Approximate TF-IDF based on Topic Extraction from Massive Message Stream using the GPU

Ugo Erra

Dipartimento di Matematica, Informatica ed Economia,
Universitá della Basilicata, Potenza, Italy

Sabrina Senatore*

Dipartimento di Informatica
Universitá degli Studi di Salerno, 84084 Fisciano (SA), Italy

Fernando Minnella

Dipartimento di Matematica, Informatica ed Economia,
Universitá della Basilicata, Potenza, Italy

Giuseppe Caggianese

Scuola di Ingegneria, Universitá della Basilicata, Potenza, Italy

Abstract

The Web is a constantly expanding global information space that includes disparate types of data and resources. Recent trends demonstrate the urgent need to manage the large amounts of data stream, especially in specific domains of application such as critical infrastructure systems, sensor networks, log file analysis, search engines and more recently, social networks. All of these applications involve large-scale data-intensive tasks, often subject to time constraints and space complexity. Algorithms, data management and data retrieval techniques must be able to process data stream, i.e., process data as it becomes available and provide an accurate response, based solely on the data stream that has already been provided. Data retrieval techniques often require traditional data storage and processing approach, i.e., all data must be available in the storage space in order to be processed. For instance, a widely used relevance measure is Term Frequency-Inverse Document Frequency (TF-IDF), which can evaluate how important a word is in a collection of documents and requires to a priori know the whole dataset.

To address this problem, we propose an approximate version of the TF-IDF

*Corresponding author
Email addresses: ugo.erra@unibas.it (Ugo Erra), ssenatore@unisa.it (Sabrina Senatore), fernando.minnella@unibas.it (Fernando Minnella), giuseppe.caggianese@unibas.it (Giuseppe Caggianese)

measure suitable to work on continuous data stream (such as the exchange of messages, tweets, sensor-based log files, etc.). The algorithm for the calculation of this measure makes two assumptions: a fast response is required, and memory is both limited and infinitely smaller than the size of the data stream. In addition, to face the great computational power required to process massive data stream, we present also a parallel implementation of the approximate TF-IDF calculation using Graphical Processing Units (GPUs).

This implementation of the algorithm was tested on generated and real data stream and was able to capture the most frequent terms. Our results demonstrate that the approximate version of the TF-IDF measure performs at a level that is comparable to the solution of the precise TF-IDF measure.

1. Introduction

Over the past twenty years technological advances in communication infrastructure, the miniaturization of computing devices, and Web networking platforms have led to the growth of data at an astonishing rate. Data comes from various sources: Internet-based companies acquire vast amounts of information about their customers and suppliers, network sensors are embedded in physical devices such as smart phones and capture physical or environmental information, and more recently, user-generated content is growing faster than ever. This explosive increase in information is most evident on the Web: every day, millions of new documents, posts, messages, and web pages are added by users to the data space.

The management of large pools of information is becoming crucial for several sectors of the global economy. The challenge is to capture, aggregate, store, and analyze this huge amount of data that exceeds available storage space (volume), comes from different sources (variety) and requires immediate, real-time solutions (velocity). For example, data from sensor networks or monitoring systems must be processed, and the correct action provided, as soon as possible, especially in critical situations. Recent studies predict that in the future, the behavior of individual customers will be tracked by Internet click streams, used to update their preferences, and predict their behavior in real time [21]. The term 'big data' is widely used to refer to the explosion in the amount of digital data that is increasing in terms of volume, variety and velocity worldwide. Big data offers new opportunities that affect how the global economy makes decisions, how it responds to customers' requests, and how it identifies new markets and monitors operations. Hence, big data mining relates to the extraction of knowledge structures represented by continuous streams of information. The real-time consumption of data streams is increasingly important and requires continuously updated data processing; an example is data from the last hour's news feed, which must be made available and ranked at the top of the results that search engines provide to users' queries.

In the data mining domain, a well-known measure that is often used for scoring and ranking the relevance of documents is the term frequency-inverse document frequency, commonly referred to as the TF-IDF measure [30]. It provides a simple model to evaluate the relevance of keywords within a corpus or large collection of documents. Although the TF-IDF measure is a relatively old weighting factor, its simplicity makes it a popular starting point for more sophisticated ranking algorithms. It is widely used in information retrieval and text mining, and its variants are used in many other domains of application where it is important to know the relevance of a term. A recent use of the TF-IDF measure is in business and content marketing for advertising campaigns and return on investment tracking [34]. However, the TF-IDF measure requires a traditional data storage and processing approach, i.e. the whole corpus must be available in the storage space in order to be processed. The problem is that big data often takes the form of massive data streams that require real-time computation, i.e., data can only be examined in a few passes (typically just one) and then it is no longer available.

Stream processing has recently been recognized as an emerging paradigm in data-intensive applications. Data is modeled as a transient stream consisting of relational tuples [3]. This paradigm forces us to rethink traditional algorithms and introduces new challenges. Input data streams arrive at a very high rate; they require intense computation and consume high levels of resources. The rate at which data must be produced means that it must be temporarily captured and consumed at the same rate it arrives, as the cost of archiving it on a long-term basis is usually prohibitive. One way to build workable systems that can meet these challenges is to use parallelism. As computations can be performed independently for each data item in the stream, a high degree of parallelism is possible, i.e., the same function can be applied to all items of an input stream simultaneously. Stream processing has increased in recent years thanks to new programmable processors such as Graphics Processing Units (GPUs) [5] and Field Programmable Gate Arrays (FPGAs) [19], which make it easy to exploit the characteristics of data stream using low-cost parallel architectures.

This paper describes a parallel implementation of an algorithm to process massive data streams using GPUs. The algorithm takes its inspiration from the TF-IDF measure, and provides an approximate ranking of terms from a continuous stream in real-time. In nutshell, our contribution is twofold:

- An approximate TF-IDF measure for streamed data processing.
- A parallel implementation of the calculation of this measure based on programmable Graphics Processing Units.

The proposal has been tested on generated and real datasets. Particularly, the real dataset is composed of a large Twitter collection. The results of the case studies are used to assess the speed up with respect to the sequential implementation. We then compare the results of the approximate TF-IDF measure with its exact counterpart, and demonstrate that our implementation is both efficient and effective.

The remainder of the paper is structured as follows. Section 2 provides an overview of related work; it addresses the role of the TF-IDF measure (and its variants) in information retrieval and data mining. Section 3 presents the theoretical background and the formalization of the problem. Section 4 describes the approximate version of the TF-IDF measure. Implementation and configuration details are given in Section 5. A comparison of the performance of GPUs vs. CPUs is presented in Section 6, which also provides details of the experiments. We end our paper with some final remarks and future directions for our research.

2. Related Work

Automatic topic/term extraction is a crucial activity in many knowledge-based processes such as automatic indexing, knowledge discovery, text mining, and monitoring. Other emerging domains such as biomedicine or news broadcasting also have an interest, as new terms emerge continuously.

Most research on topic extraction takes two distinctive approaches: the first looks at the distributional properties of terms, such as frequency and the TF-IDF measure, and the second aims for a deeper textual analysis based on Natural Language Processing (NLP) principles [22]. Our work reflects the first approach, as it returns the most relevant terms found in the data stream through the modeling of an approximate version of the TF-IDF measure.

The following work has similarities to our algorithm (particularly with respect to the use of the TF-IDF measure). To the best of our knowledge, there has been no other work that aims to compute the TF-IDF measure from a massive data stream.

The approach described in [6] shares certain similarities with our work, although the research goals are different. It provides a weekly summary of the main topics found in archived newswire sources on the Web. The approach can analyze as many channels as there are newswire sources. It uses a modified version of the TF-IDF measure, called the TF-PDF (Term Frequency-Proportional Document Frequency) that gives significant weight to terms related to the hot topics carried by the main channels. The measure reflects the weight of a term taken from a data channel; it is linearly proportional to the frequency of the term in the channel, and exponentially proportional to the ratio of documents containing the term found in the channel. Unlike our approximate TF-IDF measure, the TF-PDF measure depends on the number of channels: the more channels, the more accurately the TF-PDF measure is able to recognize terms that reflect emerging topics. The TF-PDF is a precise measure, computed from the entire data stream composed of all channels. Once the TF-PDF measure has been computed, the approach builds sentence clusters in order to create a summary of the relevant topics. Experiments were carried out on samples of around a thousand news documents, and performed well in terms of recall and precision.

An approach closer to deep textual analysis is described in [36]. Linguistic and temporal features are exploited to extract topic-level conversations in text message streams; the approach uses the cosine measure to calculate the degree of content similarity between messages. It demonstrates that when the TF-IDF measure is used to represent message content, similarities between messages may be lost because of the sparsity of terms. The approach was tested on a sample of around ten thousand instant messages, subject to limited memory and CPU resources. The results were good in terms of the F-measure.

An interesting approach is described in [28], which addresses a real-time unsupervised document clustering problem. The computation of the TF-IDF measure on streamed documents is translated into a new term weighting scheme, called the TF-ICF measure (where C stands for Corpus) that takes advantage of empirical evidence. The authors demonstrate that the document frequency distribution derived from a training dataset can be approximated to that of an unknown data stream; consequently, the IDF (in this case, the ICF) computation can be applied to a carefully chosen static corpus that makes it possible to approximate information about unknown documents. At the same time, benefits are derived in terms of algorithmic and computational complexity. However, the traditional IDF computation needs a priori knowledge of the entire static collection of documents; the authors overcome this problem by defining an ICF measure that is not dependent on any of the global features of the set. We address the same issue in our work, but in our case, we compute an approximate TF-IDF measure from a continuous, dynamic data stream.

There are other notable examples in the literature that exploit the TF-IDF measure (or variants thereof) to achieve different results: in [35] two parallel streaming algorithms are used to classify HTTP requests in order to detect attacks on websites. Their algorithms are based on machine learning techniques, and implement a real-time document similarity classifier that is based on the TF-IDF measure to separate malicious HTTP requests from normal ones. They are shown to be highly accurate and achieve optimal throughput. Other algorithms, implemented on a GPU, exploit the TF-IDF measure to achieve massive document clustering [37], [33] and document searching [10]. For example, the latter implements an algorithm that, along with the TF-IDF measure, exploits Latent Semantic Analysis (LSA), Multi-objective algorithms, Genetic algorithms and Quad Tree Pareto Dominance techniques. Notably, it is able to parallelize mathematical operations typically used in the TF-IDF and LSA techniques, as a result of its CUDA-based implementation.

Although most of these approaches have some similarities with our method, in that they address research issues using the TF-IDF measure (or variants), the evaluation of the measure is accurate and requires a priori knowledge of the data. In the era of big data, handling streaming high-rate data as well as real-time (or at least rapid) interactions with large datasets is an open problem which requires a rethinking of traditional data storage approaches to fit data-intensive applications, which nowadays are increasingly widespread.

3. Background

This section introduces the frequent items problem and briefly presents the algorithm detailed in [11] that, like well-known counter-based algorithms, provides an approximate solution with a preset memory size.

3.1. Frequent Items Problem

The frequent items problem [7] is one of the most studied questions in data stream mining. It is a popular and interesting problem that is simple to explain informally: given a sequence of items, find those items that occur most frequently. It can be more formally expressed, according to [7] as: given a stream S of n items $t_1 \ldots t_n$, the frequency of an item i is $f_i = |\{t_j = i\}|$. The *exact ϕ-frequent* items comprise the set $\{i|f_i > \phi n\}$, where the parameter ϕ is called the *frequency threshold*. For example, given a stream $S = (w, x, w, u, y, w, x, u)$, we have $f_w = 3$, $f_x = 2$, $f_y = 1$, and $f_u = 2$. If we set $\phi = 0.2$, the exact ϕ-frequent items is the set $\{w, x, u\}$. Since the frequent items problem requires a space that is proportional to the length of the stream [7], an approximate version is defined, based on an error tolerance ϵ. The ϵ-approximate problem returns a set F of items so that $\forall i \in F, f_i > (\phi - \epsilon)n$ and there is no $i \notin F$ such that $f_i > \phi n$. As a consequence, there can be false positives but no false negatives.

Most algorithms for frequent items in data stream mining are classified as counter-based algorithms. Frequent and SpaceSaving [7] are two well-known examples. They maintain a small subset of items together with relative counters that store the approximate frequency of these items in the stream. For each new, incoming item the algorithm decides whether to store the item or not, and if so, what counter value to associate with it. Both algorithms maintain a set T entries that represent the most frequent items computed so far. Specifically, given a data stream of n items, the set T stores $k-1 < item, counter >$ pairs in the Frequent algorithm and $k < item, counter >$ pairs in the SpaceSaving algorithm, while processing all incoming items. At runtime, each new item is compared against the stored items in T. If the item exists, the corresponding counter is incremented by 1. Otherwise, the new item is stored and the corresponding counter is set to 1. If the set T is full, the two algorithms take different strategies. The Frequent algorithm decrements all counters by 1, while in the SpaceSaving algorithm, the $< item, counter >$ pair with the smallest count is replaced by the new item, whose counter is incremented by 1.

3.2. Sort-Based Frequent Items

In order to provide a comprehensive view of our approach, first we briefly present the approach outlined in [11] that inspired the solution to the problem described in this paper. The pseudo-code of the *Sort-based Frequent Items* algorithm (Algorithm 1) provides an approximate solution to the discovery of frequent items, when buffer memory is limited. It must be emphasized that simply counting the number of items is not feasible with limited memory; as we stated before, the frequent items problem requires a space proportional to the length of the stream [7].

The algorithm processes the continuous stream of data as blocks of substream S. In order to compute the frequency of each item, a buffer B is created to contain the $< item, counter >$ pairs where, *counter* is the number of times that the *item* has appeared. The buffer B is split into two parts. The first part

Algorithm 1 SORT-BASED FREQUENT ITEMS(k)

1: $B \leftarrow \emptyset$
2: **for all** S **do**
3:     **for** $i \leftarrow 1$ to $|S|$ **do**
4:         $B_{k+i}.item \leftarrow S_i.item$
5:         $B_{k+i}.counter \leftarrow 1$
6:     **end for**
7:     $min \leftarrow B_k.counter$
8:     **for** $i \leftarrow 1$ to k **do**
9:         $B_i.counter \leftarrow B_i.counter - min$
10:     **end for**
11:     `Sort_by_key`($B.item$)
12:     `Reduce_by_key`($B.item$)
13:     `Sort_by_key`($B.counter$)
14:     **for** $i \leftarrow 1$ to k **do**
15:         $B_i.counter \leftarrow B_i.counter + min$
16:     **end for**
17: **end for**

of size k holds the most frequent items found in the items received so far and the number of occurrences. The second part of size $|S|$ contains the incoming sub-stream S to be processed. For each new sub-stream S that is received, its items are copied into the buffer B, starting from position $k+1$ (line 4). As each new incoming item occurs exactly once, each relative *counter* value is initialized to 1 (line 5).

Next, the item with minimum counter value (that is in the k-th position of buffer B) is selected (line 7). This value is subtracted from the first k items (lines 8-10). Sorting and reduction by key operations, called respectively `Sort_by_key` and `Reduce_by_key` are used to update the current k most-frequent items (lines 11-13). The reduction operation brings together identical elements and sums their counter values. Consequently, buffer B is ordered in descending order and reduced with respect to the item (lines 11-12). Then, in order to calculate the k most-frequent items, a further descending sort operation is applied on buffer B, with respect to the counter (line 13). Finally, the previously subtracted minimum is added to all items in the first k positions (lines 14-16).

Therefore, while previously the minimum was subtracted from the first k-items, by allowing new items to be candidates for the first k most-frequent items, the last for loop (lines 14-16) restores the correct values of the *counter* once new items in the incoming sub-stream S have been evaluated. Specifically, the basic idea behind the algorithm is that the frequency of new items is overestimated by the minimum value min of all counters of items in the first k positions. Each new item could have already occurred anywhere between 0 and min times. This is true because if it has occurred more than min times, the second `Sort_by_key` operation places it in a position such that is not substituted by new items, and thus cannot be a new item. As we do not know the exact number of occurrences in the range $[0, min]$, we overestimate the frequency by choosing the value min. By overestimating frequencies, the genuinely frequent items satisfy the condition $counter > \epsilon n$. As demonstrated by [11], the frequency estimation error is negligible.

3.3. Term Frequency - Inverse Document Frequency

TF-IDF (Term Frequency - Inverse Document Frequency) [30] is a well-known measure that is often used to construct a vector space model in information retrieval. It evaluates the importance of a word in a document. The importance increases proportionally with the number of times that a word appears in a document, compared to the inverse proportion of the same word in the whole collection of documents. Roughly speaking, the TF-IDF measure associated with a term t takes:

- a higher value when the term t appears several times within a small number of documents;
- a lower value when the term t occurs fewer times in a document, or occurs in many documents;
- a lower value if the term t appears in almost all documents.

More formally, let $D = \{d_1, d_2, \ldots, d_n\}$ be a comprehensive collection of documents and t a term in the collection. The term frequency-inverse document measure is computed as follows:

$$tf - idf(t, d, D) \;=\; tf(t, d) * idf(t, D) \qquad (1)$$

Specifically, $tf(t, d)$ represents the frequency of term t in a document d (i.e., the number of times a term occurs in a document), expressed by:

$$tf(t, d) \;=\; \frac{f(t, d)}{|d|} \qquad (2)$$

where $f(t, d)$ is the number of times the term t appears in the document d and the denominator is the dimension of d, expressed as the cardinality of its own terms. The inverse document frequency $idf(t, D)$ is described as follows:

$$idf(t, D) \;=\; log\frac{|D|}{|\{d|t \in d\}|} \qquad (3)$$

where the denominator represents the number of documents in which the term t appears.

4. The Approximate TF-IDF Measure

The definition of the TF-IDF measure described in the previous section requires a complete knowledge of the number of documents to process and consequently all the terms appearing in those documents. In real data stream problems, these conditions are not always practicable due to environmental constraints. Limitations on the size of memory available to store the stream of terms prevent the computation of the exact TF-IDF measure, and requires an approximate solution. This section introduces the formal implementation of the calculation of the approximate TF-IDF measure on a continuous data stream.

4.1. A High Level Abstraction Description

The Sort-Based Frequent Items algorithm (Algorithm 1) is the first step of our implementation. It is described in [11], where it is used in the frequent items computation. However, different constrains led to a different design. The main difference is in how the item is represented. The calculation of the TF-IDF measure requires each item in a data stream to be composed of two elements: the term and the document in which it appears (rather than a simple numeric stream). Consequently, Algorithm 1 was revised to work with buffer B that contains pairs of the form $< (t, id), counter >$, where (t, id) represents the "atomic" item, composed of the document identifier *id* and the term t that appears in the document *id*; the *counter* represents the number of times that the item has appeared so far. In practice, B contains triples. Thus, the first step of the TF-IDF implementation is to compute the most frequent k items, as shown in Figure 1, assuming that B already contains ordered pairs.

Let us notice that the pseudocode presented in Algorithm 1 has been thought to be easily translated in a parallel implementation code. Moreover, there are many parallel implementations of the two algorithms that we named `Sort_by_key` and `Reduce_by_key`, such as, for example, the parallel implementation available in the Thrust library (that will be described in Section 5) .

We assume that at time τ, we have the most frequent pairs (t_i, d_j) with $i = 1, \ldots, n$ and $j = 1, \ldots, m$, computed by Algorithm 1 . Let $D' = \{d_1, d_2, \ldots, d_n\}$ and $T' = \{t_1, t_2, \ldots, t_m\}$ be the set of documents and terms from these pairs, with $D' \subseteq D$ and $T' \subseteq T$, where D and T are respectively the comprehensive collection of documents and the set of all the terms that are in D. We note that memory size constraints mean that we can only maintain a part of the data flow.

Let *count* be a function that associates a numeric value with each (term, document) pair. Expressed more formally: $count : T' \times D' \to \mathbb{N}$, such that $\forall (t, d)$ with $t \in T'$ and $d \in D'$, $count(t, d) \geq 0$. The value $count(t, d)$ represents the number of occurrences of term t in a document d: $count(t, d)$ is equal to 0 when the term t does not appear in the document d. We note that $count(t, d)$ corresponds to the numerator of the ratio in Equation 2. Therefore, taking the approximate version of the TF-IDF measure, Equations 2 and 3 can be rewritten respectively in the following form:

$$A - tf(t,d) \; = \; \frac{count(t,d)}{\sum_{i=1}^{m} count(t_i,d)} \tag{4}$$

$$A - idf(t,D') \; = \; log \frac{|D'|}{\sum_{j=1}^{n} \delta_j} \tag{5}$$

where

$$\delta_j = \begin{cases} 1 \quad if & count(t,d_j) > 0 \\ 0 & otherwise \end{cases} \tag{6}$$

Equation 4 is the ratio between the occurrence of term t in document d and the total number of terms that appear in the document (in other words, its cardinality). Equation 5 differs from Equation 3 in the denominator: the set of all the documents containing the term t in Equation 3 is computed by summing the value (0 or 1) returned by the binary function δ that determines whether a term t appears (or not) in a certain document.

Algorithm 2 APPROXIMATE TF-IDF(B, k)

```
V ← Sort_by_key(B.id)
C ← Reduce_by_key(V.id)
for i ← 1 to k do
    id = B_i.id
    TF_i = B_i.counter / Get_total(C, id)
end for

for i ← 1 to k do
    C_i.total = 1
end for
D ← Count(C.id)

V ← Sort_by_key(B.t)
for i ← 1 to k do
    C_i.t = V_i.t
    C_i.total = 1
end for
C ← Reduce_by_key(C.t)

for i ← 1 to k do
    t = B_i.t
    IDF_i = log(D / Get_total(C, t))
end for

for i ← 1 to k do
    TFIDF_i ← TF_i * IDF_i
end for
```

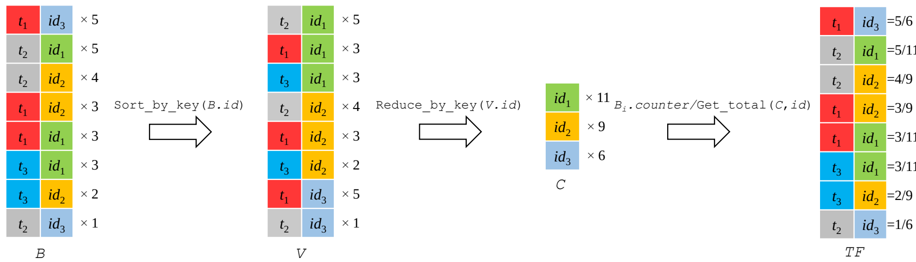

Figure 1: Example computation of approximate term frequency values $A - tf(t, d)$.

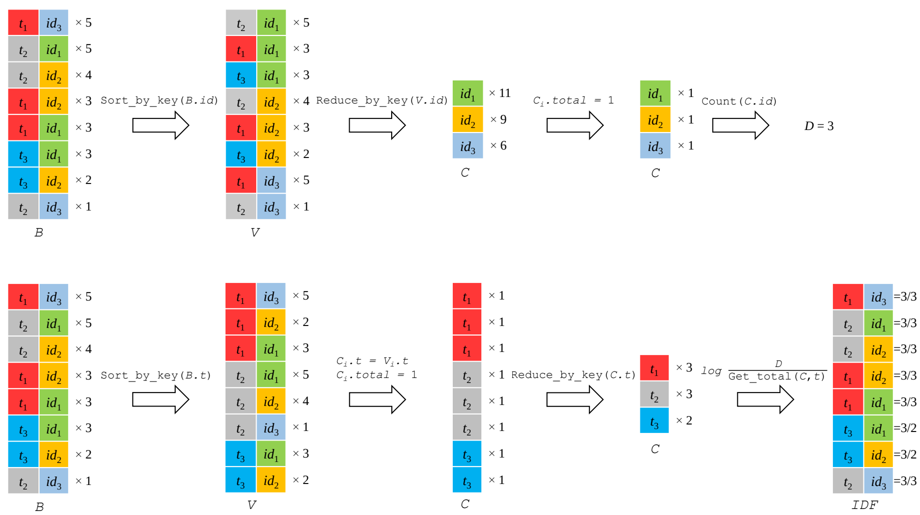

Figure 2: Example computation of inverse document frequency values. The top part of the figure shows the approach used to count the total number of documents contained in buffer B. This value is used in the bottom part of the figure to compute the inverse document frequency value $A - idf(t, D)$.

4.2. Computing the Approximate TF-IDF Measure

Algorithm 2 shows the pseudo-code for computing the approximate TF-IDF measure. Like Algorithm 1, the approximate TF-IDF measure is created on-demand in buffer B and returns the k items (t, id) with the highest TF-IDF values in the given data stream. The algorithm has two phases. In the first phase, it computes $A - tf$ and in the second phase $A - idf$. These values are used to compute $A - tf - idf$. Note that the pseudo-code assumes a parallel implementation. We discuss this further in the implementation section, where we use parallel foreach loops and parallel sorting and reduction.

Figure 1 shows the computation of $A-tf$. Initially, B is ordered with respect to the document identifier (line 1) and saved in buffer V; next, buffer C is created to reduce V (line 2). Buffer C contains $< id, total >$ pairs, where $total$ represents the number of terms found in the document identified by id. According to the definition given in Equation 4, lines 3-6 compute the term frequency for the first k items of buffer B. We note in particular that `Get_total`(C, id) returns the *total* number of terms associated with the document *id*. The frequency of the term t_i is given by $B_i.counter$ that provides the number of occurrences of the item (t_i, id).

At this point, according to Equation 5, $A - idf$ must be computed. Figure 2 shows the computation of $A - idf$: the top part of the figure shows an example computation of the size of $|D|$, while the bottom half shows the computation of $A - idf$. Lines 7-10 of Algorithm 2 calculate the whole number of documents $|D|$. Lines 11-16 compute the denominator of the ratio in Equation 5. Buffer B is ordered with respect to terms (line 11); then the loop (lines 12-15) generates a buffer C composed of pairs $< t, 1 >$. At this point, the number of terms in C is reduced, leaving only pairs $< t, total >$ where, for each term t, *total* is the number of documents in which t appears (see Figure 2). Like the term frequency computation, lines 17-20 calculate the inverse document frequency. Finally, the approximate TF-IDF measure for the first k items of buffer B is returned (lines 21-23). Like Algorithm 1, this algorithm has been designed to allow easily the parallelization of some parts of it (i.e., most of the `for`cycles can be processed in parallel).

5. Implementation

The implementation relies upon the computational capability of a modern Graphics Processing Unit (GPU). Here we adopt NVIDIA's GPU architecture with its CUDA parallel programming model. This section starts by describing the main features of NVIDIA's GPU, in order to highlight the role of parallel programming in our approach. Then, we introduce the parallel template library, *Thrust*, and focus on the parallel and high performance operations used in our implementation.

5.1. GPU Programming Model

Modern NVIDIA GPUs are fully programmable multi-core chips known as CUDA processors [24]. In a GPU, a streaming multiprocessor (SMX) is a cluster of streaming core processors. Each core can only execute one thread at a time, but when a thread performs an operation with high latency it is put into a waiting state and another thread executes. This feature, and the application of a scheduling policy means that the core can run many threads at the same time. In the current generation of GPUs (codenamed Kepler), the number of SMXs ranges from 1 to 14. The GPU used in our experiments (GK110) consisted of 14 SMXs, each containing 192 single-precision CUDA cores. Because each SMX is capable of supporting up to 2048 threads, this GPU manages up to 28672 resident threads. All thread management, including creation, scheduling and barrier synchronization is performed entirely in hardware by the SMX with virtually zero overheads.

In terms of the software model, CUDA [25] provides software developers with facilities to execute parallel programs on the GPU. To use CUDA, a programmer needs to write their code as special functions called kernels, which are executed across a set of parallel threads. The programmer organizes these threads into a hierarchy of blocks and grids. A thread block is a set of concurrent threads that can cooperate via shared memory (which has a latency similar to that of registers) and can be coordinated using synchronization mechanisms. A thread grid is a set of thread blocks executed independently. All threads have access to the same global memory space. Each thread block is mapped to an SMX and executes concurrently. SMX resources (registers and shared memory) are split across the mapped thread block. As a consequence, this limits the number of thread blocks that can be mapped onto one SMX.

5.2. Sorting on the GPU

We use a GPU in the implementation of our approach because of its excellent performance in parallel sorting and reduction. Following the introduction of programmable GPUs and NVIDIA's CUDA framework, many sorting algorithms have been successfully implemented on the GPU in order to exploit its computational power. The first programmable GPUs were suited to implement sorting algorithms [12], [27] although they were far from optimal in terms of execution time $O(n \log^2 n)$. Successive improvements in GPU technology have enabled the implementation of other comparison sorts with algorithmic complexity below $O(n \log n)$, such as the merge sort and the radix sort [31].

The radix sort is currently the fastest approach to sorting 32- and 64-bit keys on both CPU and GPU processors [32]. It is based on a positional representation of keys, where each key is an ordered sequence of digits. For a given input sequence, this method produces a lexicographic ordering iteration over the digit-places from the least significant to the most significant. Given an n-element sequence, the algorithmic complexity of radix sorting is $O(n)$. In [23], the authors demonstrate a radix sorting approach that is able to exceed 1 billion 32-bit keys/sec on a single GPU microprocessor.

Reduction has also been successfully implemented on the GPU. In general, reduction is any operation that computes a single result from a set of data. Examples include min/max, average, sum, product, or any other user-defined function. Several implementations has been developed in past years, such as [14], and [29].

5.3. Detailed Implementation: the Thrust Library

Thrust [4] is a parallel template library that implements high performance applications with minimal programming effort. It is based on CUDA and the Standard Template Library (STL).Thanks to Thrust, developers can take advantage of a collection of fundamental high-level parallel algorithms such as scan, sort, and reduction. For instance, the radix sort discussed above has been implemented. One of the greatest benefits of Thrust is that it exploits the computational capability of the GPU without making fine-grained decisions about how computations are decomposed into parallel threads, and how they are executed on the target architecture. Moreover, its parallel algorithms are generic and can be used with arbitrary user-defined types and operators. There are many features in the Thrust Library; here we only explain those functions that are directly relevant to our implementation.

According to the description of the algorithm for the approximate TF-IDF measure, the main structures to maintain the data stream are vector containers. Thrust provides two types of vector container: `host_vector` and `device_vector`. Declaring a container as a `host_vector` means that it resides on the CPU host memory; otherwise, the `device_vector` container is resident in the GPU device memory. These vectors are generic containers that are able to store any data type and can be resized dynamically, simplifying the data exchange process between the CPU and the GPU. Depending on which vector containers the algorithms use, Thrust can automatically map computational tasks onto the CPU or the GPU. In this way, changing the type of the container enables developers to write executable code for either the GPU or the CPU with minimal effort.

In our implementation, each item in the data stream corresponds to a pair $< (id, t), count >$; we store these values in three separate containers using a Structure of Arrays (SoA) layout. The SoA is a parallel design pattern that promotes the optimization of parallel data access inside the arrays, by using separate arrays for each element of a data structure. The advantage of the SoA technique is that it gives all threads access to contiguous global memory addresses, in order to maximize memory bandwidth [24]. Thrust provides a design pattern that can traverse one or more containers that exploit the SoA layout. In our approach, the aforementioned buffer B (see Algorithm 2) is allocated on the GPU device memory as an SoA layout, and each incoming sub-stream is copied to GPU memory. Therefore, the CPU manages each sub-stream as three separate arrays before the host-to-device memory transfer.

A Thrust program acts on the containers by adopting the STL convention of describing a vector position through an iterator. The iterator works like a pointer that can point to any element in the array. Each algorithm typically takes a pair of iterators as arguments that define the range on which the algorithm must act. Thrust provides some standard iterators, that are defined as primitive types (e.g., `char`, `int`, `float`, and `double`).

As we have three separate `ints` containers, we need three iterators. Two of these iterators must have simultaneous access to the key (t, id) (through the SoA layout); therefore, we need a way to parallel-iterate over the two containers. To achieve this, Thrust provides a `zip_iterator`, which is composed of a tuple of iterators and enables it to parallel-iterate over two or more containers simultaneously. Thus, by increasing the `zip_iterator`, all the iterators in the tuple are increased in parallel. We note that through using the `zip_iterator` design pattern, we logically encapsulate the elements of each container into a single entity. It is clear that our key (id, t) is implemented by the zip-iterator.

Thrust's functionality is derived from four fundamental parallel algorithms: `for_each`, `reduce`, `scan` and `sort.` In our implementation we use the `sort_by_key` algorithm to perform a key-value sort and a `reduce_by_key` algorithm to obtain, for each group of consecutive keys, a single value reduction by using a sum operator over the corresponding values. As for sorting, Thrust statically selects a highly optimized *radix sort* algorithm for sorting primitive types, while it uses a general *merge sort* algorithm for user-defined data types. We also use a `Count` algorithm, which is an additional Thrust function based on the `reduce` algorithm: it returns the number of occurrences of a specific value in a given sequence (e.g., we used `Count` to obtain the total number of documents in buffer B, see Algorithm 2).

Moreover, Thrust provides several algorithms that can perform parallel operations over each element of one or more input containers. For instance, the `sequence` algorithm is designed to fill a container with a sequence of numbers, while a more general `transform` operation applies a binary function to each pair of elements from two input containers. These algorithms are generic in both the type of data to be processed, and the operations to be applied to the input containers. They can easily handle all the loop iterators presented in the pseudo-code of Algorithm 2; moreover, they can be efficiently executed in parallel.

6. Evaluation

In order to evaluate the performance of our algorithm, we have performed the experimentation on a generated and a real dataset. Precisely, we assess the quality of the approximate TF-IDF measure for both the datasets. In particular, for the real dataset, we consider the performance when applied to topic extraction from a Twitter data stream. Moreover, on the real dataset we also evaluate the CPU vs. GPU performance. To the best of our knowledge, there is no standardized benchmark.

The quality of our approximate TF-IDF measure has been estimated by comparing the results of the exact TF-IDF and our approximate version. Precisely, we compared the most frequent term-document pairs $< t, id >$ ranked by our approach with those calculated by the brute force approach.

For this purpose, we consider two measures: the Kendall rank correlation coefficient [15] and the recall measure.

The Kendall coefficient, also referred to as Kendall's tau (τ) coefficient is a statistical measure that evaluates the degree of similarity between two sets of ranked data. Specifically, it considers all possible pairwise combinations of a first set of values, and compares them with a second set of values. Given two sets of size n, the Kendall coefficient measures the difference between the number n_c of concordant pairs and the number n_d of discordant pairs, as a ratio of the total number of possible pairings, based on the following formula:

$$\tau = \frac{n_c - n_d}{n(n-1)/2}$$

Kendall's coefficient τ ranges from $[-1, 1]$. The value is 1 if the agreement between the two rankings is perfect (perfect positive correlation). On the other hand, a value of -1 means there is total disagreement (perfect negative correlation). This coefficient allows us to assess the correlation between the most frequent pairs $< t, id >$, resulting from our approximate solution and those from the brute force approach.

A typical measure of Information Retrieval is instead, the recall: it returns the retrieved pairs that are relevant, from the returned pairs. In our context it measures the number of coincidences of the top-k terms.

More formally, we define $recall(k)$ as the intersection between the top-k terms, defined as follows:

$$recall(k) = top(k)_{tf-idf} \cap top(k)_{A-tf-idf}$$

where $top(k)_{tf-idf}$ and $top(k)_{A-tf-idf}$ are the top-k terms computed by the exact TF-IDF algorithm and the approximate TF-IDF algorithm, respectively.

These two measures supply two different criteria to estimate how much the results provided by our approximate TF-IDF version are comparable to that exact.

The hardware configuration was based on a CPU Intel Core i7-3820@3.6Ghz (quad-core HT) with 16GB of RAM and a GPU NVIDIA GeForce GTX TITAN (2688 CUDA cores) with 6GB of RAM running Microsoft Windows 8.1. The code was compiled using Microsoft's Visual Studio 2012 and the NVIDIA CUDA Toolkit 5.5. Thrust version 1.7.0 was used.

6.1. Generated Datasets

The generated data stream has been designed as a sequence of term-document pairs $< t, id >$. Specifically, we have considered a maximum number of unique documents and unique terms: 100 documents and 100 terms. This way, The maximum number of possible distinct pairs $< t, id >$ is 10,000. Without loss of generality, we have assumed a uniform distribution for the documents. For the terms generation instead , a Zipf distribution has been used. It is shown that the distribution of word frequencies for randomly generated texts is very similar to Zipf's law observed in natural languages such as English [18]. Moreover Zipf's law governs many others domains: email archives, newsgroups [16], caching [13], hypermedia-based communication [17], web accesses [9] and many other Internet features [2]. In addition, massive data streams are rarely uniform, and real data sources typically exhibit significant skewness. They lend themselves well to be modeled by Zipf distributions, which are characterized by a parameter that captures the amount of skew [8].

6.1.1. Methodology and Evaluation

We have generated several datasets, by considering three different sizes: 10K, 100K, and 1,000K; then, the terms have been generated from a skewed distribution, varying the skew from 0.8 to 2 (in order to obtain meaningful distributions that produce at least one heavy hitter per run), with a step equal to 0.2. In total, we had 21 datasets (7 for each dataset size).

The dataset size affects the input stream size: with the dataset of size 10K, we consider a stream size s = 1K; with the dataset of size 100K, we consider two streams with size s = 1K and 10K. Finally, with a 1,000K dataset, the

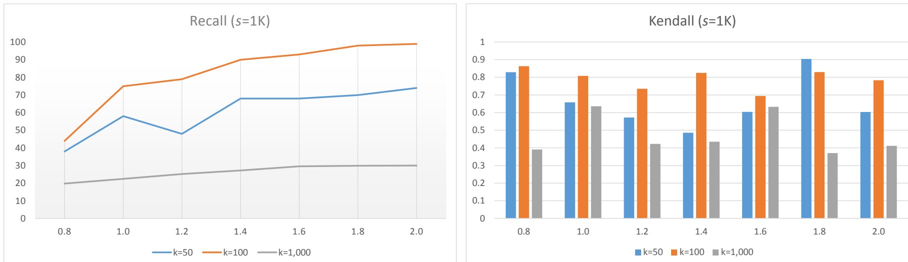

Figure 3: Recall and Kendall for a generated dataset of size 10K. The input streams is of size 1K and is tested on the 50, 100, and 1,000 most frequent items.

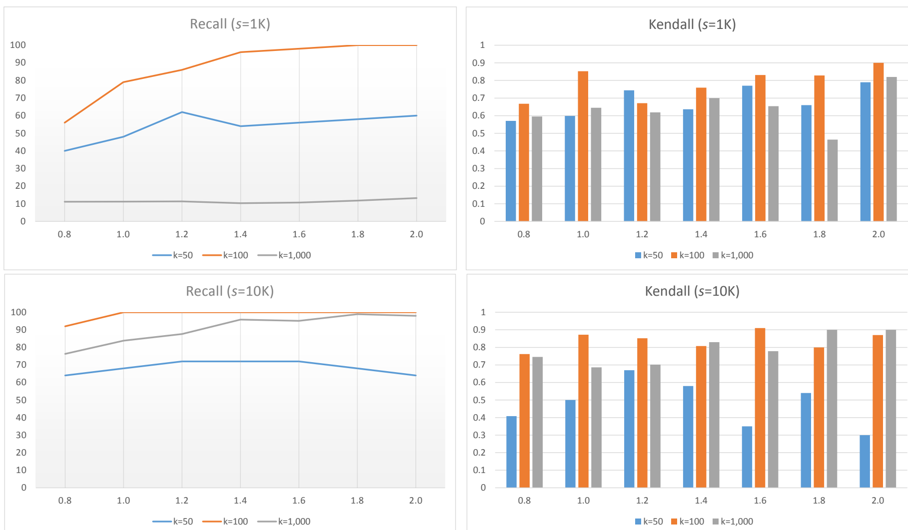

Figure 4: Recall and Kendall for a generated dataset of size 100K. The input streams are of size 1K and 10K. Each input stream is tested on the 50, 100, and 1,000 most frequent items.

conceivable stream sizes are s = 1K, 10K, and 100K. In total, there are six different (dataset, stream) size-configurations.

For each (dataset, stream) size-configurations, there are 7 datasets (by varying the skew value); consequently, there are 42 (7 datasets $\times$ 6 (dataset, stream) size-configurations) experiments. Finally, since for each experiment, we use three different sizes for the most k frequent items: 50, 100, and $1,000$, we have in total as many as 126 (42 experiments $\times$ 3 k-size) experiments.

Figures 3-5 show the performance of our algorithm in terms of recall measure (on the left) and Kendall coefficient (on the right). Fixed the size of the dataset and the input stream, each figure describes 21 experiments, by varying the skew from 0.8 to 2 and the size of k among 50, 100 and $1,000$. By analyzing the recall, when k is lower than s, the recall is generally good. Our algorithm indeed, computes the k most frequent items, by processing a stream of size s.

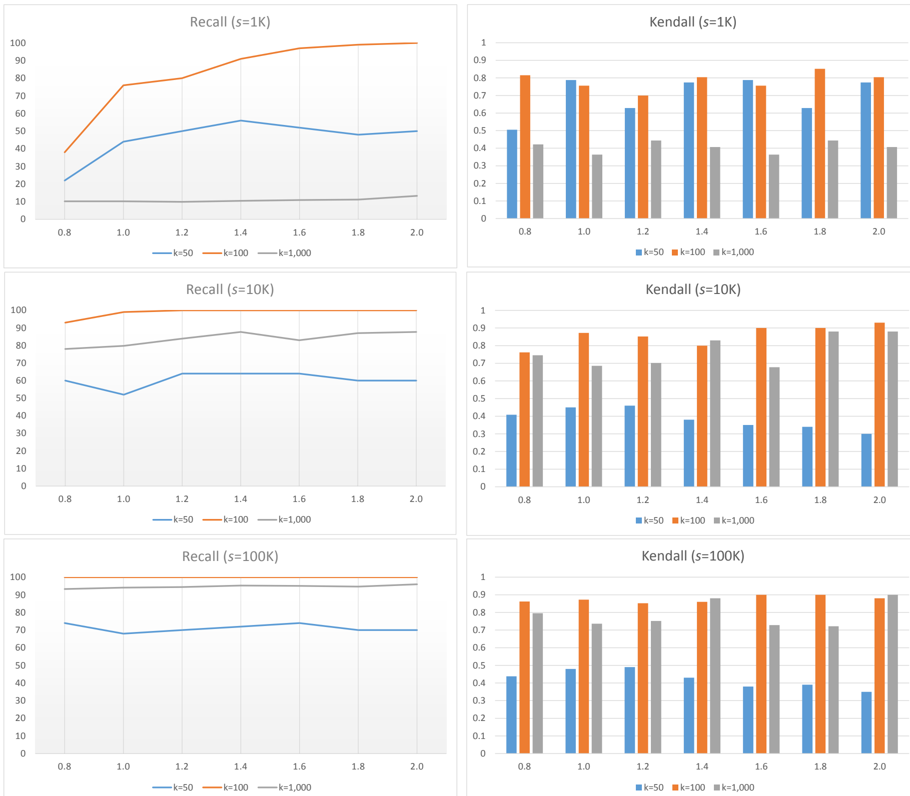

Figure 5: Recall and Kendall for a generated dataset of size 1,000K. The input streams are of size 1K, 10K, and 100K. Each input stream is tested on the 50, 100, and 1,000 most frequent items.

When a new stream arrives, it preserves the most k frequent items computed so far and discards the remaining items in the current stream in order to process the incoming stream. When the input stream s is greater than the value of k, the algorithm has more chances to accurately compute the most frequent items, since the size of s provides a wider "window" to seek candidate frequent items. Just to give an example, when s = 1K and the dataset size is 10K (Figure 3), the best recall values are obtained for $k = 100$. In any case, the recall for $k = 50$ is better than for $k = 1,000$ (i.e., the same order of magnitude of the input stream), demonstrating that $k = 50$ is more selective in identifying the most frequent elements. Similar observations are also for Figures 4 and 5, when s = 1K. When the input size is greater, i.e., s = 10K, (see Figures 4 and 5), the best recall is still for $k = 100$, even also $k = 1,000$ provides good recall (quite close to $k = 100$ for some skew value). By increasing the input stream size indeed, the algorithm handles a window with more items, for each incoming stream; then it can select more frequent items. In these cases, $k = 50$ becomes too small to guarantee a good recall. Again, with s = 100K (Figure 5), the best recall value is for $k = 100$ and $k = 1,000$. In general, $k = 100$ is the best compromise among all the data streams in all the datasets generated, since it provides the better results in the experimentation. Moreover, from the analysis of the results, it is evident that the best recall values can be obtained with the increase of the skew value. (i.e., when only the frequency of a small number of pairs with respect to the dataset size, tends to increase).

The Kendall coefficient τ values show an equally good rank correlation between the most k frequent items from the exact and the approximate TF-IDF measure. From the analysis of the results, τ seems to be affected by the input stream size. When the input stream is small (i.e., $s = 1\text{K}$), the returned τ values are slightly lower: it is due to the fact that probably, after some processed streams, an item that was a candidate frequent item, because it appeared many times, does not still appear in the next incoming streams and thus, at one point, it is eliminated from the candidate frequent items. But, in the successive incoming streams, it reappear and goes into the most frequent item. This way, the frequency of that item will be lower that the actual value, and then it will get lower rank position.

In general, with the higher input stream size, τ reveals a good level of rank correlation, especially when the skew value is greater than 1.4. Like in the recall measurement, the results confirm that τ assumes higher value when $k = 100$ and $k = 1,000$ provided that $k < s$.

6.2. Real Dataset

Other experiments were carried out on a Twitter dataset[1]. Twitter is a social network that connects people through the exchange of *tweets*, which are small messages of up to 140 characters. Its popularity has increased dramatically in recent years, and it has become one of most well-known ways to chat, microblog and discuss current topics. Each tweet can contain one or more hashtags i.e., the # symbol used to mark keywords or topics. Through this mechanism, a conversation about a specific topic can be identified by one or more hashtags associated with a message.

We extracted a dataset composed of 4 million public tweets posted by users from August 2012 to September 2012 in the cities of Washington, New York, and San Francisco. The dataset was transferred to a local machine using the command line tool cURL [1].

All tweets were in English and could have one or more hashtags. Tweets with the same hashtag were grouped to represent a document about a specific topic, identified by the hashtag. Consequently, tweets without hashtags were discarded because they could not be associated with a document. The dataset was preprocessed to discard "noise" using simple NLP principles [20]. We used the Porter stemming algorithm [26] to obtain the *stem* or base forms of each term, then stop-words were removed. Irregular syntactic slang and non-standard

[1] Source and data are available at http://graphics.unibas.it/atfidf

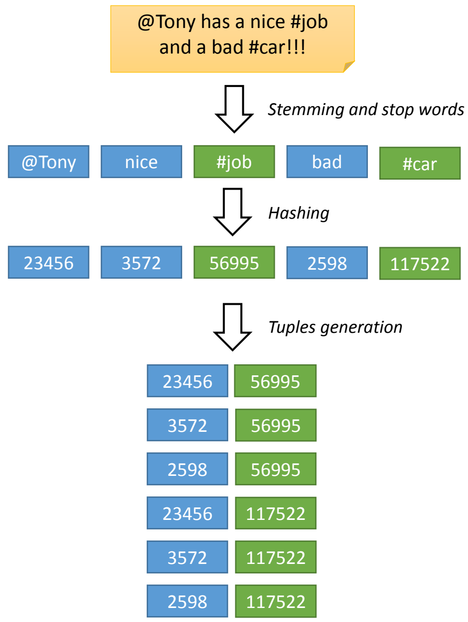

Figure 6: Tuple generation from a tweet. The green label represents the document and the blue label the word.

vocabulary (widespread in phone text messaging due to the necessary brevity) was not discarded and was included in the data to be processed.

Finally, to facilitate data processing, a hash function was defined in order to map each hashtag to a numeric identifier (as shown in Figure 6). Likewise, all the terms that remained following text processing were mapped into a numeric identifier. The pair, composed of the document identifier and the term identifier (Figure 6) represents the minimal unit to process. A total of 40 million $< term, document >$ pairs were generated.

6.2.1. Methodology and Evaluation

The analysis of the dataset led to the creation of 40 million pairs. These pairs were split into four batches of input stream of size s = 1M, 5M, 10M, and 20M. For each of these batches, k (the number of most frequent items) was varied with k = 1K, 10K, 100K, and 1,000K. Once we had computed the most frequent items for all s and all k, we applied the approximate TF-IDF algorithm. In total, we performed 16 experiments.

We assessed our approach on three criteria:

- Speed up and memory footprint: the CPU implementation was compared with the GPU implementation. We exploited the Thrust portability between GPUs and multicore CPUs to obtain a running version of the approximate TF-IDF measure for both architectures.

s	k	Times (sec.)	Memory (MB)
1M	1,000	1.062	30.54
	10,000	0.982	30.82
	100,000	1.070	33.56
	1,000,000	1.342	61.03
5M	1,000	0.773	152.61
	10,000	0.814	152.89
	100,000	0.795	155.64
	1,000,000	0.886	183.10
10M	1,000	0.802	305.20
	10,000	0.768	305.48
	100,000	0.801	308.22
	1,000,000	0.857	335.69
20M	1,000	0.788	610.38
	10,000	0.774	610.65
	100,000	0.827	613.40
	1,000,000	0.791	640.86

Table 1: GPU performance for input streams of size s, and k most frequent items. In all experiments, we measured the total time to process all 40M document-term pairs. Times include CPU to GPU data transfers (and vice-versa).

- Frequent items similarity: the output of the CPU brute force approach was compared with the output of the approximate TF-IDF measure. In particular, we compared the most highly ranked $term, document$ pairs resulting from our approximate solution to those of the brute force approach.

- TF-IDF quality: the TF-IDF measure calculated using the approximate solution was compared with values calculated using the brute force approach, which provides the exact frequency of each pair. For all k, we compared the output of a percentage of the overall corpus of tweets.

GPU vs CPU performance. We analyzed performance in two steps. In the first step, we measured the total time and the amount of memory required to process all of the 40M document-term pairs, based on input streams of size s and a calculation of the k most frequent items, as illustrated in Table 1. These times include both memory transfer time (CPU to GPU) and Thrust execution time. The results show that execution times were not affected by the size of the input stream. In particular, execution time remained stable when the input stream was equal to or greater than 5M, and was independent on its size. In the second step, we measured the time required to compute the approximate TF-IDF measure with different values of k. We evaluated the performance of GPU and CPU implementations by comparing how long each call to the approximate TF-IDF algorithm took on buffer B returned by the Sort-Based Frequent Item of the previous step. Figure 7 compares performance with different sizes of k. With k greater than $10,000$ the GPU implementation is more efficient than the

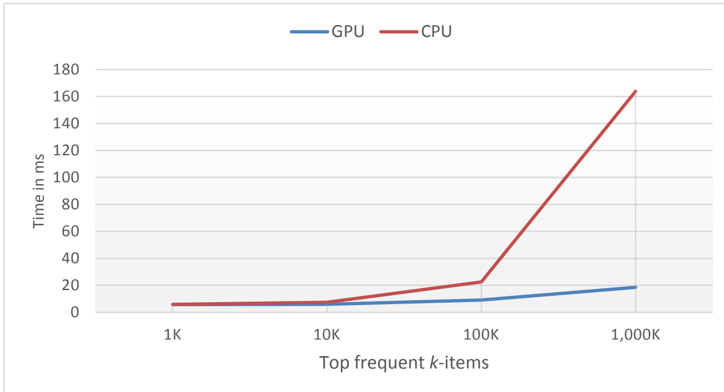

Figure 7: GPU vs CPU scalability with varying sizes of most frequent k items.

CPU implementation. With lower values of k the CPU implementation is more efficient due to overheads that dominate overall performance.

Finally, in Figure 8 we report the total GPU speed up compared to the CPU implementation, by summing the times obtained in the two steps. For a 1M input stream, the speed up ranges from 5× to 8×, and when it is equal to or greater than 5M it is around 10×.

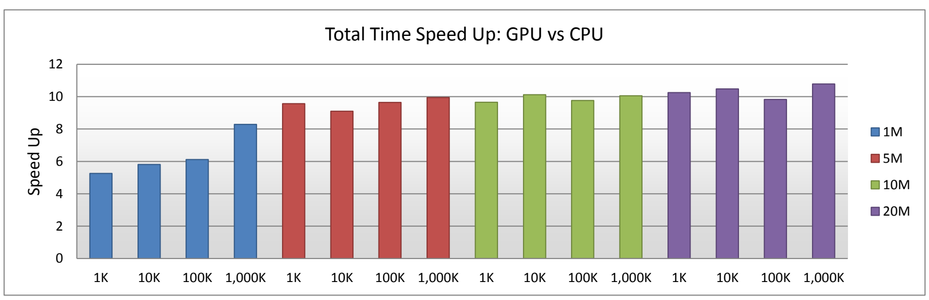

Figure 8: GPU speed up compared to the CPU implementation with input streams of 1M, 5M, 10M, and 20M. Each input stream is tested on the 1K, 10K, 100K, and 1,000K most frequent items.

Let us observe that the size of input stream is very important for guaranteeing a high speed up: in our experiments, when the input stream size is 1M, there are many "CPU to GPU" data transfers; precisely, on our dataset composed of 40M document-term pairs, there are as many as 40 data transfers. As stated above, the high number of transfers produces high overhead, penalizing the GPU performance. The number of data transfers required for the input streams of 5M, 10M and 20M, instead considerably drops to 8, 4, and 2 respectively, guaranteeing an high speed up (around 10×). Therefore, when input streams are large, the GPU offers stable and better performance than the CPU.

s	k	Kendall coefficient
1M	1,000	0.9582
	10,000	0.8986
	100,000	0.8392
	1,000,000	0.7796
5M	1,000	0.9446
	10,000	0.8853
	100,000	0.8257
	1,000,000	0.7659
10M	1,000	0.9340
	10,000	0.8749
	100,000	0.8144
	1,000,000	0.7557
20M	1,000	0.9230
	10,000	0.8634
	100,000	0.8022
	1,000,000	0.7443

Table 2: Change in Kendall coefficient as the input stream size s and the k most frequent items increase.

This is due to the fact that GPUs perform better on repetitive tasks using large data blocks, and when data transfers are limited.

The importance of these results is twofold. First, the time to compute the approximate TF-IDF measure on the GPU is not affected by the type of data source (it may be a stream of tweets, a news feed, etc.) and second, the parallel implementation works well when GPU or graphic card memory is limited. In general, with input stream size equals or greater than 5M, our GPU implementation fully exploits the benefits of the data buffering and performs much better than the CPU implementation. Moreover, the GPU performance evidences a good scalability with varying the input data stream. The obtained values of speed up indeed, outline the validity of our parallel implementation for the calculation of the approximate TF-IDF, when the data collection is a real-time continuous data stream.

Rank Correlation. As stated above, we use the Kendall coefficient to assess the correlation between the most frequent pairs $< t, id >$, resulting from our approximate solution and those from the brute force approach.

Table 2 shows the results computed for different sizes of input stream s, and the k most frequent items. The table shows that our approach performs well: its accuracy in identifying the most frequent items is comparable with that of the brute force approach. In particular, we note that when $k = 1,000$, the τ coefficient is very close to 1 (i.e., greater than 0.9), even as the size of the input data stream grows to 20M. We also note that as k increases, the Kendall coefficient tends to decrease.

	1M		5M		10M		20M	
k	ρ	*recall*	ρ	*recall*	ρ	*recall*	ρ	*recall*
1K	0.9331	81.50%	0.9913	84.00%	0.9928	86.00%	0.9964	99.00%
10K	0.9513	75.05%	0.9850	79.40%	0.9621	79.90%	0.9981	99.20%
100K	0.9854	96.89%	0.9879	67.35%	0.9883	68.31%	0.9994	71.17%
1,000K	0.9994	93.50%	0.9876	59.07%	0.9764	61.73%	0.9301	61.18%

Table 3: Top 20% of terms.

	1M		5M		10M		20M	
k	ρ	*recall*	ρ	*recall*	ρ	*recall*	ρ	*recall*
1K	0.9331	67.75%	0.9872	78.25%	0.9865	81.25%	0.9965	99.50%
10K	0.9529	59.45%	0.9821	71.85%	0.9732	74.23%	0.9945	94.83%
100K	0.9742	62.27%	0.9700	33.67%	0.9780	67.37%	0.9984	93.27%
1,000K	0.9789	56.13%	0.9689	55.34%	0.9745	53.44%	0.9923	64.34%

Table 4: Top 40% of terms.

	1M		5M		10M		20M	
k	ρ	*recall*	ρ	*recall*	ρ	*recall*	ρ	*recall*
1K	0.9055	68.67%	0.9812	78.50%	0.9500	82.00%	0.9919	95.83%
10K	0.9566	48.73%	0.9824	65.25%	0.9768	70.38%	0.9947	90.60%
100K	0.9655	60.15%	0.9700	22.45%	0.9718	66.83%	0.9970	91.00%
1,000K	0.9673	63.56%	0.9712	47.63%	0.9612	88.35%	0.9832	86.12%

Table 5: Top 60% of terms.

	1M		5M		10M		20M	
k	ρ	*recall*	ρ	*recall*	ρ	*recall*	ρ	*recall*
1K	0.9015	66.38%	0.9675	75.50%	0.9409	76.63%	0.9894	91.50%
10K	0.9579	44.48%	0.9821	61.75%	0.9786	68.50%	0.9945	86.25%
100K	0.9596	57.86%	0.9655	64.79%	0.9670	66.90%	0.9956	89.07%
1,000K	0.9689	56.40%	0.9612	63.20%	0.9611	64.19%	0.9913	87.28%

Table 6: Top 80% of terms.

	1M		5M		10M		20M	
k	ρ	*recall*	ρ	*recall*	ρ	*recall*	ρ	*recall*
1K	0.8996	61.60%	0.9630	70.40%	0.9423	71.60%	0.9881	87.60%
10K	0.9577	43.23%	0.9807	59.14%	0.9785	66.34%	0.9941	81.94%
100K	0.9566	54.23%	0.9639	63.86%	0.9667	66.30%	0.9944	87.77%
1,000K	0.9632	56.42%	0.9523	62.84%	0.9632	56.09%	0.9432	79.83%

Table 7: Top 100% of terms.

Figure 9: Recall computed for the top 20%, 40%, 60%, 80% and 100%, for the sizes of the input streams s = 1M, 5M, 10M, and 20M, by varying k = 1K, 10K, 100K, and 1,000K.

TF-IDF quality. As stated, the analysis of TD-IDF quality is based on a comparison of the approximate TF-IDF values returned by our implementation and the exact values returned by the brute force algorithm. We use the Spearman's rank correlation coefficient ρ, to measure the strength of the relationship between the two variables. This coefficient provides a nonparametric measure of statistical dependence between paired data, in our case, the TD-IDF values computed by the two algorithms. Thus, we calculate the number of coincidences of the k most frequent terms obtained by the exact brute force algorithm with those found using the approximate TF-IDF measure.

We exploit the recall defined in Section 6, to compute the top-k terms common to the exact and approximate TF-IDF version. Once the top-k common terms are obtained, Spearman's rank correlation coefficient ρ can be computed.

Tables 3-7 illustrate the results from the Twitter dataset, with the 20%, 40%, 60%, 80%, and 100% of the k most frequent terms. In total, 16 experiments have been carried out. Given k and s, each experiment has been analyzed for the 20%, 40%, 60%, 80%, and 100% of the k most frequent terms. Just to give an example, with $k = 1,000$ and s=1M, the results (expressed in terms of recall and ρ) of the corresponding experiment is described by the first column and the first row of the each of the five tables (with varying the percentage). Increasing the percentage allows enlarging the "window" of the top terms from the most k frequent items, in order to better evaluate how many terms are in common with the exact TF-IDF, with varying the dimension of the window.

We note in general, that, given k, the recall value tends to decrease with the increase of the percentage of terms taken into account. Figure 9 shows the recall trend for each input stream size s, by varying the k frequent items. It is a summary view of the recall values described in Tables 3-7.

Moreover, we also note that in general, recall is lower for smaller sizes of s (i.e., s equal to 1M and 5M), compared to 10M and 20M. This result is due to the fact that the accurate identification of frequent items is affected by the size of the input stream. Our algorithm computes the k most frequent items, by processing a stream of size s. As stated, when a new stream arrives, it discards the items in the current stream in order to process the incoming stream. For small input streams, it is probable that some items that are discarded from the current stream also appear in the following stream, but in both cases, the occurrence is not high enough to included in the k most frequent items. By increasing the size of the stream, it is possible that these items appear in the same stream; in this case, the number of occurrences make them appropriate candidates to one of the top k items. In other words, when the input stream is large, the exact item count may be more accurate, because more items are kept in memory before they are discarded. Therefore, when s is small, the algorithm can only accurately count a few items. The recall indeed, is very good in the top 20% of the terms (see Table 3). As this percentage increases, recall values tend to decrease, as shown in Tables 4-7. Increasing the size of the input stream overcomes this problem for streams of size 10M or 20M.

There is some result that deserves to be further explored: the output generated by the experiment with input stream s =5M and $k = 100,000$ (Tables 3-7: second column, third row) shows a decreasing recall value, from the initial discrete recall value (67.35% on 20% of terms) until to get a very low recall, at 40% and 60% of the top terms (Tables 4, 5). It seems that by enlarging the percentage of the top k terms, the recall is lower, i.e., the common terms are dispersed. In other words, the common terms are mainly concentrate in the first 20%. The recall improves when the top percentage of terms is close to $k = 100,000$ (i.e., 80%, 100% of the top terms). From the analysis of the recall results of this experiment, it seems that the top most frequent items are accurately computed. Perhaps due to an unhappy sequence of data stream, the recall degrades (with middle percentages) and then back to acceptable values when it is calculated on the whole k size.

It also is interesting to notice that, fixed the input stream s = 20M, the recall tends to decrease, by varying the size of k (Tables 3-7). This fact outlines that, in general, the algorithm can correctly identify the most frequent items that appear in the top portion of the k frequent items. Yet, when the size of k is big (i.e., k = 1,000,000), the first 20% of terms is not enough to gather the most frequent items, that are in common with the precise TF-IDF measure. Just increase the percentage of k, in order to improve the recall (Tables 4 - 7). This situation emphasizes the fact that when $k < s$, the recall values are generally more accurate. Completely opposite situation happens for s=1M: the recall tends to increase by varying k (for example, see Table 3, but it hold also for Tables 4-7).

Tables 3-7 show also the values of the Spearman correlation coefficient ρ. It is always around 0.90, evidencing that a good positive correlation exists between the exact and approximate TF-IDF measure. In particular, for input stream

HASHTAG	TERMS
#VMA	tweet mtv vote lead video shareworthi
#bing	tweet mtv vote lead video shareworthi
#votebieber	justinbieb time hit bed button difficult morn
#rageofbahamut	tweet card ref code free friend
#voteonedirection	home daniellepeaz
#StopChildAbuse	donat tweet join campaign twitter
#Gameinsight	play start paradis island
#Kiss	excit list track reveal carlyraejepsen
#fashion	show largest voguemagazin
#votebeyonce	tweet vote mtv lead

Table 8: From top to bottom: the most frequent hashtags contained in tweets for the period August 2012 to September 2012. From left to right: the terms used most often in tweets with same hashtags.

size $s = 20M$, iρ is close to 1, showing a strong correlation.

Additionally, Tables 3-7 seem to show that bigger recall values cause better correlation value. To confirm this, we have assessed how the correlation varies using different recall values. For each experiment of the top-k common terms, we take into account the minimum and maximum recall. Using this data, we have performed a Wilcoxon rank sum test to check if the differences are or not significant. The test yields a p-value of 1.8165e-04, thus rejecting the null hypothesis (the medians are equal) and confirming the alternative hypothesis at the default 5% significance level, i.e., bigger recall values result in a better correlation.

The experiments confirm the validity of our implementation of the approximate TF-IDF measure, evidencing that, in general, the algorithm works well by varying the size s of input stream (as stated, the results are better for larger s). Moreover, the results computed for the different execution of the approximate TF-IDF outlines the effectiveness of our implementation choices (parallel GPU computation), supported also by a good efficacy.

6.2.2. Analysis of Trending Topics

Table 8 shows an ordered list of the most frequent document-term pairs. Hashtags represent documents. For each hashtag in the first column of Table 8, the second column shows the associated terms. Their order reflects the ranking of the most frequent hashtag-term pairs. It is interesting to note that the same terms often appear in the top positions of the ranked list, although they are associated with different hashtags. This is seen most clearly at the top of the list; they become more scattered as they appear lower down.

The tweets dataset was posted in the period August 2012 to September 2012 in the cities of Washington, New York, and San Francisco. We note that the most frequent hashtags were VMA (Video Music Awards), bing, votebieber, etc., these correspond to the 2012 MTV Video Music Awards that took place in the United States at the time. This emphasizes the coherence and relevance of our results. For example, our analysis showed that the hashtags VMA and bing always appeared together in tweets. This is clearly shown in Table 8, where these hashtags appear in the first two positions with the same terms (second column of the table).

Our results so far are rough, and our experiments could be improved with further textual processing and refinements.

7. Conclusions

The exact TF-IDF measure is typically used to retrieve relevant terms from a corpus of documents, given that the contents of the entire corpus are a priori known for the purposes of the calculation. This paper presents a revised TF-IDF measure, which can be used for processing continuous data streams. Specifically, our contribution is twofold: we provide 1) an approximate TF-IDF measure and 2) a parallel implementation of the calculation of the approximate TF-IDF, based on GPUs. The parallel GPU architecture meets fast response requirements and overcomes storage constraints when processing a continuous flow of data stream in real time.

The proposal was tested on both generated and real datasets. The generated dataset is composed of pairs $< t, id >$ and it has been created from a skewed and uniform distribution in order to study in details how our approximate TF-IDF works under a know distribution of terms and documents, respectively. The real dataset instead, was composed of a large Twitter collection, where it detected trending topics in the data stream. Both experimentations evidence that the approximate version of TF-IDF yields good results. We notice that the best results are when k is smaller (some order of magnitude) than the input stream size. We think that this is due to the frequency of distinct pairs inside the datasets. From the analyze of generated and real datasets, we think that when the data distribution presents a skew value close to 2 (a few pairs that appear many times, with respect to the dataset size), it preferable to choose a small k, in order to capture all the actual frequent items. For a large k, the most k frequent items will enclose also terms with a low frequency, that, in the successive input streams have been encountered, but are not effectively relevant.

In terms of performance, our approximate TF-IDF measure performs satisfactorily. One interesting aspect is the GPU implementation is stable and performs well even with limited memory. We found good scalability and a significant speed up of the GPU over the CPU implementation in computing the approximate TF-IDF. Furthermore, the time to compute the approximate TF-IDF measure on the GPU is not affected by the data source. In terms of quality, we correlated our results with those computed using the exact TF-IDF measure. Here again, the approximate solution is reasonably good when used in data mining scenarios.

With respect to future developments, we are working on an implementation using multiple GPUs. This may provide many benefits, although it is not immediately possible as the Thrust library is designed for a single GPU. To achieve this requires a complete software architecture redesign, in order to exploit multiple GPUs during the computation of the approximate measure.

In a nutshell, our approximate TF-IDF measure can be considered as an extension of the exact version. It can be applied in data stream data contexts: for instance, from sophisticated sensors, batch query processing in search engines, and in text stream classification techniques applied to social networks, news feeds, chat, and e-mail.

References

[1] cURL and libcurl. `http://http://curl.haxx.se/`.

[2] L. A. Adamic and B. A. Huberman. Zipf's Law and the Internet. *Glottometrics*, 3:143–150, 2002.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[4] Nathan Bell and Jared Hoberock. Thrust: A Productivity-Oriented Library for CUDA. *in GPU Computing Gems Jade Edition*, 2:359–371, 2011.

[5] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, August 2004.

[6] Khoo Khyou Bun and Mitsuru Ishizuka. Topic Extraction from News Archive Using TF*PDF Algorithm. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering*, WISE '02, pages 73–82, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Graham Cormode and Marios Hadjieleftheriou. Finding the frequent items in streams of data. *Communications of the ACM*, 52:97–105, October 2009.

[8] Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SIAM Conference on Data Mining (SDM)*, pages 44–55, 2005.

[9] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. A Practical Guide to Heavy Tails. chapter Heavy-tailed Probability Distributions in the World

Wide Web, pages 3–25. Birkhauser Boston Inc., Cambridge, MA, USA, 1998.

[10] Jason P. Duran and Sathish A. P. Kumar. CUDA Based Multi Objective Parallel Genetic Algorithms: Adapting Evolutionary Algorithms for Document Searches. In *Proceedings of the 2011 Intl Conf on Information and Knowledge Engineering, (IKE 2011)*, volume 52 (29), pages 36–49, 2011.

[11] Ugo Erra and Bernardino Frola. Frequent Items Mining Acceleration Exploiting Fast Parallel Sorting on the GPU. *Procedia Computer Science*, 9(0):86 – 95, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[12] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 611–622, New York, NY, USA, 2005. ACM.

[13] Raffaella Grieco, Delfina Malandrino, and Vittorio Scarano. A Scalable Cluster-based Infrastructure for Edge-computing Services. *World Wide Web*, 9(3):317–341, 2006.

[14] Daniel Horn. Stream Reduction Operations for GPGPU Applications. In Matt Pharr, editor, *GPU Gems 2*. Addison-Wesley, 2005.

[15] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, 1938.

[16] Mark Kot, Emily Silverman, and Celeste A. Berg. Zipf's law and the diversity of biology newsgroups. *Scientometrics*, 56(2):247–257, 2003.

[17] V.V. Kryssanov, K. Kakusho, E.L. Kuleshov, and M. Minoh. Modeling hypermedia-based communication. *Information Sciences*, 174(12):37 – 53, 2005.

[18] W. Li. Random texts exhibit Zipf's-law-like word frequency distribution. *Information Theory, IEEE Transactions on*, 38(6):1842–1845, Nov 1992.

[19] K.F. Lysakov and M.Yu. Shadrin. FPGA-based hardware accelerator for high-performance data-stream processing. *Pattern Recognition and Image Analysis*, 23(1):26–34, 2013.

[20] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[21] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big Data: The Next Frontier for Innovation, Competition, and Productivity. Technical report, McKinsey Global Institute, June 2011.

[22] Diana Maynard, Yaoyong Li, and Wim Peters. NLP Techniques for Term Extraction and Ontology Population. In *Proceedings of the 2008 Conference on Ontology Learning and Population: Bridging the Gap Between Text and Knowledge*, pages 107–127, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[23] Duane Merrill and Andrew Grimshaw. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[24] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, Nvidia Corporation, 2009.

[25] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide 5.5*. NVIDIA Corporation, 2010.

[26] Martin F. Porter. An Algorithm for Suffix Stripping. *Program*, 14(3):130–137, 1980.

[27] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 41–50, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[28] Joel W. Reed, Yu Jiao, Thomas E. Potok, Brian A. Klump, Mark T. Elmore, and Ali R. Hurson. TF-ICF: A New Term Weighting Scheme for Clustering Dynamic Data Streams. In *Proceedings of the 5th International Conference on Machine Learning and Applications*, ICMLA '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[29] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Efficient Stream Reduction on the GPU. In David Kaeli and Miriam Leeser, editors, *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[30] Gerard Salton and Christopher Buckley. Term-weighting Approaches in Automatic Text Retrieval. *Information Processing & Management*, 24(5):513–523, August 1988.

[31] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[32] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[33] Benjamin E. Teitler, Jagan Sankaranarayanan, Hanan Samet, and Marco D. Adelfio. Online Document Clustering Using GPUs. In Barbara Catania, Tania Cerquitelli, Silvia Chiusano, Giovanna Guerrini, Mirko Kmpf, Alfons Kemper, Boris Novikov, Themis Palpanas, Jaroslav Pokorn, and Athena Vakali, editors, *New Trends in Databases and Information Systems*, volume 241 of *Advances in Intelligent Systems and Computing*, pages 245–254. Springer International Publishing, 2014.

[34] S. Thomaidou and M. Vazirgiannis. Multiword Keyword Recommendation System for Online Advertising. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 423–427, 2011.

[35] Craig Ulmer, Maya Gokhale, Brian Gallagher, Philip Top, and Tina Eliassi-Rad. Massively parallel acceleration of a document-similarity classifier to detect web attacks. *Journal of Parallel and Distributed Computing*, 71(2):225 – 235, 2011. Data Intensive Computing.

[36] Le Wang, Yan Jia, and Yingwen Chen. Conversation Extraction in Dynamic Text Message Stream. *JCP*, 3(10):86–93, 2008.

[37] Yongpeng Zhang, Frank Mueller, Xiaohui Cui, and Thomas Potok. Data-intensive document clustering on graphics processing unit (GPU) clusters. *Journal of Parallel and Distributed Computing*, 71(2):211 – 224, 2011. Data Intensive Computing.