks, network pruning and compression techniques [6, 7, 8] enable DL applications to run on low memory devices, and sketch [9] based methods produce data summaries in a single pass over the data while using sub-linear memory. While presenting a comprehensive survey of all related work in this area is beyond the scope of this thesis, the important takeaway is that currently, the community has a keen interest in developing and advancing approximate data mining methods.

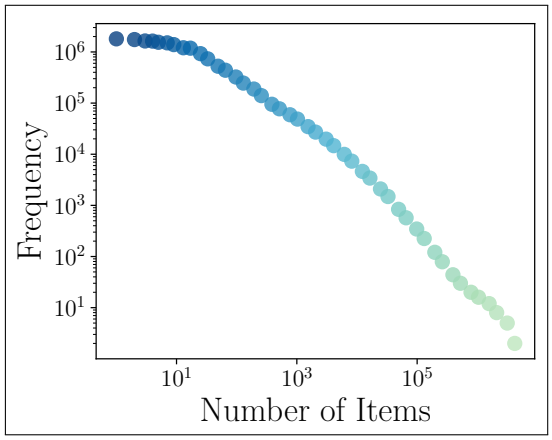
It is noteworthy to mention that we have reached a point where more and more community efforts are focusing on adopting data-centric approaches. Examples include sparsity-



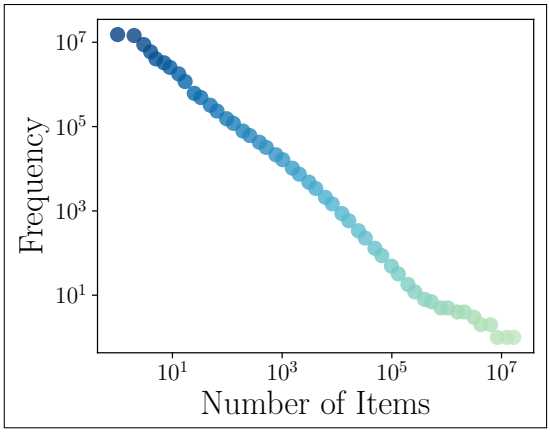
(a) *Kosarak* [10]



(b) *Webdocs* [10]



(c) *CAIDA* [11]



(d) *Criteo* [12]

Figure 1.1: Frequency distributions of elements in different real-world datasets

aware compression of Neural Network parameters [13, 8], exploiting the power-law behavior of data in frequent elements mining [14, 15], and data-centric workload optimizations [16, 17]. In the context of this thesis, we consider the power-law distribution of frequencies of elements present in data, which is commonly found in a large class of data, such as text, internet traffic, click-stream data, etc. As motivation, we show the frequency distribution of elements in log-log scale for representative real-world datasets from different classes of data in Figure 1.1. From a broad classification viewpoint, *Kosarak* [10], *Webdocs* [10], *CAIDA* [11], and *Criteo* [12] datasets represent click-stream, web text, internet traffic, and search log datasets respectively. A general intuition behind power-law behavior is that we perform computations associated with frequent elements exponentially more often compared to the rare items.

Given the current interest in approximate data mining algorithms and data-centric performance optimization approaches, we explore opportunities for improvement in performance characteristics of important data mining tasks on power-law data. Given the multitudes of parallelism in contemporary processors, an important question that follows is: how well do these algorithms map on to current parallel architectures when executed with power-law data? Another way to look at this problem is to analyze the available parallelism in the algorithms and how well they scale to different types of parallelisms employed in various architectures. Better yet, can we exploit the power-law property of data in our parallel algorithm design to further improve the performance? Are these efforts worthwhile, i.e., how much performance gain can we achieve through careful parallel data mining algorithm designs? We answer these questions in our thesis by performing a deep analysis on three very important and frequently faced problems in data mining community - a) frequency estimation [18, 14, 19] b) identification of most frequent elements [20, 21], and c) word embedding [22, 23].

1.1 Thesis Statement

"Data-centric algorithmic and mapping optimizations can deliver significant improvements in the performance of approximate data mining applications with power-law data on modern parallel architectures."

CHAPTER 2

MATRYOSHKA: FREQUENCY ESTIMATION WITH GPU PARALLELISM FOR SKEWED DATA

One of the most fundamental operation in large-scale data stream processing is frequency estimation of elements. To get an exact solution for the frequency estimation task, one has to store all the items from the data stream and then sort them to get the respective counts, or use counters for all the unique items to track their counts. However, big data scenarios involve a massive amount of data streams, which make such a "store and sort" strategy impractical from both computational and memory requirement perspective. Furthermore, the number of unique items in these scenarios is also rather very high, and the "counter" based approach with dynamic counter creation requires a high computational cost along with linear memory in terms of number of unique items which become prohibitive.

An alternative approach that has been pursued for improved scalability is to develop approximate stream processing algorithms with probabilistic data structures [18, 14, 24, 25], which usually require a single pass over the data and sub-linear memory space. Sketches are a popular choice among these approximate solutions, mainly due to their improved time and space bounds [18] in tasks associated with summarizing data streams. We can find their application in a wide variety of areas, including click-through prediction [26], real-time IP traffic measurements [27, 28, 29], feature selection [30], semi-supervised learning [31, 32, 33], and natural language processing [34].

Over the last decade, there has been a significant effort in improving the accuracy of sketch-based algorithms. Since many real world data sets obey the power law [35], the frequency distribution of items in a data stream is often highly skewed. That means few items have a very high frequency (referred to as *"hot items"* or *"heavy hitters"*) and most of the remaining items have a low frequency (referred as *"cold items"*). While applying

sketching techniques, significant frequency estimation errors occur for every hash collision between two *"hot items"* and also between a *"hot item"* and a *"cold item"*. To improve accuracy, recent works on sketches, such as Augmented sketch [36], Pyramid sketch [37], HeavyGuardian [38], and Learning-Based Frequency Estimation [19] try to exploit the skewness by handling *"hot items"* and *"cold items"* separately. All these approaches significantly reduce the average frequency estimation error compared to base-line sketches.

Interestingly, most of the past work on improving the accuracy of sketches also improve performance. However, the "performance" metric for sketching methods primarily focuses on sequential execution on CPUs [38, 37, 39]. A few cases [40, 36] also considers parallel algorithms for multi-core CPUs. However, if we look into architectures of current generation computing resources, including mobile processors, it is almost impossible to find one without parallelism. GPUs represent one of the most popular platforms in the machine learning community due to their throughput oriented architecture with massive parallelism (80 streaming multi-processor for nVidia®Volta GPU [2], each capable of running 2048 threads). Current trend [41] indicates we are going to see more new architectures [42, 43] with increasing levels of parallelism in the future. So, there is a strong motivation for designing effective parallel sketching strategies for current and future generation massively parallel hardware.

Although sketches appear to be easily parallelizable, we argue that scaling it to massive parallelism is a daunting task. For example, pipeline parallelism explored in Augmented sketch [36] scales only to two cores. Another popular SPMD style parallelism strategy with sketches [44] on multi-core CPUs executes the sketching kernel sequentially within each parallel worker. We can see from Augmented sketch [36], when we increase the number of cores from 2 to 16, we gain roughly 3x improvement on throughput using this strategy. It clearly indicates that achieving good scalability for parallel sketching algorithms is a challenging problem.

Taking GPU scale parallelism into consideration, we currently have a lack of effective

strategies for designing parallel sketching techniques. If we opt for exploiting data parallelism with *reducible* [40] or *mergeable* [45] sketches, we have to create local copies of the sketch for each parallel threads and at the end, we merge those local copies to get the final sketch. It is quite clear that this strategy [46] is not scalable to GPU level parallelism since we are dealing with hundreds of thousands of threads. Another solution is converting updates to buckets in a sketch to "`atomic`" updates if an `atomic` operation can replace the bucket update step. Using this strategy, we implement Count-Min Sketch [18], one of the most popular choice for sketching algorithms, for GPU. Figure 2.1 represents its performance analysis with *kosarak* [10] data set on nVidia®V100 GPU. As we can see, the efficiency of this approach is particularly bad (details of the metrics are in Section 2.6.5).

In this work, we address this hard problem by introducing the *Matryoshka* sketching strategy which enables efficient data parallelism on sketches and scales sketches to highly parallel architectures, such as GPUs. We argue that, in addition to improving accuracy, skewness in data can also be exploited for reducing contention in high-throughput parallel execution. Thus, we have an excellent opportunity to harness the high performance of the throughput-oriented massive architectures if we enable data-aware parallelization of the sketching algorithms.

Our Contributions. 1) We propose *Matryoshka* sketching, a nested strategy which hierarchically exploits skewness present in the data to improve contention when exploiting fine-grain data parallelism. 2) We coupled our strategy with the popular Count-Min Sketch algorithm [18]. Compared to a reference Count-Min Sketch implementation on GPU, we achieve roughly 1.2x to 5.74x throughput improvement on real datasets and 5.95x to 32x improvement on synthetic datasets following the *Zipf* distribution. 3) We provide precise mathematical quantification of contention reduction and also prove the soundness of our approach. 4) To enable *Matryoshka* sketching, we propose a lightweight online *"heavy-hitter"* detection algorithm that works in practice. 5) Our work also provides empirical evidence that it is worthwhile to invest in designing parallel sketching techniques for current and

future generation highly parallel architectures.

2.1 Frequency Estimation Problem

The frequency estimation problem can be formalized as follows: given a stream S consisting of M unique elements $\{e_1 ... e_m\}$ from some universe U, estimate frequency f_i, the number time e_i appears in S. Skewed data or stream refers to the frequency distribution, i.e. distribution of f_i being skewed. Many real world frequency distributions follow power law or Zipf's law and the corresponding data refers to as "power law data" or "Zipf data" respectively. Given a parameter α for skewness, $f_i = ci^{-\alpha}$ (where c is a constant) for power law data. With Zipf data, $f_i = \frac{N}{i^{\alpha}\zeta(\alpha)}$ where $N = \sum_{i=1}^{M} f_i$ and $\zeta(\alpha)$ is Riemann's zeta function with value $\zeta(\alpha) = \sum_{i=1}^{\infty} \frac{1}{i^{\alpha}}$.

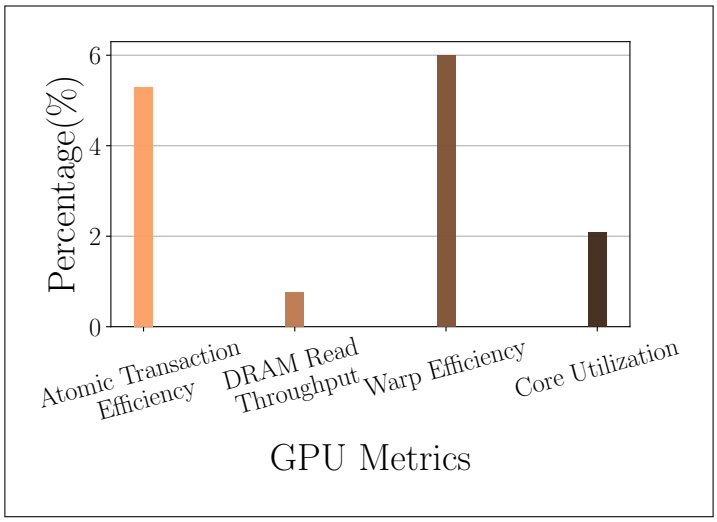


Figure 2.1: Naive *CMS* Parallelization Efficiency on GPU

As mentioned earlier, the exact solutions to the frequency estimation problem involves high computational cost and memory requirement, rendering quite impractical for large data sizes. People usually resort to approximate solutions with sub-linear memory requirements. These approximate approaches come in mainly two flavors - a) sketch-based and b) counter-based.

Hashing-based approximate solutions, such as sketches usually guarantee a small error

bound with high probability. In a trade-off, they offer high memory efficiency and fast processing. Due to this nice property and theoretical guarantees, they are a popular choice for the frequency estimation task. There has been an extensive effort to improve on its memory efficiency [37] and accuracy [18, 44, 36, 38, 19, 47]. Some very popular sketching solutions are Count-Sketch [14], Count-Min Sketch (*CMS*) [18], and multi-stage filters [48].

2.2 Sketch - Overview & Parallelism

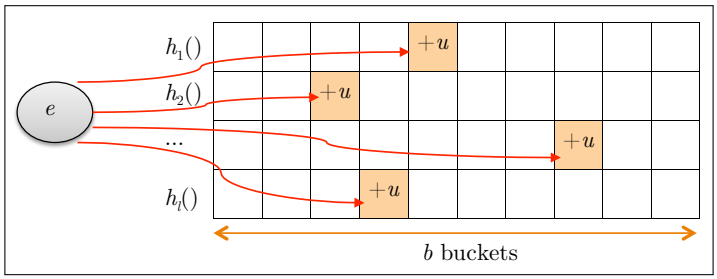


Figure 2.2: Sketch Data Structure

Briefly, as depicted in Figure 2.2, a sketch data structure consists of l arrays, $M_1, M_2, ..., M_l$, where each array contains b buckets. We have l pair-wise independent uniformly random hash functions, $h_1(), h_2(), ..., h_l()$, each associated with its respective array. These hash functions maps an element $e \in U$ to B, i.e. $h_i : U \rightarrow B \ \forall \ i \in \{1, 2, ..., l\}$ where B represents the set $\{1, 2, ..., b\}$. During stream processing, for each element e in stream S, sketching algorithms compute $h_1(e), h_2(e), ..., h_l(e)$. Then, it performs update u on the counters of the mapped buckets, $M_1[h_1(e)]$, $M_2[h_2(e)], ..., M_l[h_l(e)]$ according to the specifics of the algorithm. During query, an element is hashed the same way and based on the related l counter values, the algorithm gives an estimate of its frequency. [9] gives a good summary on different sketching methods.

Count-Min Sketch (*CMS*). This widely adopted sketching technique, proposed by Cormode and Muthukrishnan [18], increments the counters in each mapped bucket by the count associated with each element during performing update u. So, it always overestimates the counts. During query phase, it reports the minimum of the l counts of the buckets a

element hashes into. [18] proved that the expected error in frequency estimation using *CMS*, is always an overestimate $\leq \left(\frac{N}{b}\right)$ where N is the total number of elements and the error reduces exponentially with l. Hence, given an error parameter ϵ and an error probability δ, if we set $l = O(log\frac{1}{\delta})$ and $b = O(\frac{1}{\epsilon})$, the error in frequency estimation is $\leq \epsilon N$ with probability $(1 - \delta)$.

If we think of designing a parallel sketching algorithm for multi-core CPU, we find that we have abundant data parallelism. As long as the sketch data structure is *reducible* [40] or *mergeable* [45], we can easily exploit the data parallelism. We can create a separate sketch data structure for each parallel worker to process different chunks of data independently. At the end of the sketch update process, we merge the local copies sketches to produce the final sketch. [45] proved that the merged sketch would give the same error guarantees as the original sketch we would get from running the algorithm sequentially.

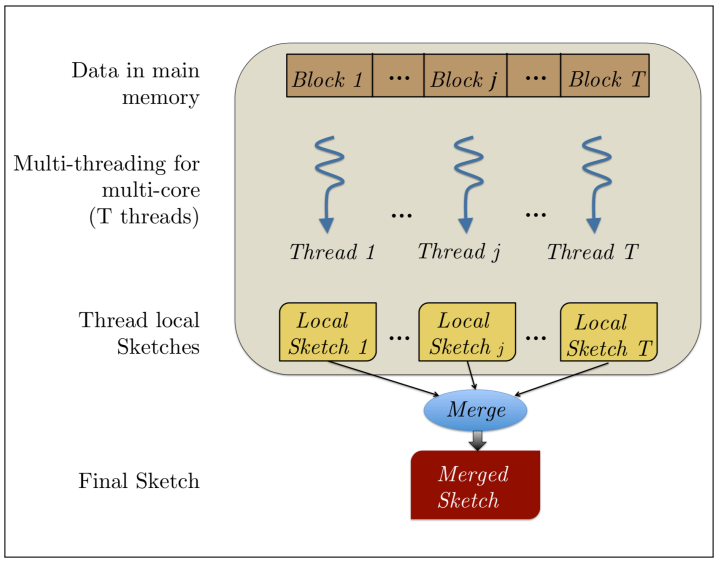


Figure 2.3: Parallel Sketch on Multi-core CPU

This parallelization strategy, as depicted in Figure 2.3, is suggested for *CMS* in [46] and used in [44, 40]. However, one thing to note that there are two overheads with this approach - a) memory overhead of creating local copies of sketches which grows linearly with the parallelism employed and b) a computational overhead of merging the thread-local copies of sketches which grows at least logarithmically with the parallelism used. We can easily see,

this approach is not scalable for hundreds of thousands of threads, such as in GPU, since the overhead would forfeit the benefits from parallelism.

2.2.1 Overview of GPU Parallelism

GPU is a throughput oriented architecture with Single Instruction Multiple Thread (SIMT) parallelism. For ease of discussion, we consider nVidia® GPUs. The fundamental unit in GPU is Streaming Multi-processor (`SM`). Each `SM` privately owns several streaming processors or simple cores, `Register File`, `L1 Cache`/`Shared Memory`. `SM`s have *in-order* issue pipeline and use `warp scheduler` for scheduling hardware threads. GPUs have a device wide `L2 Cache` and high bandwidth `Global Memory`, shared across all `SM`s. In the CUDA parallel programming model, threads are partitioned into `threadblock`s. Smallest execution unit is `warp`, which usually consists of 32 threads. So, threads inside a `threadblock` are scheduled in `warp`s. Multiple `threadblock`s are assigned to each `SM`. Thus, `Shared Memory` and `Register File` are partitioned among `threadblock`s, which in turn restricts the number `threadblock`s that can be assigned to a `SM`. For more details on the GPU architecture, one can look into [2]. A reference to CUDA programming model can be found in [49].

2.2.2 Problem: Sketch on GPU

Considering feasible parallelization of sketches on GPU, one straight-forward solution is converting each update to buckets in a sketch to "`atomic`" update if an `atomic` operation can replace the bucket update. This guarantees that if multiple threads try to modify a bucket, the accesses get serialized and the results are the same as doing the computation sequentially (we assume the operation being associative, i.e., the order does not influence the final result). The strategy is shown in Figure 2.4. The sketch data structure is shared among threads. Since we want to leverage data parallelism, different threads process different shards of data in parallel. In Figure 2.4, thread T_1 and T_3 encounter the same element e_1 and try to

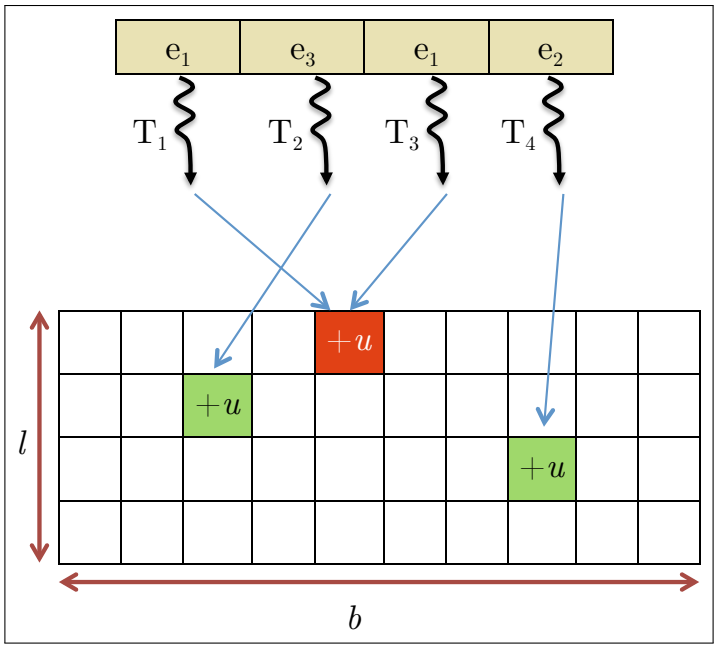


Figure 2.4: Parallel Sketch on GPU

modify the same bucket in row 1. The red-colored update u refers to the updates getting serialized. On the other hand, thread T_2 and T_4 do not face any contention and thus perform their respective updates in parallel, represented as green color update u.

In this strategy, we lose parallel performance for the updates which get serialized. Clearly, this strategy would be helpful if the number of serialized accesses is smaller compared to independent or parallel accesses. However, in the case of skewed data following Zipf or power-law frequency distribution for elements, the situation is exactly the opposite. The majority of the active threads encounter "*heavy-hitters*", and their attempts to update the respective counters get serialized. Consequently, the performance degradation due to contention becomes dire.

Using this strategy, we implement *CMS* on GPU. Figure 2.1 represents its performance analysis with *Kosarak* [10] dataset on nVidia® V100 GPU. As we can see from the values of the GPU performance metrics, the efficiency of this approach is particularly bad. One important thing to notice in Figure 2.1 is that the `atomic` updates are facing serious contention which results in retrying the `atomic` updates many times before getting

successful. It directly correlates to our hypothesis that the contention will be very high on Zipf or power-law data. This observation motivates us to design an efficient parallel sketching strategy for highly parallel architectures.

2.3 Our Proposal: Matryoshka

As mentioned in Section 2.2.2, one of the main source of inefficiency in `atomic` update based parallel sketching approach is the heavy contention on `atomic` transactions. Now, let us think of a skewed stream. If a large number of threads are working concurrently on the stream, chances are many threads will encounter *"heavy hitters"*. Apparently, the number threads encountering a common element will be proportional to the frequency of that element in the stream. When these threads try to update the same bucket corresponding to the common element, they face contention. Seemingly, the main sources of contention are the *"heavy hitters"* since they have the highest frequencies in the stream.

An obvious question followed after the above insight is that how we can address this skewness in a highly parallel system where it is impractical, or worse, impossible to make local copies of the sketch. Fortunately, most data follow power law [35], and the number of *"heavy hitters"* is much smaller compared to the total size of the element set. So, conceptually, if we create local copies of the buckets for only these *"heavy hitters"*, we will be able to avoid a significant amount of contention.

With the above-mentioned strategy in our hand, we face two stumbling blocks - 1) how do we identify *"heavy hitters"* and make local copies of the respective buckets in a one-pass frequency estimation algorithm, and 2) even if we are able to identify *"heavy hitters"*, how to determine the cut-off point in the frequency distribution to define the *"heavy hitters"* set. We target the second problem in the next section and address the first problem in Section 2.3.2.

2.3.1 Matryoshka - Hierarchical Exploitation of Skewness

A *"heavy hitters"* set HH contains all elements whose frequency is greater than some threshold frequency f_k. The size of HH or the number of *"heavy hitters"* in HH is determined by the position of f_k in the frequency distribution. Since f_k is generally high, we have a small number of *"heavy hitters"* in HH. As a result, making copies of corresponding buckets is very feasible. However, unless the frequency is ultra-skewed, we will be missing many elements with moderately high frequencies, and thus, we will fail to avoid a great portion of contention in parallel execution. Whereas, if we choose f_k to be moderately low, we will be able to accommodate most of the *"heavy hitters"* of interest. But, the size of HH may become too large for us to make local copies.

A very similar problem was faced by researchers in the computer architecture community when they tried to bridge the gap between memory speed and processor speed. As a practical and effective solution, memory cache system was invented. We take inspiration from this hierarchical memory system. In theory we can have n number of *"heavy hitters"* sets $HH_1, HH_2, ..., HH_n$ where $HH_i \subset HH_j \;\; if \;\; i < j \;\; \forall\, i, j \in \{1, 2, ..., n\}$. We can then make C_i local copies of HH_i, where $C_i > C_j \;\; if \;\; i < j \,\forall\, i, j \in \{1, 2, ..., n\}$. The number of threads sharing a local copy at i-th level is T_i, where $T_i < T_j \;\; if \;\; i < j \,\forall\, i, j \in \{1, 2, ..., n\}$. A sample adoption of the strategy is depicted in Figure 2.5. An interesting observation, the cache system is designed for latency, whereas we are using similar kind of hierarchical system to improve throughput.

The data structure we choose for HH can be called a Counting Bloom Filter with one hash function or a sketch with only one row. However, each bucket not only has a $counter$ for frequency estimation, but also an id field for identifying the corresponding *"heavy hitter"*. If the elements require more bits than the number of bits assigned to id, we can store a hash fingerprint in the id.

Algorithm 1 presents pseudocode for updating *CMS* using our proposed *Matryoshka* sketching strategy. For ease of discussion and algorithmic representation, we assume two

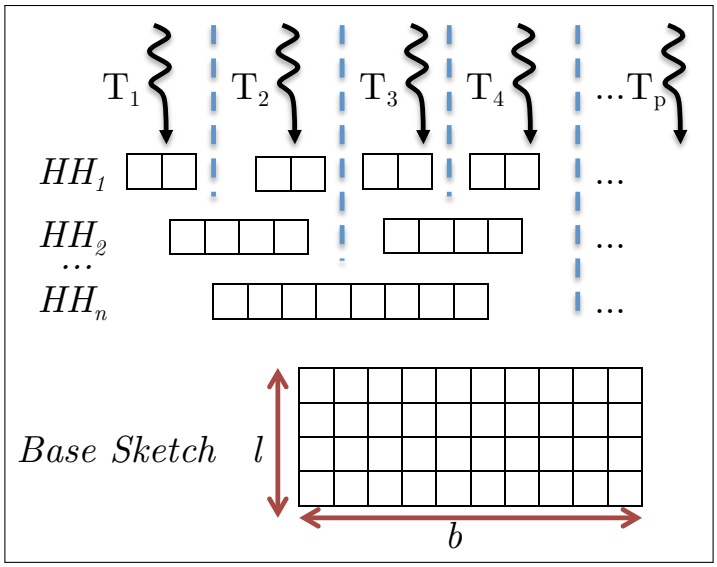


Figure 2.5: *Matryoshka* Sketching Strategy

levels of HH, i.e. we have HH_1 and HH_2. Let HH_1 and HH_2 contain b_1 and b_2 buckets respectively. The hierarchy here is as follows: $HH_1 \rightarrow HH_2 \rightarrow$ Base Sketch (*CMS*). If the base sketch contains b buckets per row, then $b_1 << b_2 << b$.

In the beginning, we initialize *CMS* and HH sets. We set the $count$ fields in all the buckets in *CMS* and HH sets to zero. Now, assuming the identity of the "*heavy-hitters*" are given, i.e., id fields of HHs are already set, we start processing elements from the given stream. After applying the first hash function $h_1()$, if we find that the id of the element matches the id in the corresponding bucket in HH_1, we increment the respective counter in HH_1. If the id check fails, we do the same for HH_2. After checking HH_2, if there is no match in id, we go to *CMS* and perform the regular *CMS* update procedure. After the stream processing completes, we need to fold the counts of the elements tracked by HH sets to their respective counters in *CMS*. For that, we first fold the values of HH_1 to HH_2. This phase presented in lines 16 to 19 in Algorithm 1. For ease of presentation and correctness, we have mentioned that we first merge the C_1 copies of HH_1 in groups of $\frac{C_1}{C_2}$ and then fold the counts from the merged HH_1s to the respective HH_2 copy among the C_2 copies. However, in actual computation, this is done by a group of parallel threads, and they directly

Algorithm 1: Matryoshka Sketching

Data: Input stream S
Result: Updated Count-Min Sketch CMS

1 $b \longleftarrow \lceil \frac{e}{\epsilon} \rceil$ //ϵ is error parameter
2 $l \longleftarrow log\frac{1}{\delta}$ //δ is error probability
3 $CMS \longleftarrow l \times b$ counters
4 $HH_1.count[i] \longleftarrow 0 \quad \forall i \in \{1, 2, .., b_1\}$
5 $HH_2.count[i] \longleftarrow 0 \quad \forall i \in \{1, 2, .., b_2\}$
6 //Assume $HH_1.id$ and $HH_2.id$ are given
7 **for** $e \in$ *stream* S **do**
8 **if** $e == HH_1.id[h_1(e)\%b_1]$ **then**
9 $HH_1.count[h_1(e)\%b_1] \longleftarrow HH_1.count[h_1(e)\%b_1] + 1$
10 **else if** $e == HH_2.id[h_1(e)\%b_2]$ **then**
11 $HH_2.count[h_1(e)\%b_2] \longleftarrow HH_2.count[h_1(e)\%b_2] + 1$
12 **else**
13 **for** $i \in 2, 3, ..., l$ **do**
14 *calculate* $h_i(e)$
15 $CMS[i][h_i(e)] \longleftarrow CMS[i][h_i(e)] + 1$
16 *perform* `reduction` *on batch of* $\frac{C_1}{C_2}$ *copies of* HH_1
17 **for** $j \in 1, 2, ..., b_1$ **do**
18 $bucket \longleftarrow h_1(HH_1.id[j])\%b2$
19 $HH_2.count[bucket] \longleftarrow HH_2.count[bucket] + HH_1.count[j]$
20 *perform* `reduction` *on* C_2 copies of HH_2
21 **for** $j \in 1, 2, ..., b_2$ **do**
22 **for** $k \in 1, 2, ..., l$ **do**
23 $bucket \longleftarrow h_k(HH_2.id[j])$
24 $CMS[k][bucket] \longleftarrow CMS[k][bucket] + HH_2.count[j]$
25 `return` CMS

update the counts in HH_2 through `atomics`. We fold the counts from HH_2 to *CMS* in similar fashion.

2.3.2 Head-first Scan - Light Weight Heavy Hitter Detection

Now that we have *Matryoshka* sketching strategy, which exploits *"heavy hitters"* to reduce contention in a massive parallelism scenario, the critical part left is *"heavy hitters"* identification. It is indicated in line 6 of Algorithm 1 where we mention that the identities of the *"heavy hitters"* are given. As discussed in the beginning of this chapter, we have a rich

Algorithm 2: Head-first scan

Data: Input sub stream S_{sub}
Result: Updated HH_1 and HH_2

$HH_1.count[i] \longleftarrow 0 \quad \forall i \in \{1, 2, .., b_1\}$
$HH_2.count[i] \longleftarrow 0 \quad \forall i \in \{1, 2, .., b_2\}$
for $e \in stream\ S_{sub}$ **do**
    $CMS[1][h_1(e)] \longleftarrow CMS[1][h_1(e)] + 1$
    if $HH_2.count[h_1(e)\%b2] < CMS[1][h_1(e)]$ **then**
        $HH_2.count[h_1(e)\%b2] \longleftarrow CMS[1][h_1(e)]$
        if $HH_2.id[h_1(e)\%b2] \neq e$ **then**
            $HH_2.id[h_1(e)\%b2] \leftarrow e$
    do rest of the CMS *update*
for $r \in \frac{b_2}{b_1}$ **do**
    Find `max` *among* $HH_2.count[r * \frac{b_2}{b_1} : (r+1) * \frac{b_2}{b_1}]$
    place corresponding $HH_2.id$ *entry in* $HH_1.id[r]$
`return` HH_1 *and* HH_2

literature in improving the accuracy of sketches by separately handling *"heavy hitters"*. For example, we can employ the recurrent neural network based strategy mentioned in [19] for a robust solution or a table look-up type strategy used in [36] for quick identification.

Additionally, in this work, we propose a lightweight online strategy for *"heavy hitters"* detection to aid *Matryoshka* sketching in improving parallel performance for frequency estimation. We call it *head-first scan*. Our aim here is not to give another competing *"heavy hitters"* detection algorithm. Rather, we want to design a very lightweight technique to keep thread divergence (a big performance bottleneck in GPU) as low as possible. If the streaming data is not adversarial and the data is uniformly random, i.e., it is not towards sorted or clustered type data, then we can use *head-first scan* for our purpose of identifying *"heavy hitters"* to apply *Matryoshka* sketching.

As we can see in Algorithm 2, for the small portion of a stream on which we use *head-first scan*, we perform the regular *CMS* updates for all elements. For an element, after doing the update in *CMS* for the first row, we check whether the corresponding count in HH_2 is less than the count in *CMS*. If the check succeeds, we set the count of HH_2 to that of *CMS*. Then we check if the corresponding id field of HH_2 bucket matches with the

current element id. If not, we replace the id field with the current element id. At the end of substream processing, we divide the range of HH_2 into equal-sized slots of number the same as the range of HH_1. We take the `max` counts and corresponding ids from these slots to fill HH_1 fields.

The idea is to run the *head-first scan* algorithm for a small subset of data to build the HH sets and then rely on the HH set for the rest of the data. There can be two strategies - a) run *head-first scan* at the very beginning for few iterations and then use the *"heavy hitters"* info, and b) periodically run *head-first scan* for few iterations to build HH and then use this HH for several iterations. A pseudocode for *head-first scan* algorithm is given in Algorithm 2. We carried out extensive experiments on real datasets along with synthetic datasets generated using *Zipf* distribution and found that our *head-first scan* method works in practice. In the worst case, we can always resort to the sketch itself periodically to identify the heavy hitters.

2.4 Theoretical Analysis

In this section, we provide some theoretical analysis and fundamental properties of our approach.

2.4.1 Matryoshka Sketching Analysis

Theorem 2.4.1. *The heavy hitter sets, i.e., HH sets in Matryoshka sketching do not have any error in their counter array, i.e., $HH.count$.*

Proof: Since, HH sets contain *id*s of the *"heavy hitters"* they are tracking, a counter only gets incremented when the corresponding *id* matches. Hence, the counters embedded in HH buckets give an accurate count of the number of times the element is observed.

Theorem 2.4.2. *Matryoshka sketching does not introduce any additional errors; it gives the same error guarantee as the base sketching algorithm does.*

Proof: As Theorem 2.4.1 proves that the counters in HH sets contain an accurate measure of the partial counts, when *Matryoshka* strategy adds these partial counts to all counter arrays of the base sketch, it does not introduce any error. The counts in base sketch are exactly the same as they have been without applying *Matryoshka* sketching. This theoretical result is also confirmed by the accuracy results in Section 2.6.3.

Definition 2.4.1. Contention. *We define "contention" as the situation where more than one parallel worker (for example, threads) encounters same unique element from a stream and thus the attempts to modify the corresponding bucket in the shared sketch data structure gets* linearized *for correctness (for example, using an* `atomic` *operation). We define "contention factor" as the product of the number of parallel workers participating in the contention and the expected value of the contention.*

Definition 2.4.2. Parallel Update. *Here "T-way data parallel" update means T parallel counter updates are being performed at a given time instance. Naturally, this number will be less than or equal to the number of active threads.*

Theorem 2.4.3. *A T-way data parallel CMS update, over a stream of length N consisting of M unique elements $\{e_1, e_2, .., e_M\}$ with frequencies $\{f_1, f_2, ..., f_M\}$, encounters contention of factor $\sum_{j=2}^{T} \sum_{i=1}^{M} j \times \binom{T}{j} \times (f_i/N)^j$ without considering the hash collisions in CMS.*

Proof: Assuming the stream is uniform, the probability of any 2 threads among T threads processing the same element e_i is $(f_i/N)^2$. As there can be $\binom{T}{2}$ such scenarios, the expected value for contention in this case is:

$$\binom{T}{2} \times (f_i/N)^2 \tag{2.1}$$

If we consider the situation for all elements in $\{e_1, e_2, .., e_M\}$ instead of a specific element e_i, the expected value from Equation 2.1 becomes:

$$\sum_{i=1}^{M} \binom{T}{2} \times (f_i/N)^2 \tag{2.2}$$

Now, the degree of contention is determined by the number of threads participating in the contention. So, the contention factor here will be:

$$2 \times \sum_{i=1}^{M} \binom{T}{2} \times (f_i/N)^2 \tag{2.3}$$

We can have contention for any scenario where two or more threads are processing the same element. Hence, we can have contention among 2 to T threads here. Consequently, the complete contention factor becomes:

$$\sum_{j=2}^{T} \sum_{i=1}^{M} j \times \binom{T}{j} \times (f_i/N)^j \tag{2.4}$$

Theorem 2.4.4. *Applying Matryoshka sketching using one level of HH with K buckets and C copies on top of CMS in the scenario mentioned in Theorem 2.4.3, reduces the contention factor by $\sum_{j=2}^{T} \sum_{i=1}^{K} j \times \binom{T}{j} \times (f_i/N)^j - C \times \sum_{j=2}^{\frac{T}{C}} \sum_{i=1}^{K} j \times \binom{T}{j} \times (f_i/N)^j$ where $\{f_1, f_2, ..., f_k\}$ represents the approximate top-K frequencies.*

Proof: As each HH has K buckets, they are tracking K unique elements. Also, we have C copies of HH. So, a group of $\frac{T}{C}$ threads accessing each HH. Replacing M with K and T with $\frac{T}{C}$ in Equation 2.4, we get the contention for each HH:

$$\sum_{j=2}^{\frac{T}{C}} \sum_{i=1}^{K} j \times \binom{T}{j} \times (f_i/N)^j \tag{2.5}$$

Since, HH tracks approximately the top frequent elements, $\{f_1, f_2, ..., f_k\}$ will represent the approximate top-K frequencies from set $\{f_1, f_2, ..., f_M\}$.

Assuming a simplification that all C copies of HH are tracking same K elements, the accumulated contention factor from all HH will be:

$$C \times \sum_{j=2}^{\frac{T}{C}} \sum_{i=1}^{K} j \times \binom{T}{j} \times (f_i/N)^j \tag{2.6}$$

Apart from the K elements tracked by HH, all other elements will contribute to the contention factor the same way as they did in Equation 2.4. So, the contention factor for CMS will be:

$$\sum_{j=2}^{T}\sum_{i=k+1}^{M} j \times \binom{T}{j} \times (f_i/N)^j \tag{2.7}$$

To get the contention factor reduction due to application of *Matryoshka* sketching strategy, we sum equation 2.6 and 2.7, and then subtract it from Equation 2.4, which is:

$$\sum_{j=2}^{T}\sum_{i=1}^{K} j \times \binom{T}{j} \times (f_i/N)^j - C \times \sum_{j=2}^{\frac{T}{C}}\sum_{i=1}^{K} j \times \binom{T}{j} \times (f_i/N)^j \tag{2.8}$$

Theorem 2.4.5. *For Zipf data, the upper bound on contention reduction from Matryoshka sketching becomes* $T(2^{T-1} - 2^{(T/C-1)})$.

Proof: If we consider Zipf data with parameter α, f_i in Equation 2.8 becomes $\frac{N}{i^\alpha \zeta(\alpha)}$ where $\zeta(\alpha)$ is Riemann's zeta function and $\zeta(\alpha) = \sum_{i=1}^{\infty} \frac{1}{i^\alpha}$. After replacing the value of f_i and approximating $\sum_{i=1}^{K} \frac{1}{i^\alpha}$ as $\zeta(\alpha)$, Equation 2.8 becomes:

$$\sum_{j=2}^{T} j \times \binom{T}{j} - C \times \sum_{j=2}^{\frac{T}{C}} j \times \binom{T}{j} \tag{2.9}$$

After replacing the values for the summation over binomial coefficients, Equation 2.9 becomes:

$$T \times \left(2^{T-1} - 1\right) - C \times \frac{T}{C}\left(2^{\frac{T}{C}-1} - 1\right) \tag{2.10}$$

After simplification of Equation 2.10, the final expression we get for contention factor reduction is:

$$T(2^{T-1} - 2^{(T/C-1)}) \tag{2.11}$$

Remark. *We can apply Theorem 2.4.3 to 2.4.5 on HH set at any level of an arbitrarily deep Matryoshka sketching strategy by replacing CMS with the HH set situated just below the level we are considering.*

Theorem 2.4.6. *Assuming t_1, t_2 and t_S as the update time for HH_1, HH_2, and CMS respectively, the average update time for Matryoshka sketching is* $(t_1 \sum_{i=1}^{b_1} f_i + t_2 \sum_{i=b_1+1}^{b_2} f_i + t_S \sum_{i=b_2+1}^{M} f_i)/N$ *where* $t_1 \ll t_2 \ll t_S$.

Proof: If we assume we have perfect information on top frequent elements, the elements tracked by HH_1 will be top frequent b_1 elements in the stream. Similarly, HH_2 will track top frequent b_2 elements. Rest of the elements will be handled by *CMS*. Consequently, the update time for b_1 elements will be t_1. Since $HH_1 \subset HH_2$, (b_1+1)th to b_2th most frequent elements will have update time associated with HH_2, which is t_2. The rest of the elements, i.e. (b_2+1)th to Mth elements (assuming M unique elements in the stream) will update *CMS* counters with update time t_S. Now, to get the average update time per element, we need to perform a weighted average of the respective update times based on the frequency of elements. Hence, the average update time will be:

$$\left(t_1 \sum_{i=1}^{b_1} f_i + t_2 \sum_{i=b_1+1}^{b_2} f_i + t_S \sum_{i=b_2+1}^{M} f_i \right) /N \tag{2.12}$$

Theorem 2.4.7. *Considering Zipf data with parameter α, the average update time for Matryoshka sketching becomes* $t_1 + \frac{t_2}{\zeta(\alpha)} \sum_{i=b_1+1}^{b_2} i^{-\alpha} + \frac{t_S}{\zeta(\alpha)} \sum_{i=b_2+1}^{M} i^{-\alpha}$.

Proof: For Zipf data with parameter α, the value for frequency f_i becomes $\frac{N}{i^\alpha \zeta(\alpha)}$ where $\zeta(\alpha) = \sum_{i=1}^{\infty} i^{-\alpha}$. Replacing f_i with this value in Equation 2.12 and approximating $\sum_{i=1}^{b_1} i^{-\alpha}$ as $\zeta(\alpha)$, the average update time becomes:

$$t_1 + \frac{t_2}{\zeta(\alpha)} \sum_{i=b_1+1}^{b_2} i^{-\alpha} + \frac{t_S}{\zeta(\alpha)} \sum_{i=b_2+1}^{M} i^{-\alpha} \tag{2.13}$$

2.4.2 Head-first Scan Analysis

Theorem 2.4.8. *For sequential version of Head-first Scan on Zipf or power law data, the average accuracy in HH_2 is* $\sum_{i=1}^{b_2} \left(\frac{i}{b_2}\right)^{(b_2-i+1)} \left(1 - \frac{i-1}{b_2^2}\right)$ *assuming random order arrival*

in input data and suitable sample size.

Proof: If we assume random arrival order in input data and a sufficiently large sample size, *Head-first Scan* encounters a similar frequency distribution (assumed Zipf with parameter α) and the same b_2 most frequent elements, as the complete data.

Now, the accuracy for HH_2 can have b_2 distinct values, i.e. $\{\frac{1}{b_2}, \frac{2}{b_2}, ..., 1\}$ with respectively $1, 2, ..., b_2$ tracked elements being accurate. Considering Zipf or power law data and assuming the case of no collisions in the hash buckets of *CMS*, the probability associated with these accuracy values are respectively $\left(\frac{1}{b_2}\right)^{b_2-1}, \left(1-\frac{1}{b_2^2}\right) * \left(\frac{2}{b_2}\right)^{(b_2-2)}, ..., \left(1-\frac{(b_2-1)}{b_2^2}\right)$, ignoring higher order terms of b_2. So, the expected value for accuracy in HH_2 is:

$$\sum_{i=1}^{b_2} \left(\frac{i}{b_2}\right)^{(b_2-i+1)} \left(1-\frac{i-1}{b_2^2}\right) \tag{2.14}$$

If we consider collisions in the hash buckets of *CMS*, the load factor for each bucket is (M/b). Picking an element e will be $\propto f(e)/\sum_{i=1}^{M/b} f(e_i)$ where $f(e)$ represents frequency of element e. As we are dealing with most frequent elements from Zipf data in HH_2, this ratio becomes 1 for average case. Hence, the average accuracy is the same as Equation 2.14.

Theorem 2.4.9. *Considering HH_2, the parallel version of Head-first Scan will have the same accuracy as the sequential version if each threadblock gets a minimum sample size of $(\zeta(\alpha)(b_2+1)^{\alpha})^2$.*

Proof: Using Hoeffding's inequality, for θ-heavy hitters, a sample size of $(1/\theta^2)$ is required. Here, HH_2 tracks b_2 most frequent elements, or in other words, θ corresponds to (b_2+1)th frequency. Hence, a loose lower bound on the sample size for HH_2 will be $(\zeta(\alpha)(b_2+1)^{\alpha})^2$. We can ensure that the parallel version of *Head-first Scan* has same accuracy as the sequential version, if for each copy of HH_2, the group of threads building HH_2, encounters the minimum sample size. Hence, the minimum sample size for each *threadblock* will be $(\zeta(\alpha)(b_2+1)^{\alpha})^2$.

Remark. *One can do a similar analysis for HH_1.*

2.5 Implementation

To show the effectiveness of our parallel sketching strategy on massively parallel architecture, we have implemented our approach *Matryoshka* sketching along with *head-first scan* for nVidia®GPUs using CUDA. We choose *CMS* as our base sketching algorithm on which we apply *Matryoshka*. The main reason behind this decision was to show improvement on a widely adopted sketching algorithm and *CMS* fits this criterion well.

Matryoshka Mapping on GPU. To implement *Matryoshka* sketching on GPU, we mapped the first level of *"heavy hitters"* set, i.e. HH_1 to GPU `Register File`. So, each thread will have its own HH_1 copy and there is no contention in updating counters in HH_1. The next level, HH_2 is mapped to GPU `Shared Memory`. So, each `threadblock` will have its own HH_2 summary and the group of threads inside a `threadblock` will share HH_2. Finally, the *CMS* buckets map to GPU `Global Memory` and shared by all the threads system wide. The b_1 and b_2 parameters are constrained by GPU architecture parameters. From a purely parallel algorithm point of view, the higher the values of b_1 and b_2, the better. So, once we fix the target GPU *occupancy*, we can set b_1 to be $(\textit{register-size}/(\textit{maximum-number-of-threads} \times \textit{occupancy}))/\textit{size-of-each-bucket-in-}HH_1$. Similarly, we can set b_2 to be $(\textit{shared-memory-size}/\textit{number-of-thread-blocks-per-}SM)/\textit{size-of-each-bucket-in-}HH_2$.

We can see here, the multiple levels of HH data structures are important for exploiting parallelism hierarchies in GPUs. A single HH set at the `Register File` level is sub-optimal for a GPU, because increasing the registers in a thread can decrease the number of threads that can run in parallel in a `threadblock`. Likewise, a single HH set at the `Shared Memory` level can limit the number of `threadblocks` that can run in a `SM` while incurring more serialization overhead from `Shared Memory atomics`.

2.6 Evaluations

2.6.1 Experimental Setup

2.6.1.1 Hardware Configuration

We carried out all our GPU experiments on a machine with nVidia® Tesla® V100 GPU. This GPU is based on Volta™ GV100 architecture which has 80 Streaming Multiprocessors (SM), 16GB High Bandwidth Memory as main memory, `Shared Memory` per SM configurable up to 96KB, and 256KB `Register File` per SM. The operating system of the machine is Red Hat Enterprise Linux 6.7. On the other hand, all our CPU experiments were carried out on a machine with Intel® Xeon® Platinum 8180 CPU (Skylake architecture) having 28 cores @ 2.5GHz with maximum Turbo frequency being 3.8GHz. The machine has a total of 756 GB DRAM and runs on CentOS Linux 7 operating system.

2.6.1.2 Software Configuration

We implemented our work in C++ using CUDA [49] APIs for GPU code. Codes for all the methods we compare with are also in C++. In the case of parallel *CMS* on CPU, we use OpenMP for exploiting multi-threaded shared memory parallelism. We incorporate as many common frameworks (such as hash functions) as possible, across the implementations, to ensure the comparisons are meaningful. Finally, we compile our GPU codes using `NVCC` version 9.1.85 and for compilation of CPU codes, we use `GCC` version 4.8.5 with `O3` optimization level.

2.6.1.3 Datasets

In our experiments, we have used synthetic datasets as well as real datasets. The details of the data sets are as follows:

Zipf - It is synthetic stream data generated following Zipf distribution with the skewness parameter varying from 1.25 to 2 with steps of 0.25. These datasets contain 720 million

items.

Kosarak [10] - An anonymized click-stream data of a Hungarian online news portal. It is a relatively small dataset with around 8 million elements, of which 41720 is unique.

Webdocs [10] - This data set is built from a spidered collection of web html documents and contains roughly 300 million items belonging to a set of 5.3 million unique items.

CAIDA - It is an internet traffic dataset from CAIDA UCSD Anonymized Internet Traces Dataset 2016 [11]. It contains IP packets with source IP addresses and destination IP addresses. Our dataset includes 472 million items comprising 6.6 million unique items.

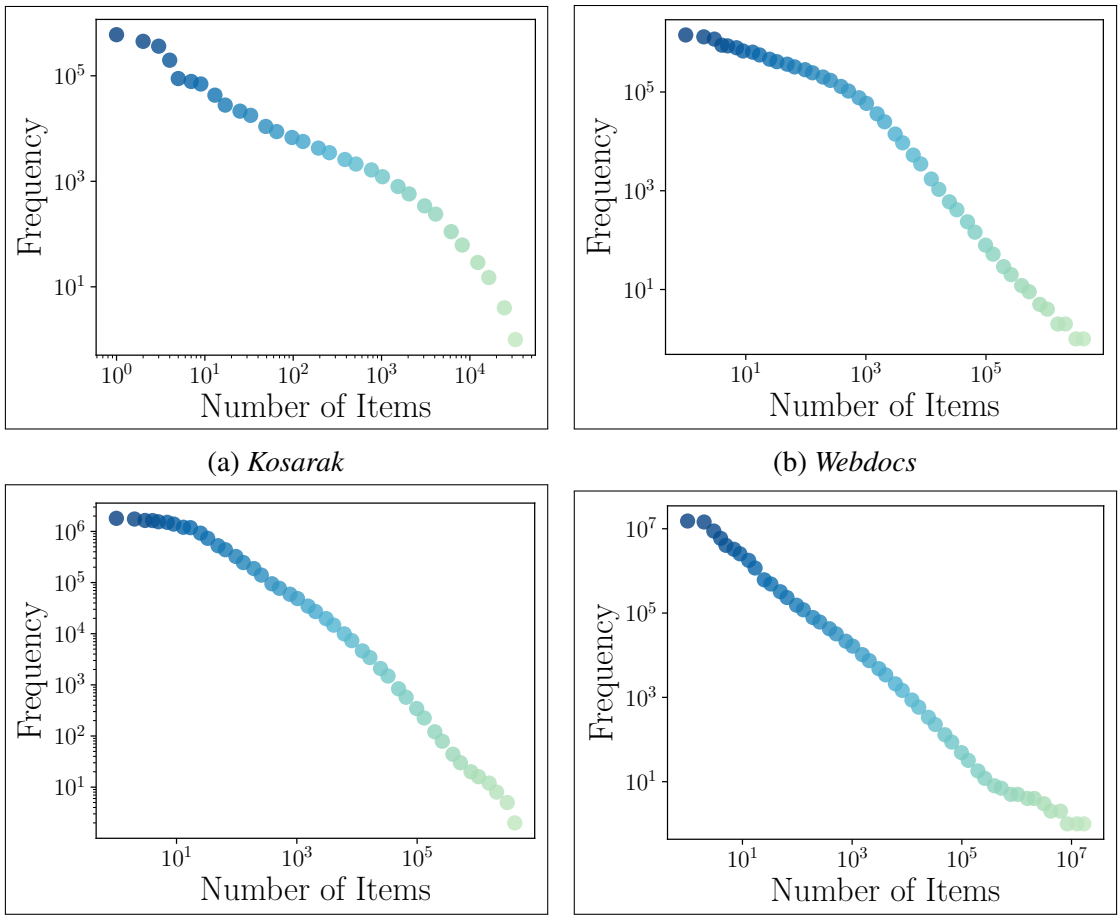
Criteo - Criteo search conversion log dataset [12] contains anonymized logs from Criteo Predictive Search. Each entry includes conversion feedback and feature values, such as product information, timestamp of click, user characteristics for clicked display advertisements sampled over a two month period from Criteo live traffic data. It contains near 398 million elements composed of 23 million unique elements.

Figure 2.6a, 2.6b, 2.6c, and 2.6d represent the frequency distributions of elements in *Kosarak*, *Webdocs*, *CAIDA*, and *Criteo* datasets respectively.

2.6.1.4 Metrics for Empirical Analysis

We have mainly used three metrics for our empirical study in this work, namely Insertion Throughput (*Mips*), Query Throughput(*Mqps*), and ARE. We give the details of these metrics here. Metrics used in `performance counter` based performance analysis is given separately in the respective location (Section 2.6.5) of discussion to maintain locality of reference.

Insertion Throughput(*Miqs*). This is a performance measurement metric for insertion or update operation in sketches. As sketches require one pass over the data to have estimated counts for all the elements, the performance for this phase can be measured in terms of time spent. If we want to compare the performance across different datasets, it is meaningful to normalize the performance with respect to data size or the total number of items. Throughput



(a) *Kosarak* (b) *Webdocs*

(c) *CAIDA* (d) *Criteo*

Figure 2.6: Frequency distributions of items in different real datasets (presented again for ease of reference)

is a metric inverse of time spent per item. Specifically, we measure the insertion or update throughput as millions of insertions per second or *Mips*.

Query Throughput(*Mqps*). Similar to the insertion phase, we measure the performance of the query phase in sketching algorithms in terms of throughput or more specifically millions of queries processed per second (*Mqps*).

***ARE*.** To compare accuracy of methods, the error metric we used in our experiments is average relative error or *ARE*. Relative error for a frequency estimation query is given as the difference between the estimated frequency ($\hat{f}$) and the true frequency (f), divided by the true frequency, or in other words the ratio presented by $\frac{|\hat{f}-f|}{f}$. Hence, *ARE* can be defined as follows: for a query stream S consisting of Q query elements $\{e_1, e_2, ..., e_Q\}$, the *ARE*

for frequency estimation will be $\frac{1}{Q}\sum_{e_i}\frac{|\hat{f}_i - f_i|}{f_i} \quad \forall\, i \in \{1, 2, .., Q\}$, where $\hat{f}_i$ is the estimated frequency for e_i and f_i is the true frequency of e_i.

2.6.1.5 Methods of Comparison

For a comparative evaluation of our method, we include the following sketching approaches:

CMS CPU. An implementation of basic sequential *CMS* on CPU.

HeavyGuardian CPU. This method refers to the *HeavyGuardian* [38] algorithm on CPU. To the best of our knowledge, *HeavyGuardian* gives the state-of-the-art throughput for Sketching-based methods for single-thread CPU execution. So, we compared our performance results against *HeavyGuardian* to get a notion of performance improvement over the best sequential CPU method.

CMS Multi-thread CPU. A shared-memory based multi-threaded parallel implementation of *CMS* on CPU. The strategy used for exploiting parallelism here is the first approach mentioned in Section 2.2, i.e., each thread with its local copy of *CMS*, processes different chunks of data. In the end, thread-local copies of sketches get merged to produce final *CMS*.

CMS GPU. Reference implementation of vanilla *CMS* on GPU following the parallelization strategy mentioned in section 2.2.2.

Matryoshka$_{CMS}$ GPU. It is our *Matryoshka* sketching strategy with *CMS* as the base sketch. It uses *head-first scan* to build the heavy-hitter sets in *Matryoshka* sketching. The implementation details are in Section 2.5.

2.6.2 Performance Results

Here, we present throughput based performance evaluation of the methods on the real and synthetic datasets mentioned in Section 2.6.1.3. For consistency of performance comparison, we set the sketch parameters as follows - 1) the number of buckets or b set to 32K and 2) the number of rows or l set to 8. For all the graphs referenced in this section, the error bars represent the standard deviation in performance results. We repeated the experiments

for each scenario 10 times and then took the average and standard deviation of the results. As for execution, both *CMS CPU* and *HeavyGuardian CPU* run sequentially on Intel® Xeon® 8180 CPU. We execute *CMS Multi-thread CPU* with 28 threads on the same CPU. For GPU based approaches, i.e. *CMS GPU* and *Matryoshka$_{CMS}$ GPU*, we use nVidia® Tesla® V100 GPU.

2.6.2.1 Performance on Synthetic Datasets

Figure 2.7 shows the comparison of achieved throughput in sketch update computation on *Zipf* datasets with varying skewness of 1.25, 1.5, 1.75, and 2. For skewness 1.25, *Matryoshka$_{CMS}$ GPU* attains roughly 5.95x, 28x, 251x, and 438x higher throughput compared to respectively *CMS GPU*, *CMS Multi-thread CPU*, *HeavyGuardian CPU*, and *CMS CPU*. With 1.5 skewness, the performance improvement is 14x, 45x, 176x, and 655x over respectively *CMS GPU*, *CMS Multi-thread CPU*, *HeavyGuardian CPU*, and *CMS CPU*. The same throughput improvements become 24x, 69x, 241x, and 925x when we increase the skewness to 1.75. Finally, for the extremely skewed *Zipf* dataset with skewness 2, the respective performance gains are 32x, 82x, 253x, and 1090x. The large improvement gains clearly show the effectiveness of our solution for sketch updates with skewed data on massively parallel architectures.

Furthermore, we see from Figure 2.7, as the skewness increases from 1.25 to 2, the performance of *Matryoshka$_{CMS}$ GPU* increases by a factor of 2.7x. In fact it consistently gives better throughput with increasing skewness. Whereas, the performance of *CMS GPU* monotonically deteriorates as the skewness increases from 1.25 to 2. This empirical evidence backs up our intuition behind the design of our hierarchical sketching strategy, i.e., *CMS GPU* faces higher contention with increasing skewness, while *Matryoshka* exploits the skewness to give better performance. *CMS CPU* and *CMS Multi-thread CPU* both show similar performance over varying skewness because their execution does not depend much on skewness. On the other hand, *HeavyGuardian CPU* shows a small improvement in

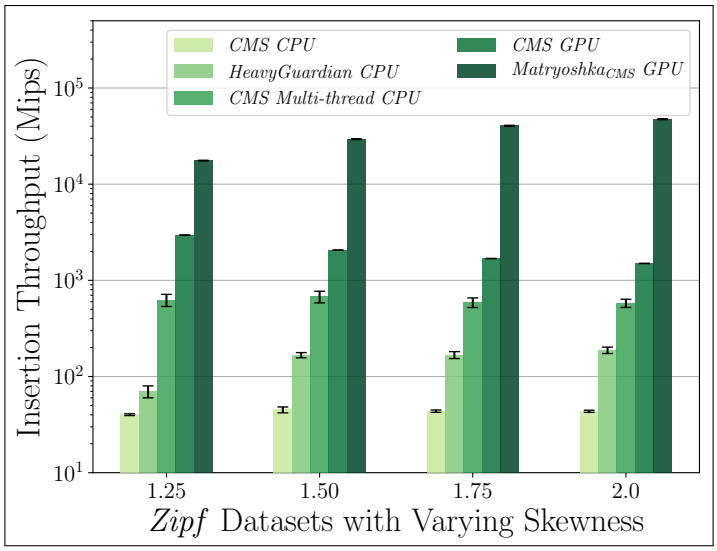


Figure 2.7: Performance comparison for updates on *Zipf* datasets

throughput as the skewness increases since it exploits skewness for performance.

We present the comparison of query processing throughput on *Zipf* datasets with varying skewness of 1.25, 1.5, 1.75, and 2 in Figure 2.8. We consider query tasks as reading an array of elements and returning an array of corresponding estimated frequencies. Since this task involves a very regular access pattern for elements and frequency array and as we are only reading from the sketch (thus no contention for updating buckets), we have the same query kernel for both naive *CMS GPU* and *Matryoshka$_{CMS}$ GPU*. So, in Figure 2.8, we see almost the same throughput for both of them. Now, in comparison to *CMS Multi-thread CPU*, we get 35x, 47x, 53x, and 52x higher throughput respectively for the skewness of 1.25, 1.5, 1.75, and 2. The improvements are respectively 260x, 148x, 211x, and 219x for *HeavyGuardian CPU*, and for *CMS CPU*, we have 551x, 578x, 693x, and 779x increase in throughput. This shows the potential of performance gain from GPU scale parallelism when we have regular memory access patterns as in query processing.

2.6.2.2 Performance on Real Datasets

In Figure 2.9, we present update throughput of the methods on four real datasets, namely *Kosarak*, *Webdocs*, *CAIDA*, and *Criteo*. On *Kosarak* dataset, *Matryoshka$_{CMS}$ GPU* gives

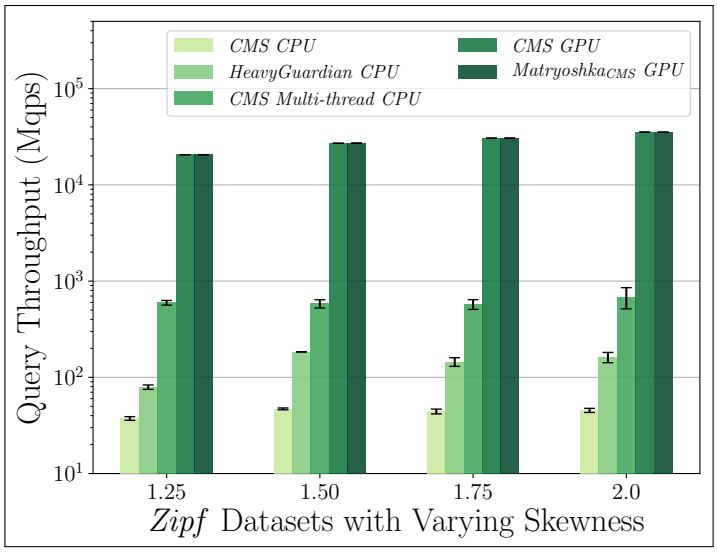


Figure 2.8: Performance comparison for query on *Zipf* datasets

1.96x, 30x, 90x, and 183x higher insertion throughput compared to *CMS GPU*, *CMS Multi-thread CPU*, *HeavyGuardian CPU*, and *CMS CPU* respectively. *Kosarak* is a relatively small dataset and our significant performance gain over other methods show that our solution is quite effective on relatively small size of data. Now, for the same comparison on *Webdocs* dataset, we see throughput improvements of factor 1.5x, 15x, 151x, and 250x from *Matryoshka$_{CMS}$ GPU*. *Webdocs* is a moderately large size dataset among the real datasets we consider in this work and our methods performs well on it, as evident from the throughput results.

If we consider *CAIDA* dataset, we find in Figure 2.9 that, *Matryoshka$_{CMS}$ GPU* attains roughly 1.2x, 12x, 176x, and 193x improvement in update throughput over *CMS GPU*, *CMS Multi-thread CPU*, *HeavyGuardian CPU*, and *CMS CPU* respectively. *CAIDA* is a relatively big dataset, and as mentioned in [19], the heavy-hitters change dynamically over the data length, which corresponds to the time of collection of internet traffic. Due to this dynamic nature of heavy-hitters, our *Head-first Scan* is less effective on this dataset. Hence, we get relatively lower performance gain over *CMS GPU* compared to other datasets. However, we still get large improvements over CPU based methods, including the state-of-the-art *HeavyGuardian*. For *Criteo* dataset, the same comparison shows a stark insertion throughput

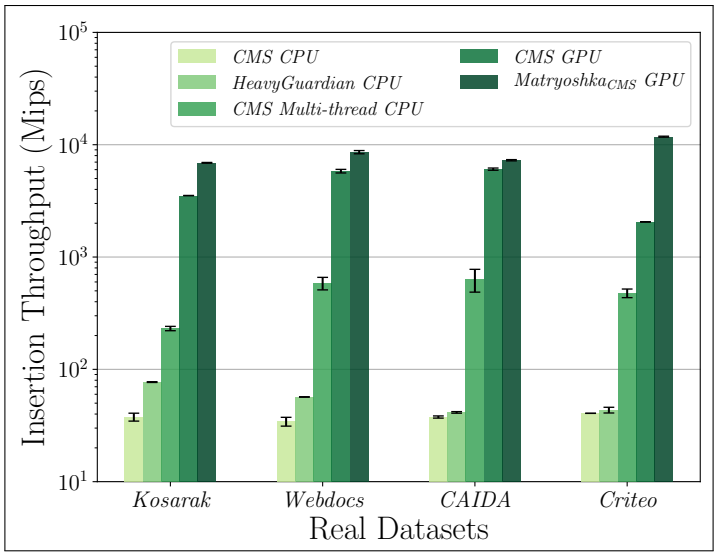


Figure 2.9: Performance comparison for updates on real datasets

improvement of 5.74x, 25x, 272x, and 290x over the respective methods as mentioned in the previous comparison. *Criteo* is also a relatively big dataset and our large performance gain shows that our solution works great on large data if we have relatively small dynamic changes in heavy-hitters.

Figure 2.10 presents the comparison of throughput in query processing phase on real datasets. As mentioned and reasoned in Section 2.6.2.1, we have same query processing kernel in both *Matryoshka$_{CMS}$ GPU* and *CMS GPU*. So, similar to Figure 2.8, we also see almost same query throughput here for both the methods on different real datasets. Compared to query throughput of *CMS Multi-thread CPU*, *HeavyGuardian CPU*, and *CMS CPU* respectively, we get 31x, 61x, and 236x improvement on *Kosarak* dataset, 15x, 67x, and 199x improvement on *Webdocs* dataset, 15x, 183x, and 256x gain on *CAIDA* dataset, and finally, a factor of 25x, 291x, and 444x higher throughput on *Criteo* dataset. This shows the impressive performance gain that can be obtained in query processing by exploiting massive parallelism on GPU.

One important thing to note from all the performance results, even though the state-of-the-art *HeavyGuardian* beats vanilla sequential *CMS CPU* in all the cases, both parallel versions of *CMS*, i.e. *CMS Multi-thread CPU* and *CMS GPU* surpasses it in performance

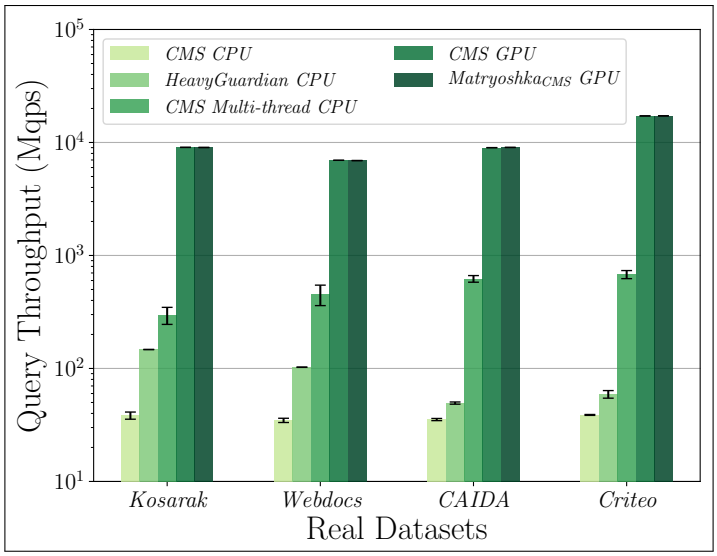


Figure 2.10: Performance comparison for query on real datasets

in a significant manner. We recognize that, from the perspective of errors in frequency estimation, there is a large gap between *HeavyGuardian* and *CMS*. However, from the performance viewpoint, it reveals an important point - the significance of designing parallel algorithms for achieving practical performance on current and future generation computing architectures.

2.6.3 Accuracy Comparison with CMS

Our metric of accuracy measurement is average relative error or *ARE*. Details of this metric is given in Section 2.6.1.4. Figure 2.11 gives the accuracy comparison between *CMS* and *Matryoshka$_{CMS}$ GPU* on *Zipf* data with varying skewness of 1.25, 1.5, 1.75, and 2. Similarly, Figure 2.12 presents the same comparison on four real datasets - a) *Kosarak*, b) *Webdocs*, c) *CAIDA*, and d) *Criteo*. In Section 2.4, we theoretically shown that *Matryoshka* sketching strategy does not incur any extra error, it produces the same error as the base sketch. With the base sketch being *CMS* here, we wanted to validate our theoretical results with empirical study. As both the Figures 2.11 and 2.12 indicate, *Matryoshka$_{CMS}$ GPU* and *CMS* produces same *ARE* for all the synthetic and real datasets. This is a great property since, we do not sacrifice on the accuracy of the base sketch, and yet achieves much higher throughput.

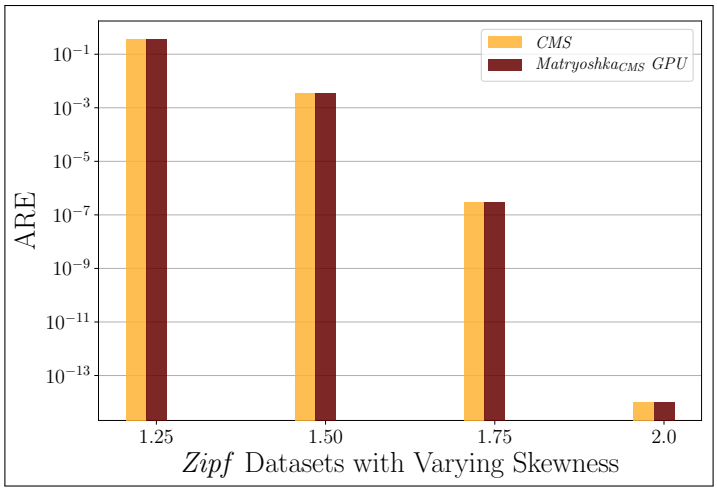


Figure 2.11: Average relative error comparison on *Zipf* datasets

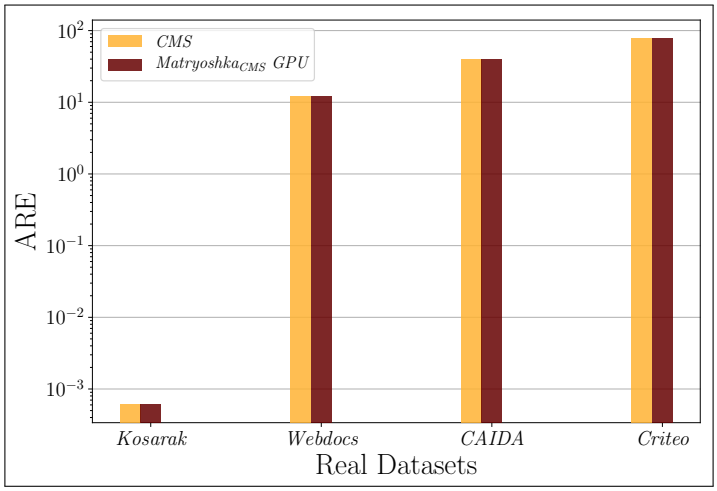


Figure 2.12: Average relative error comparison on real datasets

2.6.4 Effects of Parameters

In this section, we present the sensitivity of our approach on two main sketch parameters, specifically - 1) number of buckets or b and 2) number rows or depth or l.

The results in Figure 2.13 shows the effects we get by varying b. For this experiment, we used the *Zipf* data with skewness factor of 1.5. From Figure 2.13, we see that the *ARE* reduces significantly as we increase b while the insertion throughput remains quite similar. This is helpful since we can reduce the error by increasing b while maintaining the performance.

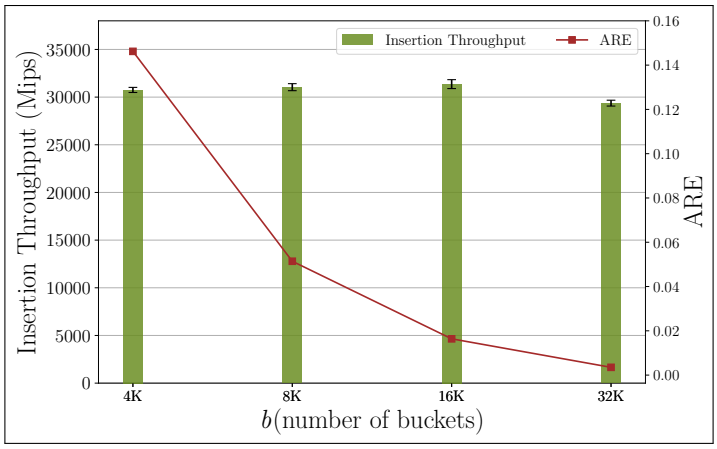


Figure 2.13: Effect of b

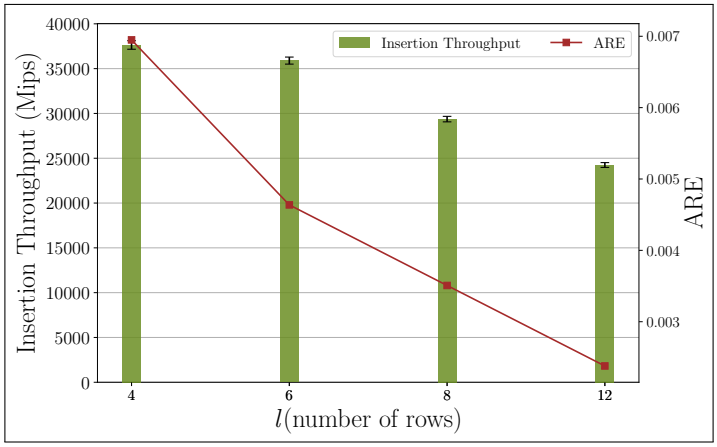


Figure 2.14: Effect of l

Figure 2.14 represents the effect of depth parameter (l) of sketch. We used *Zipf* data with a skewness factor of 1.5 for this experiment too. Here, *ARE* reduces as we increase l while the update throughput degrades with increasing l. As there is a trade-off between accuracy and throughput here, we have to choose a suitable point based on our requirements.

2.6.5 Performance Analysis on GPU

In Section 2.2.2, we showed and discussed inefficiencies in terms of GPU performance metrics in a reference *CMS* implementation on GPU. Now, to give reasoning behind our immense throughput gain, we present the percentage improvement we achieve in terms of various GPU performance parameters by applying *Matryoshka* sketching on *CMS*. The

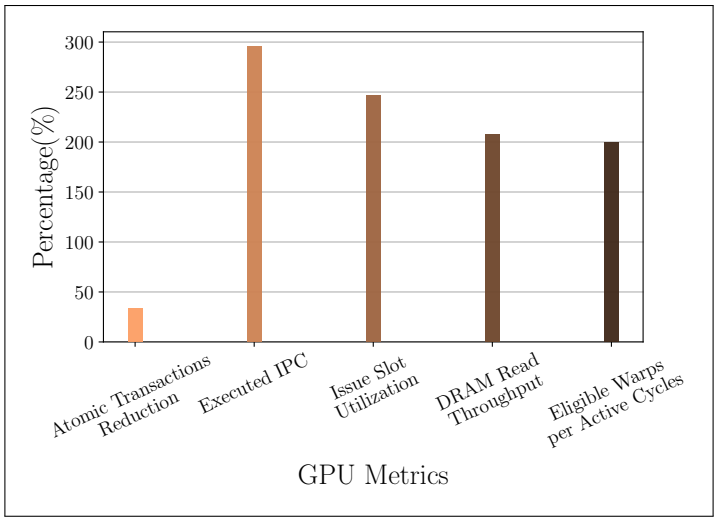


Figure 2.15: Percentage improvement over reference *CMS GPU* in GPU performance metrics

resulting plot is in Figure 2.15. The performance metrics are for the update phase on the *Kosarak* dataset. We collected the GPU performance metrics using *nvprof* on V100 GPU.

"Atomic Transaction Reduction" is derived from the metric `atomic_transactions`. We defined it as the percentage reduction in `atomic_transactions` in *Matryoshka$_{CMS}$ GPU* from *CMS GPU*. `atomic_transactions` refers to serialized access to a shared value for correctness. Reduction in `atomic_transactions` relates to the contention reduction mentioned in Theorem 2.4.4. As we are hierarchically reducing the number threads accessing a shared counter, the number of `atomic` accesses for counter updates becomes lower.

"Executed IPC" is derived from `ipc` metric which refers to *instructions executed per cycle*. This a popular performance metric as it indicates the efficiency of execution. The more `stalls` a computing resource faces, the less `ipc` becomes for the execution. Close to 3x improvement on "Executed IPC" means our *Matryoshka$_{CMS}$ GPU* is far efficient than *CMS GPU* from the execution viewpoint.

"Issue Slot Utilization" refers to the same *nvprof* metric `issue_slot_utilization`. It indicates the number of cycles the warp scheduler issued at least one instruction and relates to how efficiently the units in GPU are being utilized. Almost 2.5x improvement on

this metric shows that our method uses the GPU much more efficiently.

"DRAM Read Throughput" refers to the same *nvprof* metric `dram_read_throughput`, which indicates how much bandwidth of the device memory is being used on average. Figure 2.15 indicates that our *Matryoshka$_{CMS}$ GPU* is able to use twice as much of the memory bandwidth of the `HBM2` device memory of V100 GPU as used by *CMS GPU*.

"Eligible Warps per Active Cycles" is the number of warps that can issue instructions at a given cycle. It has same value as the metric `eligible_warps_per_cycle`. There can be several reasons for a warp becoming stalled (opposite of eligible). However, one main difference between our approach and the reference *CMS* is the efficiency of memory operations. So, a 2x improvement here indicates the effectiveness of our method in the reduction of stalled memory operations.

2.7 Related Works

We give a brief overview of some related works here.

Exploiting Skewness for Accuracy of Sketches. Interestingly, most of the works on accuracy improvement of sketching techniques, take into consideration the skew present in the frequency distribution of elements. For example, Augmented sketch [36] adds a filtering stage before the original sketch data structure to separate *"heavy hitters"* from the rest of elements. Pyramid sketch [37] assigns gradually increasing amounts of additional counters according to the current frequency of an element. So, *"hot items"* get more memory assigned as required while the *"cold items"* take small memory. HeavyGuardian [38] modifies each bucket of the sketch with a heavy part which tracks *"hot items"* and a light part for tracking *"cold items"*. Due to having separate counters for a limited number of *"hot items"* and *"cold items"*, collision is greatly reduced. Learning-Based Frequency Estimation [19] uses learned neural networks (Recurrent Neural Networks) as an oracle to separate *"heavy hitters"*.

Other Frequency Estimation Methods. There are several bloom filter [50] based methods for frequency estimation. These work essentially extend bloom filters from an-

swering set queries to estimate frequencies over multiset (set having repetitions). Popular choices include Counting Bloom Filter (*CBF*) [51], Spectral Bloom Filter [25], and Dynamic count filters [52]. Some "*non-hash*" based methods employ only counters, for example, Randomized Counters [29] and Counter Braids [53]. In this work, we mainly focus on hash-based methods.

CHAPTER 3

TOPKAPI: FREQUENT ELEMENTS FINDING WITH SHARED AND DISTRIBUTED MEMORY PARALLELISM

Counting and identifying frequently occurring items, or "heavy hitters", is one of the most important and intuitive metrics to gain insight into large-scale data. The naive way to extract top-K items from a data stream is to count the exact number of occurrences of each distinct item, then sort the histogram to obtain the most frequent items. This naive but popular approach suffers from a time complexity of $O(n \log n)$, in which n is the total number of elements in the dataset, and also a space requirement of $O(n)$, assuming sorting is performed in linear space. In a distributed environment, where data sharding is common, the problem is quite severe. We have to keep a local frequency histogram on each node, which is usually of size n itself. These local histograms are communicated across before final merging followed by sorting. Thus, each node needs to communicate $O(n)$ sized histogram, which can be prohibitively large. Consider the simple task of keeping track of most popular phrases, of up to 4 words, on twitter feeds. With a vocabulary of over a million 10^6, the total number of items we need to keep track of becomes $n = (10^6)^4 = 10^{24}$. Similarly, counts of the number of clicks on "Amazon.com", given specific user's features and their combinations, in the past hour, are common in clickthrough prediction [54]. $O(n)$ time complexity becomes unacceptably large for "big data".

Fortunately, approximations often suffice in practice. Frequencies in most real word applications follow the Power Law [18], and therefore even approximately knowing the counts are enough to identify frequent items, also known as *heavy hitters*, efficiently. This feasibility for approximations allows for a significant reduction in computational and memory requirements. As a result, approximate counting is a very active and widely studied research area. There has been a remarkable success in obtaining algorithms for finding

heavy hitters with exponential improvements in memory requirements, and a lot is known about the theoretical complexity of these algorithms [20]. Several of these algorithms are deployed in practice. Two notable algorithms include *Count-Min Sketch* (*CMS*) [18] which is hashing based and *Frequent* algorithm (*FA*) [21] which is based on maps (or dictionaries).

However, even after 30 years of research on approximate counting over data streams, developing a practical algorithm that can fully utilize the massive amounts of available parallelism in the form of multi-core and multi-node (or distributed parallelism) is still an active area of research. Prior algorithms, such as [55], only rely on the theoretical reduction in communication, but require synchronized updates, for every increment, making them expensive. In [56], the authors identify mergeable or *reducible* as a critical property that eliminates the need for synchronization. With the *reducible* property, every node can create their summarization of the local data and transmit this exponentially small summary. Each of these little sketches can be merged to obtain the global summary of the data from which global heavy hitters can be identified.

It was argued in [56] that most popular algorithms, including *CMS*, are not suitable for the distributed setting because they lose the *reducible* property, i.e., it is not possible to identify top-K by merging local top-K and their *CMS* summaries. Our experiments (sec 3.6.5) confirm the significantly poor precision for *CMS* in distributed settings. Fortunately, the same paper [56] showed that *FA* is *reducible* and thus suitable for distributed computing. However, *FA* is costly to update which requires operation linear in the size of the summary. Slow updates are also one of the main reason why *CMS*, despite being theoretically inferior, is preferred [18]. On the contrary, *CMS* has only logarithmic update cost, which is desirable, but local *CMS* summaries cannot be combined (not *reducible*). Thus, even if *CMS* is known to be faster than *FA*, it is not a suitable option in distributed setting.

To summarize, the popular hashing based *CMS* has logarithmic update cost but do not have the crucial *reducibility* property required for utilizing massive parallelism. On the other hand, non-hashing based *FA* summaries are reducible, but updates are significantly costly.

In this work, we propose a theoretically sound and superior algorithm which combines both *CMS* and *FA* in a novel way that achieves the best-of-the-both worlds – logarithmic (efficient) updates as well as *reducibility* needed for parallelism. Our experiments show that the new proposal is on average 2.5x faster in practice than *FA* for distributed and multi-threaded execution.

Our Contributions. The problem we address here is to find the identities of top-K frequent items in a given data (formal definition in Section 3.1). For this problem, we present *Topkapi*, a fast and parallel approximate algorithm. 1) *Topkapi* combines *CMS* and *FA* in a novel way that makes the summary *reducible* and at the same time enabling parallelism. 2) We show that *Topkapi* retains the provable probabilistic error guarantees analogous to popular sketching algorithms in the literature. 3) We provide optimized parallel implementations for *FA*, *CMS* and our proposed *Topkapi*. Our implementation is optimized to overlap communication with computation and is capable of exploiting both multi-node and multi-core parallelism effectively. 4) We provide rigorous evaluations, profiling, and comparisons of two popular algorithms *CMS* and *FA* with *Topkapi* on large-scale word counting benchmarks. Our experiments indicate significant performance gains with *Topkapi* compared to existing approximate heavy hitters problem. 5) Our work also provides empirical quantification of the benefits of using approximate algorithms over exact state-of-art distributed implementation in Spark. Our results show disruptive performance gains, with *Topkapi*, over some of the fastest known exact implementations, at the cost of small approximations.

3.1 Problem Statement

We will refer the problem of finding the top-K most frequent items in the data stream as the "top-K problem". Let's assume we have D distributed data streams $\{S_1 ... S_D\}$, for example, D text streams. Let us assume that there are in total M words $\{w_1 ... w_m\}$. Our goal here is to find K most frequent words in these streams as an aggregate, i.e., $\cup_{i=0}^{D} S_i$ where the

union represents concatenation (or aggregation) of the streams. We represent the frequency of a word w by f. Also, let N denotes the summation of all the frequencies, i.e., $N = \sum f$. If the K-th most frequent word has frequency f_K, then we want to report all the words for which $f \geq f_K$.

3.1.1 ϕ-Approximate Heavy Hitters

Several approximate formulations of the *heavy hitter* problem were proposed to overcome the linear memory barrier. We use the standard formulation given in [56]. Given an approximation parameter ϵ, the approximate heavy hitters solution returns a set of words (items) HH that satisfies the following two conditions with high probability $(\geq 1 - \delta)$ - a) All words w having $f > \phi \times N$ is present in the returned set HH and b) every word in the set HH is guaranteed to have $f > (\phi - \epsilon)N$.

We collectively call the algorithms solving this approximation as "approximate algorithms". Approximation breaks the linear complexity barrier and allows us to work with only logarithmic memory, with an insignificant loss in accuracy.

We will interchangeably use the word sketches and summary. They mean the same thing. Approximate algorithms for heavy-hitters produce a summary output which is typically much smaller than the data. This summary can be used to answer the heavy-hitters or other estimation queries.

Since we will be using approximate (lossy) algorithms over distributed clusters, where we will need to merge different summaries from different nodes, we need to define *reducibility* of the summaries (or sketches). *Reducibility* will ensure that the algorithm can be parallelized efficiently. Our definition of *reducibility* is inspired from the definition of mergeability in [56]. However, our definition is simpler and more generic for better readability.

Reducible Summary: Given the output summary O_1 from running algorithm A on data stream S_1 and output summary O_2 with running the same A on data S_2. We call an algorithm reducible if we can recover some summary $\hat{O}$ directly from the two output summaries O_1

and O_2, such that, if we use the combined summary $\hat{O}$ to replace O, which is a summary obtained after running A on $S_1 \cup S_2$, we still retain all theoretical guarantees of algorithm A. In addition, we want two more conditions: – 1) The computation cost of calculating $\hat{O}$ from O_1 and O_2 should be less than the cost of running A over $S_1 \cup S_2$ and 2) The space required by $\hat{O}$ should not be more than that of O.

Note that sometimes the algorithm A, such as *FA* (defined later), is sensitive to the order in which it sees the input data. In such cases, we cannot guarantee that the combined summary O will be equal to $\hat{O}$, but so long as the final outputs have same accuracy guarantees and computation time, we can distribute it efficiently.

3.2 Previous Solutions & Their Limitations

3.2.1 Exact Algorithms

Exactly solving the top-K problem requires $O(M)$ memory and have $O(MlogM)$ runtime complexity. One can compute all the frequencies f using standard word count or histogram computation. Then sort the words based on the frequencies f as the key and report the top-K words. We can utilize hash-maps to store words and update frequencies as we read the data. Finally, we sort the map.

A unique advantage of this exact method is that it is easy to parallelize. We can perform separate hash map updates with separate data in parallel, and at the end, we perform reduction by key to get the final frequencies. Then we sort the words to get the top-K frequent words. Several state-of-the-art implementations, such as Spark based `wordcount() + sort()` use this method. However, our experiments in sec 3.6.6 reveal that $O(M)$ storage and communication, even with the best possible distributed implementation can be orders of magnitude slower compared to approximate solutions in a distributed setting.

3.2.2 Approximate Algorithms

Algorithms for finding approximate heavy hitters is a heavily studied topic in database and theory community. These algorithms mainly come in two flavors - 1) counter-based and 2) sketch-based.

Counter-based Algorithms: Counter-based algorithms maintain a set of counters (maps) associated with a subset of words (or maps with counters) from the data stream it has traversed. This subset of words is called the monitored set. There are several variants, such as Frequent [21], Lossy Counting [57], and Space Saving [58]. Please see [39] for a good survey on them. Note that, [39] explored only sequential version of these algorithms whereas we are mainly interested in parallelism here. In our work, for comparison with counter-based approach in general from the perspective of parallelism, we consider one of the most popular variant – *Frequent Items* or simply *Frequent* algorithm (*FA*). The main advantage of this approach is the summaries are *reducible* whereas the main disadvantage is high update time.

Frequent Algorithm. In 1982, Misra and Gries [59] first proposed a generalization of *Majority* algorithm (finds the most frequent element) to extend it for "top-K problem". The same algorithm was rediscovered in 2002 by Demain *et al.* [60] and Karp *et al.* [21]. We refer to these algorithms by the general term "*Frequent*" algorithm (*FA*). *FA* keeps $(1/\phi)$ number of counters for finding all words with $f > \phi \times N$. During stream traversal, each new word is compared against the monitored set. If the element exists in the monitored set, then its count is incremented. Else, if there is some non-allocated counter, i.e., counter with count zero, then allocate the counter for the new item and set its count to 1. If all counters are already allocated, decrement all counters. In this process, if the count of any counter becomes 0, declare the counter as non-allocated and remove the associated word from the monitored set. As observed by Bose *et al.* [61], setting the number of counter to $(1/\epsilon)$ for *FA* solves the approximate frequency estimation problem. This algorithm is deterministic and achieves optimal theoretical guarantees.

The algorithm requires maintaining a map from strings to integer of size $(1/\epsilon)$. We briefly highlight three important aspects of this algorithm which will be used to contrast it with other algorithms

1. $(1/\epsilon)$ **per Update** The update cost of addition is $(1/\epsilon)$ in the worst case as we have to decrement counters.

2. **Reducible** It was shown in [56] that maps used in *FA* is reducible, and hence can be easily parallelized across multiple nodes.

3. **Map Overheads** To identify the items exactly, we need a map of strings to counters. Addition to maps creates additional overheads of resolving the hash collisions [62].

Sketch-based Algorithms: Instead of maintaining counters for a monitored set of words, sketch-based algorithms use lossy hashes to create a summary which can be used to estimate the frequency of any given item. For this study, we consider one of the most popular and efficient among the sketching algorithms – *Count-Min Sketch* (*CMS*), which is widely adopted in practice. Sketch-based approach provides fast update of summary but has significant disadvantage when it comes to *reducibility* because heap, which is not reducible, is needed for recovering identity of counters.

Count-Min Sketch. The Count-Min Sketch (*CMS*) algorithm proposed by Cormode and Muthukrishnan [18] is inspired by widely popular data structure called *Bloom Filter*[50] which is used for estimating counts of items over data stream while using high level of compression. The sketch is a two-dimensional array M of $l \times b$ counters. Here we use l 2-universal hash functions $h_1, h_2, ..., h_l$ which map words to $\{1, 2, ..., b\}$. These hash functions are pair-wise independent. For each occurrence of word w in data stream, we increment the counter $h_i(w)$, in the i^{th} row for $\forall i : 1 \leq i \leq l$. Any query of frequency estimation $\hat{f}$ of any word w returns $min\{h_i(w) \forall i : 1 \leq i \leq l\}$. [18] proved that the expected error in frequency estimation is always an overestimate $\leq \left(\frac{N}{b}\right)$ and using l hash functions reduces the error

exponentially with l. So, using $l = O(log\frac{1}{\delta})$ and $b = O(\frac{1}{\epsilon})$ ensures the error in frequency estimation is $\leq \epsilon N$ with probability $1 - \delta$.

Since the algorithm only needs lossy hash functions, it does not require a map and can work with arrays. However, as the hash functions are not invertible, sketch-based methods do not preserve the identity of words associated with specific counters. Thus, to identify heavy hitters, we need additional data structures. There are two workarounds – 1) Dyadic interval trick [14] and 2) use of Heaps. The dyadic interval trick requires a tree of individual count-sketches and the memory overhead of tree is prohibitive in practice. The common workaround is to maintain a heap of top-K words along with the sketch while reading the data stream. We will focus on this practical variant.

The sketch keeps track of the number of words processed so far (n). For each word w in the data stream, we first update the sketch and then query the frequency estimation $\hat{f}$ of that word. If $\hat{f} \geq \phi \times n$, we search the word in heap. If the word already exists in a heap, then we update its count. Otherwise, we insert the word to the heap. If the heap is already full, we check if $\hat{f}$ is greater than the min count in a heap. If so, we do delete-min on the heap and insert the word.

Count-Min Sketch with Heap has the following key properties

1. $\max(\log\frac{1}{\delta}, logK)$ **per Update:** The update cost only requires adding to $\log\frac{1}{\delta}$ counters. If we need to update the heap, it requires additional $\log K$ operation. The total cost is logarithmic and hence significantly smaller than $(1/\epsilon)$ in practice.

2. **Not Reducible:** Since only the identities of top-K items are stored in a heap, we cannot merge top-K over two different streams to obtain the global top-K.

3. **Heap Overheads:** Although, the sketch only consists of arrays, and 2-universal hash functions are cheap, to identify top-K items we have to use the heap data.

Although there has been a significant development in past years on approximate heavy hitters [20, 15, 18, 57, 60], little focus has been given on the parallelism aspects except a

very few, such as [63, 64, 65, 66]. When it comes to parallelism, there are several choices. Parallelizing the individual updates is not a good option as the computation is too low to justify parallelism. Data parallelism, i.e., performing computation for different blocks of data in parallel, is more preferred because we have a much better granularity of parallelism. Thus, with enough data, it is always preferred to have each parallel worker work on its own summary and later perform a one-time merge. We also get a very high degree of parallelism due to the large size of the data. Thus, it is essential for the algorithm to be *reducible*. However, with data parallelism, the algorithmic update time becomes a factor with a significant impact on performance. [63, 64, 65] discuss parallel counter-based Space Saving [58] algorithm over CPU, GPU, and distributed environment respectively. However, none of them addresses distributed environment with multi-threading. Also, we can see in [64] that the counter-based approach has significant update time even on massively-parallel architecture such as GPU. Interestingly, [66] explored fine grain parallelism to speedup Space Saving on modern CPUs with advanced vector instructions. This kind of exploitation of fine grain parallelism is complementary to coarse grain parallelism which is the main focus of this work.

3.3 Our Proposal: Topkapi

3.3.1 Intuition

Consider the *CMS* matrix M (sec 3.2.2) without the overhead of updating the heap for identifiability. Note that every row of this matrix is a simple hashed counter, and all rows are independent. Thus, without the heaps, *CMS* are *reducible* summaries, i.e., different summaries with the same hash functions can be merged by simply adding the sketches. The update time is mere $\log \frac{1}{\delta}$ (δ is failure probability) which is also the number of independent hash functions needed. Following [26], in all our experiments, only 4 hash functions suffice in practice. An important observation is that the sketch matrix M is enough to estimate the counts of any given item accurately but cannot identify the frequent items on its own.

Thus, without identifiability, we need another pass over every item, estimate its count, and then report top-K. Given the number of unique items is astronomical, this is prohibitive. However, if we can somehow efficiently identify a small enough set of candidates CS which likely contains the most frequent elements then we just have to check every element in CS, instead of all the items.

It should be noted that due to simple hashing, every cell of *CMS* will count the total occurrence of a small set of items (ϵN in expectation where ϵ is approximation parameter). If a heavy hitter item HH with $f \geq \phi \times N$ hashes to this counter, it is very likely to be the most frequent item in the cell. Thus, if we can identify the heaviest element in the subset of stream in every cell efficiently, then there is hope of getting a good enough candidate set CS.

FA keeps the identity of the heavy hitters in a map. The update time is equal to the size of the map, which needs to be $\frac{1}{\epsilon}$ for reporting all the heavy hitters. However, if we are interested in just the heaviest item, then we don't need maps and the update time will be constant. We just need two cells; one stores the identity of the heaviest element and another a counter to increment/decrement.

The above observations form the basis of our proposal. We propose to associate a *FA* summary of size 1 to each counter of *CMS*. We later show that it has sound theoretical guarantees analogous to *CMS* for solving approximate heavy hitters problem. Furthermore, this modification eliminates all the issues mentioned in sec 3.2.2.

3.3.2 Topkapi: Algorithm Descriptions

Topkapi contains a *CMS* summary, i.e., a two-dimensional array $l \times b$ M. As a reminder, b represents number of buckets for a hash function and l represents the number of hash functions. We have l pair-wise independent hash functions $h_1, h_2, ..., h_l$ to map words to the range $\{1, 2, ..., b\}$. b is set to $(\frac{1}{\epsilon})$ and l is set to $log\frac{2}{\delta}$. Now, each cell $M_{i,j}$ has in addition two more components: - 1) $LHHcount_{ij}$ representing the count of frequent item associated

with M_{ij} (Local Heavy Hitter count) and 2) LHH_{ij} containing the word (identity) whose frequency is stored in the $LHHcount_{ij}$. This LHH_{ij} will ideally be the most frequent item mapping to M_{ij}. Note, each item is mapped to l cells in M.

During initialization, all the $LHHcounts$ as well as M are set to 0. During processing of data stream, we do the usual update of M, the *CMS*. In addition, for each word w, we compare w with the LHH of the cell at $h_i(w)$. If it matches, then we increment the corresponding $LHHcount$ of the cell at $h_i(w)$. Otherwise, we decrement the $LHHcount$. If the decrement causes the $LHHcount$ to become 0, then we replace the LHH of $h_i(w)$ with w and set the corresponding $LHHcount$ to 1. We do this $\forall i : 1 \leq i \leq l$.

In the end, we consider the union of all the unique LHH values as the candidate set CS. We estimate their counts using the *CMS* and finally report all elements with the count higher than some threshold like $\phi \times N$ for ϕ-heavy hitters problem.

3.3.3 Topkapi: Properties

Here, we summarize the main algorithmic properties of *Topkapi*. Detailed theoretical analysis of *Topkapi* is given in sec 3.4. An important thing to note here is that we do not require any heap for *Topkapi*.

1. *Topkapi* with size $l = \log[\frac{2}{\delta}]$ and $b = \frac{1}{\epsilon}$ solves the ϕ-approximate heavy hitter problem provided ($\epsilon < \phi$).

2. *Topkapi* data structure is reducible. As a result, *Topkapi* can exploit parallelism easily.

3. *Topkapi* data structure has update cost of $\log \frac{2}{\delta}$ which is similar to logarithmic update cost of *CMS*.

It is noteworthy to mention that if we want to get the frequency estimates along with the identities of top-K frequent elements, we can use both *CMS* count (overestimates) and *LHH* count (underestimates) to take an average and decrease the error constants, else we can always use the estimate from *CMS*. So, we are strictly better.

Algorithm 3: Topkapi

Data: Input text stream S, parameter K
Result: top-K frequent words in HH

$b \longleftarrow \lceil \frac{1}{\epsilon} \rceil$
$l \longleftarrow log\frac{2}{\delta}$
$C \longleftarrow l \times b$ *counters*
$C[i][j].LHHcount \longleftarrow 0 \quad \forall i \in \{1, 2, .., l\}$ *and* $\forall j \in \{1, 2, ..., b\}$
for $w \in$ *stream* S **do**
    for $i \in 1, 2, ..., l$ **do**
        calculate $h_i(w)$
        if $C[i][h_i(w)].LHH == w$ **then**
            $C[i][h_i(w)].LHHcount \longleftarrow C[i][h_i(w)].LHHcount + 1$
        else
            $C[i][h_i(w)].LHHcount \longleftarrow C[i][h_i(w)].LHHcount - 1$
            if $C[i][h_i(w)].LHHcount == 0$ **then**
                $C[i][h_i(w)].LHH \longleftarrow w$
                $C[i][h_i(w)].LHHcount \longleftarrow 1$
for $j \in 1, 2, ..., b$ **do**
    if $C[1][j].LHH$ *OR* $C[i][h_i(C[1][j].LHH)].LHH > Threshold \; \forall i \in \{2, .., l\}$ **then**
        CS.`insert`$(C[1][j])$
`sort`(CS) *in descending order of* $LHHcount$
`report` LHH *of* CS *entries with top* K *highest* $LHHcount$

3.3.4 Practical Considerations

In *Topkapi*, the only use of *CMS* counters in M is estimation. It turns out that in practice $LHHcount$ itself is also a good estimator of the true frequency of LHH. This is because we are using *FA* summary of size 1 on a tiny stream. Thus, if our goal is only to get the identities of top-K frequent elements, we can altogether get rid of *CMS* counters and reduce the memory overhead significantly.

Finally, towards the end, instead of considering all the unique LHHs, we can be little smarter. Note that every item is mapped to every row and all the rows are independent. The idea is to perform a linear scan over only the 1st array ($l = 1$) of counters and add LHH into CS if the corresponding LHH is greater than a threshold in any of the l rows. Then we sort the candidate set CS to identify top-K candidates according to their $LHHcounts$ and report the LHHs associated with highest $LHHcounts$. Pseudocode of this practical version of *Topkapi* is given in Algorithm 3. We will use this algorithm in experiments.

3.4 Topkapi: Theoretical Analysis

Before we argue about *Topkapi*, we review one useful known theoretical fact about *CMS* which we will use in the proofs.

Theorem 3.4.1. *For every w with frequency f and its estimate $\hat{f}$ using CMS of size $l = \log[\frac{1}{\delta}]$ and $b = \frac{1}{\epsilon}$, we have the following with probability $1 - \delta$*

$$f \leq \hat{f}^{CMS} \leq f + \epsilon N \tag{3.1}$$

Note, we need $l = \log[\frac{1}{\delta}]$ to ensure the above for all N after union bound.

Using the theorem above, we can show the following for *Topkapi*.

Theorem 3.4.2. *Topkapi with size $l = \log[\frac{2}{\delta}]$ and $b = \frac{1}{\epsilon}$ solves the ϕ-approximate heavy hitter problem provided ($\epsilon < \phi$) (Definition in sec 3.1.1)*

Proof: Follows from two lemmas below combined with the definition of approximate heavy hitters instance.

Lemma 3.4.3. *Topkapi with* $l = \log[\frac{2}{\delta}]$ *and* $b = \frac{1}{\epsilon}$ *(*$\epsilon < \phi$*) misses to report* w *with* $f \geq \phi \times N$ *with probability at most* $\frac{\delta}{2}$

Proof: w is missed if it is not in $h_i(w).LHH \ \forall i$. For any i, $h_i(w).LHH \neq w$ implies that the *CMS* counter for $h_i(w)$ given by $M_{i,h_i(w)}.CMScounter \geq 2f$, otherwise local *FA* summary will not miss w. Thus, w is not reported by any of the i rows implies $h_i(w).CMScounter \geq 2f \ \forall i$. Since the *CMS* estimate is the minimum of all i rows, it means the estimate of *CMS* is at least $2f$ or $\hat{f}^{CMS} > f + f > f + \epsilon N$ which happens with probability at most $\frac{\delta}{2}$ from Theorem 3.4.1.

Lemma 3.4.4. *Topkapi with* $l = \log[\frac{2}{\delta}]$ *and* $b = \frac{1}{\epsilon}$ *reports* w *with* $f \leq (\phi - \epsilon) \times N$ *with probability at most* $\frac{\delta}{2}$.

Proof: We report w only when its estimate $\hat{f}^{CMS} \geq \phi N$. Thus, if we report w and $f \leq (\phi - \epsilon) \times N$, it implies that $\hat{f}^{CMS} \geq \phi N \geq f + \epsilon N$. Thus, the error of *CMS* estimate exceeds ϵN which happens with probability at most $\frac{\delta}{2}$ from Theorem 3.4.1.

The following is immediately clear from the description of the algorithm

Theorem 3.4.5. *Topkapi data structure has update cost of* $\log \frac{2}{\delta}$.

Finally, we can easily show that *Topkapi* is reducible

Theorem 3.4.6. *Topkapi data structure is reducible.*

Proof: The counters in *CMS* is reducible, and furthermore, *FA* is reducible. The proof follows from the fact that every cell of *Topkapi* (*CMS* counter and *FA* of size 1) is reducible.

3.5 Implementation

It is imperative that we use multi-core parallelism along with distributed parallelism to make effective use of current and future computing systems.

3.5.1 Multi-core Parallelism

When considering intra-node parallelism using multi-threaded execution, we have several options for *Topkapi*. We can use different threads for different hash functions in $\{h_1, h_2, ..., h_l\}$. However, this limits the number of threads to the number of hash functions which is usually quite low. Another option is to use different threads to process different chunks of data and use a single sketch shared across different threads. The threads will then have to use locks or atomic variables to perform the shared update of counters in the sketch. The use of locks or atomic variables can create significant contention due to the distribution of word frequencies. As the heavy hitters are most frequent, it is highly likely that many threads encounter the same heavy hitter word and try to update the same counter in the sketch.

We can mitigate the problems mentioned in the previous options by exploiting high level of data parallelism at the cost of extra local memory. We can create thread-local copies of sketches and use different threads to process different chunks of data.Then we exploit the reducibility property of the sketch and merge the thread-local sketches at the end of the data traversal to produce a single sketch for a node. We observe that even for a large dataset, we only need a small sketch. For example, with $l = 4$ and $b = 1024$, the size of the *count* array is 16KB and the size of the *id* array is 64KB. So, the amount of extra memory required is quite low. As different threads are working on their own local copies of the sketch, we do not need locks to update a counter anymore.

3.5.2 Distributed Parallelism

Since our algorithm is *reducible*, distributed parallelism is quite straightforward. We start with multi-threaded execution of *Topkapi* on each node following the method mentioned in sec 3.5.1. When we have the final summaries ready at each node, we perform a parallel reduction or merging of the summaries to get a final summary at the root node. Once we have that, we use the final summary at the root node to perform the potential top-K candidate set (CS) construction, `sort` CS, and `report` top-K words steps from the sequential *Topkapi*

pseudocode mentioned in Algorithm 3.

Communication cost - One important factor considering distributed computation is the communication overhead. The communication traffic for merging summaries between two nodes is the size of a single summary. As we use a parallel reduction strategy to merge the summaries at different nodes, we perform $logD$ such merging steps between different pairs of nodes, where D is the total number of nodes.

Overlapping Communication with Computation - In distributed computing, one can hide some of the communication overhead by carefully coordinating the communication so that it overlaps with the computation. In our implementations, we also exploit such opportunities. The reduction algorithm merges all the counters of a summary independently, i.e., a merged counter only depends on the respective two counters from the two summaries being merged. Hence, we can overlap the communication for a specific row of b counters with the computation of merging the previous rows of b counters. We use MPI non-blocking communication to achieve this overlapping.

Algorithm 4: Topkapi_Parallel(S[][], K, N, T)

```
1 for i ∈ nodes N do
2     for j ∈ threads T do
3         create thread local copies of Topkapi summary;
4         execute Topkapi for data S[i][j] in parallel using summary j with only the
            summary update phases;
5     merge thread local summary_j ∀j ∈ {1, ..., T} to produce node final
        summary_i;
6 use parallel reduction strategy to merge node final
    summary_i ∀i ∈ {1, ..., N} to produce a final summary at root node;
7 construct CS using final summary at root node;
8 sort CS and report top-K words from root node
```

For an overview of distributed and multi-threaded implementation of *Topkapi*, we present the pseudocode in Algorithm 4 which extends the pseudocode from Algorithm 3.

3.5.3 Parallelizing Baselines: Frequent Algorithms and Count-min Sketch

For the purpose of performance comparison, we choose the two most popular algorithms, namely "*FA*" and "*CMS*" as representatives from counter-based algorithms and sketch-based algorithms respectively.

As mentioned in sec 3.2.2, *CMS* requires a heap for finding top-K and is not *reducible*. Due to this exact reason, [56] instead used *FA* for mergeability. Unfortunately, without *reducibility*, it is hard to exploit massive data parallelism independently, and the implementations are unlikely to be efficient. We made a simplifying assumption that each subsample of the stream is uniformly distributed and hence merging two top-K still make sense.

There were two main quest behind making this dumb assumption with *CMS*. **1) Does Reducibility Matters in Practice?** Subsampling streams is one of the most popular ways of reducing computation. The assumption is that the frequent item in the whole stream is also a frequent item in any small subsample of the stream. If this holds, then merging top-K across substreams should be possible and reducibility may not matter much in practice for accuracy. We aimed to check this hypothesis. **2) In the most lucky world, is CMS still the fastest?** *CMS*, even with heaps, has significantly faster update time compared to *FA* (experimental results in Figure 3.6). Can *Topkapi* beat this cheap *CMS* variant on performance?

Thus, to understand the performance benefits, we ignored the accuracy aspect and merged the heaps. To merge the heaps, we perform naive merge where we take two heaps and sort them to make a final heap containing top-K candidates. One can argue that increasing the heap size (e.g., $2K$) would improve the accuracy of *CMS*. So, we give *CMS* more room to get better accuracy by using a heap size of $4K$. It should be noted that only the sketch (counters) in *CMS* is *reducible* and the reduction is performed similarly as *Topkapi*.

3.6 Evaluations

3.6.1 Code and Experimental Setup

The implementations of our algorithm[1]and competing algorithms are in C++ under a common framework to ensure as much of an apples-to-apples comparison as possible when presenting relative performance results. We would like to mention that we have used a heap size of ***4K*** for *CMS* to allow better accuracy since the heap containing top-K frequent words lacks the *reducibility* property. We used MurmurHash3 [67] for hash functions in all of the implementations to maintain comparability across different algorithms.

We compiled all codes using GCC 6.2.0 with the following flags: a) GNU C++11 extension, b) "O3" optimization flag, and c) OpenMP flag because we used OpenMP for multi-threading inside a node. We also used Boost 1.64.0 and OpenMPI 1.10.3 libraries for our code. To evaluate performance scalability for multi-node distributed computing with multi-threaded execution on each node, we ran many of our experiments on cluster of Intel®Westmere nodes with 12 processor cores per node running at 2.83 GHz. All of these nodes are connected via QDR InfiniBand (40 Gb/s) to each other. We used 8 threads per node for all of these experiments. Further, to show performance scalability in executions with large numbers of threads, we ran our experiments on a cluster of IBM POWER7®(P750) processors with 32 cores per node running at 3.8 GHz. IBM POWER7®processor supports 4-way SMT (simultaneous multi-threading) which let us launch up to 128 hardware threads per node.

3.6.2 Performance Metrics

Here, we define the performance metrics used in our work and also in past work:

Precision - The metric "Precision" here represents the ratio of number of correct top-K frequent words reported to the total number of words reported.

[1]https://github.com/ankushmandal/topkapi.git

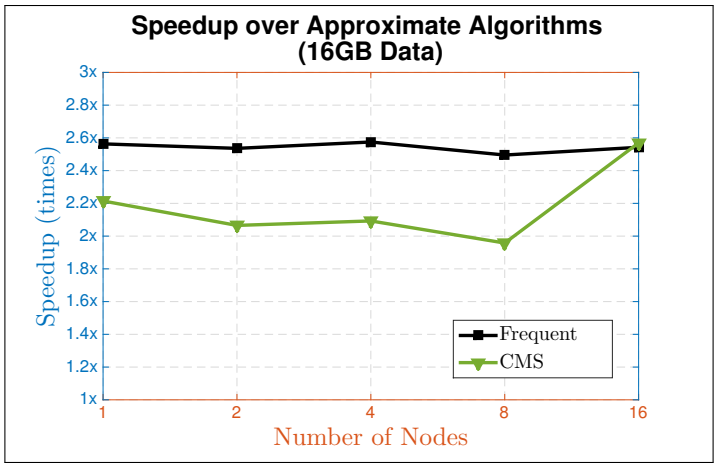


Figure 3.1: Performance comparison with *FA* and *CMS* for 16GB data. Number of threads per node is 8. Used a cluster of Intel®Westmere processors with each node having 12 cores.

Speedup - When we say performance "Speedup", we refer to the following ratio:

$$\frac{execution\ time\ of\ referred\ algorithm}{execution\ time\ of\ Topkapi}$$

3.6.3 Datasets

We give a thorough performance evaluation on standard large-scale word counting benchmarks evaluating all possible aspects of the algorithms. We used two sources to compose our data of different sizes:

Gutenberg - This is text data from the Project Gutenberg [68] corpus. The data consists of text from eBooks in the English language. The data used in our experiments of size up to 16GB are from this source.

PUMA Dataset - We also used "Dataset2" of size 150GB under description "Wikipedia" from PUMA Datasets [69]. We created data of size 32GB, 64GB, and 128GB from this data set to use in our experiments.

The task is to identify the top-100 most frequent words in the data, i.e. we use ***K*=100** for all the experiments unless otherwise stated explicitly.

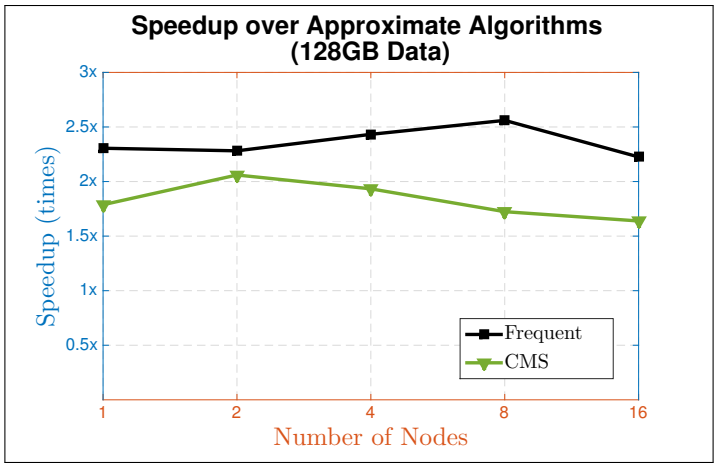


Figure 3.2: Performance comparison with *FA* and *CMS* for 128GB data. Number of threads per node is 8. Used a cluster of Intel®Westmere processors with each node having 12 cores.

3.6.4 Performance Comparison with Approximate Methods

3.6.4.1 Scalability over Number of Nodes

We present strong scaling (fixed data size) performance results over varying number of nodes for two different data sizes: a) 16GB (Gutenberg dataset) and b) 128GB (Puma dataset). Figure 3.1 and Figure 3.2 represents the speedup of *Topkapi* over *Frequent*(*FA*) and *Count-Min Sketch*(*CMS*) for 16GB and 128GB data sizes respectively for 1 to 16 nodes with each node running 8 threads. We see that our proposal consistently get roughly 2.5x speedup over *FA* for both the data types whereas we usually get sightly lower speedup over *CMS*. It should be noted that we used the dumb merging of top-K heap for *CMS* which loses significant accuracy (see sec 3.6.5). Despite this cheap approximation with *CMS*, we still observe 2x-2.6x speedup for 16GB data and 1.6x-2x speedup for 128GB data over *CMS*.

3.6.4.2 Scalability over Number of Threads

Figure 3.3 represents the performance improvement of *Topkapi* over *FA* and *CMS* for 1 to 64 threads on a single node with 32 cores. We used 16GB data for this experiment. The plot shows that we get around 2x speedup over *CMS* for all the data points whereas we get similar performance improvement over *FA* till 8 threads; after that speedup over *FA* increases steeply

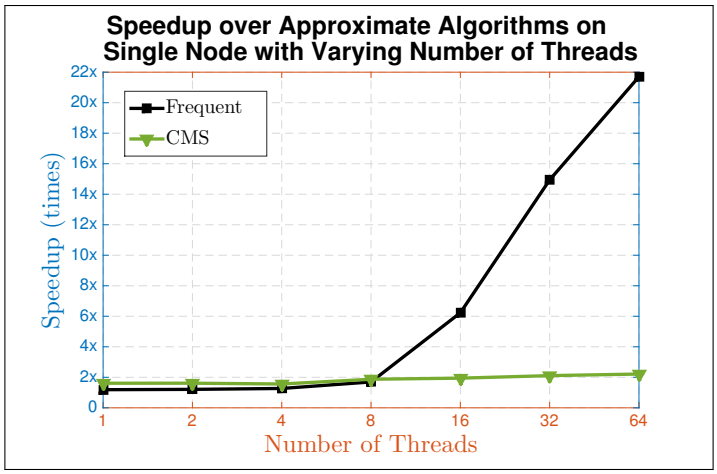


Figure 3.3: Performance comparison with *FA* and *CMS* for varying number of threads. Data Size=16GB and Number of Nodes=1. Used a single node with 32 cores from four IBM Power®7 chip.

and we get 22x speedup with 64 threads. As an optimized implementation of *FA* requires two hash-maps with size being in the order of number of counters, the memory footprint of *FA* is quite high. This negatively affects the performance after a threshold when L3 cache can not contain all the data footprint of two or more threads in the same processor chip. This performance degradation becomes more pronounced when more than one hardware thread is executed on the same core. For example, the configuration with 64 threads uses the SMT feature of Power®7 and executes 2 threads on each core.

3.6.4.3 Scalability over Data Size

To see the effects of data size on performance, we fix the number of nodes to 8 and vary the data size from 16GB to 128GB. The resulting plot with speedup over *FA* and *CMS* is given in Figure 3.4. The figure represents around 2.5x speedup over *FA*, and 1.5x-2x speedup over *CMS*. Beside these good performance improvements, the consistency of speedup indicates that *Topkapi* performs well for a wide range of data sizes.

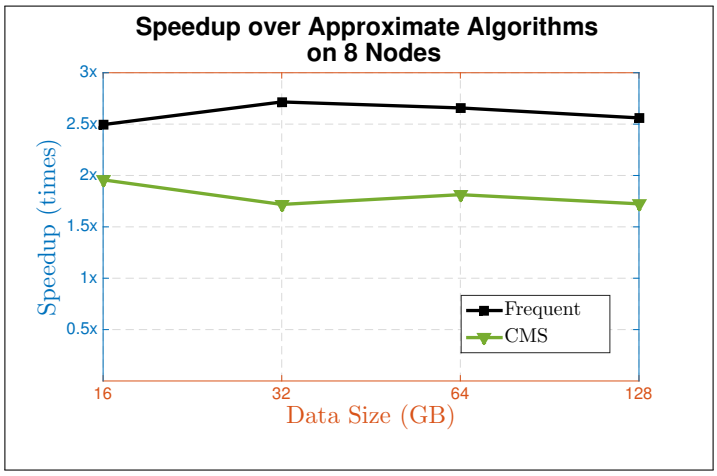


Figure 3.4: Performance comparison with *FA* and *CMS* for varying data size on 8 nodes. Number of threads per node is 8. Used a cluster of Intel®Westmere processors with each node having 12 cores.

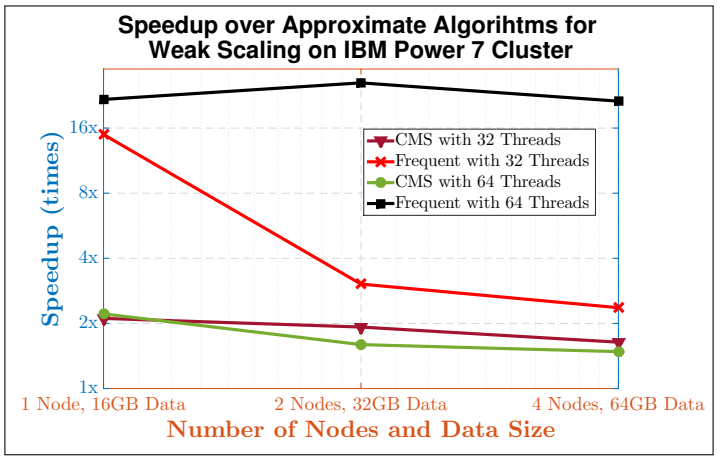


Figure 3.5: Performance comparison with *FA* and *CMS* for high number of threads (32 and 64) in distributed setting. Used a cluster of IBM Power®7 processors where each node has 32 cores from four processor chips.

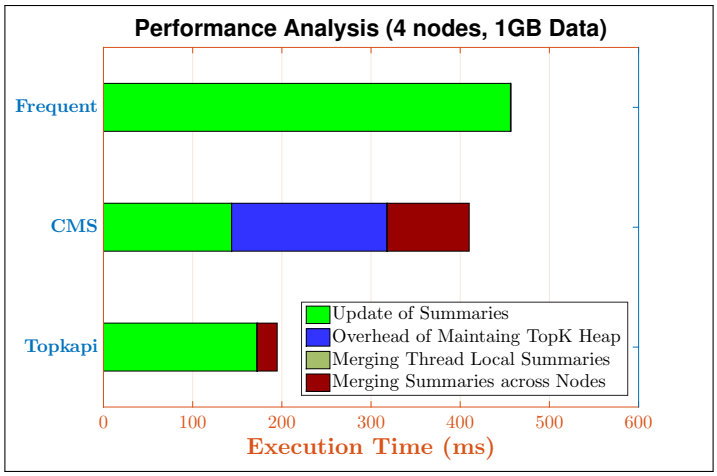


Figure 3.6: Execution time break down for *Topkapi*, *FA*, and *CMS* for 4 nodes and 1GB data size. Number of threads per node is 8. Used a cluster of Intel®Westmere processors with each node having 12 cores.

3.6.4.4 Scaling over Number of Nodes with Increasing Data Size

Now, we increase the data size along with the number of nodes and use high number of threads (32 and 64 threads) on each node to find out how we perform in terms of weak scaling. Figure 3.5 presents the resulting plot. As we can find from the plot, we get consistent speedup of roughly 2x for *CMS*. However, we see some interesting pattern for *FA*. For 32 threads, the speedup over *FA* decreases significantly as move from one node to 2 nodes setting. On the other hand, the speedup remains high (more than 16x) for 64 threads through out all data points. In case of *FA*, the merging of summaries has lower computational overhead compared to *CMS* and *Topkapi*. So, when we move to distributed setting with 2 or more nodes, it boils down to which factor has more impact - the performance gain from low overhead merging step or the performance degradation from high level of multi-threading.

3.6.4.5 Performance Analysis

Figure 3.6 represents the performance break down of *Topkapi*, *FA*, and *CMS* execution. The plot supports our analysis that *FA*, among all three algorithms, has the highest update time for the summary but lowest cost when it comes to merging summaries across nodes.

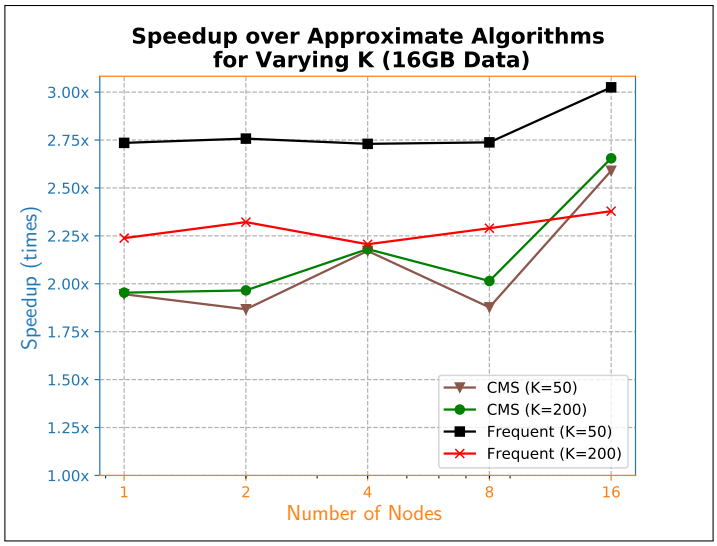


Figure 3.7: Performance comparison with *FA* and *CMS* for K=50, 200 on 16GB data. Number of threads per node is 8.

Undoubtedly, *CMS* has lowest update time for the summary because it involves only calculating the bucket through hashing and then incrementing the respective counter. However, its performance for "top-K problem" is highly thwarted by the overhead of maintaining probable top-K words summary. So, the effective update time for *CMS* becomes quite high. While *Topkapi* needs a slightly higher update time than *CMS*, its effective update time is much lower because it does not involve any overhead from maintaining heap. Furthermore, *Topkapi* has quite low computational cost for merging summaries across nodes whereas *CMS* has the highest cost in this regard.

3.6.4.6 Performance over Varying K

We carried out the experiments related to Figure 3.1 for K=50 and K=200, and represented the results in Figure 3.7. We used 512 and 2048 buckets or counters respectively for K=50 and K=200. Speedup of *Topkapi* over *FA*, for K=50, increases to the range 2.73x-3.01x and for K=200, it decreases to 2.21x-2.36x compared to K=100. However, the speedup over *CMS* remained almost the same. When K is smaller, *FA* should slow down since it now has a lesser number of counters ($1/\epsilon$ or O(K)) or tracked elements. So, it will more

Table 3.1: Precision Comparison between Approximate Methods

	Precision(%)			
Data Size	*Topkapi* (1024 Counters)	*CMS* (1024 Counters)	*CMS* (2048 Counters)	*FA* (1024 Counters)
16GB	96	64.4	68.33	87
128GB	95	11.6	49.66	94

frequently perform the computation related to element not found, which is costly. For the same reason, *FA* will be faster when K is larger. For each match, it only has to increment the corresponding counter, which is cheap. On the other hand, we do not expect the performance of *Topkapi* and *CMS* to change much apart from slight slowdown with increasing sketch size.

3.6.4.7 Comparing CMS with Separate top-K Pass

In batch processing environment, one may employ a two-pass algorithm where the first pass consists of pure *CMS* to get frequency estimates and a separate second pass for hash-based top-K identification. In our experiments using 1 to 16 nodes (8 threads on each node) with 16GB data, we find that the execution time of this two-pass algorithm is on an average 0.97x of single-pass *CMS*+heap based approach. It is noteworthy to mention that the comparison is not fair since in a streaming setting, remembering the items itself, for the second pass, is of linear cost which is prohibitive.

3.6.5 Precision for Reported top-*K*

As *Topkapi* is *reducible*, it is expected to give good precision and Table 3.1 shows us exactly the same thing. *Topkapi* outperforms *CMS* and *FA* for precision over 16GB and 128GB data. Moreover, the poor precision observed for *CMS* indicates that the simplification we assumed in sec 3.5.3 to favor better performance for *CMS* does not hold true.

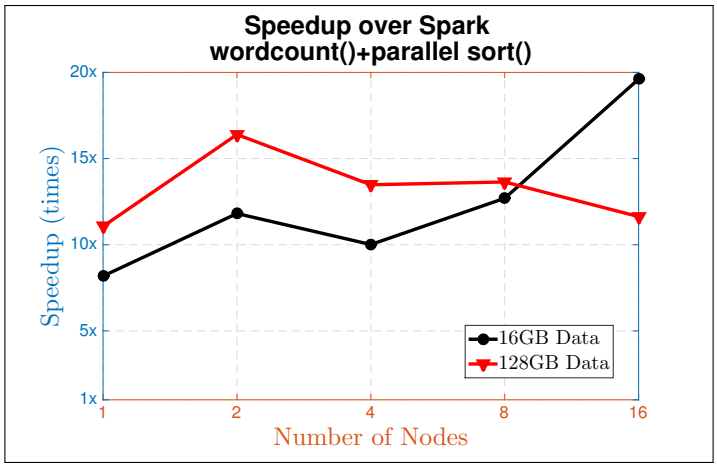


Figure 3.8: Performance comparison with *Exact Method* - `Spark wordcount() + parallel_sort()` for 16GB and 128GB data. Number of threads per node is 8. Used a cluster of Intel®Westmere processors with each node having 12 cores.

3.6.6 Performance Comparison with Exact Method

Here, we compare the performance of *Topkapi* against "exact methods" which give completely accurate results at a cost of linear memory space and communication. Representative from this class of algorithms, we select the popular *Spark wordcount() + parallel_sort()* method.

3.6.6.1 Scalability over Number of Nodes

We present strong scaling (fixed data size) performance results over varying number of nodes for two different data sizes: a) 16GB (Gutenberg dataset) and b) 128GB (Puma dataset). Figure 3.8 gives the overview of speedup variation of *Topkapi* over *Spark wordcount() + parallel_sort()* method for 1 to 16 nodes with each node running 8 threads. As expected, we see significant speedups across the board. *Topkapi* gives 8x-20x speedup over *Spark wordcount() + parallel_sort()* method for both the data sizes. In this case, the costly sorting step associated with the exact method incurs a huge performance penalty.

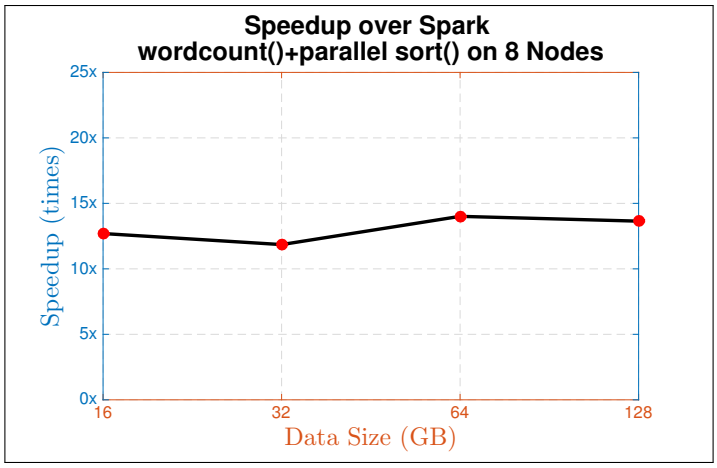


Figure 3.9: Performance comparison with *Exact Method* - `Spark wordcount() + parallel_sort()` for varying data size on 8 nodes. Number of threads per node is 8. Used a cluster of Intel®Westmere processors with each node having 12 cores.

3.6.6.2 Scalability over Data Size

To see the effects of data size on performance, we fix the number of nodes to 8 and vary the data size from 16GB to 128GB. The resulting plot with speedup over *Spark wordcount() + parallel_sort()* is given in Figure 3.9 which represent 10x-15x speedup over *Spark wordcount() + parallel_sort()*.

CHAPTER 4

WORD EMBEDDING WITH EFFICIENT FINE GRAIN PARALLELISM

In this era of Artificial Intelligence (AI), enabling machines to understand human language is one of the crucial tasks. The applications associated with such goal belongs to the fields of either Natural Language Processing (NLP) or Machine Learning (ML) or some intersection of them. As one can imagine, processing textual data and extracting meaning from them is of high importance towards the goal of automating human language decoding. Example of such applications with significant community attention are machine translation [70], named entity recognition [71], sentiment analysis [72], and document classification [73]. Interestingly, input to all of these applications are some distributed representations of words in a vector space rather than raw textual data. The main reason being high quality vector representations of words help learning algorithms perform better on NLP tasks [23].

We find one of the earliest use of word representation in [74]. This idea has its application in statistical language modeling [75], where the neural network based language models [76, 75] predict word pairs with syntactic and semantic similarity. We can see its successfully adoption in a wide range of applications [72, 70, 71, 73, 77]. Recently, Word2Vec [22] model gained considerable attention in the ML and NLP community. It is a neural network based model which learns high quality word representations in vector space from a large amount of unstructured data. [23] further Word2Vec by introducing Skip-gram model with negative sampling (SGNS). This method provides state-of-the-art performance on word similarity word analogy tasks [23, 78].

The objective of the Word2Vec model is to capture large number of syntactic and semantic relationship between words in their vector representation. The distributional hypothesis states that words with similar contexts tends to have similar meaning. To group similar words together, Skip-gram model maximizes average log probability of

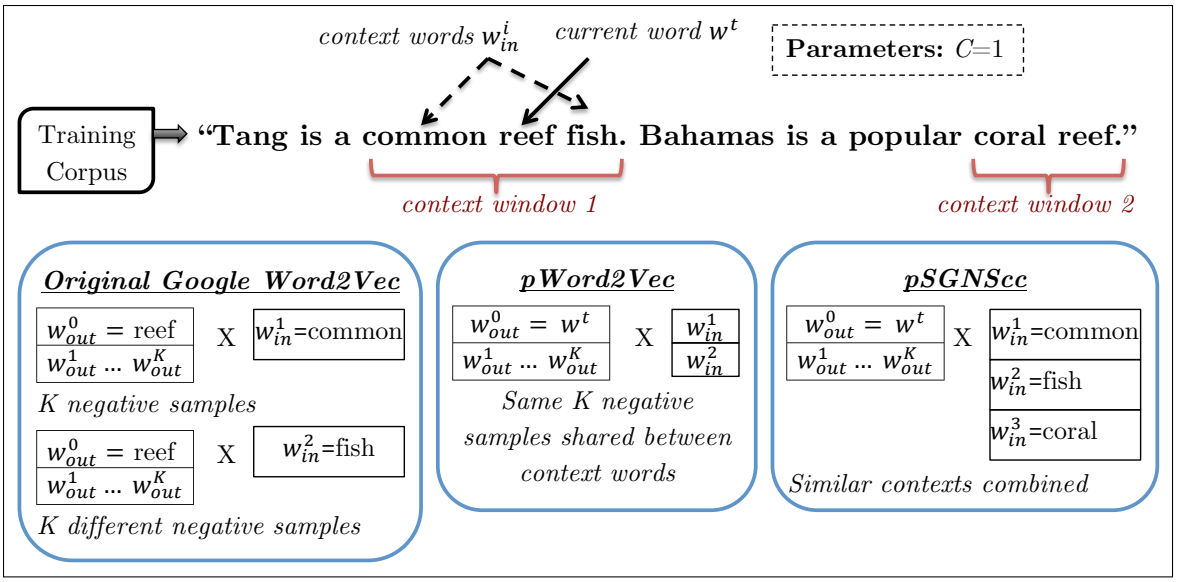


Figure 4.1: Strategies used in different Word2Vec algorithms

getting a context word as output given a current word. SGNS simplifies and reduces the computation significantly with negative sampling. To solve the optimization problem, Stochastic Gradient Descent (SGD) is used. However, original Word2Vec [23] formulation has severe computational drawback - the SGD computation involves vector-vector operations which correspond to level-1 BLAS routines. It is well known that this kind of operations are limited by memory bandwidth and not amenable to high performance.

pWord2Vec [78] addressed the problems in original Word2Vec formulation of SGD by applying mini-batching with "negative sample sharing" strategy. Basically, the main objective is to transform the level-1 BLAS operations inside SGD to level-3 BLAS operations (matrix-matrix operations), which has better arithmetic intensity per memory operation. The strategy is depicted in Figure 4.1. *pSGNScc* [79] improves on *pWord2Vec* by using "context combining" strategy, as represented also in Figure 4.1. This strategy essentially, increases the number of context words at a given instance, and thus increases the matrix size.

One of the main problem in *pWord2Vec* is that the level-3 BLAS calls inside the SGD computation typically involve extremely skewed and small size matrices. It is well known that general BLAS routines do not perform well on the matrices with small dimensions [80,

81] and skewed size [82]. We can see empirical evidence in Figure 4.4b. *pSGNScc* improves the situation only slightly, but not completely mitigates the problem. Furthermore, the formulation of SGD computation in both *pWord2Vec* and *pSGNScc* involves three BLAS library calls and activation function application in between. This strategy inhibits us from fusing the loop bodies corresponding to BLAS routine calls and activation function application. Thus, we do not exploit register reuse among the computations.

For our proposed solution, namely *NinjaVec*, we take a comprehensive approach involving both code optimization (specifically compiler optimization) and algorithmic modification. We address the shortcomings of *pWord2Vec* and *pSGNScc* in *NinjaUpdate*, the code optimization component of our work, by introducing our own code generation strategy for the SGD computation. To handle the broad range of unusual matrix dimensions in Word2Vec, we perform code specialization through multi-versioning at static compilation time. *NinjaUpdate* is equipped with a novel vector register blocking tactic carefully designed to handle extreme cases of skewed and small matrix dimensions. It is noteworthy to mention that, even though one can consider the vectorization and loop fusion strategy in *NinjaUpdate* as standard compiler optimization techniques, it is not so straightforward to apply in our case due to extremeness of the loop bounds, and thus differ significantly from standard practices and heuristics seen in GEMM optimization.

In our work, we also look into algorithmic opportunities for improvement in performance. We come up with *FrequentSkip* method, which aggressively discards frequent words from consideration in SGD computation. Although *FrequentSkip* is somewhat similar to subsampling of frequent words in the original Word2Vec [23], the main differences are - a) we discard the frequent words not only from sentence but also from negative samples, b) we devise *FrequentSkip* for shared-memory based parallel execution, keeping in mind the goal of improving memory locality. As a result, *FrequentSkip* effectively prunes computation to aid speedup in training and at the same time, improves cache locality in the shared-memory multi-threaded execution.

4.1 Background on Word2Vec

4.1.1 Word2Vec: Learning Model

Given a large amount of unstructured text data, the learning objective in Word2Vec is to find good quality distributed vector representations for words that capture the syntactic and semantic word relationships well. The learning model is based on the distributional hypothesis [83], which states that the words from the same syntactic and semantic categories tend to have similar meanings if they appear in similar contexts. Essentially, Word2Vec maps each word w appearing in the training corpus having vocabulary V, to a D dimensional dense vector $\vec{v}_w$ in an embedding space $\mathbb{R}^D$ such that a distance metric encodes many linguistic patterns and regularities. The mapping : $w \rightarrow \vec{v}_w \ \forall w \in V$ is learned by considering sentence contexts, following the distributional hypothesis.

To reduce the computational complexity of neural network based language models, [22] proposes two log-linear model architectures for learning continuous vector representation of words from very large datasets, namely - a) Continuous Bag-of-Words (CBOW) model, and b) Continuous Skip-gram model. CBOW model tries to predict the current word based on the context words. In contrast, the Skip-gram model predicts the surrounding words given the current word. Skip-gram with Negative Sampling (SGNS) [23] extends the continuous Skip-gram model to improve the quality of word vectors as well as training speed. As mentioned in [23, 78, 79], the SGNS model gives the state-of-the-art performance and is widely adopted in the NLP community. Hence, in our work, we focus on the SGNS model.

4.1.1.1 Skip-gram with Negative Sampling

As depicted in Figure 4.2, the Skip-gram model is a neural network with a single hidden layer, which is a log-linear classifier with a continuous projection layer. We feed each current word as input to the hidden layer and predict words within a given context range, i.e., words before and after the current word for a defined range. More formally, given a

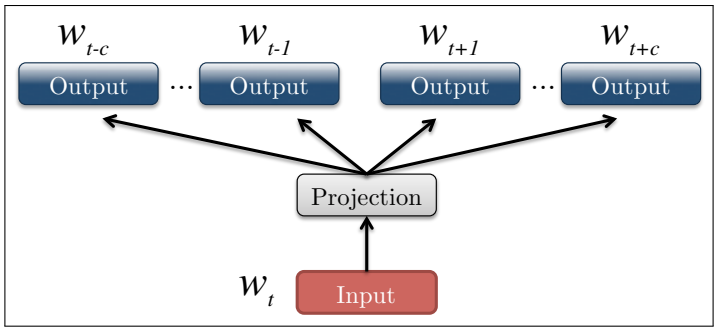


Figure 4.2: Skip-gram model architecture

training corpus with word sequence $\{w_1, w_2, w_3, \ldots, w_T\}$, the goal of the Skip-gram model is to maximize the average log probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} \log p(w_{t+j} \mid w_t) \tag{4.1}$$

where c is the training context size, which can be a function of the current word w_t. One can imagine the context for w_t being a variable length sliding window. Based on the assumption that distant words are less related to the current word than the ones close to it, the training procedure gives less weight to distant words by sampling less from those words. The conditional probability $p(w_{t+j} \mid w_t)$ denotes the probability of seeing word w_{t+j} in the context given the center word is w_t. In the Skip-gram formulation, we define $p(w_{t+j} \mid w_t)$ using the softmax function:

$$p(w_O \mid w_I) = \frac{\exp(\langle \vec{v}_{in}^{w_I}, \vec{v}_{out}^{w_O} \rangle)}{\sum_{w=1}^{V} \exp(\langle \vec{v}_{in}^{w_I}, \vec{v}_{out}^{w} \rangle)} \tag{4.2}$$

where $\vec{v}_{in}^{w}$ and $\vec{v}_{out}^{w}$ are respectively "input" and "output" vector representations of word w, and $\langle \cdot, \cdot \rangle$ denotes the inner product. This formulation is impractical from the perspective of computational cost because the cost is proportional to V or vocabulary size, which is often very large.

A computationally efficient approximation to maximize the log of softmax function is negative sampling, which is based on Noise Contrastive Estimation [84]. We define negative

sampling by the approximation

$$\log p(w_O \mid w_I) \approx \log \sigma(\langle \vec{v}_{in}^{w_I}, \vec{v}_{out}^{w_O} \rangle) + \sum_{k=1}^{K} \mathbb{E}_{w_k \sim P_n(w)} [\log \sigma(-\langle \vec{v}_{in}^{w_I}, \vec{v}_{out}^{w_k} \rangle)] \tag{4.3}$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ is a sigmoid (the logistic) function. So, we are separating out w_O from noise words using logistic regression. The expectations for noise words are computed by sampling K random words from the noise distribution $P_n(w)$ and we call these samples "negative samples". As the number of negative samples K is much smaller compared to the vocabulary size V, this approach is very efficient.

4.1.1.2 Subsampling of Frequent Words

Based on the observation that frequent words occur many more times compared to the rare words and they provide less information value, [23] proposed a subsampling strategy based on word frequency. Any word w_i in training corpus is discarded with probability

$$P(w_i) = 1 - \sqrt{\frac{\lambda}{f(w_i)}} \tag{4.4}$$

where $f(w_i)$ is the frequency of word w_i and λ is a chosen threshold parameter (typically $\sim 10^{-5}$). This heuristically chosen subsampling formula aggressively subsamples words with frequency greater than λ. In practice, it improves learning speed and generates significantly better accuracy for the learned vectors of the rare words.

4.1.2 Word2Vec Algorithms

To solve the optimization problem represented by Equation 4.1 & 4.3, Stochastic Gradient Descent (SGD) is commonly used. For shared memory parallelism with multi-threaded execution, usually the parallel SGD employs Hogwild [4] strategy for exploiting data

parallelism. Hogwild parallelism ignores the conflicts between shared model updates from different threads while processing different chunks of data and allow updates to be carried out in presence of conflicts. Here, we give brief descriptions of three algorithms on CPU - a) original *Google Word2vec* implementation [85, 23], b) *pWord2Vec* [78], and c) *pSGNScc* [79].

Algorithm 5: *Google Word2Vec*

Data: training corpus S
Result: updated word vectors $\vec{v}^{w}_{in}$ & $\vec{v}^{w}_{out}$ $\forall w \in V$

$\alpha \longleftarrow$ learning parameter
$C \longleftarrow$ context window size
$K \longleftarrow$ number of negative samples
for *each* $w^t \in S$ **do**
  target word $w^0_{out} \longleftarrow w^t$
  $b \longleftarrow$ random integer between 0 and C
  for $i \in \{b, b+1, \ldots, 2*C+1-b\} \wedge i \neq C$ **do**
    input word $w^i_{in} \longleftarrow w^{t-C+b}$
    $\vec{v}_{temp} \longleftarrow \vec{0}$
    for $k \in \{0, 1, \ldots, K\}$ **do**
      if $k \neq 0$ **then**
        target word $w^k_{out} \longleftarrow$ a negative sample from V
      $label \longleftarrow (k = 0)\ ?\ 1\ :\ 0$
      $prod \longleftarrow \langle \vec{v}^{w^i_{in}}_{in}, \vec{v}^{w^k_{out}}_{out} \rangle$
      $\Delta \longleftarrow label - \sigma(prod)$
      $\vec{v}_{temp} \mathrel{+}= \Delta * \vec{v}^{w^k_{out}}_{out}$
      $\vec{v}^{w^k_{out}}_{out} \mathrel{+}= \alpha * \Delta * \vec{v}^{w^i_{in}}_{in}$
    $\vec{v}^{w^i_{in}}_{in} \mathrel{+}= \alpha * \vec{v}_{temp}$

Google Word2Vec. In this implementation, each thread updates the word vectors using the strategy represented in Algorithm 5. As we can see from the pseudocode, in each iteration, we choose an input word w^i_{in} from a decided context range, and a target word w^k_{out}, which is either current word w^t or a negative sample from vocabulary V. Then we calculate gradient of the objective function given in Equation 4.3 w.r.t. the word vectors for w^i_{in} and w^k_{out}. Finally, we apply small updates, based on learning parameter α, to the respective word

vectors.

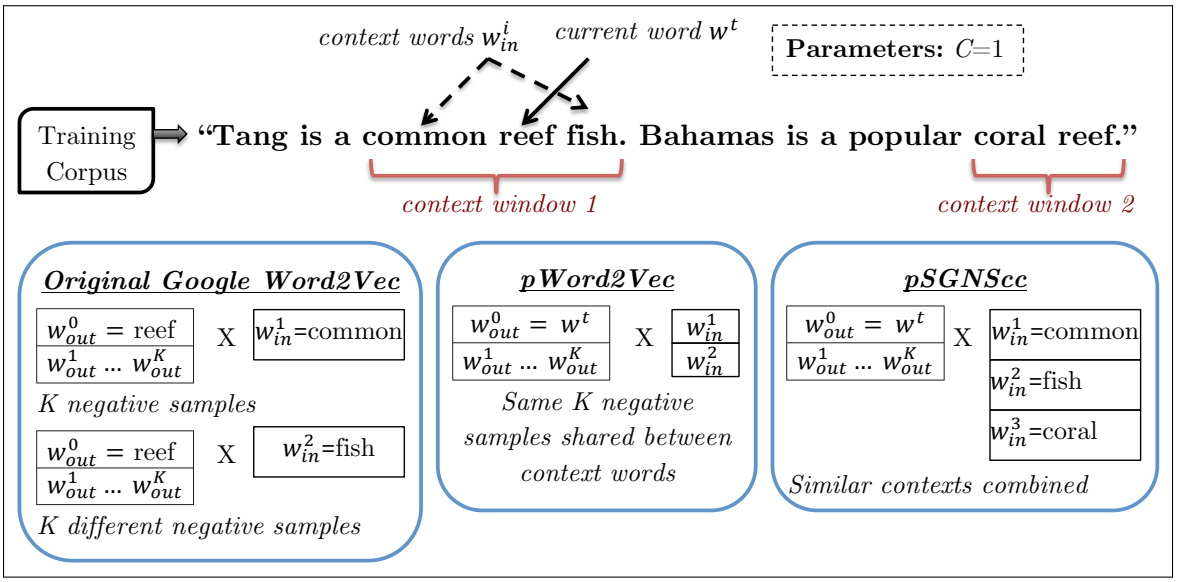


Figure 4.3: Strategies used in different Word2Vec algorithms (presented again for ease of reference)

pWord2Vec. As shown in Algorithm 5, *Google Word2Vec* involves vector-vector operations in line 14, 16, 17, and 18 of the pseudocode. To improve the computational efficiency, *pWord2Vec* converted these vector-vector operations to matrix-matrix operations (GEMM) using "negative sample sharing" strategy. The transition from *Google Word2Vec* to *pWord2Vec* is presented in Figure 4.3. In *Google Word2Vec*, we choose different negative samples for each of the context words. In contrast, *pWord2Vec* shares a set of negative samples between all the context words for a given current word. This GEMM formulation, implemented with Intel® MKL calls, gives 2.6x speedup [78] over *Google Word2Vec* on *One Billion Words*[86] data till 8 threads and even higher speedup for higher number of threads since *Google Word2Vec* shows poor scaling compared to *pWord2Vec*.

pSGNScc. Figure 4.3 depicts the strategy adopted by *pSGNScc*. In order to increase the floating point throughput for GEMM operations in *pWord2Vec*, *pSGNScc* aims to increase the matrix size. This is achieved by employing "context combining" approach where similar contexts are combined to increase the number of context words sharing same current word and negative samples. To find similar contexts, *pSGNScc* performs reverse indexing of

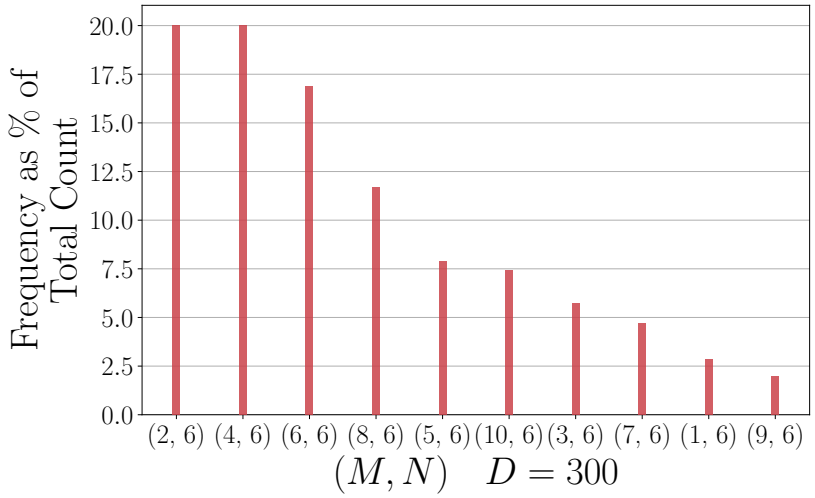
words in training data. This method provides a 1.28x speedup [79] over *pWord2Vec*.

4.2 Shortcomings of Current Solutions

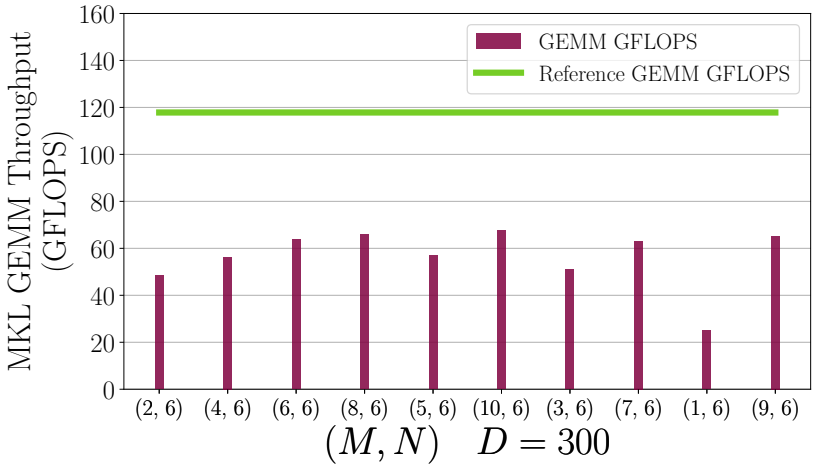
As we see, *pWord2Vec* formulates the SGD computation in word2vec as three GEMM calls - 1) line 14 in Algorithm 5 is replaced by a matrix multiply between $\mathbf{X}_{in}^t$ and $(\mathbf{X}_{out}^t)^T$ and to produce matrix $\mathbf{prodX}^t$, where $\mathbf{X}_{in}^t$ is comprised of $\vec{v}_{in}^{w_{in}^i}$ as row vectors and has maximum size $2C \times D$, $\mathbf{X}_{out}^t$ is comprised of $\vec{v}_{out}^{w_{out}^k}$ and has maximum size $(K+1) \times D$, and $\mathbf{prodX}^t$ contains the inner product values of $\langle \vec{v}_{in}^{w_{in}^i}, \vec{v}_{out}^{w_{out}^k} \rangle$ and has maximum size $2C \times (K+1)$. After applying activation to $\mathbf{prodX}^t$ as per line 15, and then performing scalar multiplication with α, the remaining two GEMM calls are - 2) line 16 is replaced by GEMM between $\mathbf{prodX}^t$ and $\mathbf{X}_{out}^t$, and 3) line 17 is replaced by GEMM between $(\mathbf{prodX}^t)^T$ and $\mathbf{X}_{in}^t$.

The size of $\mathbf{X}_{in}^t$ varies with b because the number of context words is $2(C-b)$ and also sentence beginning and end. Let us denote the number of context words at step t as M. Consequently, $\mathbf{X}_{in}^t$ is a $M \times D$ matrix. Now, repetitions in choosing K negative samples cause the size of $\mathbf{X}_{out}^t$ to vary. If we represent the number of output words, which is $\leq (K+1)$, as N, the size of $\mathbf{X}_{out}^t$ becomes $N \times D$. Thus, $\mathbf{prodX}^t$ is of size $M \times N$.

Since, typically D is in hundreds while $C, K < 20$, $\mathbf{X}_{in}^t$ and $\mathbf{X}_{out}^t$ have very skewed sizes. If we consider the parameter settings from [78] for training on *One Billion Words*[86] dataset, where $D = 300$ and $C, K = 5$, the bounds on dimensions M and N become: $1 \leq M \leq 10$ and $1 \leq N \leq 6$. It is well known that general BLAS routines do not perform well on the matrices with small dimensions [80, 81] and skewed size [82]. Figure 4.4 presents a performance analysis of Intel® MKL on the first GEMM call in *pWord2Vec* for training on *One Billion Words* dataset. Details about the experimental setup can be found in section 4.4.1. We have the frequency distribution of different cases of (M, N) combinations in Figure 4.4a. The showed cases account to 99.14% of total number of calls. Next, Figure 4.4b gives the throughput of MKL in these cases. The "Reference GEMM" is a GEMM between two regular size matrices of size 32×64 and 48×64 (having



(a) Frequency distribution



(b) Performance

Figure 4.4: Statistics for the first GEMM call in *pWord2vec* over *One Billion Words* dataset

similar memory footprint as matrices of size 6×300 and 10×300 respectively). As we can see, the throughput for GEMM over regular sized small matrices is 117.84 GFLOPS whereas the weighted average (according to frequencies) of throughput for different cases in *pWord2Vec* is 57.32 GFLOPS. This large gap in performance indicates a significant room for improvement.

Furthermore, in *pWord2Vec*, we have three separate GEMM calls with an application of activation function between first and second GEMM call. As the computations reuses $\mathbf{X}_{in}^t$, $\mathbf{X}_{out}^t$, and their inner product $\mathbf{prodX}^t$, there is a good opportunity for register reuse. However, the strategy here, involving separate library calls, inhibits us from fusing the loop bodies corresponding to GEMM calls and activation function application. Thus, we do not exploit register reuse among the computations. This is a significant performance limiter for long length vector instructions, such as `AVX-512`, because vector `load` and `store` are costly.

pSGNScc improves the throughput of GEMM calls by employing "context combining" strategy as depicted in Figure 4.1. With this approach, only the number of context words or M is increased. However, the degree of increment is not to the point where $\mathbf{X}_{in}^t$ changes to regular size. On the other hand, $\mathbf{X}_{out}^t$ remains same for a given K. Moreover, the "context combining" strategy requires reverse indexing of words, which has a significant overhead [79]. Apart from increasing M, *pSGNScc* uses the same BLAS routine based three GEMM call approach as *pWord2Vec*, which prevents register reuse through loop fusion.

4.3 Proposed Approach

In this section, we first describe our proposed solution to optimize SGD computation - *NinjaUpdate*. It addresses all the problems mentioned in section 4.2. Next, we describe another strategy - *FrequentSkip*, which exploits the power-law frequency distribution of the words in textual data to accelerate the training process further.

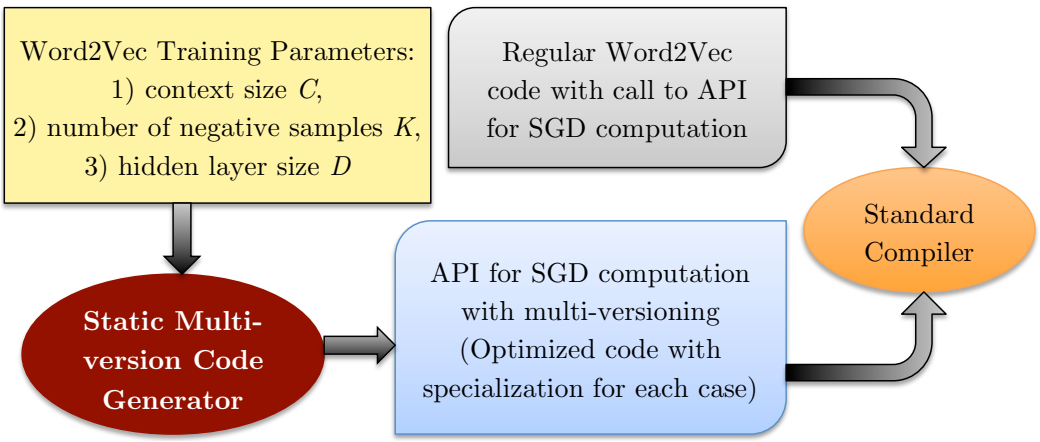


Figure 4.5: Workflow for *NinjaUpdate*

4.3.1 NinjaUpdate

Figure 4.5 depicts a high-level overview of the workflow strategy used in our *NinjaUpdate* approach. Training Word2Vec model usually takes a large amount of time, often tens of hours for big datasets, of which SGD computation is of highest importance [78, 79]. Hence, it makes sense to optimize the SGD computation to accelerate the training process, as is done in previous works [78, 79]. Towards this goal, we adopt a static multi-version code generation approach.

In our experience, the training parameters affecting the SGD performance (context size C, the number of negative samples K, and hidden layer size D) do not change much for similar-sized datasets. Moreover, the compilation time is insignificant compared to the training time. So, we kept our code specialization approach static. In our workflow, we abstracted out the SGD computation with an API, for which our static multi-version code generator generates the code. Finally, we use a standard compiler to compile the Word2Vec code along with our specialized code for SGD. For reference purposes, we present the SGD code in Figure 4.6. Next, we give the details about our code generator.

```
// first GEMM
for (int m=0; m<M; ++m) {
for (int n=0; n<N; ++n) {
for (int d=0; d<D; ++d) {
prodX^t[m][n] += X^t_in[m][d] * X^t_out[n][d];
}}}
// applying activation
for (int m=0; m<M; ++m) {
for (int n=0; n<N; ++n) {
if (n == 0) label=1;
else label=0;
prodX^t[m][n] = label - sigmoid(prodX^t[m][n]);
prodX^t[m][n] *= α;
}}
// second GEMM
for (int m=0; m<M; ++m) {
for (int d=0; d<D; ++d) {
for (int n=0; n<N; ++n) {
ΔX^t_in[m][d] += prodX^t[m][n] * X^t_out[n][d];
}}}
// third GEMM
for (int n=0; n<N; ++n) {
for (int d=0; d<D; ++d) {
for (int m=0; m<M; ++m) {
ΔX^t_out[n][d] += prodX^t[m][n] * X^t_in[m][d];
}}}
```

Figure 4.6: SGD code

4.3.1.1 Vectorization

Needless to say, vectorization is one of the most critical optimizations for achieving peak single-thread performance for compute-bound tasks (GEMM being one of them) on CPUs. As mentioned in section 4.2, typically M and N have very small and irregular values, whereas D is in hundreds. Hence, for vectorization, it is only worthwhile to consider the `d-loop` inside the loop nests associated with SGD code, as presented in Figure 4.6.

In the loop nest of first GEMM, the `d-loop` is a reduction loop. Hence, targeting vectorization of this loop means performing parallel reduction. Considering each $\mathbf{prodX}^t$`[m][n]` element as a scalar, and then applying scalar expansion to each such element enables vectorization of the `d-loop`. Later, we perform reduction of those temporary array variables to get the corresponding $\mathbf{prodX}^t$`[m][n]` elements.

Now, for loop nest in second GEMM, we interchange the `d-loop` with the innermost

`n-loop`, i.e. an interchange between loops in line 17 and 18 for second GEMM. This is legal since there is only output dependence along `n-loop`, and loop interchange between `d-loop` and `n-loop` does not violate this dependence. This loop interchange enable vectorization of the `d-loop` in innermost position. Similarly, we apply loop interchange between the `d-loop` and the innermost `m-loop` at line 23 and 24 respectively for the third GEMM, and then perform vectorization on the `d-loop`.

4.3.1.2 Loop Fusion

As we can see from the code in Figure 4.6, the computations reuse all three matrices - $\mathbf{X}_{in}^t$, $\mathbf{X}_{out}^t$, and $\mathbf{prodX}^t$. To exploit these reuse, we consider loop fusion after we apply vectorization. If we consider the first three loop nests, we can fuse the loop bodies inside a single loop nest of `m-loop` followed by `n-loop` (loops at line 17 and 18 are interchanged after vectorization). However, the fourth loop nest has `m-loop` and `n-loop` in exactly opposite order. Now, we can fuse all four loop nests by interchanging `m-loop` and `n-loop` in fourth loop nest, which increases the reuse distance for $\Delta\mathbf{X}_{out}^t$ from D to $(N * D)$ while decreasing the reuse distance for $\mathbf{X}_{in}^t$ from $(M * D)$ to D.

Another option for fusing four loop nests is to interchange `m-loop` and `n-loop` in the first three loop nests. This increases the reuse distance for $\Delta\mathbf{X}_{in}^t$ from D to $(M * D)$ while decreasing the reuse distance for $\mathbf{X}_{out}^t$ from $(N * D)$ to D in third loop nest. We also see similar changes in reuse distance for $\mathbf{X}_{in}^t$ and $\mathbf{X}_{out}^t$ respectively, in the first loop nest. Additionally, we have strided access for $\mathbf{prodX}^t$ which hurts its spatial locality. Although one can argue that we can replace $\mathbf{prodX}^t$ with its transpose to mitigate the problem. Since, both the strategies are quite equivalent, it boils down to values of M and N to decide which strategy to choose. In our experience, usually $M > N$ in Word2Vec parameter settings. So, we choose the first strategy with loop order `m` $\rightarrow$ `n` $\rightarrow$ `d` (`d-loop` being the innermost) to fuse four loop nests.

4.3.1.3 *Vector Register Blocking*

In general, register blocking helps improve instruction level parallelism (ILP). In the case of vectorized code, vector register blocking is of high importance because vector arithmetic instructions usually take longer cycles to complete compared to scalar instructions. For example, `vFMADD` (fused multiply add) from AVX-512 ISA takes 4 cycles on Intel® Skylake CPU. If we have two vector processing units (VPU) in a core, then we need at least 8 independent `vFMADD` instructions to keep the VPU pipeline busy. This can be achieved by applying a vector register blocking of factor 8. In our case, apart from keeping VPUs busy, the register blocking factor along `m-loop` and `n-loop` also determines the degree of register reuse from loop fusion in the previous section 4.3.1.2. As we are dealing with extremely irregular and small dimensions (M and N), this critical optimization becomes very challenging.

A Case of Code Specialization - Let's consider the scenario from section 4.2 for Word2Vec training on *One Billion Words* dataset. We have the bounds on `m-loop` and `n-loop` as follows - $1 \leq M \leq 10$ and $1 \leq N \leq 6$. Consequently, $(M * N)$ can vary from 1 to 60. In comparison, we have 16 vector registers in the AVX-256 scheme and 32 vector registers in the AVX-512 scheme. One can easily see that, we can not apply a fixed blocking factor on `m-loop` and/or `n-loop` for all (M, N) cases, which is big enough to make full use of the available vector registers. Furthermore, even if we were able to find a fixed relatively good blocking factor for (M, N) values closer to the upper bounds, the blocked loop would be very small, and the peeling loop would significantly degrade the performance. Mainly, for this reason, we adopt multi-version code generation, which specializes in M and N values.

Register Blocking in Specialized Code - As reasoned above, a fixed blocking factor is not suitable in our case due to irregularity and extremeness of loop bounds. As a solution, we consider partitioning the iteration space of a loop into different sized chunks, i.e., a variable number of blocking iterations where each of them can have different loop bounds.

So, instead of a scalar value for the blocking factor, now we will have a vector:

$$\overrightarrow{bf} = (lb_1, lb_2, \ldots, lb_B) \text{ where } lb_i \in \mathbb{Z}^+ \; \forall \; i \in \{1, 2, \ldots, B\}$$
$$\text{s.t. } \sum_{i=1}^{B} lb_i = \textit{OLB} \tag{4.5}$$

variable dimension B denotes the loop bound of the blocked loop, lb_i denotes the loop bound of i-th blocking loop, and *OLB* is the original loop bound before we apply blocking. We call $\overrightarrow{bf}$ as "blocking vector".

Essentially, to form $\overrightarrow{bf}$, we are looking for partitions of *OLB* where summation is non-commutative. For a given $p \in \mathbb{Z}^+$, the number of ways *OLB* can be represented as a sum of p positive integers is $\binom{OLB-1}{p-1}$. Now, if we want the number of ways *OLB* can be represented as sum of two or more positive integers or itself, it becomes $\sum_{p=1}^{OLB} \binom{OLB-1}{p-1}$. This is exponential. Consequently, if we consider blocking both `m-loop` and `n-loop`, the total number of possible cases would be $\sum_{p=1}^{M} \binom{M-1}{p-1} \times \sum_{q=1}^{N} \binom{N-1}{q-1}$. The situation looks quite dire.

Fortunately, we can leverage several simplifications here. First of all, we now get the advantages of M and N having small values. Even if we go for an exhaustive search, the search space is not absurdly huge or non-tractable. In our experience, the upper bound on N is usually very small. Thus, we can have $\overrightarrow{bf}_n$ as (N), i.e. a vector of dimension 1. We completely unroll-and-jam the `n-loop`. Hence, we only have to search for optimal $\overrightarrow{bf}_m$ for `m-loop`. We can further prune the search space heavily by applying a simple constraint related to the first GEMM - the sum total of a blocking loop bound from `m-loop`, the value of N, and their multiplication has to be less than or equal to the total number of vector registers.

Cost Function - Before describing the algorithm for finding an optimal blocking vector, we give an overview of our cost model associated with a specific blocking vector. We define a cost function $F_{cost}(\,(\overrightarrow{bf}_m)$ which gives an estimated execution cost for a given blocking

Algorithm 6: *Find Optimal Blocking Vector*

Input: $M = \mu$, $N = \nu$ (specific values)
Output: Optimal blocking vector $\overrightarrow{bf}_m^{opt}$

```
OptCost ⟵ INF
Ω ⟵ GetBlockingVectors(0⃗, μ, μ, ν)
for bf_m ∈ Ω do
    CurCost ⟵ F_cost(bf_m)
    if CurCost < OptCost then
        OptCost ⟵ CurCost
        bf_m^opt ⟵ bf_m
```

vector $\overrightarrow{bf}_m$ by considering latency of instructions. This cost function helps in the evaluation of a specific blocking vector. It thus provides a means to compare two blocking vectors for the purpose of finding an optimal blocking vector. The calculation of execution cost inside the cost function consider latency of operations, and comprised of the following symbolic costs:

- *VReg*($\overrightarrow{bf}_m$) - number of vector registers required for each computation step, i.e. three GEMMs in Figure 4.6. We use this cost in enforcing register constraint.
- *LdSt*($\overrightarrow{bf}_m$) - estimated cycles for `load` and `store` instructions.
- *FP*($\overrightarrow{bf}_m$) - estimated cycles for all types of floating-point arithmetic operations. Our code involves vector operations for multiply, add, fused multiply add, and division in the arithmetic category.
- *Shuff*($\overrightarrow{bf}_m$) - estimated cycles for all types of vector register shuffle operations.

Based on the symbolic costs, the cost function is calculated as follows:

$$F_{cost}\,(\overrightarrow{bf}_m) = \begin{cases} \infty \quad \text{if } \textit{VReg}(\overrightarrow{bf}_m) > \textit{\# vec regs} \\ \textit{LdSt}(\overrightarrow{bf}_m) + \frac{\textit{FP}(\overrightarrow{bf}_m)}{\textit{\# VPUs}} + \textit{Shuff}(\overrightarrow{bf}_m) \end{cases} \tag{4.6}$$

Algorithm 7: *Find Set of Blocking Vectors*

Input: current blocking vector $\overrightarrow{bf}_m^{cur}$, μ_{rem} is remainder from μ, μ is value of M, ν is value of N

Output: Ω - set of candidate blocking vectors $\overrightarrow{bf}_m$

1 **Function** `GetBlockingVectors` ($\overrightarrow{bf}_m^{cur}$, μ_{rem}, μ, ν) **:**
2 $\Omega \longleftarrow \emptyset$
3 *nVecReg* $\longleftarrow$ max number of vector registers
4 **for** $i \in 1, 2, \ldots, \mu$ **do**
5 **if** $(i + \nu + i * \nu) >$ *nVecReg* **then**
6 break
7 `insert` i at the end of $\overrightarrow{bf}_m^{cur}$
8 **if** $|\overrightarrow{bf}_m^{cur}|_1 > \mu$ **then**
9 break
10 **if** $|\overrightarrow{bf}_m^{cur}|_1 == \mu$ **then**
11 $\Omega \longleftarrow \Omega \cup \{\overrightarrow{bf}_m^{cur}\}$
12 **else**
13 $\Omega' \longleftarrow$ `GetBlockingVectors` ($\overrightarrow{bf}_m^{cur}$, $\mu_{rem} - i$, μ, ν)
14 $\Omega \longleftarrow \Omega \cup \Omega'$
15 `delete` last element of $\overrightarrow{bf}_m^{cur}$
16 **return** Ω

Now that we have the cost function to compare across different blocking vectors, we find the optimal blocking vector using the method depicted in Algorithm 6. First, we enumerate a set of candidate blocking vectors and then iterating over them, we calculate the cost function for each of them to select the blocking vector with least cost since the cost presents estimated latency of execution. Algorithm 6 uses function `GetBlockingVectors()` to get the candidate set of blocking vectors for a given (M, N) pair. We present the function `GetBlockingVectors()` in Algorithm 7. This is a recursive function that starts from the most fragmented iteration space or highest length blocking vector possible. As mentioned earlier, we use vector register capacity constraint from the first GEMM (line 5 in Algorithm 7) to prune the search space. The function also exits when the L-1 norm of blocking vector exceeds the original loop bound of `m-loop` as this constraint is monotonic with the increase in loop bound of blocking loops.

After applying all the optimization and code specialization, the code from Figure 4.6 transforms to the code shown in Figure 4.7. As we can see, we unroll-and-jam both `n-loop` and blocking `m-loop` represented now by `mm-loop` to perform vector register blocking. We further unroll the blocked `m-loop` represented in Figure 4.7 as `i-loop`, which makes our code branch free. One thing we haven't shown here is that, when μ and ν are extremely small, we further employ vector register blocking along `d-loop` to boost the ILP. This is mainly for better readability and clarity of presentation.

```
// Given M = μ and N = ν
find optimal blocking vector bf_m^opt = (lb_1, lb_2, ..., lb_B)
// unrolled completely
for (int i=1; i≤B; ++i) {
// complete unroll-and-jam
for (int mm=0; mm<lb_i; ++mm) {
m = mm + (i > 1) ? (∑_{j=1}^{i-1} lb_j) : 0;
// complete unroll-and-jam
for (int n=0; n<ν; ++n) {
// vectorized
for (int d=0; d<D; ++d) {
// first GEMM
prodX^t[m][n] += X^t_in[m][d] * X^t_out[n][d];
}
// applying activation
if (n == 0) label=1;
else label=0;
prodX^t[m][n] = label - sigmoid(prodX^t[m][n]);
prodX^t[m][n] *= α;
// vectorized
for (int d=0; d<D; ++d) {
// second and third GEMM
ΔX^t_in[m][d] += prodX^t[m][n] * X^t_out[n][d];
ΔX^t_out[n][d] += prodX^t[m][n] * X^t_in[m][d];
}
}}}
```

Figure 4.7: Optimized SGD code with specialization

4.3.1.4 Static Multi-versioning

As depicted in Figure 4.5, we abstract out the SGD computation with an API in the main Word2Vec code. Our static code generator implements this API. From the parameters of the Word2Vec training, we first deduct all possible (M, N) cases. Next, we generate specialized code for each of those cases. Finally, we implement multi-versioning based on M and N

```
// specialized for (M,N) = (1,1)
func_1x1();
// specialized for (M,N) = (1,2)
func_1x2();
. . .
switch((M-1) * N_max + N) {
case 1:
func_1x1(); break;
case 2:
func_1x2(); break;
. . .
default:
. . .
}
```

Figure 4.8: Outline of code generated for multi-versioning

values with a `switch-case` inside the main API for SGD. For each of those cases, we call the corresponding optimized code.

4.3.2 FrequentSkip

As mentioned previously in section 4.1.1.2, in Word2Vec model training, we discard any word w_i in the training corpus with probability $P(w_i) = 1 - \sqrt{\lambda/f(w_i)}$, where $f(w_i)$ is the frequency of word w_i. It is argued in [23] that frequent words have less information value, and the respective word vectors for frequent words remain unchanged for a large amount of time. Discarding them gives better accuracy for rare words while accelerating the training process. [87] argued that the subsampling of frequent words has the effect of implicitly increasing the effective context size. So, meaningful context words get included, which improves the accuracy.

In many real world datasets, the frequency distribution of words follows power law or Zipf's law [35]. Given a parameter α for skewness and sorted words in descending order according to their frequencies, $f(w_i) = ci^{-\alpha}$ (where c is a constant) for power law distribution, and with Zipf distribution, $f(w_i) = \frac{N}{i^{\alpha}\zeta(\alpha)}$ where $N = \sum_{i=1}^{M} f_i$ and $\zeta(\alpha)$ is Riemann's zeta function with value $\zeta(\alpha) = \sum_{i=1}^{\infty} \frac{1}{i^{\alpha}}$. For example, [88] reported Zipf distribution with $\alpha \geq 1.4$ in real-world datasets. The question that comes to our mind is, can we do a more aggressive discarding of frequent words in these extremely skewed frequency

distributions?

With the subsampling strategy mentioned earlier, the expected frequency of word w_i with $f(w_i) > \lambda$ becomes:

$$\mathbb{E}(f(w_i)) = \sqrt{\lambda * f(w_i)} \tag{4.7}$$

If we consider training on *One Billion Words* dataset, λ is set to 10^{-4} and the most frequent words have frequency in the range $\sim 10^8$. Hence, the expected frequency becomes $\sim 10^2$. Which is still rather large compared to a large number of rare words.

Furthermore, frequent words are good candidates for negative sampling and, consequently, chosen more frequently as negative samples since we draw from the unigram distribution of words raised $U(w)$ raised to the $3/4$-th power. In a multi-threading execution where threads update word vectors without `locks` or *atomics*, such as Hogwild, there is a high chance of getting cache line ping-ponging [78] for these frequent words in the negative samples.

To address the two issues mentioned above, we define a parameter θ where $\theta \leq \lambda^{-1}$, and distribute the index set $\{1, 2, \ldots, \theta\}$ among Γ threads in round-robin fashion. If we consider $\{f(w_i)\}\ \forall\ 1 \leq i \leq \theta$, the θ most frequent words are evenly distributed between threads. For the set of $\frac{\theta}{\Gamma}$ frequent words associated with a thread, the frequency distribution is similar to original one.

Now, we employ further discarding frequent words in two ways - a) if a thread γ faces a frequent word $w_i \ni i \leq \theta$ and i is not in its assigned index set, it skips the target word, and b) if a thread γ gets a frequent word $w_i \ni i \leq \theta$ as negative sample, and i is not in its assigned index set, it discards that negative sample. The first strategy has an effect of lowering the expected frequency of word $w_i \ni i \leq \theta$ in output words set by a factor Γ, i.e

$$\mathbb{E}(f(w_i)) = \frac{\sqrt{\lambda * f(w_i)}}{\Gamma} \text{ for target word} \tag{4.8}$$

The second strategy effectively lowers the frequency of word $w_i \ni i \leq \theta$ appearing in a

negative sample set by a factor Γ, and thus giving more weight to the negative samples of lower frequency. One thing to note, our strategy assumes that the corpus does not have an adversarial word arrival order, which is true in most of the real text corpus. We show the effectiveness of our strategy empirically on real text datasets in section 4.4.3.2.

4.4 Evaluation

4.4.1 Experimental Setup

4.4.1.1 Hardware Configuration

We carried out all our experiments on a machine with two socket Intel®Xeon®Platinum 8280 CPU @ 2.70GHz with maximum Turbo frequency being 4.00GHz. The machine has a total of 56 cores (each socket having 28 cores) and 756 GB DRAM. The operating system of the machine is CentOS Linux 7.

4.4.1.2 Software Configuration

We implemented our work in C++. Our work exploits shared memory parallelism using OpenMP. We compiled our software framework with Intel®C++ Compiler 18.0.0 with `O3` optimization level. We used `-xCORE-AVX512` compiler flag to target `CORE-AVX512` instruction set. We used the same compiler and Intel®MKL 2018.0.128 for the compilation of methods we compare with here.

4.4.1.3 Datasets

Training Datasets - we perform the training of Word2Vec models on three datasets, namely:

- *Text8*[89] - a small dataset of 17 million words consisting primarily of English text from Wikipedia dump.
- *One Billion Words* Benchmark[86] - a popular dataset for evaluating language modelling techniques.

- *UMBC* webbase corpus[90] - a dataset of 3 billion words containing English paragraphs.

Different datasets encompassing a large range of sizes and statistics are used to show that our approach is versatile enough. However, we primarily use *One Billion Words* for most of the analysis works. Hence, one can safely assume the dataset for any experiment to be *One Billion Words* unless otherwise stated explicitly.

Testing Datasets - In order to test the accuracy of trained Word2Vec models, we use two evaluation methods: a) finding *word similarity* with reference to human judgement and b) finding *word analogy* for questions of the form *A* is to *B* as *C* is to (?), where model has to fill in the (?). For *word similarity* tasks, we use very popular WordSim353[91] (WS353) dataset. In this case, the accuracy is measured as Spearman's rank correlation co- efficient[92] between human similarity judgement and cosine similarity of word vectors. We use Google analogy dataset[22] for *word analogy* task. It contains 19544 word-analogy queries, among them 8869 are semantic type, and 10675 are syntactic type. An answer to a query is only correct if it matches exactly with the correct word. Then the accuracy is measured as the fraction of queries answered correctly.

4.4.1.4 Word2Vec Parameters

Following *pWord2Vec* [78], we used the following parameter settings for Word2vec in all our training tasks:

- Vector Dimension or hidden layer size $D = 300$
- Context size or window size $C = 5$
- Number of negative samples $K = 5$
- Threshold for subsampling $\lambda = 10^{-4}$
- Number of epochs or iterations is 5

4.4.1.5 *Performance Parameters*

We have extensively used the parameter "speedup" for our performance study. Here, speedup for a given approach over a specific baseline method is:

$$\frac{Execution\ time\ of\ the\ baseline\ method}{Execution\ time\ of\ the\ given\ approach} \tag{4.9}$$

4.4.2 Performance Comparison

We present two types of performance measurements concerning Word2Vec training. One is the time spent in the SGD step, while the other one is the time required for the complete model training. The former gives a clear picture of the performance improvement we get from employing different strategies proposed by different Word2Vec algorithms. In contrast, the later indicates an overall performance gain in the training process. We compare the performance achieved through our approach *NinjaVec* with three other methods mentioned in section 4.1.2, namely - a) *Google Word2Vec* [23], b) *pWord2Vec* [78], and c) *pSGNScc* [79]. We have used 8, 16, and 32 threads for training on *Text8*, *One Billion Words*, and *UMBC* datasets, respectively, mainly because the sizes of the datasets cover a wide range (the respective sizes are 97MB, 4GB, and 17GB).

4.4.2.1 *Speedup in SGD*

Figure 4.9 represents the speedup achieved in SGD computation by all methods of comparison over *Google Word2Vec* on different datasets. *pWord2Vec* gives 7.93×, 7.54×, and 10.47× speedup on respectively *Text8*, *One Billion Words*, and *UMBC* datasets. *pSGNScc* achieves a moderate improvement over *pWord2Vec* by delivering respectively 8.44×, 9.27×, and 14.49× speedup on the same datasets. One thing to note here is that, in case of *pSGNScc*, we take into account the overhead of reverse indexing while calculating the time spent in SGD. On the previously mentioned datasets, our method *NinjaVec* provides 21.12×,

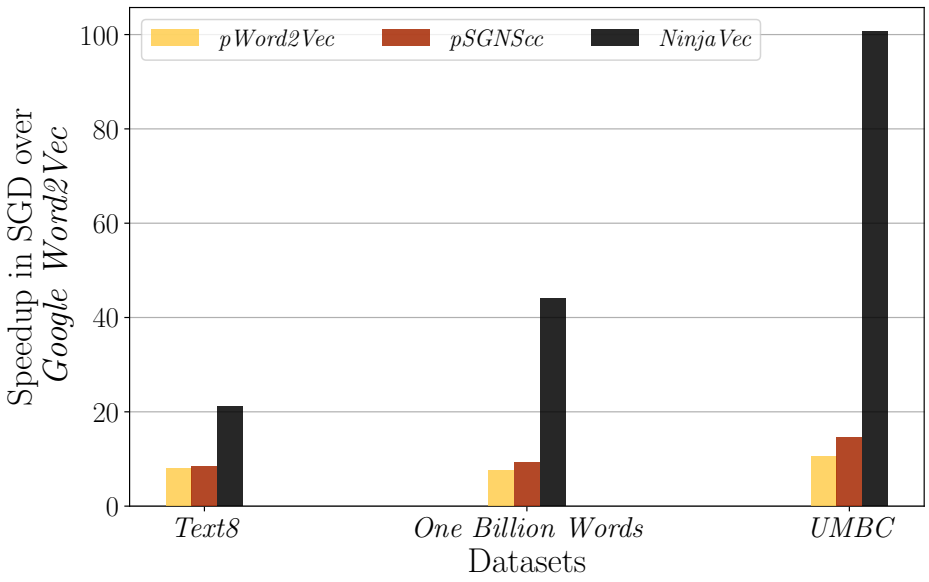


Figure 4.9: Comparison of speedups achieved in SGD

44.09×, and 100.7× speedup respectively. As we can see, *NinjaVec* significantly improves performance in SGD computation of Word2Vec compared to the state-of-the-art methods on CPUs, namely *pWord2Vec* and *pSGNScc*.

The large increases ($\sim 2\times$) in speedup from *Text8* to *One Billion Words* and further to *UMBC* dataset can be explained by weak scaling characteristics. Here, we are increasing the data size by large factors as we double the number of threads in each step, which is a weak scaling experiment by nature. *NinjaVec* shows much better weak scaling of performance compared to *pWord2Vec* and *pSGNScc*, which is a combined effect of *FrequentSkip* and *NinjaUpdate*. As the data size and the number of threads increase, *FrequentSkip* prunes a larger amount of computations while contributing to cache locality. Meanwhile, *NinjaUpdate* efficiently computes the SGD step for the extreme corner cases of dimensions, which arises from a higher level of pruning induced by *FrequentSkip*.

4.4.2.2 Improvement in Training Time

Now that we have looked into the performance improvement for SGD inside Word2Vec training, in particular, the next step would be to shed some light on the overall training

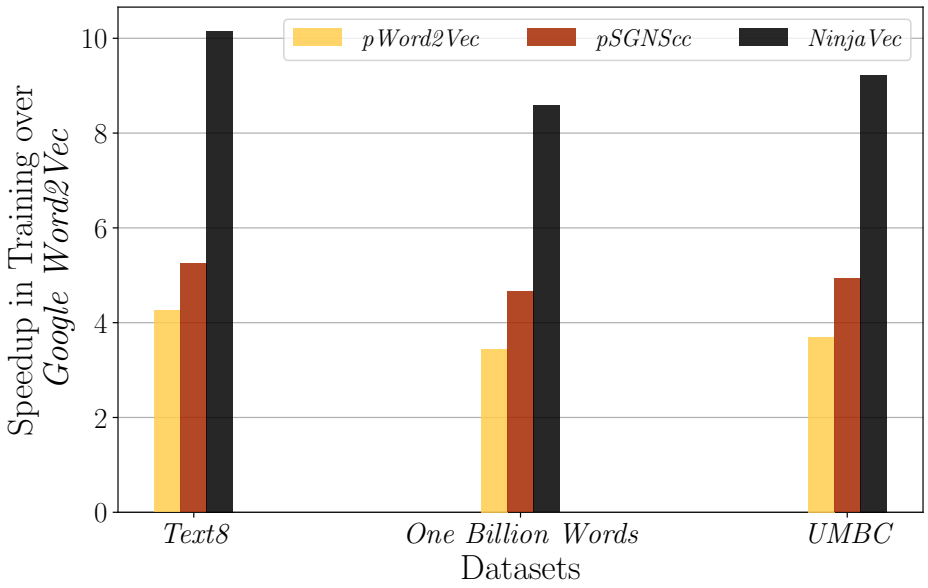


Figure 4.10: Comparison of speedups achieved in training time

time improvement. Figure 4.10 exactly addresses that. In coherence with the SGD speedup analysis, we measure speedup in training time also by considering *Google Word2Vec* as the baseline. On *Text8*, *One Billion Words*, and *UMBC* datasets, *pWord2Vec* achieves a speedup of 4.26×, 3.45×, and 3.7× respectively. Similar to SGD, *pSGNScc* moderately improves those speedups to 5.25×, 4.66×, and 4.93× respectively. Finally, our approach *NinjaVec* attains, respectively, 10.15×, 8.58×, and 9.22× speedup on the previously mentioned datasets. The large gap between speedup in SGD and training time is mainly due to the overheads of memory copy and random number generation, which is common to all methods. A possible future improvement could be overlapping memory copy with computation by assigning two separate threads for computation and memory copy, and allocating extra buffers for matrices associated with SGD computation.

4.4.2.3 Model Accuracy

Table 4.1 & 4.2 presents the accuracy of the trained word2vec model or word vectors on *word similarity* and *word analogy* tasks respectively. We see that, on *One Billion Words* and *UMBC* datasets, *NinjaVec* gives superior accuracy for *word similarity* task while achieving similar

Table 4.1: Accuracy for *word similarity* (WS353 dataset)

Methods	Training Datasets		
	Text8	*One Billion Words*	*UMBC*
Google Word2Vec	64.9%	64.1%	68.6%
pWord2Vec	66.5%	64.9%	68.2%
pSGNScc	68.5%	64.9%	68.4%
NinjaVec	65.7%	67.8%	71.7%

Table 4.2: Accuracy for *word analogy* (Google analogy dataset)

Methods	Training Datasets		
	Text8	*One Billion Words*	*UMBC*
Google Word2Vec	23.7%	33.3%	36.7%
pWord2Vec	23.8%	33.3%	36.6%
pSGNScc	25.3%	33.5%	36.7%
NinjaVec	22.1%	33.1%	36.2%

accuracy for *word analogy* task compared to other methods. The improvement in accuracy from *NinjaVec* can be contributed to *FrequentSkip* strategy. The reasoning is similar to the argument behind subsampling of frequent words in original Word2Vec [23]. We can improve the accuracy of learned vector representations of the rare words by aggressively skipping frequent words through *FrequentSkip*. For *Text8* dataset, *NinjaVec* achieves accuracy close to *Google Word2Vec* and *pWord2Vec*. However, *pSGNScc* provides better accuracy on *Text8* dataset compared to other methods. This is because, as mentioned in [23], bigger context size helps in improving accuracy for very small datasets, such as *Text8*. The "context combining" strategy in *pSGNScc* essentially does this by increases the number of context words for a specific instance.

4.4.3 Empirical Analysis of NinjaVec

After depicting the general system performance landscape in the previous section, we delve deeper and give a thorough performance analysis on different aspects of our *NinjaVec* approach in this section.

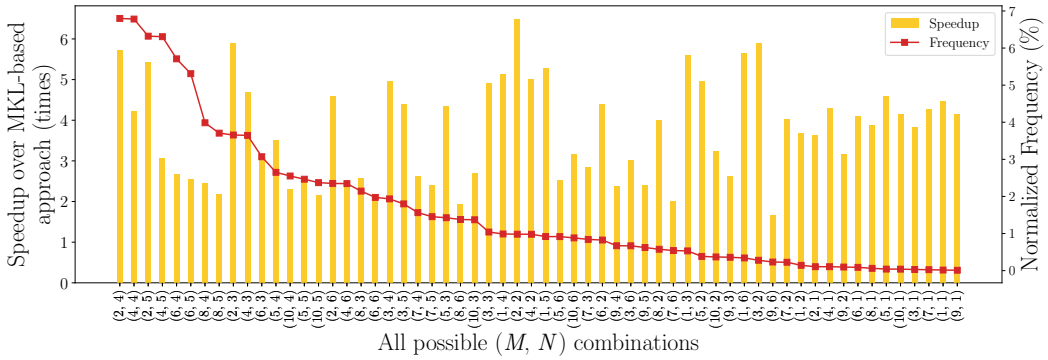


Figure 4.11: Performance gain for gradient update step in different scenarios

4.4.3.1 *Performance of NinjaUpdate*

In Figure 4.11, we give a detailed breakdown of speedup achieved from our *NinjaUpdate* over different (M, N) combinations, where M represents the number of output words (a target word and at most K negative samples), and N represents the number of context words. The baseline is Intel® MKL-based approach used in *pWord2Vec*. The scenarios are sorted based on their normalized (w.r.t. total count) frequency of appearance on *One Billion Words* dataset after using our *FrequentSkip* strategy. As we can see, the speedups vary quite significantly across the board. The geometric mean speedup is 3.71, and the weighted average of speedups according to their relative frequencies is 3.61. One thing to notice in Figure 4.11 is that the frequencies are now much more distributed compared to Figure 4.4a. This is an effect of *FrequentSkip* strategy. As we are skipping frequent words, the chances of appearing lower values for M are greatly increased.

4.4.3.2 *Analysis of FrequentSkip*

We present the effect of *FrequentSkip* in Figure 4.12. We vary the threshold θ for *FrequentSkip* from 160 to 1600 in steps of 160, where a threshold θ indicates that θ most frequent words will be considered in *FrequentSkip* strategy. The respective gradient update time and accuracy presented are after applying only *FrequentSkip*, not *NinjaUpdate*. All executions are done with 16 threads on *One Billion Words* dataset. Figure 4.12 shows that

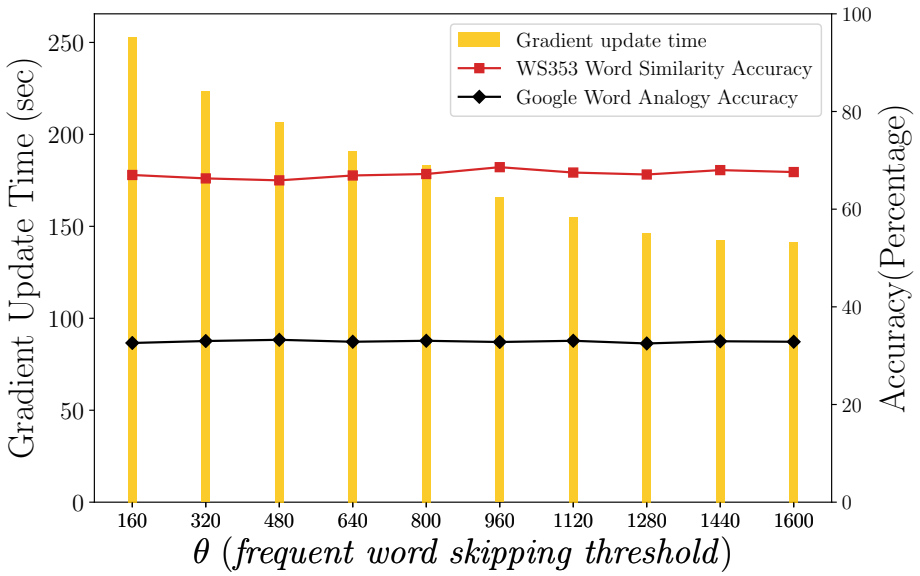


Figure 4.12: Execution time and accuracy with varying threshold for *FrequentSkip* on *One Billion Words* dataset

the gradient update time decreases with increasing θ while the accuracy remains quite the same. Although, the reduction in gradient update time becomes less for higher values of θ.

4.4.3.3 Performance Scaling

Figure 4.13 gives an overview of how our strategies scale with the number of threads. In this case, we measure the speedup w.r.t. *pWord2Vec*. Theoretically speaking, the *NinjaUpdate* strategy should not have any effect on multi-thread performance, because it optimizes performance for fine-grain or vector parallelism. The performance will be affected only when multiple threads execute on single core through simultaneous multi-threading since, at that point, Vector Processing Unit(s) and L1 and L2 caches of a single core will be shared among multiple threads. As we did not come across any background work on Word2Vec, which exploits such a high level of parallelism for single node multi-core CPU, here we present scaling results for 4 to 16 threads. As expected, the *NinjaUpdate* performance remains almost the same with the increasing number of threads.

However, we expect some variation in the performance of the *FrequentSkip* strategy as we change the number of threads. Because, in this strategy, the frequencies of the elements

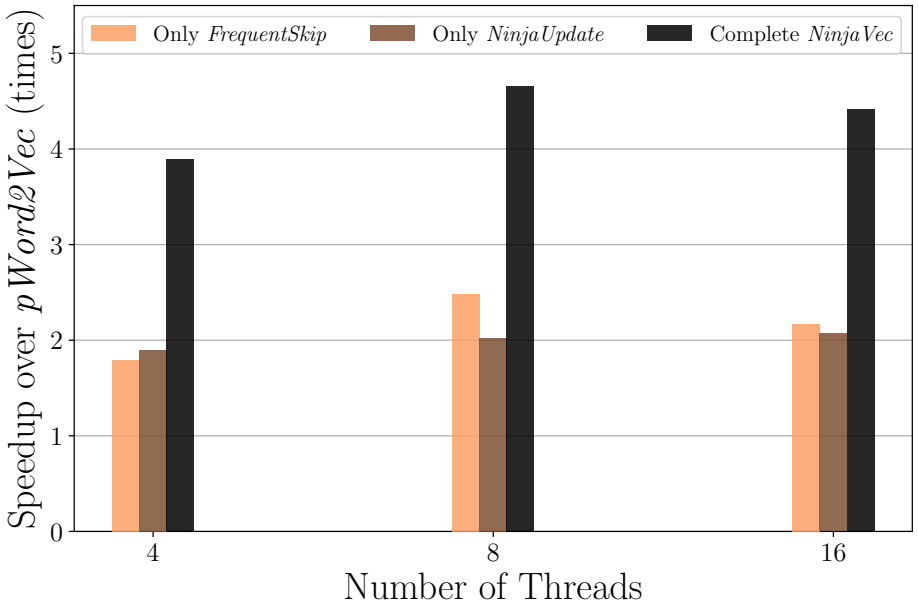


Figure 4.13: Performance scaling with number of threads on *One Billion Words* dataset

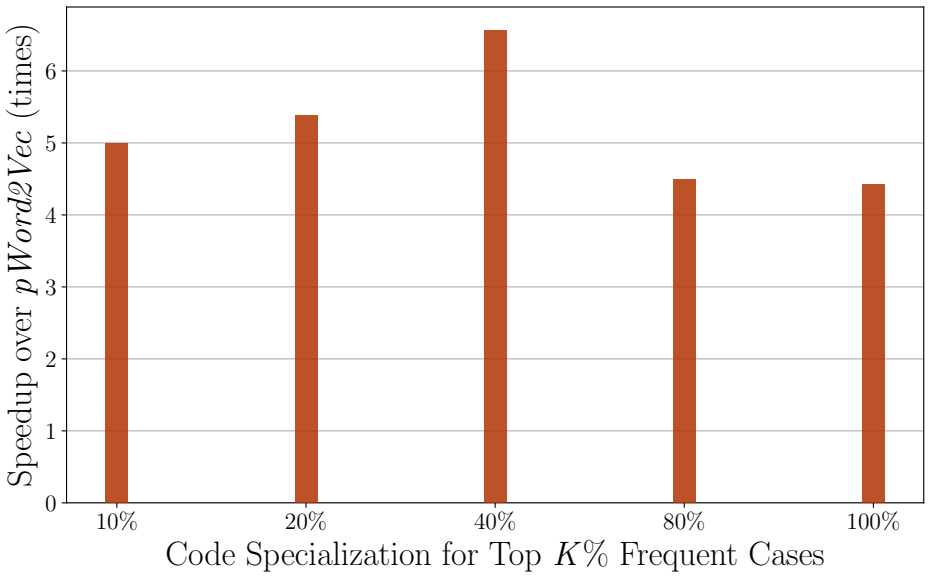


Figure 4.14: Performance with varying number of case specialization based on the frequency on *One Billion Words* dataset

skipped by a specific thread changes with changing number of threads. We see the same in Figure 4.13. Consequently, the speedup for *NinjaVec* as a whole varies a bit as we increase the number of threads from 4 to 16.

4.4.3.4 Varying Degree of Specialization

One interesting investigation on the code specialization of *NinjaVec* would be to analyze how much specialization is sufficient. We present the experimental results for such a study in Figure 4.14. We first sort the cases, or (M, N) tuples based on their frequency in *One Billion Words* dataset, i.e., the order previously presented in Figure 4.11. Then apply *NinjaUpdate* for the top $\tau\%$ cases and use the three GEMM call based approach used in *pWord2Vec* for the rest of the cases. We measure the speedup of each scenario w.r.t. *pWord2Vec* as baseline.

We see that the performance improvement from specialization increases steadily till 40% cases. After that, it drops and remains the same for 80% and 100% cases. This drop is from an increase in code size crossing the L1-I cache size, which is a well-known limiting factor [93] in code specialization. The flat tail of the speedup curve over 80% and 100% follows from the fact that these cases are very infrequent, and hence, specializing them does not affect the performance much. However, we have to remind one point here that, the frequencies of (M, N) cases are unavailable before the training process. Hence, it is a naive yet practical approach to specialize in all possible cases.

CHAPTER 5

CONCLUSIONS

We need efficient data mining and learning algorithms to extract meaningful pieces of information from a large amount of data in a reasonable amount of time. As we deal with more and more data in an increasing number of applications, such as internet search, network traffic analysis, e-commerce, and so on, designing fast data mining methods becomes more important. Mostly, there have been two paths for improving the execution time of the algorithms of concern - a) designing approximate algorithms to reduce computation significantly and b) hardware-centric advancement enabling a rapid increase in processing power. Observing that parallelism is omnipresent in today's processors, we see different types of parallelisms adopted by various architectures. In this thesis, we considered several important applications and show how a carefully designed parallel algorithm, often exploiting power-law data properties, can lead to significant performance gains on current computing resources.

For the first work, we examined the classical problem of frequency estimation, on throughput optimized parallel GPGPUs. We find that if we exploit the natural skew present in the data with a novel hierarchical sketching strategy tailored for the fine-grain parallelism in GPGPU, we attain impressive performance gain over the standard sketching method. For the next work, we focused on the problem of identifying the most frequent elements in distributed data streams. As the current generation servers largely deploy multi-core CPUs for their multi-node infrastructure, we consider both multi-core and distributed parallelism. We show how we can combine a counter-based method with a sketch-based method to achieve the best of both methods, which is parallelism for the counter-based algorithms and fast update time for the sketch-based algorithms. As a result, compared to both the methods we combined, our method provides significant performance gains on distributed multi-core

settings while preserving the accuracy.

Finally, we studied a popular word embedding method called Word2Vec. This time we consider fine-grain SIMD parallelism used in current generation CPUs. As the vector length increases, such as a progression from SSE to AVX-512 in the x86 architecture, the importance of efficiently using vector processing units becomes more relevant. We investigated the limitations of current approaches and to address them, we proposed a static multi-version code generation strategy coupled with an algorithmic approximation based on the power-law frequency distribution of words.

Future Directions. A major direction for future work would be to explore automating the key steps in our work, as far as possible. Given a target parallelism model or architecture and some high-level specifications for power-law distribution or other data properties, one can aim to automatically generate the optimized code that was created manually in this thesis work. Doing so will reduce the programming burden and make it easier for data mining domain experts to adopt our approach. Apart from this, it is worth noting that the frequency distribution need not specifically be power-law or Zipfian for our work to be applicable. As long as there is a steep cut-off between frequencies of heavy hitters and cold items, one can apply our approach. Some examples of such related distributions include the log-normal distribution and Gibrat's distribution. However, one needs to verify whether some additional challenges arise in the case of these related distributions. On that note, it would be helpful for the research community to have a formal way to identify the separation of heavy hitters in frequency distributions from real-world data. Another area of future exploration would be to exploit the skewness in the frequency distribution hierarchically for different parallelism models at the same time. In recent supercomputers such as Summit, we have multi-socket CPUs employing SIMD parallelism and multi-core parallelism, accompanied by multiple GPUs with SIMT parallelism, all in a single compute node. Now, if we consider multiple such nodes, we have a large hierarchy of different parallelism models. There is a potential opportunity to exploit the skewness in the frequency distribution in tandem with

this parallelism hierarchy to efficiently use the computing resources.

REFERENCES

[1] AMD, *AMD EPYC Architecture*, http://developer.amd.com/wordpress/media/2013/12/AMD-EPYC-Performance-White-Paper-Final-Jun-2017.pdf.

[2] NVIDIA Corporation, *NVIDIA Tesla V100 GPU Architecture*, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[3] Qualcomm, *Qualcomm 630 Mobile Platform*, https://www.qualcomm.com/products/snapdragon-630-mobile-platform.

[4] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, pp. 693–701.

[5] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Springer, 2010, pp. 177–186.

[6] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems*, 2017, pp. 2181–2191.

[7] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian, "Variational convolutional neural network pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2780–2789.

[8] R. Spring, A. Kyrillidis, V. Mohan, and A. Shrivastava, "Compressing gradient optimizers via count-sketches," *arXiv preprint arXiv:1902.00179*, 2019.

[9] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, 2011.

[10] *Frequent itemset mining dataset repository*, http://fimi.ua.ac.be/data/.

[11] *The caida ucsd anonymized internet traces dataset - 2016*, http://www.caida.org/data/passive/passive_dataset.xml.

[12] M. Tallis and P. Yadav, "Reacting to variations in product demand: An application for conversion rate (cr) prediction in sponsored search," *arXiv preprint arXiv:1806.08211*, 2018.

[13] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[14] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Automata, languages and programming*, pp. 784–784, 2002.

[15] A. Metwally, D. Agrawal, and A. E. Abbadi, "An integrated efficient solution for computing frequent and top-k elements in data streams," *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1095–1133, 2006.

[16] E. Ogasawara, J. Dias, D. Oliveira, F. Porto, P. Valduriez, and M. Mattoso, "An algebraic approach for data-centric scientific workflows," *Proc. of VLDB Endowment*, vol. 4, no. 12, pp. 1328–1339, 2011.

[17] G. Kougka, A. Gounaris, and A. Simitsis, "The many faces of data-centric workflow optimization: A survey," *International Journal of Data Science and Analytics*, vol. 6, no. 2, pp. 81–107, 2018.

[18] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[19] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms," in *International Conference on Learning Representations*, 2019.

[20] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss, "Space-optimal heavy hitters with strong error bounds," *ACM Trans. Database Syst.*, vol. 35, no. 4, 26:1–26:28, 2010.

[21] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 51–55, 2003.

[22] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[23] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[24] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[25] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM, 2003, pp. 241–252.

[26] A. Shrivastava, A. C. Konig, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2016, pp. 1417–1432.

[27] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ACM, 2016, pp. 101–114.

[28] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 29–42.

[29] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Transactions on Networking (ToN)*, vol. 20, no. 5, pp. 1622–1634, 2012.

[30] A. Aghazadeh, R. Spring, D. LeJeune, G. Dasarathy, A. Shrivastava, and R. G. Baraniuk, "Mission: Ultra large-scale feature selection using count-sketches," *arXiv preprint arXiv:1806.04310*, 2018.

[31] S. Ravi and Q. Diao, "Large scale distributed semi-supervised learning using streaming approximation," in *Artificial Intelligence and Statistics*, 2016, pp. 519–528.

[32] Y.-B. Kim, K. Stratos, and R. Sarikaya, "Scalable semi-supervised query classification using matrix sketching," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, vol. 2, 2016, pp. 8–13.

[33] P. Talukdar and W. Cohen, "Scaling graph-based semi supervised learning to large number of labels using count-min sketch," in *Artificial Intelligence and Statistics*, 2014, pp. 940–947.

[34] A. Goyal, H. Daumé III, and G. Cormode, "Sketch algorithms for estimating point queries in nlp," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Association for Computational Linguistics, 2012, pp. 1093–1103.

[35] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[36] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2016, pp. 1449–1463.

[37] 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*, vol. 10, no. 11, pp. 1442–1453, 2017.

[38] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "Heavyguardian: Separate and guard hot items in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM, 2018, pp. 2584–2593.

[39] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *The VLDB Journal*, vol. 19, no. 1, pp. 3–20, 2010.

[40] A. Mandal, H. Jiang, A. Shrivastava, and V. Sarkar, "Topkapi: Parallel and fast sketches for finding top-k frequent elements," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 898–10 908.

[41] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[42] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 1–12.

[43] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[44] D. Thomas, R. Bordawekar, and C. Aggarwal, "A frequency-aware parallel algorithm for counting stream items on multicore processors," Tech. Rep.

[45] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, p. 26, 2013.

[46] G. Cormode and S Muthukrishnan, "Approximating data with the count-min data structure,"

[47] J. Chen and Q. Zhang, "Bias-aware sketches," *Proceedings of the VLDB Endowment*, vol. 10, no. 9, pp. 961–972, 2017.

[48] C. Estan and G. Varghese, *New directions in traffic measurement and accounting*, 4. ACM, 2002, vol. 32.

[49] NVIDIA Corporation, *NVIDIA CUDA C Best Practices Guide*, `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.

[50] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[51] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[52] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J.-L. Larriba-Pey, "Dynamic count filters," *Acm Sigmod Record*, vol. 35, no. 1, pp. 26–32, 2006.

[53] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 121–132, 2008.

[54] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, *et al.*, "Ad click prediction: A view from the trenches," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2013, pp. 1222–1230.

[55] K. Yi and Q. Zhang, "Optimal tracking of distributed heavy hitters and quantiles," *Algorithmica*, vol. 65, no. 1, pp. 206–223, 2013.

[56] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '12, Scottsdale, Arizona, USA: ACM, 2012, pp. 23–34, ISBN: 978-1-4503-1248-6.

[57] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02, Hong Kong, China: VLDB Endowment, 2002, pp. 346–357.

[58] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th International Conference on Database Theory*, ser. ICDT'05, Edinburgh, UK: Springer-Verlag, 2005, pp. 398–412.

[59] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[60] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *European Symposium on Algorithms*, Springer, 2002, pp. 348–360.

[61] P. Bose, E. Kranakis, P. Morin, and Y. Tang, "Bounds for frequency estimation of packet streams.," in *SIROCCO*, 2003, pp. 33–42.

[62] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[63] X. Yang, J. Liu, and W. Zhou, "A parallel frequent item counting algorithm," in *2016 8th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, vol. 02, 2016, pp. 225–230.

[64] M. Cafaro, I. Epicoco, G. Aloisio, and M. Pulimeno, "Cuda based parallel implementations of space-saving on a gpu," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, 2017, pp. 707–714.

[65] M. Cafaro, M. Pulimeno, and P. Tempesta, "A parallel space saving algorithm for frequent items and the hurwitz zeta distribution," *Information Sciences*, vol. 329, pp. 1 –19, 2016, Special issue on Discovery Science.

[66] P. Roy, J. Teubner, and G. Alonso, "Efficient frequent item counting in multi-core hardware," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12, Beijing, China: ACM, 2012, pp. 1451–1459.

[67] Austin Appleby, *MurmurHash3*, `https://github.com/aappleby/smhasher`, 2016.

[68] *Project Gutenberg*, `https://www.gutenberg.org/`, 2017.

[69] Faraz Ahmad, *Puma Dataset*, `https://engineering.purdue.edu/~puma/datasets.htm`.

[70] W. Y. Zou, R. Socher, D. Cer, and C. D. Manning, "Bilingual word embeddings for phrase-based machine translation," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 2013, pp. 1393–1398.

[71] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," *arXiv preprint arXiv:1603.01360*, 2016.

[72] X. Glorot, A. Bordes, and Y. Bengio, "Domain adaptation for large-scale sentiment classification: A deep learning approach," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 513–520.

[73] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *International conference on machine learning*, 2015, pp. 957–966.

[74] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1,

[75] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[76] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 160–167.

[77] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2013, pp. 746–751.

[78] S. Ji, N. Satish, S. Li, and P. Dubey, "Parallelizing word2vec in shared and distributed memory," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[79] V. Rengasamy, T.-Y. Fu, W.-C. Lee, and K. Madduri, "Optimizing word2vec performance on multicore systems," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ACM, 2017, p. 3.

[80] A. Heinecke, H. Pabst, and G. Henry, "Libxsmm: A high performance library for small matrix multiplications," in *Poster and Extended Abstract Presented at SC'15*, 2015.

[81] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched gemm for gpus," in *International Conference on High Performance Computing*, Springer, 2016, pp. 21–38.

[82] J. Chen, N. Xiong, X. Liang, D. Tao, S. Li, K. Ouyang, K. Zhao, N. DeBardeleben, Q. Guan, and Z. Chen, "Tsm2: Optimizing tall-and-skinny matrix-matrix multiplication on gpus," in *Proceedings of the ACM International Conference on Supercomputing*, ACM, 2019, pp. 106–116.

[83] G. A. Miller and W. G. Charles, "Contextual correlates of semantic similarity," *Language and cognitive processes*, vol. 6, no. 1, pp. 1–28, 1991.

[84] M. U. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 307–361, 2012.

[85] Google, *Google word2vec*, `https://code.google.com/archive/p/word2vec/source/default/source`, 2015.

[86] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson, "One billion word benchmark for measuring progress in statistical language modeling," *arXiv preprint arXiv:1312.3005*, 2013.

[87] Y. Goldberg and O. Levy, "Word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

[88] N. Manerikar and T. Palpanas, "Frequent items in streaming data: An experimental evaluation of the state-of-the-art," *Data & Knowledge Engineering*, vol. 68, no. 4, pp. 415–430, 2009.

[89] M. Mahoney, *Text8 Dataset*, `http://mattmahoney.net/dc/text8.zip`, 2011.

[90] T. F. J. M. Lushan Han Abhay L. Kashyap and J. Weese, "UMBC_EBIQUITY-CORE: Semantic Textual Similarity Systems," in *Proceedings of the Second Joint Conference on Lexical and Computational Semantics*, Association for Computational Linguistics, 2013.

[91] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin, "Placing search in context: The concept revisited," *ACM Transactions on information systems*, vol. 20, no. 1, pp. 116–131, 2002.

[92] C. Spearman, "The proof and measurement of association between two things," *American journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904.

[93] P. Zangerl, P. Thoman, and T. Fahringer, "Characterizing performance and cache impacts of code multi-versioning on multicore architectures," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, IEEE, 2017, pp. 209–213.