Approximate Frequency Counting on Single-Board Computers

ABSTRACT

One of the biggest challenges today is analyzing enormous chunks of data gathered from different sources. Traditional approaches for data analysis fail to scale as their memory and time requirement are too high to be practical. Sketches are probabilistic data structures that can provide approximate results within mathematically proven error bounds while having orders of magnitude less memory and time complexity than traditional approaches. They are tailored for big, streaming data analysis on architectures even with limited memory such as single-board computers. Today, these architectures are widely exploited for IoT and edge computing since they offer multiple cores and with efficient parallel sketching schemes, they are able to manage high volumes of data streams; e.g., they can be deployed at the edge of a network to perform frequency analysis on streaming data obtained by many sensors.

In this paper, we work on the frequency estimation problem via single-board computers and evaluate the performance of a high-end server, a 4-core Raspberry Pi and an 8-core Odroid for this problem. As a sketch, we employed Count-Min Sketch which has been widely used for frequency estimation. An important tool to build a sketch is an efficient and effective hash function; we chose tabulation hashing which provides strong statistical guarantees without too much of a price in memory footprint and runtime. To hash the stream in parallel and in a cache-friendly way, we applied a novel technique and rearranged the auxiliary tables into a single one. To improve the performance further, we modified the stream processing workflow and applied some form of buffering between hash computations and sketch updates. Today, many single-board computers have heterogeneous processors in which slow and fast cores are equipped together. To utilize all these cores to their full potential, we proposed a dynamic load-balancing mechanism which significantly increased the performance of frequency estimation.

ACM Reference Format:

. 2019. Approximate Frequency Counting on Single-Board Computers. In *Proceedings of SIGMOD 2019 (SIGMOD'19).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SIGMOD'19, June 2019, Amsterdam, Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Today, many applications such as financial market [8] and network traffic analysis [16, 17] have a streaming nature. For these applications, the data can rapidly scale up to massive amounts, making them impossible to store and allowing only a single pass over the stream. Furthermore, since the data arrive at real-time, it may be impractical to process it and answer the queries, such as membership [5] and/or frequency [9], with 100% accuracy. Although an exact computation can be possible for some streams by processing the data on cutting-edge servers equipped with a large memory and powerful processor(s), enabling less power hungry devices such as single-board computers (SBC), e.g., Arduino, Raspberry Pi, Odroid etc., with smarter algorithms and data structures yields much more energy efficient and cheaper solutions for the same task. These devices are indeed cheap, they are equipped with multicore processors and they can be located at the edge of a data analysis ecosystem, which is where the data is actually generated. Furthermore, SBCs can be easily enhanced with various hardware such as cameras, sensors, microphones and software such as network sniffers to gather and process the data. Hence, exploiting their superior price/performance ratio for stream analysis is a promising path to follow. A comprehensive survey of data stream applications can be found in [21].

Sketching is a probabilistic data summarization technique; there exist various sketches in the literature tailored for different applications. These data structures can work on streaming data and help us process a query with small, usually sublinear, amount of memory for many problems [1, 9, 15, 19]. Furthermore, sketches are very useful for distributed data streams since each stream can independently be sketched and then these sketches can be combined to obtain the final sketch. To remove the bias on the data distribution, sketches frequently employ hashing. This maps the actual data to a set on which various statistical analysis can be performed for problems such as heavy-hitters, distinct element counts, rare

counts etc. Due to the implicit compression performed during this process, there is almost always a trade-off between the accuracy of the final result and the size of the sketch. A complete analysis and comparison of various sketches can be found in [12].

Sketches are tailored for different applications; one of the most popular and well-known sketches today is *Bloom Filter* [2] which is designed to answer membership queries. Despite allowing false-positives, i.e., falsely confirming the existence of a non-existent item, Bloom Filters have proven to be useful for many applications employing membership queries [4]. The simple Bloom Filter cannot perform any kind of counting on the item frequencies. However, Bloom Filter variants designed for counting tasks also exist [3, 18].

Count-Min Sketch (CMS) is a probabilistic sketch that helps to estimate the frequencies, i.e., the number of occurrences, of distinct items in a stream [12]. The frequency information is crucial to find finding heavy-hitters or rare items or detecting anomalies within a large computer network [10, 12]. Unlike Bloom Filters which use bits, CMSs store a small counter table to keep track of the frequency of each distinct element. Intuitively, the reduction to sub-linear space decreases the accuracy in the sense that the frequencies of some elements can be overestimated due to the collisions of hash functions. An important property of a CMS is that the error is always one sided; that is, the sketch never underestimates the frequencies.

In this work, we focus on the frequency estimation problem on single-board computers. We use Count-Min Sketch and evaluate stream processing performance of a high-end server and two multicore SBCs; Raspberry Pi 3 Model B+ and Odroid-XU4. As the hashing function, we used *tabulation hashing* which is simple and can provide strong statistical guarantees [22, 25]. Our contributions can be summarized as follows:

- We first propose a cache-friendly hashing scheme to compute multiple hashes at a time which is a crucial operation for CMS. The proposed scheme is also useful for applications hashing the same element multiple times. This is a common technique in many sketch-based data processing algorithms used to reduce the estimation error.
- As our second contribution, we restructure the CMS construction in a way to process the data stream in parallel while avoiding possible race-conditions on a *single* CMS table. Such a synchronization mechanism is necessary, especially for sketches, since race-conditions not only degrade the performance but also increase the amount of error on estimation.
- Recently, many devices including SBCs have both fast and slow CPU cores to reduce the energy consumption by assigning compute intensive tasks to the faster cores and the rest to the slower ones. The relative performance of the fast and slow cores differ for various devices/processors. Furthermore, this relative performances changes even on the same device but for different computations. Under this heterogeneity, it is hard to use all the available cores efficiently while maintaining the CMS. As our third contribution, we propose a parallelization and load-balancing technique that distributes the work evenly to all the available cores and uses them as efficient as possible. The proposed technique is dynamic; it is not specialized for a single device and can be employed on various devices having heterogeneous cores.

The rest of the article is organized as follows: Section 2 presents the background and describes the notation that will be used later. In Sections 3 and 4, the cache-friendly tabular hashing mechanism and corresponding parallel CMS construction are described. Section 5 describes the load balancing technique used for Odroid-XU4 and Section 6 presents the experimental results. The related work is summarized in Section 7. Section 8 concludes the paper.

2 NOTATION AND BACKGROUND

Let $\mathcal{U} = \{1, \cdots, n\}$ be the universal set where the elements in the stream are coming from. Let N be size of the stream $\mathtt{s}[.]$ where $\mathtt{s}[i]$ denotes the ith element in the stream. We will use f_x to denote the frequency of an item. Hence,

$$f_x = |\{x = \mathtt{s}[i] \,:\, 1 \le i \le N\}|.$$

Given two parameters ϵ and δ, a Count-Min Sketch is constructed as a two-dimensional counter array with $d = \lceil \ln(1/\delta) \rceil$ rows and $w = \lceil e/\epsilon \rceil$ columns. Initially, all the counters inside the sketch are set to 0.

There are two fundamental operations for a CMS; the first one is *insert*(x) which constructs/updates internal sketch counters to process the items in the stream. To insert $x \in \mathcal{U}$, the counters $\mathtt{cms}[i][h_i(x)]$ are incremented for $1 \le i \le d$, i.e., a counter from each sketch row is incremented where the column IDs are obtained from the hash values. Figure 1 illustrates a single insertion, and Algorithm 1 gives the pseudocode to sequentially process a stream $\mathtt{s}[.]$ of size N and construct a CMS.

The second operation for CMS is *query*(x) which estimates the frequency of $x \in \mathcal{U}$ as

$$f'_x = min_{1 \le i \le d}\{\mathtt{cms}[i][h_i(x)]\}.$$

With $d \times w$ memory, the sketch satisfies that $f_x \le f'_x$ and

$$\Pr\left(f'_x \ge f_x + \epsilon N\right) \le \delta.$$

Hence, the error is additive and always one-sided, i.e., the estimation is always larger than the actual frequency. Furthermore, for ϵ and δ small enough, the amount of the error is also bounded with high probability. Hence, especially for

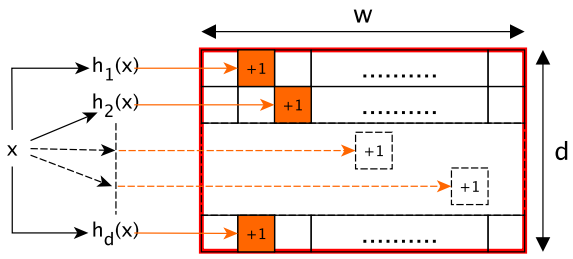


Figure 1: Illustration of the insert operation for Count-Min Sketch.

ALGORITHM 1: CMS-CONSTRUCTION

Input: ϵ: error factor
δ: error probability
s[.]: a stream with N elements from n distinct elements
$h_i(.)$: pairwise independent hash functions where for $1 \le i \le d$, h_i: $\mathcal{U} \rightarrow \{1, \cdots, w\}$ and $w = \lceil e/\epsilon \rceil$
Output: cms[.][.]: a $d \times w$ counter sketch where $d = \lceil 1/\delta \rceil$

```
for i ← 1 to d do
    for j ← 1 to w do
        cms[i][j] ← 0
    end
end
for i ← 1 to N do
    x ← s[i] for j ← 1 to d do
        col ← h_j(x) cms[j][col] ← cms[j][col] +1
    end
end
```

frequent items, i.e., when f_x is large, the ratio of the estimation and the actual frequency approaches to one.

2.1 Tabulation Hashing

CMS uses pairwise independent hash functions to provide the desired properties stated above. There exists a separate hash function for each row of the CMS with a range equal to the range of columns. In this work, we use Tabulation Hashing [27] which has been recently analyzed by Patrascu and Thorup et al. [22, 25] and shown to provide strong statistical guarantees despite of its simplicity. Furthermore, it has been shown that a slightly more sophisticated version of tabulation hashing is almost as fast as the classic multiply-mod-prime scheme, i.e., $(ax + b) \bmod p$.

Assuming each element in $\mathcal{U}$ is represented in 32 bits and the desired output is also 32 bits, the tabulation hashing works as follows: first a 4×256 table is generated and filled with random 32-bit values. Given a 32-bit input x, each character, i.e., 8-bit value, of x is used as an index for the corresponding row. Hence, four 32-bit values, one from each row, are extracted from the table. The XOR of these 32-bit values are returned as the hash value. The implementation we use in this work is given in Figure 2.

```
uint32_t hash(uint32_t x, uint32_t table[4][256]) {
  uint32_t h = 0;
  for (uint32_t i = 0; i < 4; i++) {
    uint32_t c = x & 0x000000ff;
    h ^= table[i][c];
    x = x >> 8;
  }
  return h;
}
```

Figure 2: Naive tabulation hashing in C++

3 MULTIPLE TABULATION AT ONCE

Hashing the same item multiple times is a common technique applied in sketch-based data processing to reduce the error of the estimation. One can use a single row for CMS, i.e., set $d = 1$ and answer the query by reporting the value of the counter corresponding to the hash value. However, using multiple rows reduces the probability of having large estimation errors.

Although the auxiliary data used in tabulation hashing are usually small and can fit into a cache, the spatial locality of the accessed table elements, i.e., the distance among them in memory, is not good enough since each access is performed to a different table row. A naive, cache-friendly rearrangement of the entries in the tables is also not possible for applications performing a single hash per item; the indices for the table rows are obtained from adjacent chunks in the binary representation of the hashed item which are usually not correlated. Hence, there is no relation whatsoever among them to help us to fix the access pattern for all possible stream elements.

In addition to CMS, for many frequency sketches, e.g., Count Sketch [9] and Count-Mean-Min Sketch [20], the same item is hashed more than once. For tabulation hashing, this yields an interesting optimization; there exist multiple hash functions and hence, more than one hash table. Although, the entries in a single table is still accessed in a somehow irregular fashion, the accessed coordinates in all the tables are the same for different tables. This can be observed on the left side of Figure 3. Hence, the columns of the tables can be combined in an alternating fashion as shown in the right side of the figure. In this approach, when only a single thread is responsible from computing the hash values for a single item to CMS, the cache can be utilized in a better way since the memory locations accessed by that thread are adjacent. Hence, the computation will pay the penalty for a cache-miss only once for each 8-bit character of a 32-bit item. This proposed scheme is called *merged tabulation*.

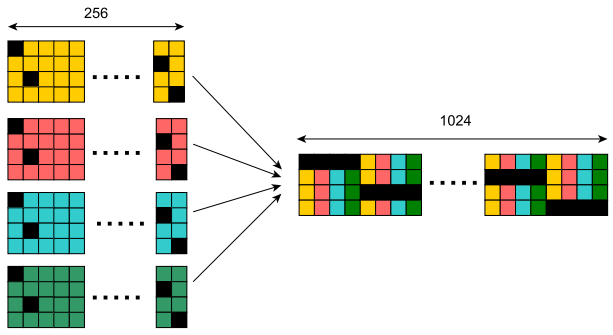


Figure 3: Memory access patterns for naive and merged tabulation for four hashes. The hash tables are colored with different colors. The accessed locations for both approaches are shown in black.

With merged tabulation, the consecutive elements in the merged table belong to different hash values. Since these values require multiple XOR operations, we need to store the partial results in the memory. A simple C++ implementation of merged tabulation which uses a single 4 × 1024 table to compute four hashes at once is given in Figure 4. We provide actual implementations here for clarity as well as reproducibility.

```
void hash(uint32_t x, uint32_t hashes[8],
          uint32_t table[4][1024]) {
  uint32_t c = (x & 0x000000ff) << 2;
  x = x >> 8;

  hashes[0] = table[0][c];
  hashes[1] = table[0][c + 1];
  hashes[2] = table[0][c + 2];
  hashes[3] = table[0][c + 3];

  for (uint32_t i = 1; i < 4; i++) {
    c = (x & 0x000000ff) << 2;
    x = x >> 8;

    hashes[0] ^= table[i][c];
    hashes[1] ^= table[i][c + 1];
    hashes[2] ^= table[i][c + 2];
    hashes[3] ^= table[i][c + 3];
  }
}
```

Figure 4: Merged tabulation hashing in C++.

4 PARALLEL COUNT-MIN SKETCH CONSTRUCTION

Since multiple CMS sketches can be combined, on a multi-core hardware, each thread can process a different part of the data (with the same hash functions) to construct a partial CMS. These partial sketches can then be aggregated by adding the counter values in the same locations to obtain the final value on the combined CMS sketch. Although this approach has been already proposed in the literature and requires no synchronization until all partial sketches are constructed, compared to constructing the final CMS table directly, it consumes more memory. Furthermore, with this straightforward parallelization, the amount of the memory is increasing with increasing number of threads. Hence, the multi-sketch approach is not scalable.

Constructing a single CMS sketch in parallel is not a straightforward task. The naive approach is assigning each item in the stream to a single thread and let it do the updates (i.e., increment operations) on CMS counters. The pseudocode of this parallel CMS construction is given in Algorithm 2. However, to compute the counter values correctly, the approach presented in Algorithm 2 requires a significant synchronization overhead; when a thread processes a single data item, it accesses an arbitrary column of each CMS row. Hence, race conditions may reduce the estimation accuracy. In addition, these memory accesses are probable causes of false sharing which prevents the implementation to utilize the hardware to its full potential. To avoid the pitfalls stated above, one can allocate locks on the counters before every increment operation. However, this lock-based manual synchronization is too costly to be applied in practice.

As an alternative approach, one can assign each item to multiple threads which evaluate different hash functions hence touch different rows of the CMS table. However, this approach cannot use the merged tabulation described in the previous section since, each thread performs only a single hash of an item.

ALGORITHM 2: NAIVE-PARALLEL-CMS

Input: ϵ: error factor
δ: error probability
s[.]: a stream with N elements from n distinct elements
$h_i(.)$: pairwise independent hash functions where for $1 \le i \le d$, h_i: $\mathcal{U} \rightarrow \{1, \cdots, w\}$ and $w = \lceil e/\epsilon \rceil$
τ: no threads

Output: cms[.][.]: a $d \times w$ counter sketch where $d = \lceil 1/\delta \rceil$
Reset all the cms[.][.] counters to 0 (as in Algorithm 1).
for $i \leftarrow 1$ **to** N **in parallel do**
  $x \leftarrow s[i]$ hashes[.] $\leftarrow$ MERGEDHASH(x)
  for $j \leftarrow 1$ **to** d **do**
    $col \leftarrow$ hashes[j] cms[j][col] $\leftarrow$ cms[j][col] +1
  end
end

In this work, we propose a *buffered parallel* execution to alleviate the above mentioned issues; we (1) divide the data into batches and (2) process a single batch in parallel in two phases; a) multi-hashing and b) CMS counter updates. In the proposed approach, after each batch, the threads synchronize

and process the next one. For batches with b elements, the first phase requires a buffer of size $b \times d$ to store the hash values, i.e., column ids, which then will be used in the second phase to update corresponding CMS counters. Such a buffer allows us to use merged tabulation effectively during the first phase. In our implementation, the counters in a row are updated by the same thread hence, there will be no race conditions and probably much less false sharing. We also experiment with extra padding between the table rows which we found to be useful on a server (but not SBCs), especially for CMS tables with a small w value. Algorithm 3 gives the pseudocode of the proposed buffered CMS construction approach.

ALGORITHM 3: BUFFERED-PARALLEL-CMS

Input: ϵ: error factor
δ: error probability
s[.]: a stream with N elements from n distinct elements
$h_i(.)$: pairwise independent hash functions where for $1 \le i \le d$, h_i: $\mathcal{U} \rightarrow \{1, \cdots, w\}$ and $w = \lceil e/\epsilon \rceil$
b: batch size (assumption: divides N)
τ: no threads (assumption: divides d)
Output: cms[.][.]: a $d \times w$ counter sketch where $d = \lceil 1/\delta \rceil$
Reset all the cms[.][.] counters to 0 (as in Algorithm 1)

```
for i ← 1 to N/b do
    j_end ← i × b  j_start ← j_end − b + 1
    for j ← j_start to j_end in parallel do
        x ← s[j]  ℓ_end ← j × d  ℓ_start ← ℓ_end − d + 1
        buf[ℓ_start, ··· , ℓ_end] ← MergedHash(x)
    end
    Synchronize the threads, e.g., with a barrier
    for t_id ← 1 to τ in parallel do
        for j ← 1 to b do
            nrows ← d/τ  r_end ← t_id × nrows
            r_start ← r_end − nrows + 1 for r ← r_start to r_end do
                col ← buf[((j − 1) × d) + r]
                cms[r][col] ← cms[r][col] + 1
            end
        end
    end
end
```

5 MANAGING HETEROGENEOUS CORES

A recent trend on SBC design is heterogeneous multiprocessing which had been widely adopted by mobile devices. Recently, some ARM-based devices including SBCs use the *big.LITTLE* architecture equipped with power hungry but faster cores, as well as battery-saving but slower cores. The faster cores are suitable for compute-intensive, time-critical tasks where the slower ones perform the rest of the tasks and save more energy. In addition, tasks can be dynamically swapped between these cores on the fly. One of the SBCs we experiment in this study has an 8-core Exynos 5422 Cortex processor having four fast and four relatively slow cores.

Assume that we have d rows in CMS and d cores on the processor; when the cores are homogeneous, Algorithm 3 works efficiently with static scheduling since, each thread performs the same amount of merged hashes and counter updates. When the cores are heterogeneous, the first inner loop (for merged hashing) can be dynamically scheduled: that is the batch can be divided into smaller chunks and these chunks can be dynamically assigned to the threads on the fly resulting faster cores seamlessly hash more items. However, *dynamic scheduling* will incur a runtime overhead. Furthermore, the same technique is not applicable to the second inner loop where the counter updates are performed: in the proposed buffered approach, Algorithm 3 divides the workload among the threads by assigning each row to a different one. When the fast cores are done with the updates, the slow cores will still be working. Furthermore, faster cores cannot help to the slower ones by stealing a portion of their remaining jobs since when two threads work on the same CMS row, race conditions will yield more errors on the final counter values.

To alleviate these problems, we propose to pair a slow core with a fast one. In this approach, each core pair is responsible from two rows where the batch is processed in two stages as shown in Figure 5. In the first stage, the items on the batch are processed in a way that the threads running on faster cores update the counters on even numbered CMS rows whereas the ones running on slower cores update the counters on odd numbered CMS rows. When the first stage is done, the thread/core pairs exchange their row ids and resume from the item their mate stopped in the first stage. In both stages, the faster threads process $fastBatchSize$ items and the slower ones process $slowBatchSize$ items where $b = fastBatchSize + slowBatchSize$.

To avoid the overhead of dynamic scheduling and propose a generic solution, we start with $fastBatchSize = b/2$ and $slowBatchSize = b/2$ and by measuring the time spent by the cores, we dynamically adjust them to distribute the workload among all the cores as fairly as possible. Let t_F and t_S be the times spent by a fast and slow core, respectively, on average. Let

$$s_F = \frac{fastBatchSize}{t_F} \quad \text{and} \quad s_S = \frac{slowBatchSize}{t_S}$$

be the speed of these cores for the same operation, e.g., hashing, CMS update etc. We then solve the equation

$$\frac{fastBatchSize + x}{s_F} = \frac{slowBatchSize - x}{s_S}$$

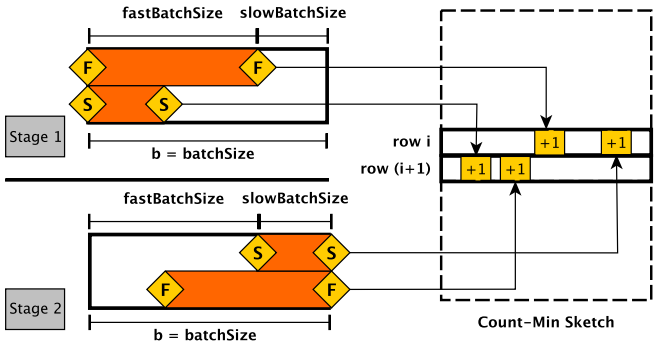


Figure 5: For a single batch, rows i and $i+1$ of CMS are updated by a fast and a slow core pair in two stages. In the first stage, the fast core performs row i updates and the slow core processes row $i+1$ updates. In the second stage, they exchange the rows and complete the remaining updates on the counters for the current batch.

for x and update the values as

$$fastBatchSize = fastBatchSize + x$$
$$slowBatchSize = slowBatchSize - x$$

for the next batch. One can apply this method iteratively for a few batches and use the final values or their average to obtain a generic and dynamic solution for such computations. In this study, we applied this technique both for hashing and counter update phases of the proposed buffered CMS generation algorithm, i.e., for both of the inner loops in Algorithm 3.

6 EXPERIMENTAL RESULTS

To understand the performance of the proposed approach, we perform the experiments on three architectures:

- **Arch-1** is a server running on 64 bit CentOS 6.5 equipped with 64GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz where each socket has 15 cores (30 in total). Each core has a 32KB L1 and a 256KB L2 cache, and each socket has a 30MB L3 cache.
- **Arch-2** (Raspberry Pi 3 Model B+) is a quad-core 64-bit ARM Cortex A-53 clocked at 1.4 GHz equipped with 1 GB LPDDR2-900 SDRAM. Each core has a 32KB L1 cache, and the shared L2 cache size is 512KB.
- **Arch-3** (Odroid XU4) is an octa-core heterogeneous multi-processor. There are four A15 cores running on 2Ghz and four A7 cores running on 1.4Ghz. The SBC is equipped with a 2GB LPDDR3 RAM. Each core has a 32KB L1 cache. The fast cores have a shared 2MB L2 cache and slow cores have a shared 512KB L2 cache.

For the multicore implementations, we use C++11 and OpenMP. We use `gcc 5.3.0` with the `-O3` optimization flag enabled on **Arch-1**. On **Arch-2**. the compiler is `gcc 6.3.0`, and on **Arch-3**, the compiler is `gcc 7.3.0`. For both SBCs, `-O3` optimization flag is also enabled.

To generate the datasets used in the experiments, we used a *Zipfian* distribution [26]. Although this study mainly focuses on the performance which is not directly related to the distribution of the values in the stream s[.], many data in real world such as number of paper citations, file transfer sizes, word frequencies etc. fit to a Zipfian distribution. Furthermore, although the error analysis for CMS do not depend on the data, Zipfian is a common choice for the studies in the literature to benchmark the estimation accuracy of data sketches. For completeness and to cover the real-life datasets better, we used the shape parameter $\alpha \in \{1.1, 1.5, 3.0\}$ to understand how the performance of our parallel CMS construction algorithm changes when the distribution of the items, i.e., the variance on the frequencies, change. Note that when the variance between the item frequencies increase, i.e., when the frequent items become more dominant in the stream, some counters are touched much more than the others. This happens with increasing α and is expected to increase the performance since most of the times, the counters will already be in the cache. To see the other end of the spectrum, in addition to *Zipfian*, we also used *Uniform* distribution to measure the performance where all counters are expected to be touched the same number of times.

We employ two sets of (ϵ, δ) values to generate CMS tables; in the first one, we used $(\epsilon = 0.01, \delta = 0.004)$ to generate a **small** sketch with $d = \lceil \log_2(1/\delta) \rceil = 8$ and $w = 211$ where w is selected as the first prime after $2/\epsilon$. In addition, we generate a relatively **large** sketch with $(\epsilon = 0.001, \delta = 0.004)$ with $d = 8$ rows and $w = 2003$ columns.

For the experiments on **Arch-1**, we choose $N = 2^{30}$ elements from a universal set $\mathcal{U}$ of cardinality $n = 2^{25}$. For **Arch-2** and **Arch-3**, we use $N = 2^{25}$ and $n = 2^{20}$. For all architectures, we used $b = 1024$ as the batch size. Each data point in the tables and charts given below is obtained by averaging ten runs.

6.1 Parallel CMS construction with partial sketches

In the first set of experiments, we measure the performance improvement due to merged tabulation in the partial sketches setting. That is, each thread is responsible from a different part of the stream and construct a partial sketch for that part. As mentioned before, this approach consumes more memory and its memory consumption is proportional to the number of threads used. Yet, no synchronization mechanism is required since all the CMS constructions are independent

from each other. The results of this set of experiments are given in Table 1.

As the results in Table 1 show, merged tabulation significantly increases the performance of the CMS construction on both **Arch-1** and **Arch-2** even with extra overhead of managing an auxiliary array, i.e., hashes[.] in Figure 4. That is instead of using a single register as in naive tabulation (Figure 2), in merged tabulation, the intermediate hash values are first written to the auxiliary memory. Once all d hashes are computed, they are read back from the memory again. However, these extra reads as well as the hash table accesses are more cache friendly.

The impact of merged tabulation changes for different architectures and for **small** and **large** sketches. On **Arch-1**, the runtime decreases when the shape parameter α increases. With $\tau = 16$, for large sketches the improvement is in between 6%–10%. For small ones, it is in between 14%–46% and the largest improvement is obtained with the uniform distribution where all the counters, as well as the entries of the auxiliary tables are expected to be touched the same number of times.

Merged tabulation yields significant speedups on **Arch-2**; when $\tau = 4$, for both small and large sketches, the improvement is in between 8%–13%. The proposed technique also improves the runtime on **Arch-3** with $\tau = 8$, around 3%–13%. However, the improvement is usually less than 5%.

An interesting observation is the relative performances of the architectures on small and large sketches. Considering the cases with $\tau \geq 4$, **Arch-1** and **Arch-3** almost always process the stream faster when the sketch is large, i.e., 8×2003. For **Arch-1** the difference between the performance reduces with merged tabulation. On the contrary to these architectures, **Arch-2** processes the stream faster when the sketch is small. We note that this architecture has the least amount of cache available and hence, it suffers more when the memory footprint is large.

6.2 Parallel CMS construction with a single sketch

Although using partial sketches is pleasingly parallelizable and can be practical for some use cases, it is not a good approach for memory restricted devices such as SBCs. The additional memory might be required by other applications running on the same hardware and/or other types of skeches being maintained at the same time for the same or a different data stream. A straightforward parallelization with multiple sketches uses $(d \times w \times \tau)$ counters where each counter can have a value as large as N. Hence, memory consumption is

$$(d \times w \times \tau \times \log N)$$

bits. On the other hand, a single sketch with buffering consumes

$$(d \times ((w \times \log N) + (b \times \log w)))$$

bits since there are $(d \times b)$ entries in the buffer and each entry is a column ID on CMS. For instance, with $\tau = 8$ threads, $\epsilon = 0.001$ and $\delta = 0.004$), the straightforward parallelization with multiple sketches require $(8 \times 2003 \times 8 \times 30) = 3.85$Mbits whereas using single sketch requires $(8 \times ((2003 \times 30) + (1024 \times 11))) = 570$Kbits. This is why in the rest of the experiments, we focus on the single CMS case. We also keep using merged tabulation.

Table 2 shows the results of the experiments in which a single sketch is constructed in parallel. As expected, the naive parallel approach (first set of columns in Table 2) suffers from false sharing with multiple threads on **Arch-1**. However, we did not observe the such a dramatic slowdown on **Arch-2** and **Arch-3**. We think that the behavior is different since ARM-based processors use a weaker memory consistency model which reduces the impact of race conditions encountered during parallel execution.

For sequential executions, the proposed buffering technique behaves differently on these architectures; Since storing intermediate hash values has an overhead, we expect a slowdown in a sequential execution. This is what we have observed on **Arch-1**; the runtimes are increased to 67–68 seconds from 60–63 seconds both for small and large sketches. However, even with this extra overhead, the technique significantly increases the sequential performance of **Arch-2**, especially for the large sketches. Although not as significant as **Arch-2**, buffering yields a slight improvement even for **Arch-3**. We believe that the difference is due to the existence of a relatively smaller cache sizes on **Arch-2**. On **Arch-3**, we have a slightly larger cache, and **Arch-1** has the largest cache sizes. In the naive approach, an item is hashed with merged tabulation and then d counters are updated. This process is repeated in an alternating fashion for all the data items in the stream. Hence, the merged hash table and the CMS are accessed one after another which makes these structures alive in the cache at the same time as much as possible which may result cache trashing. An advantage of buffering is that it separates the accesses to these structures. Hence, it tends not to use the cache for both structures at the same time. Thus for a sequential execution, where the pressure on the cache is low, the overhead of buffering exceeds its benefits for larger cache sizes. On the contrary, it becomes useful for cache-restricted devices such as SBCs even for a sequential execution despite the overhead.

With multiple threads, the pressure on the cache increases further. In this case, the buffered parallel approach given in Algorithm 3 (the second set of columns in Table 2) is significantly better than the naive parallel one; it improves the

			Naive Tabulation					Merged Tabulation				
			τ=1	τ=2	τ=4	τ=8	τ=16	τ=1	τ=2	τ=4	τ=8	τ=16
Uniform	**Arch-1**	**small**	64.3	33.1	23.1	14.0	7.6	60.4	34.4	17.5	10.0	5.2
		large	72.3	36.8	20.0	9.8	4.9	60.6	33.3	17.1	8.8	4.5
	Arch-2	**small**	31.8	16.0	8.0			28.3	14.2	7.1		
		large	33.2	16.7	8.4			30.8	15.4	7.8		
	Arch-3	**small**	22.7	11.4	5.9	3.4		21.8	10.9	5.6	3.2	
		large	21.5	10.9	5.6	3.5		20.2	10.2	5.3	3.1	
Zipf 1.1	**Arch-1**	**small**	65.8	32.2	19.6	11.9	6.1	60.0	33.1	16.6	9.0	4.8
		large	65.7	33.2	18.4	9.4	4.6	60.0	32.5	16.8	8.4	4.2
	Arch-2	**small**	31.5	15.9	8.4			28.8	14.6	7.5		
		large	32.4	16.1	10.1			29.6	15.0	8.9		
	Arch-3	**small**	22.7	11.4	5.9	3.4		21.8	10.9	5.6	3.3	
		large	21.3	10.8	5.5	3.4		20.2	10.1	5.2	3.2	
Zipf 1.5	**Arch-1**	**small**	63.2	32.7	17.8	10.6	5.7	59.7	30.7	15.4	7.9	4.0
		large	63.8	32.0	16.3	8.2	4.1	59.7	30.3	15.3	7.6	3.8
	Arch-2	**small**	31.1	15.7	8.0			28.6	14.4	7.4		
		large	30.9	15.5	7.8			28.3	14.3	7.2		
	Arch-3	**small**	22.5	11.4	5.8	3.4		21.9	10.9	5.6	3.3	
		large	20.9	10.6	5.4	3.3		20.1	10.1	5.2	3.2	
Zipf 3.0	**Arch-1**	**small**	62.9	31.9	16.4	8.3	4.2	59.8	29.9	14.9	7.5	3.7
		large	63.0	31.5	15.7	7.9	3.9	60.0	29.9	15.0	7.5	3.7
	Arch-2	**small**	31.0	15.8	8.3			28.7	14.6	7.4		
		large	31.2	15.5	9.8			28.4	14.4	8.9		
	Arch-3	**small**	22.5	11.4	5.8	3.4		21.9	11.0	5.6	3.3	
		large	20.3	10.3	5.3	3.2		19.8	9.9	5.0	3.1	

Table 1: Parallel CMS construction times (in seconds) with naive and merged tabulation for $\tau \in \{1, 2, 4, 8, 16\}$ threads on Arch-1, $\tau \in \{1, 2, 4\}$ threads on Arch-2, and $\tau \in \{1, 2, 4, 8\}$ threads on Arch-3. The stream size for Arch-1 is $N = 2^{30}$. For Arch-2 and Arch-3, N is set to 2^{25}. The rows labeled with small and large denote the 8×211 and 8×2003 CMS tables, respectively. For this set of experiments, dynamic scheduling is used to overcome the heterogeneity on Arch-3.

runtimes and yields a better scalability. However, on **Arch-1**, the proposed approach yields much less parallel efficiency compared to the other results. This is expected since, with multiple threads performing simultaneous updates on the sketch, false sharing reduces the performance. Furthermore, the impact will be more drastic for smaller sketches which is the case in our experiments. To avoid false sharing as much as possible, we added extra padding to each row of CMS sketch. The third set of columns in Table 2 shows the execution times after padding. As the results confirm, padding helps to get rid of the problems arose from increased coherency traffic due to the concurrent updates on the same cache line by multiple threads.

6.2.1 Managing heterogeneous cores: For **Arch-3**, which equipped with a heterogeneous multi-processor, we applied the workload distribution technique described in Section 5. We pair each slow core with a fast one, virtually divide each batch into two parts and make the slow core always run on smaller part. As mentioned before, for each batch, we dynamically adjust the part sizes based on the previous runtimes of slow and fast cores. Figure 6 shows the ratio

$$F2S = \frac{fastBatchSize}{slowBatchSize}$$

for the first 256 batches. As expected, the best F2S changes w.r.t. the computation performed; for hashing a 4-to-1 division of workload yields a fair distribution. However, for CMS updates a 1.8-to-1 division is the best one yielding a fair distribution. As the figure shows, the F2S ratio becomes stable after a few batches for both phases. Hence, one can stop the update process after ~ 30 batches and use a constant F2S for the later ones.

The proposed workload distribution technique improves the runtime on **Arch-3** and increase the parallel efficiency by 15%–30% for $\tau = 8$ threads.

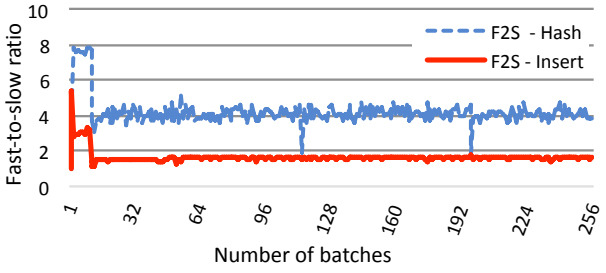
			Naive Parallel (Algorithm 2)					Buffered Parallel (Algorithm 3)					Buffered Parallel + P (Algorithm 3 + Padding)				
			τ=1	τ=2	τ=4	τ=8	τ=16	τ=1	τ=2	τ=4	τ=8	τ=16	τ=1	τ=2	τ=4	τ=8	τ=16
Uniform	**Arch-1**	**small**	61.5	299.9	304.4	201.7	156.9	68.3	60.9	45.5	30.4	28.0	68.6	43.5	22.8	13.0	10.6
		large	60.1	233.8	229.2	139.2	96.5	68.7	51.4	29.8	17.1	14.4	68.8	45.0	23.2	13.0	10.6
	Arch-2	**small**	28.5	15.4	7.8			27.5	13.8	6.9			27.6	13.8	6.9		
		large	30.8	15.4	7.7			26.8	13.3	6.6			26.8	13.3	6.6		
	Arch-3	**small**	22.0	11.2	6.0	5.3		21.8	12.2	6.3	4.5		21.8	12.2	6.2	4.5	
		large	20.3	10.3	5.5	5.1		20.6	11.1	5.7	4.3		20.6	11.1	5.7	4.3	
	Arch-3++	**small**	22.0	11.2	6.0	5.3		21.8	12.2	6.3	3.9		21.8	12.2	6.2	3.9	
		large	20.3	10.3	5.5	5.1		20.6	11.1	5.7	3.7		20.6	11.1	5.7	3.7	
Zipf 1.1	**Arch-1**	**small**	60.3	287.6	287.3	179.3	125.9	68.1	58.4	40.1	26.2	24.1	68.1	43.5	22.9	13.0	10.6
		large	60.9	146.1	222.2	159.1	130.5	68.3	51.4	28.7	16.0	13.2	68.4	46.4	24.3	13.6	10.8
	Arch-2	**small**	28.7	15.5	8.6			27.7	13.9	7.4			27.9	13.9	7.4		
		large	30.2	15.4	9.5			26.8	13.3	8.1			27.1	13.3	8.1		
	Arch-3	**small**	22.1	11.3	6.0	5.3		21.7	12.1	6.2	4.5		21.7	12.1	6.2	4.5	
		large	20.3	10.4	5.5	5.0		20.4	11.1	5.7	4.3		20.5	11.1	5.7	4.3	
	Arch-3++	**small**	22.1	11.3	6.0	5.3		21.7	12.1	6.2	3.8		21.7	12.1	6.2	3.8	
		large	20.3	10.4	5.5	5.0		20.4	11.1	5.7	3.6		20.5	11.1	5.7	3.6	
Zipf 1.5	**Arch-1**	**small**	62.8	331.8	314.2	206.8	166.1	68.0	53.0	34.2	20.6	18.5	68.1	44.6	23.5	13.1	10.6
		large	63.4	214.2	257.5	193.6	170.5	68.3	47.5	25.8	14.6	12.1	68.3	43.7	23.1	13.0	10.5
	Arch-2	**small**	30.7	16.4	8.5			27.4	13.7	6.9			27.3	13.7	7.0		
		large	30.8	16.1	10.3			26.5	13.2	7.4			26.5	13.2	7.5		
	Arch-3	**small**	22.1	11.3	6.0	5.7		21.7	12.1	6.2	4.4		21.7	12.1	6.2	4.4	
		large	20.1	10.4	5.5	5.4		19.9	10.9	5.6	4.2		20.0	11.0	5.6	4.3	
	Arch-3++	**small**	22.1	11.3	6.0	5.7		21.7	12.1	6.2	3.8		21.7	12.1	6.2	3.8	
		large	20.1	10.4	5.5	5.4		19.9	10.9	5.6	3.6		20.0	11.0	5.6	3.5	
Zipf 3.0	**Arch-1**	**small**	62.9	372.5	340.8	278.5	250.8	67.6	45.2	24.6	13.8	11.4	67.7	43.1	22.9	12.8	10.5
		large	62.9	319.2	349.0	282.1	253.5	67.7	38.6	22.6	13.0	11.0	68.1	44.4	23.1	13.1	11.2
	Arch-2	**small**	30.7	16.4	8.5			27.0	13.5	7.2			27.3	13.7	7.2		
		large	30.8	16.1	10.3			26.2	13.0	7.4			26.3	13.1	7.4		
	Arch-3	**small**	21.5	12.1	6.2	4.4		21.1	11.7	6.0	4.4		20.3	11.4	5.8	4.4	
		large	19.7	11.2	5.8	4.3		18.7	10.6	5.3	4.3		19.2	11.0	5.5	4.3	
	Arch-3++	**small**	21.5	12.1	6.2	4.4		21.1	11.7	6.0	3.6		20.3	11.4	5.8	3.6	
		large	19.7	11.2	5.8	4.3		18.7	10.6	5.3	3.3		19.2	11.0	5.5	3.4	

Table 2: Parallel CMS construction times (in seconds) with a single sketch for $\tau \in \{1, 2, 4, 8, 16\}$ threads on Arch-1, $\tau \in \{1, 2, 4\}$ threads on Arch-2, and $\tau \in \{1, 2, 4, 8\}$ threads on Arch-3. Arch-3++ represents the approach to take care of the heterogeneity which we only applied for $\tau = 8$. For less number of cores, only fast cores are employed. The stream size for Arch-1 is $N = 2^{30}$. For Arch-2 and Arch-3, N is set to 2^{25}. The rows labeled with small and large denote the 8×211 and 8×2003 CMS tables, respectively. The Naive Parallel construction does not apply buffering or padding whereas the next two sets of columns apply these techniques. For Buffered Parallel construction w/out padding, when $\tau = 16$, all threads are utilized for the first phase of Algorithm 3, i.e., hashing, but only $d = 8$ of them are used for the second phase.

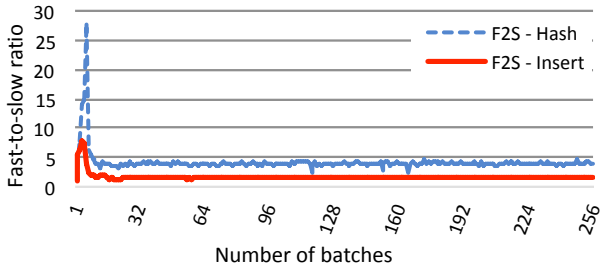
6.2.2 Analyzing the time spent for different phases: Figure 7 summarizes all the experiments on parallel construction of a single CMS sketch with individual execution times for the first and the second phases of buffered parallel executions. For all the experiment, the first phase, i.e., hashing phase is takes much less time compared to the second phase, i.e., CMS update phase. An interesting observation is that for almost of all executions, the relative execution time of CMS update phase to whole execution time is much larger for SBCs. We believe that this is due to much faster execution of this phase on **Arch-1**.

6.3 Other optimizations and details

In Algorithm 3, there is an implicit barrier between the first and second phases of the buffered construction. This barrier guarantees that the buffer is always full and ready to be processed at the beginning of the second phase. For the experiments given above, we removed this barrier on **Arch-1** and **Arch-2** but keep it on **Arch-3**. That is we implemented



(a) small on Arch-3++



(b) large on Arch-3++

Figure 6: Plots of fast-to-slow ratio $F2S = \frac{fastBatchSize}{slowBatchSize}$ of hashing and CMS update phases for consecutive batches and for small (left) and large (right) sketches.

the buffered algorithms with OpenMPs `nowait` directive on the first two architectures. This may have a negative impact on the accuracy; when threads do not wait for each other, there can exist an entry in the buffer which is not computed yet but being processed by a thread to update a counter. However, when b is large, e.g., 1024, and cores are homogeneous, the probability of a thread using a buffer entry which is not correctly set is small. To analyze this option further, we performed two experiments; first, as Table 3 shows, removing the barrier usually has a positive impact on the performance on both **Arch-1** and **Arch-2**. However, it does not change the execution time on **Arch-3**. Hence, we opt to keep the barrier for this architecture.

To understand how the estimates change if the barrier is removed, we measured how much they deviate from a sequential execution (which is the same as parallel executions with barriers). Figure 8 show the absolute differences on the estimations when the barrier is kept or removed. The most frequent 1000 items are used for both charts in the figure. As the charts show, the differences are negligible on on **Arch-1** and **Arch-2** considering that the real frequencies are in the order of tens of thousands. Hemce, we opt to remove the barrier on these architectures.

7 RELATED WORK

CMS is proposed by Cormode and Muthukrishnan structure to summarize data streams [12] and it quickly became popular. They also suggest on sequential, parallel and hardware implementations of CMS [11] where the first parallel algorithm distributes rows to the threads so each thread computes the corresponding hash value. However, the hash computations are not merged as in our study. Another suggested algorithm is using multiple CMSs from different sub-streams and merge the results later.

Cafaro et al. propose an augmented frequency sketch for time-faded heavy hitters: recent items are considered to be more important [7]. The authors extend this work via parallelization by dividing the stream into sub-streams and generating multiple sketches instead of a single one [6]. When a query is issued, the result is computed using a reduce operation (as we did with multiple CMS tables in this study). There are studies in the literature employing synchronization primitives such as atomic operations for frequency counting [14]. To the best of our knowledge, our work is the first cache-friendly, synchronization-free, single-table CMS algorithm specifically designed for limited-memory multicore architectures such as SBCs. Furthermore, it utilizes tabular hashing which is recently shown to provide good statistical properties and reported to be fast [13, 25]. When multiple hashes on the same item are required, which is the case for many sketches, our table-merging technique will be useful for algorithms using tabular hashing.

CMS has also been used as an underlying structure to design more advanced sketches. Recently, Roy et al. developed ASketch which filters high frequent items first and handles the remaining with a sketch such as CMS which they used for implementation [23]. However, their parallelization depends on SPMD which stores multiple filters/sketches. Another advanced sketch employing multiple CMSs for parallelization is [24]. If a single CMS sketch is necessary, e.g., to fit into the cache, our techniques can be employed to develop more advanced parallel sketches using table-based ones such as CMS or Count Sketch.

Overall, unlike most of the studies in the literature, we try to reduce the memory usage while staying scalable as much as possible. Furthermore, to the best of our knowledge, this is the first study focusing on memory-restricted devices such as SBCs for Count-Min Sketch.

8 CONCLUSION

In this work, we investigated parallelization of Count-Min Sketch on three multicore architectures; a high-end server and two single board computers. We proposed three main

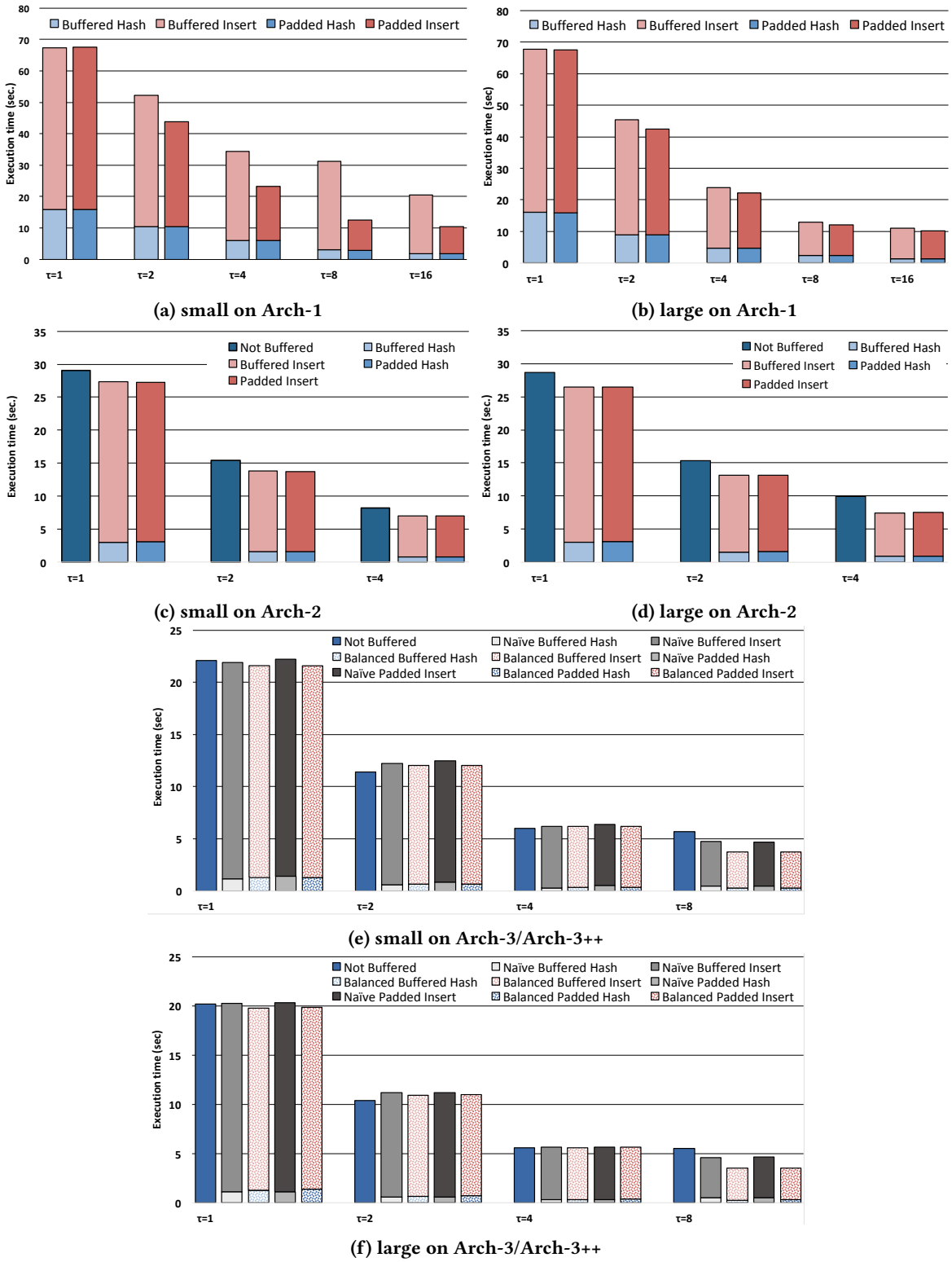


Figure 7: Execution times (in seconds) for hashing and CMS update phase for single sketch experiments. The execution times for the naive parallel approach are omitted for Arch-1 since they are relatively much larger and deteriorate the visibility of the improvements.

			Buffered Parallel + P				Buffered Parallel + PB			
			τ=2	τ=4	τ=8	τ=16	τ=2	τ=4	τ=8	τ=16
Uniform	**Arch-1**	**small**	43.5	22.8	13.0	10.6	43.8	23.5	13.5	11.1
		large	45.0	23.2	13.0	10.6	44.0	23.7	13.6	11.1
	Arch-2	**small**	13.8	6.9			14.3	10.4		
		large	13.3	6.6			13.9	9.6		
	Arch-3++	**small**	12.2	6.2	3.9		12.2	6.2	3.9	
		large	11.1	5.7	3.7		11.1	5.7	3.7	
Zipf 1.1	**Arch-1**	**small**	43.5	22.9	13.0	10.6	45.6	24.6	14.2	11.5
		large	46.4	24.3	13.6	10.8	44.7	24.1	13.8	11.3
	Arch-2	**small**	13.9	7.4			13.8	7.8		
		large	13.3	8.1			13.5	9.3		
	Arch-3++	**small**	12.1	6.2	3.8		12.1	6.2	3.8	
		large	11.1	5.7	3.6		11.2	5.8	3.7	
Zipf 1.5	**Arch-1**	**small**	44.6	23.5	13.1	10.6	44.3	24.2	13.8	11.4
		large	43.7	23.1	13.0	10.5	44.2	23.6	13.6	11.0
	Arch-2	**small**	13.7	7.0			13.7	7.0		
		large	13.2	7.5			13.1	7.5		
	Arch-3++	**small**	12.1	6.2	3.8		12.0	6.1	3.8	
		large	11.0	5.6	3.5		11.1	5.7	3.6	
Zipf 3.0	**Arch-1**	**small**	43.1	22.9	12.8	10.5	45.6	24.3	13.9	11.4
		large	44.4	23.1	13.1	11.2	44.8	23.9	13.7	11.2
	Arch-2	**small**	13.7	7.2			13.7	8.2		
		large	13.1	7.4			13.1	7.8		
	Arch-3++	**small**	11.4	5.8	3.6		11.5	5.8	3.7	
		large	11.1	5.5	3.4		11.1	5.8	3.5	

Table 3: Impact of removing the barrier between the phases of buffered parallel execution on performance. The first set of the columns is the same with the last set of columns in Table 2. The values in the table are in seconds.

techniques. The first one, merged tabulation, is useful when a single is item needs to be hashed multiple times and can be used for different sketches and other applications performing the same task. The second technique buffers the intermediate results to correctly synchronize the computation and regularize the memory accesses. The third one helps to utilize heterogeneous cores which is a recent trend on today's smaller devices. The experiments we performed show that all these techniques work for CMS construction on the architectures used for the experiments.

As a future work, we are planning to analyze the configuration options of the processors on single board computers such as how much data/instruction cache they use and how they handle coherency. Since sketches already incur inaccuracies, a freedom to have a little bit more can yield various optimizations. We also want to extend the architecture spectrum with other architectures such as FPGAs and other SBCs. We believe that similar techniques we develop here can also be used for other sketches. Finally, we are planning to work on different frequency sketches other than CMSs.

REFERENCES

[1] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing (STOC '96)*. ACM, New York, NY, USA, 20–29. https://doi.org/10.1145/237814.237823

[2] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[3] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14 (ESA'06)*. Springer-Verlag, London, UK, UK, 684–695. https://doi.org/10.1007/11841036_61

[4] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. 2002. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*. 636–646.

[5] Andrej Brodnik and J. Ian Munro. 1999. Membership in Constant Time and Almost-Minimum Space. *SIAM J. Comput.* 28, 5 (May 1999), 1627–1640. https://doi.org/10.1137/S0097539795294165

[6] Massimo Cafaro, Marco Pulimeno, and Italo Epicoco. 2018. Parallel mining of time-faded heavy hitters. *Expert Systems with Applications* 96 (2018), 115 – 128. https://doi.org/10.1016/j.eswa.2017.11.021

[7] Massimo Cafaro, Marco Pulimeno, Italo Epicoco, and Giovanni Aloisio. 2016. Mining frequent items in the time fading model. *Information*

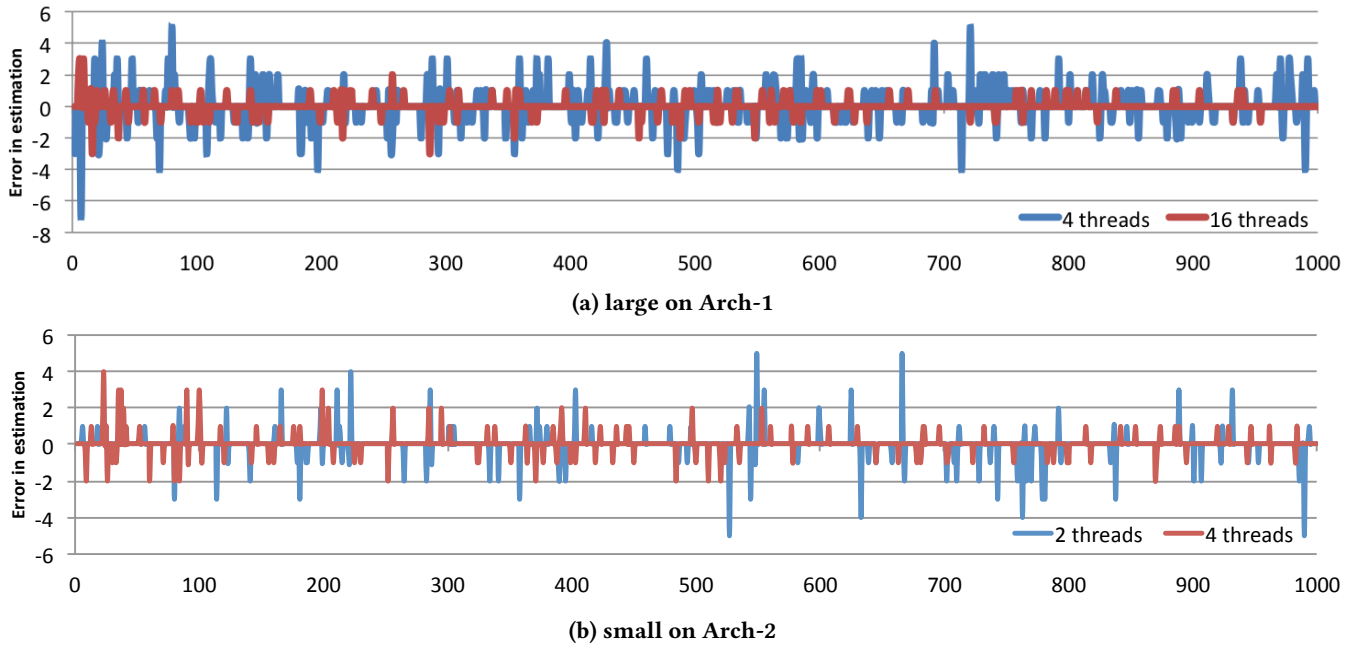


Figure 8: The difference between the frequency estimations of single and multithreaded executions of buffered parallel approach without the barrier. The charts show the differences for the top 1000 items for the large sketch on Arch-1 and small sketch on Arch-2.

Sciences 370-371 (2016), 221 – 238. https://doi.org/10.1016/j.ins.2016.07.077

[8] B. Chandramouli, M. Ali, J. Goldstein, B. Sezgin, and B. S. Raman. 2010. Data Stream Management Systems for Computational Finance. *Computer* 43, 12 (Dec 2010), 45–52. https://doi.org/10.1109/MC.2010.346

[9] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*. Springer-Verlag, Berlin, Heidelberg, 693–703. http://dl.acm.org/citation.cfm?id=646255.684566

[10] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. 2003. Finding Hierarchical Heavy Hitters in Data Streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, 464–475. http://dl.acm.org/citation.cfm?id=1315451.1315492

[11] Graham Cormode and Muthu Muthukrishnan. 2012. Approximating Data with the Count-Min Sketch. *IEEE Softw.* 29, 1 (Jan. 2012), 64–69. https://doi.org/10.1109/MS.2011.127

[12] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58 – 75. https://doi.org/10.1016/j.jalgor.2003.12.001

[13] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Mikkel Thorup. 2017. Practical Hash Functions for Similarity Estimation and Dimensionality Reduction. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 6618–6628.

[14] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. 2009. Thread Cooperation in Multicore Architectures for Frequency Counting over Multiple Data Streams. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 217–228. https://doi.org/10.14778/1687627.1687653

[15] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2002. Processing Complex Aggregate Queries over Data Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/564691.564699

[16] N. G. Duffield and Matthias Grossglauser. 2001. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM Trans. Netw.* 9, 3 (June 2001), 280–292. https://doi.org/10.1109/90.929851

[17] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 1998. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '98)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/285237.285287

[18] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.* 8, 3 (June 2000), 281–293. https://doi.org/10.1109/90.851975

[19] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. 2002. How to Summarize the Universe: Dynamic Maintenance of Quantiles. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. VLDB Endowment, 454–465. http://dl.acm.org/citation.cfm?id=1287369.1287409

[20] Amit Goyal, Hal Daumé, III, and Graham Cormode. 2012. 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 (EMNLP-CoNLL '12)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 1093–1103. http://dl.acm.org/citation.cfm?id=2390948.2391070

[21] S. Muthukrishnan. 2005. Data Streams: Algorithms and Applications. *Found. Trends Theor. Comput. Sci.* 1, 2 (Aug. 2005), 117–236. https://doi.org/10.1561/0400000002

[22] Mihai Pătraşcu and Mikkel Thorup. 2012. The Power of Simple Tabulation Hashing. *J. ACM* 59, 3, Article 14 (June 2012), 50 pages. https://doi.org/10.1145/2220357.2220361

[23] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented Sketch: Faster and More Accurate Stream Processing. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1449–1463. https://doi.org/10.1145/2882903.2882948

[24] Dina Thomas, Rajesh Bordawekar, Charu Aggarwal, and Philip S. Yu. 2007. *A Frequency-aware Parallel Algorithm for Counting Stream Items on Multicore Processors.* Technical Report. IBM.

[25] Mikkel Thorup. 2017. Fast and Powerful Hashing Using Tabulation. *Commun. ACM* 60, 7 (June 2017), 94–101. https://doi.org/10.1145/3068772

[26] George Zipf. 1935. *The Psychobiology of Language: An Introduction to Dynamic Philology.* M.I.T. Press, Cambridge, Mass.

[27] A. L. Zobrist. University of Wisconsin, Madison, Wisconsin, 1970. A new hashing method with application for game playing. *Technical Report 88* (University of Wisconsin, Madison, Wisconsin, 1970).