Document Stream Clustering using GPUs

Michael J. Szaszy Hanan Samet
Department of Computer Science, University of Maryland
College Park, MD 20742
{mszaszy, hjs}@cs.umd.edu

ABSTRACT

The Web is constantly generating streams of textual information in the form of News articles and Tweets. In order for Information Retrieval systems to make sense of all this data partitional clustering algorithms are used to create groups of similar documents. Traditional clustering algorithms, like K-means, are not well suited for stream processing where the dataset is constantly changing as new documents are published. In this paper we present a clustering algorithm designed to work with streaming documents. These documents, described by their TF-IDF (term frequency - inverse document frequency) [15] term vectors, are incrementally generated appropriate clusters based on the cosine similarity metric. We provide an efficient implementation of this algorithm on a GPU using CUDA, that achieves speedups of over 43X compared to its serial CPU implementation and has the ability to cluster a document within just .01 seconds after its term vector is received, even when there are 1.6 million clusters. Our implementation is capable to scale to clustering 5.5 million documents using a single GTX 480 GPU in 16.1 hours and can easily be extended to run on a system containing large numbers of GPUs.

Categories and Subject Descriptors

H.3.1 [**Document Representation and Content Analysis**]:

General Terms

Algorithms, Design, Performance

Keywords

Online Clustering, Stream Clustering, GPU, TF-IDF

1. INTRODUCTION

The days where people receive the majority of their news from just a few sources are long gone. For better or worse the Walter Cronkite era has been replaced by a perpetual 24/7 media machine filled with pundits and daily poll statistics. The Internet especially has created an explosion in the number of available news outlets, giving each of us a stage in the form of Blogs and Tweets. In order to stay relevant, traditional news outlets, like the New York Times, have invested heavily in maintaining an online presence. All of these News sources are accessible as data streams where the interested party subscribes to a feed and is notified when the publisher releases new content. Thousands of newspapers all over the world publish many thousands of articles in a single day. The sheer volume of data creates the necessity for technology to aid humans in making sense of it all. Our goal is to address just one particular angle of this problem. Specifically we want to be able to take newly generated documents, as they are published, and cluster them with other documents of similar content. While we use the word document throughout this paper the motivation of this work is to power the clustering of text from micro-bloggers on Twitter in the TwitterStand [16] system. Due to the very rapid stream of documents (tweets) that come from Twitter emphasis is placed on producing an online method for document clustering. The details on the integration of the work described in this paper into the larger TwitterStand system is ommitted, however, it is important to note that the this method is the current clustering method used by TwitterStand.

TF-IDF (term frequency-inverse document frequency) [15] has become a popular technique in Information Retrieval to determine the most important words of a document in a collection. A *term weight* for a given *term* is proportional to the frequency of the term in the document (e.g. TF) and inversely proportional to the frequency of that term in the entire document collection (e.g. IDF, how common it is).

There are a couple of variations on how exactly these weights are computed [13, 15]. For this work we use a weighting scheme *TF-ICF* [13], which is designed to work with streaming data by approximating the commonality (IDF) of a particular term based on a large corpora. We use the term *TF-ICF* and *TF-IDF* interchangeably throughout the rest of this document. It is also important to note that how these term weights are derived, while important for cluster quality, is not of much concern to the clustering process itself since our chosen metric works with normalized term vectors (see Equation 2).

A document, D_j, can be described by the set of its terms t_i and corresponding, term weights w_i, called a *term vector*.

$$D_j = (t_0, w_0), (t_1, w_1)...(t_n, w_n) \quad (1)$$

With corpus based IDF components, a documents term vector is constant and remains unchanged as additional documents are streamed and added to the document collection.

Once two documents have had their term vectors computed we can compute their similarity by using the cosine metric.

$$Sim(D_0, D_1) = \frac{\sum w_{0,i} * w_{1,i}}{\|D_0\| * \|D_1\|} \quad (2)$$

Note that Equation 2 devolves into a simple dot product when both D_0 and D_1 have normalized term vectors and that the range of $Sim = [0, 1]$ where 0 = no common terms and 1 = angle between vectors is 0 (e.g. term vectors are identical).

2. BACKGROUND

Graphics Processing Units (GPUs) have gained a lot of attention recently for their relatively low-cost high-throughput designs. A single high end GPU, like the NVIDIA Tesla C2075, at today's (Feb. 2012) market prices sells for about 5USD per CUDA core (448 total) and has 6GB of on card memory. However, the SIMD architecture of these platforms combined with little near core caches, as well as strict memory access patterns are the Achilles heel of these devices. Great care must be taken by the developer, especially in the case of *memory bound kernels*, to access data in a way that is efficient for the device. We will only address a few of these subtleties here as they have been extensively covered in [1, 2, 4, 18].

CUDA programs running on the device (GPU), called *kernels*, are launched by *host* processes running on the CPU. The usual workflow is that the CPU copies some data to the GPU, invokes a kernel, and reads the result back. More advanced techniques are possible through the use of co-processing. This allows a CPU to invoke a kernel and then continue executing its program in parallel. When the CPU needs to synchronize with the GPU, because it needs to read a result back and wants to know the result is ready, it can do so via a single API call.

There are many different types of memory available on the GPU. Each has its own optimized access patterns and latency characteristics. Global memory is by far the largest and slowest of these memories. A typical graphics card will have around 1-2GBs of this type of memory available for use by a CUDA application. The most optimal access pattern for global memory is when threads with consecutive ids access consecutive words from global memory. When this happens the memory access is coalesced, meaning that instead of requesting a 128 byte memory segment (in the case of 4B floats) to service the threads in a half-warp (16 threads), only 64B will be requested (1 float for each thread). This effectively doubles the memory bandwidth of the device, which in our case is the most precious resource of all. NVIDIA cards beyond compute capability 1.1 have relaxed this constraint slightly by allowing coalescing to occur as long as the 16 consecutive threads access memory from the same 64B memory segment.

Shared memory is another type of memory on the GPU, which is moderately sized and very low latency. It can be shared and used to communicate between all threads in the same thread block. The number of threads in a thread block can be chosen at runtime up to the maximum supported by the device, which is usually around 1024 threads. The fact that threads can use this low latency memory to communicate makes it ideal for parallel reduction operations where all the threads try to combine their local results into a single value based on some binary operator (e.g. sum, max, min). In the case where threads inside different blocks need to communicate, global memory needs to be used.

Texture memory, which was originally designed to store images for use in graphics applications, is also accessible from CUDA. Texture memory because of its original design for use with images is capable of providing caching mechanisms based on 1, 2, or 3 dimensional spatial locality. This special texture cache is not available to data stored in global memory and can provide valuable performance gains for smaller datasets.

3. RELATED WORK

GPUs have been shown to provide good performance for certain problem domains like bioinformatics [12], fluid dynamics, text mining [22], singular value decomposition [5], and clustering [6, 9, 17, 19, 21, 23].

[6, 9, 19, 21] all focus on K-means based clustering in a dense feature space. A document's term vector in general does not contain more than a few hundred terms even for news articles, which are comparatively longer than Tweets. Since the space spanned by these term vectors is the positive quadrant of a high-dimensional space (e.g. dimensions = the number of unique tokens in the language), they are extremely sparse. While these K-means implementations could be modified, at the cost of some performance, to deal with sparse feature spaces the bigger concern is in the nature of streaming news itself.

News, as the name implies, is contemporary and as such their contents reflect a constantly changing series of events from the recent past. This makes traditional clustering algorithms, like K-Means, ill-suited for the domain of News articles and Tweets, because it is unclear what value of K should be chosen.

Zhang et al [23] implement a *flock of birds* based clustering algorithm [8] on the GPU in order to cluster documents based on their TF-IDF term vectors. They show the feasibility of using GPUs to accelerate this task, but do so in a way that restricts their system by the network bandwidth between compute nodes. We show in this paper that by modifying existing techniques in GPU based Sparse Matrix Vector ($SpMV$) multiplication we can implement a much simpler document clustering algorithm than proposed in [23], while maintain high performance. We can do this due to a 3X speedup in similarity computations compared to [23]. We also achieve lower execution times using 1 GTX280 GPU for document collections less than 350K while the flock of birds algorithm uses 16 GTX280 GPUs. Furthermore, the proposed algorithm remains open to implementations on share nothing compute clusters containing many GPUs (see Section 6).

Our contributions apart from the clustering implementation itself is a mechanism whereby a row in a sparse matrix stored on a GPU in column major order can be updated without incurring prohibitively large penalties for frequent cudaMemcpy invocations, or writing large amounts of unnecessary bytes.

4. CLUSTERING ALGORITHM

In this paper we expand upon the work done by Teitler et al. [17]. Here the cosine similarity metric defined by Equation 2 serves as the foundation of the clustering algorithm. When a new document D is streamed from some source, then it is added to the most similar cluster C with

$Sim(D, C) > T$ where T is a user chosen value in the range $[0, 1]$. In the case where no such C exists then D serves as the start of a new cluster. Furthermore, one luxury that is granted is that a value K is chosen such that the number of terms in any C or D is $< K$.

Since these streams are constantly generating new data, being able to choose a cluster C for D without having to re-run an entire simulation from the beginning is very desirable. This algorithm allows us to do that, and in doing so, helps to optimize *latency* where latency is defined as, the time from when a published document enters the system to the time it has been added to the appropriate cluster. Also it does not require any domain knowledge for choosing the number of partitions to generate, like K-means. This is a two-sided blade in that this same lack of constraint means that the amount of memory required to store the number of resulting clusters from a document collection of size N, is $O(N)$ (e.g. none of the documents cluster together).

4.1 CPU Implementation

Algorithm 1 provides the pseudo code for the CPU based implementation.

Algorithm 1 Sequential CPU Clustering Algorithm.

```
1: procedure ClusterDocument( )
   Input: Document term vector D
   Input: Set of current clusters vectors C_all = C_0, C_1...C_n
2:     C_candidates ←⊆ C_all that share a term with D
3:     C_best ← C_i in C_candidates most similar to D
4:     if Sim(D, C_best) > T then
5:         Add D to C_best
6:     else
7:         Add D as new cluster C_{n+1}
8:     end if
9: end procedure
```

The operation on line 2 of Algorithm 1 is achieved through the use of a hash table index that maintains the set of all clusters $C_i..C_j$ that contain a given term t. A document can determine $C_{candidates}$ using this index by hashing each t in its own term vector and adding the set of returned clusters to $C_{candidates}$ without duplication. Finding C_{best} among the $C_{candidates}$ involves computing the similarity metric (Equation 2) between D and all C in $C_{candidates}$ and choosing the max.

Teitler et al. [17] show that the average-case complexity of this algorithm for N documents is $O(N * L_C * L_D)$ where L_C is the average number of clusters containing a given term t and L_D is the average number of terms in a document's term vector. The sparseness of these feature vectors leads to 2-3X gain in performance on the CPU when using $C_{candidates}$ as opposed to the brute force algorithm, which is always $O(N^2)$.

A term vector's t_i (see Equation 1), which is natively a string, is transformed into an integer id called a $term_{id}$, unique for that term. This allows for better compression when storing the data, as well as faster comparisons, and easier means for stack based allocations due to its fixed size. The penalty incurred for mapping between the $term_{id}$ and its constituent string only needs to be paid once upon a document entering the system, and whenever a user wants to display the data in its native format. In both the CPU and GPU implementation these mappings are maintained in an STL container and persisted on disk. However, for the purposes of clustering, there is never any need to transform a $term_{id}$ back into its corresponding term.

$\textsc{Sim}(D, C)$ is computed by first sorting D by increasing $term_{id}$ and normalizing the term vector, this takes $O(K * logK)$. All C in C_{all} are stored sorted and normalized along with their corresponding pre-normalized magnitude, $\|C\|$. This comes for free in the case of D starting a new cluster. With both C and D sorted by their $term_{id}$, computing the vector dot product, $C * D = O(K)$ [14].

Adding $C + D$ (line 5 of Algorithm 1) is the same as vector addition, since C is also described by a term vector. The addition processes is outlined in detail below:

1. De-normalize C and D by multiplying by their stored magnitudes
2. Add like terms of C and D
3. Choose top K term weights from $C + D$
4. Normalize and sort by $term_{id}$
5. Store the result back into C

This takes $O(K * logK)$ time due to the sorting required to choose the top K terms.

While all the sorting that is required may sound expensive, it does yields better results because K is usually small, between 20-200. So $K \ll L_C * L_D$, where the right-side of the inequality is equivalent to the number of dot products that need to be performed on average for a single D. With sorted term vectors we minimize the cost of a single dot product from $O(K^2)$ to $O(K)$.

4.2 GPU Implementation

The majority of the computation in Algorithm 1 takes place in the computation of the dot products between D and all clusters in $C_{candidates}$ (line 3). However, by using the power of the GPU we can speedup this computation by as much as 43X. This is despite the fact that our initial GPU implementation computes dot products between D and all clusters, C_{all}.

Algorithm 2 shows the pseudo code for portion of the GPU clustering algorithm that runs on the CPU (also known as the *host* process). While the GPU is very good at computing the dot products and performing local reductions, the memory access patterns involved with performing cluster updates (e.g. $D + C$) are too irregular to be done efficiently on the GPU. As such we perform these operations on the CPU in the same way as Algorithm 1.

Algorithm 2 line 8 shows when the GPU is invoked. The GPU is responsible for performing two main functions when a D needs to be clustered. It first computes all the similarity values between D and C_{all}. It then performs a *local* reduction to find the maximum similarity value. Here *local* means the maximum value in a given thread block. CUDA threads inside a *thread block* (256-1024 threads) can communicate using fast near core memory called *shared memory*. However, threads inside different thread blocks have no way of synchronizing amongst each other without going to global memory, which is 100X slower. Our GPU kernel, which is discussed in more detail later in Algorithm 3, only performs the reduction between threads in the same thread block. B is then an array of thread block local maxima. In order to find the global maximum, C_{best}, the CPU must perform

Algorithm 2 GPU Clustering Algorithm Host Process.

```
 1: procedure HOST
 2:     C ← Load Cluster Matrix from Persistent Storage
 3:     Copy C to GPU in ELL
 4:     loop
 5:         D ← read/wait for new document
 6:         Sort(D) by term_id
 7:         Copy D to GPU
 8:         B ← FINDBESTCLUSTERKERNEL(...)
 9:         C_best ← FINDMAX(B)
10:         if SIM(D, C_best) > T then
11:             Add D to C_best
12:             Copy C_best to update buffer
13:         else
14:             Add D as new cluster C_{n+1}
15:             Copy D to update buffer
16:         end if
17:     end loop
18: end procedure
```

a final scan of B. The rest of the host process remains relatively unchanged from the CPU implementation. The update buffers on line 12 and 15 are discussed later.

We turn our attention now to the GPU kernel itself and begin by first framing the problem by considering that C_{all} is a matrix where each cluster is a row and D is a column vector. All term vectors are normalized reducing $\text{SIM}(D, C)$ to a dot product computation. Algorithm 1 line 3 is then equivalent to finding $max(S)$:

$$S = \begin{bmatrix} sim_0 \\ sim_1 \\ \vdots \\ sim_{M-1} \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} & ... & C_{0,Q} \\ C_{1,0} & C_{1,1} & ... & C_{1,Q} \\ \vdots & \vdots & ... & \vdots \\ C_{M-1,0} & C_{M-1,1} & ... & C_{M-1,Q} \end{bmatrix} \cdot \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_Q \end{bmatrix} \tag{3}$$

The problem is that Q is very large (number of unique terms) while only K or fewer of these entries will be non-zero. This makes the problem identical to Sparse Matrix Vector (*SpMV*) multiplication.

There has been a considerable amount of research in implementing SpMV using GPUs [2, 4, 7, 18]. Most of this research was spurred by its application to linear solvers. While there has been some application of SpMV outside the traditional domains [20], this is, to the best of our knowledge, the first time it has been used to compute TF-IDF cosine similarities. We give a brief overview of this research, as is relevent to our algorithm, below.

Bell and Garland [2] were among the first to profile many different types of sparse matrix formats on the GPU and found they could achieve speedups of up to 30X compared to optimized CPU implementation. One of these formats, *ELLPACK* (ELL), stores at most MAX_{nzr} number of non-zero entries per row. Each of these rows is allocated exactly MAX_{nzr} entries of space regardless of the actual number of non-zeroes and 1 GPU thread is assigned to each row. A sparse matrix encoded in ELL is described by a data matrix and an indices matrix, both with dimensions R x MAX_{nzr} where R is the number of rows. Figure 1 gives an example of a matrix, A, stored in ELL.

Digitally the ELL indices and data matrices are stored as two 1D arrays, like a key-value pair defined by two associative arrays. In order for the memory accesses to these arrays to be *coalesced* by the GPU, they are stored in column-major order, shown below for matrix A:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \; indices = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

Figure 1: Example of matrix A stored in ELL

$$\text{data} = \begin{bmatrix} 1 & 2 & 5 & 6 & 7 & 8 & 3 & 4 & * & * & 9 & * \end{bmatrix}$$

$$\text{indices} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 2 & 2 & 3 & * & * & 3 & * \end{bmatrix}$$

While ELL has the largest potential for speedup, it is marred by several limitations. Namely that each row is allocated MAX_{nzr} space. SpMV kernels are known to be *memory bound kernels*, because each matrix entry is involved in just 1 floating point operation [2]. As a consequence ELL's performance becomes highly susceptible to the distribution of non-zeroes per row. This is because the effectiveness of the compression depends on the variability of non-zero terms in the rows. As the variability between rows becomes too large, the compression becomes less effective leading to sub-optimal pruning of the non-zero data, which will eventually find its way into the SpMV multiplication.

Choi et al [7] optimize ELL by partitioning A into a set of rows and storing each partition in a separate ELL matrix. This format known as Blocked ELLPACK (BELLPACK) showed up to a 1.5X speedup on average over ELLPACK for single-precision values. The added performance comes from performing row-permutations before the partition phase to group rows with similar numbers of non-zeroes into the same partition, in affect increasing the compression.

Vazquez et al [18] proposed a GPU implementation known as ELL-T, which was based on the ELLPACK-R storage format. ELLPACK-R differs from ELLPACK in that there is an added array rl, which stores the row length of a row. Continuing the example from Figure 1 then:

$$rl = \begin{bmatrix} 2 & 2 & 3 & 2 \end{bmatrix}$$

ELL-R allows the GPU kernel to move an if statement from the inner loop to the outer loop, which helps avoid branch divergence in the kernel. ELL-T expanded on ELL-R by assigning T threads to compute the value for one row. The partial values would be summed together by a reduction operation in the GPU's shared memory. Vazquez et al [18] showed performance improvements for matrices with a large row length, where T was set fairly low $T \leq 16$. Another optimization used by Vazquez is that of forcing the alignment property for the ELL encoding.

This alignment property is critical in ensuring optimal memory access to the indices and data arrays. On the GPU when threads access consecutive words of memory then the request is coalesced, meaning that if 16 threads request 16 consecutive float values, instead of the GPU issuing a request for 128 bytes of memory, the request will be coalesced into a single 64 byte request. This coalescing can

only be achieved if the threads begin their access on an aligned memory boundary. In the case of single-precision floats this means the first thread's request must lay on a 64-byte word boundary. This alignment must be maintained for all MAX_{nzr} values of the row (stored in column-major). To achieve the proper alignment *padding* is inserted between the last row's value for a particular column and the first row's value for the next column. The positions where padding would be inserted into A are noted with an '_' below:

$$\text{data} = \begin{bmatrix} 1 & 2 & 5 & 6 & \_ & 7 & 8 & 3 & 4 & \_ & * & * & 9 & * \end{bmatrix}$$

$$\text{indices} = \begin{bmatrix} 0 & 1 & 0 & 1 & \_ & 1 & 2 & 2 & 3 & \_ & * & * & 3 & * \end{bmatrix}$$

Different GPUs can have different alignment requirements but assume a 64B boundary, where each float is 4 bytes, then in the example above '_' would expand to 12 arbitrary float values. While this is counterproductive to the issue of compression, it exhibits better memory behavior because all MAX_{nzr} memory requests for a row are coalesced. Note that we do not need to pad the end of the arrays since the only goal is to ensure that each MAX_{nzr} column starts on a 64B boundary. Since no more elements will be read after the last set it is safe to omit the padding from the end.

For the purposes of our TF-IDF document clustering we already choose a K, which serves as the maximum number of terms in the term vector. This K is usually slightly less than the mean number of terms in the collections (20 - 200). Even if a cluster, when it is created, does not contain K terms, after additional documents join the cluster it will quickly reach K. This is of course a function of T which dictates how similar any joining D will be. Choosing appropriate values for K eliminates the need for more advanced permutation schemes like BELLPACK. Based on work by Vazquez et al [18] we also choose not to use ELL-T since the width of our matrices, K, is not very large. As such, we choose the basic ELL along with the additional alignment constraint as the basis for our GPU kernel.

There are two main shortcomings to the current state of GPU SpMV when applied to our problem domain and they are shared by the non-ELL formats:

1. They assume the vector V is dense
2. Updating a row is prohibitively expensive

Shortcoming 1 can be solved by either performing a binary search on V or using the method used in our CPU implementation [14]. Both methods require V to be sorted by their keys ($term_{id}$ in TF-IDF) and both pose challenges for the GPU as they are prone to branch divergence. Zhang et al [23] faced a similar problem for their similarity kernel and found binary search to be superior. We found empirically that when V is stored in the GPU's texture memory, the performance differences are minimal. We end up choosing Rieck's [14] method because it offers a thread warp the opportunity to exit the inner loop of the multiplication early if all threads in the warp find $max(V_{key}) < t$, where t is the current key in the SpM that is being searched for in V.

Shortcoming 2 is a much more serious problem. Since the cluster matrix is stored in column-major order updating a single row needs $2 * K$ invocations of cudaMemcpy (e.g. K data elements and K term_{id} elements). When performed this way, 40% of the execution time of the application is spent writing row updates to the GPU. Alternatively 2 cudaMemcpy invocations can be made pushing both matrices back onto the device in their entirety. Due to the size of the cluster indices and data matrices this is also very inefficient and yields worse results for even moderately sized matrices (200,000 x K entries).

In order to reduce the number of cudaMemcpy invocations without writing lots of unnecessary bytes, we reserve a buffer with 1 row worth of memory on the GPU to store the updated cluster in row-major order. We further allocate an equivalent amount of space on the host using cudaHostMalloc, which pins the allocated memory so that it cannot be paged out. Since K is not very large, we argue that pinning $2 * K$ integers and floats is acceptable. When a cluster needs to be created or updated, then we push it through this pinned memory to the GPU. The buffer only needs to be 1 row in size because when a document D_i, is clustered it is added to just 1 cluster or creates a new cluster.

The row update sits inside the update buffer until D_{i+1} needs to have its similarities computed. The thread in the GPU kernel assigned to compute the dot product for the

Algorithm 3

```
1: procedure FindBestClusterKernel( )
   Input: numRows - number of clusters
   Input: K - number of non-zeros
   Input: padding - padding between columns
   Input: indices - Indicies of ELL encoded cluster matrix
   Input: data - Data of ELL encoded cluster matrix
   Input: updId - update row id
   Input: updV - term vector for updated row
   Input: D - term vector of document
   Input: out - Output buffer
2:     shared ← Initialize shared memory
3:     rowId ← blockDim.x * blockIdx.x + threadIdx.x
4:     dot ← 0.0
5:     vidx ← 0 // Starting index to search for next term in D
6:     off ← numRows + padding
7:
8:     if rowId < numRows then
9:         // Does my warp contain the updating row?
10:        if _any( rowId == updId ) then
11:            for n ← 0 ... K do
12:                if rowId == updId then
13:                    // Update Cluster
14:                    indices[off*n+rowId] ← updV.term_id[n]
15:                    data[off * n + rowId] ← updV.weight[n]
16:                end if
17:                col ← indices[off * n + row]
18:                dot+ = data[off * n + row]*Seek(D, vidx, col)
19:            end for
20:        else
21:            // My warp doesn't have the updater
22:            for n ← 0 ... K AND vidx ≠ K do
23:                col ← indices[off * n + row]
24:                dot+ = data[off * n + row]*Seek(D,vidx,col)
25:            end for
26:        end if
27:    end if
28:
29:    shared[rowId] ← dot
30:    ... shared mem reduction, store max dot in shared[0] ...
31:
32:    // Write result to output buffer
33:    if shared[0] == dot then
34:        out[blockIdx.x].id ← atomicExch(rowId)
35:        out[blockIdx.x].sim ← atomicExch(dot)
36:    end if
37: end procedure
```

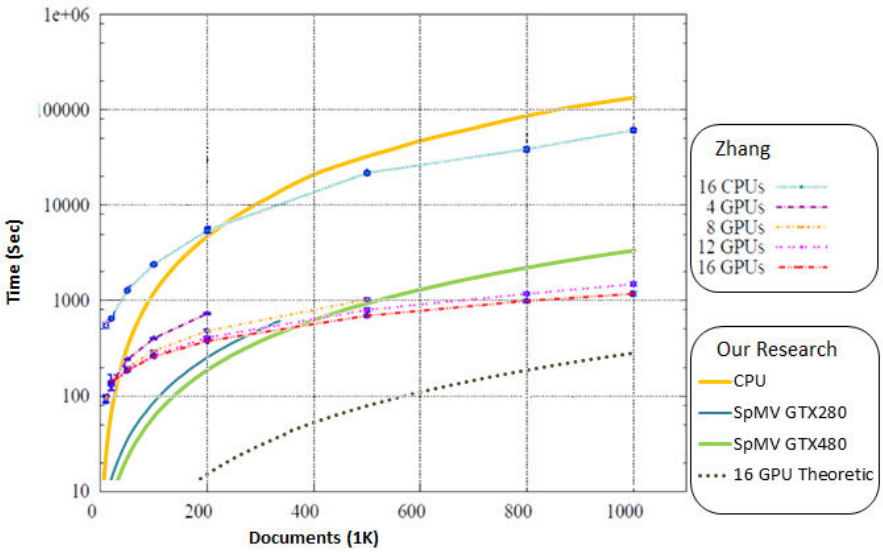


Figure 2: Throughput performance for increasing number of documents compared against [23].

row that needs to be updated pushes the values from the update buffer into the ELL encoded matrices just before it computes the dot product for D_{i+1}. Once the GPU kernel is done executing, the update buffer is no longer needed by D_i and is ready for use by D_{i+1}. This method reduces the percent of execution time spent updating rows from 40% to 3%, with a raw speedup of 10X and no significant rise in the execution time of the multiplication kernel.

Algorithm 3 shows the pseudo code for the GPU kernel. Lines 2-6 initialize some local variables. Line 10 is an operation known as a *warp vote*, where if any thread inside a thread warp (32 threads) evaluates the branch to true, then all threads in that warp will take that branch. By performing the warp vote outside the multiplication loop we can eliminate complicated boolean logic inside the multiplication loop. For instance if a thread, t_{upd}, is the updating thread than it must execute all K iterations of the multiplication loop regardless if $vidx$ has seeked to the end of D. This is because it must update all K entries for its cluster. Because t_{upd} must perform all K iterations, all threads inside its warp will be stuck waiting as well.

For the threads not in t_{upd}'s warp, repeatedly evaluating whether they are the updating thread is unnecessary, while there is also a chance that all threads in the warp can exit early (e.g. before $n = K$). The rest of the multiplication loop is similar to those described in [2, 18]. Line 30 performs the thread block local reduction using the 6^{th} variant described in [11]. Line 33-36 writes the thread block local maxima back to global memory so that the CPU can perform the global reduction. One thing to note is that access to $shared[0]$ is served via a *shared memory broadcast*, meaning that threads will not be serialized trying to access that shared memory bank. Atomic operations are needed to set the output values in the unlikely case that multiple threads computed the same maximal dot product. In this situation, the last writer to $out[blockIdx.x].sim$ wins.

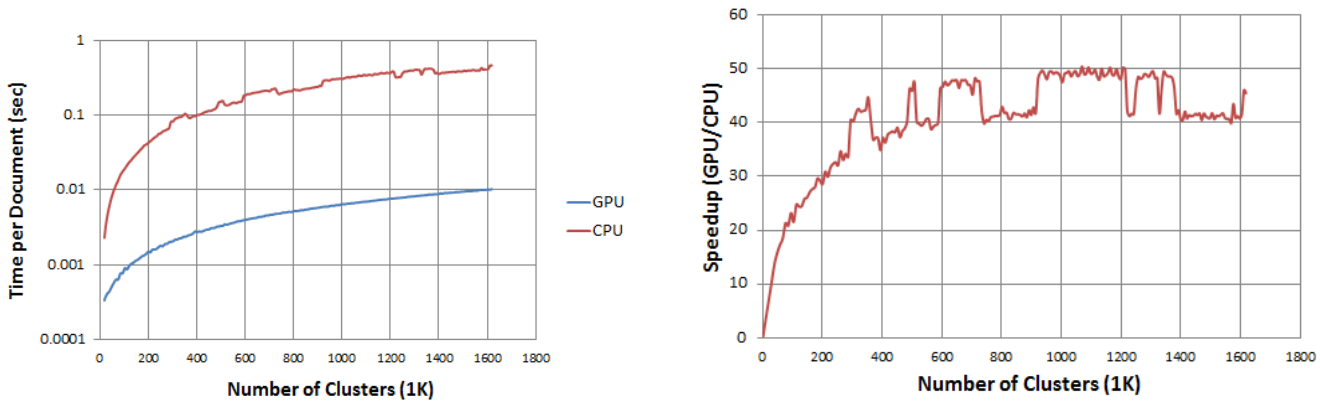
5. EXPERIMENTS

We tested our CPU and GPU implementation using datasets provided by the UCI Machine Learning Repository [10]. The second largest dataset from [10], features only 300,000 New York Times articles, which is not a large enough document collection for us to test against. In order to compare against current research in the area [23] we need to use document collections of at least 1 million. UCI's largest dataset is generated from 8.2 million PubMed articles.

This data is provided pre-tokenized with the removal of stopwords. The data is also truncated by keeping only those words that occurred more than ten times. The document data comes in sparse format, meaning that an integer id is provided for each term, term frequency pair as opposed to the word itself. A separate vocabulary dataset is provided in order to map these integers keys back to their constituent words. We use these keys directly in our system as the term vector's term_{id}. If a document has more than K terms, then we keep the top K.

The entire 8.2M document set serves as the corpus for our document collection. This makes the weighting scheme more analogous to a traditional TF-IDF term weighting scheme, than a TF-ICF term weighting scheme. However, the weighting scheme is not pertinent to the performance of the clustering algorithm itself. As such, the time spent computing a document's term vector weights is omitted from the timing data presented here. This is also true for the timing data in [23] that we compare against.

We run our CPU implementation on a machine with a Intel Core2 Quad Q9450 processor and 4GBs of RAM. The application is compiled using GCC version 4.1.2 with the -O2 flag. The GPU implementation is run on a CPU host with a Dual Core AMD Opteron 2218 processor and 2GBs of RAM. The GPU for our GPU implementation is a NVIDIA GTX 480. We build our application using the NVIDIA CUDA



(a) latency in seconds

(b) speedup of latency

Figure 3: Latency performance when computing against varying number of clusters.

SDK version 3.2 targeting Compute Capability 2.0 and GCC version 4.1.2 with the -O2 flag. We also perform some experiments with a NVIDIA GTX 280 graphics card. The GTX 280 sits on a CPU host with the same specs as the 480. The build configuration is nearly identical except we target Compute Capability 1.3, which is the highest version supported by that card. The GTX 280 is the card also used by Zhang et al [23] in Figure 2.

In order to compare against current research we measure throughput, which for us is the length of time needed to cluster N documents, starting with 0 clusters. Figure 2 shows a plot of our throughput performance against Zhang et al [23]. In order to produce this graph we overlaid our data directly on top of Zhang's graph. The graph shows the clustering times for document collections of up to 1M. We average our GPU running times from 3 independent runs for $K = 35$ and $T = .6$. Our CPU is also run for $K = 35$ and $T = .6$. The median number of terms per article inside the PubMed dataset for the first 5.5M articles is 55.

The set of plots near the top of the legend are all those from [23], while the lower set are ours. The plots from Zhang et al [23] as more GPUs are added shifts little. This is due to the bottleneck being, not the GPUs, but the communication between the different MPI compute nodes. For 16 GPUs, about 60% of their execution time is spent communicating *boids* between nodes. This is partially a result of the power of the GPUs as the communication costs for CPU clusters is only 1-7% for document collections over 100K.

Due to limitations on our access to the GTX 280 we were only able to process 350,000 documents on that device instead of 1M as in our other experiments. With the 950MBs of global memory available to the GTX 280, the card should be able to process much more (depending on the value of K), since we need only $O(M)$ space:

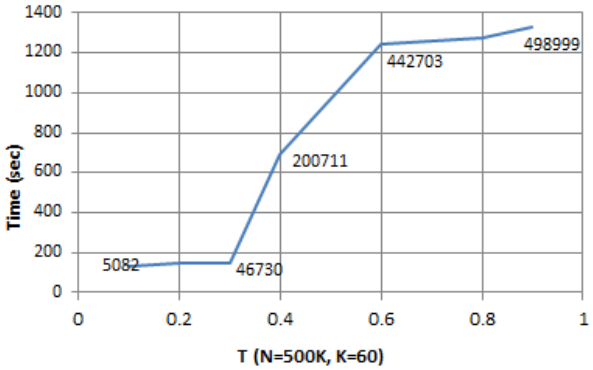
$$Max_{clusters} = \frac{950MB}{\text{SIZE}(KeyType) * \text{SIZE}(ValueType) * K} \quad (4)$$

Having data points up to 1M would have been ideal for a more direct comparison against [23]. We can, however, project it based on our timing data up to 350,000 data points by comparing it against the performance of the GTX 480, for which we collect data up to 5.5M documents. When we compare the performance of the GTX 480 against the GTX 280 we see a consistent 30% reduction in execution time for more than 50,000 documents. Under 50,000 we see 15%. This may seem surprising because the 480 has 2X as many CUDA cores as the 280. The cause is that the SpMV kernels are memory bound, so the performance is dictated by the memory bandwidth of the device. The GTX 280 has a memory bandwidth of 141.7 GB/s while the GTX 480 is 177.4 GB/s. This is a 25% increase in memory bandwidth and thus our model and observed performance differences fit nicely.

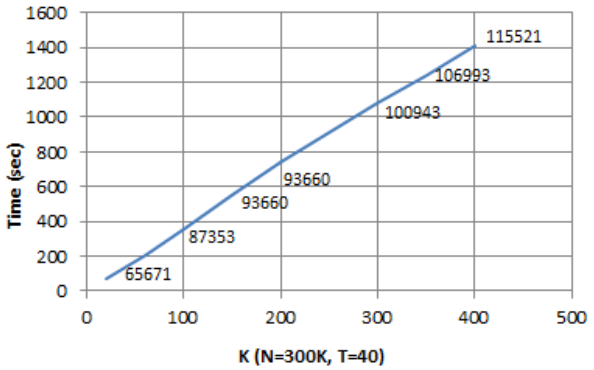
Even though our implementation using 1 GPU is not faster than Zhang et al's 16 GPU cluster, our throughput results still look promising for two reasons. With the flock of birds clustering algorithm even with 4 GTX 280 graphics cards only 200K documents can be clustered, without implementing a paging mechanism between the GPU and CPU. It isn't clear from the paper [23] whether this memory restriction rose from the clustering algorithm itself or in combination with memory allocations being used by other portions of their system. By contrast we can cluster between 300K-2M using a single GTX 280 (see Equation 4) and on the 1 GTX 480 we cluster 5.5M documents in 16.1 hours with $T = .6$, which is a high tolerance leading to rapid cluster growth (3.5M clusters at the end of the clustering). The second reason is that our algorithm is very easily extendable to a GPU/MPI compute cluster (see Section 6). We plot the *theoretical* speedup of this work as if we had 16 GTX 280 graphics cards.

The best case speedup can be achieved if the communication overhead introduced by moving to a GPU/MPI machine can be hidden behind the GPU compute kernel, which currently accounts for 95% of the clustering execution time (see Section 6). In this scenario the speedup is directly proportional to the increase in memory bandwidth, which we've already seen indications of in the performance differences of the GTX 280 and 480. With two cards of the same type the effective memory bandwidth is doubled thus yielding a 2X speedup in performance.

For the purposes of our algorithm we measure the performance in terms of latency, which is the length of time



(a) 500K documents as T is varied and $K = 60$.



(b) 300K documents as K is varied and $T = .4$

Figure 4: Performance characteristics when varying both K and T. Numbers next to the plot indicate the number of clusters after processing all N documents.

needed to cluster a single document D when there are M clusters. Due to limitations in the granularity of the clock, we measure this statistic by timing how long it takes 10,000 documents to be clustered and divide by that number.

One of the goals of our design is to provide an algorithm that can incrementally cluster documents in order to reduce latency. Figure 3 compares the latency performance of our GPU implementation against our CPU implementation. We do not plot number of documents on the X axis because the performance of these algorithms depends on the number of clusters. The number of clusters is a function of the number of documents, K and T. By graphing the X axis versus cluster size we minimize the effects of varying K or T during our analysis. Figure 3 also shows some sporadic jumps in the performance of the CPU, while the GPU remains smooth. The problem with the CPU implementation is that in order to process 2M documents it takes a whole week. The areas of the graph that show spikes in the CPU usage are caused by system maintenance or others users logging into the machine. When stating our speedup of 43X we ignore these regions of graphs.

Our best speedups from Figure 3b approach 43X. This is also close to the range of our speedup against the CPU in the throughput oriented experiments (30-40X). Teitler et al[17] obtained speedups of only 3-4X with their GPU implementation over the CPU baseline for moderate to large document collections. This make our GPU implementation 10X faster than their GPU implementation. The reason for this performance difference is in the efficient memory coalescing of the SpM in the SpMV multiplication kernel, as well as an immediate thread-block local reduction phase. Furthermore, when we look at how long it takes to cluster a single document it takes the GPU just .01 seconds even when there are 1.61M clusters. This is equivalent to 100 documents per second.

In order to see the sensitivity of our implementation to varying values of K and T, we first run experiments with 500K documents fixed at $K = 60$ and then vary T. The number of clusters generated for various configurations are shown on the graph in Figure 4a. As we'd expect as T increases the running time also increases. This is because the performance is proportional to the number of clusters generated. With large values for T cluster growth is more rapid, leading to more similarity computations. We also notice that cluster size is more sensitive in the range 0.1 -0.6 than over 0.6. Performance is most sensitive between 0.3 -0.6. The reason the 9-fold increase in cluster size between $T = .1$ and $T = .3$ does not have a drastic effect on performance is that the GPU has not yet become saturated. Changes beyond $T = .6$ produce little variation in the number of output clusters, which explains why the performance is flat on the right side of Figure 4a.

When measuring the effect of K on the algorithm we use the New York Times dataset [10], consisting of 300K documents. The benefit of this dataset over the PubMed dataset [10] is that the average number of terms per document is larger (230 as opposed to 50). When we look at the data in Figure 4b there are two noticeable things. First, we notice a weak linear scaling with respect to increasing values of K. Second, we notice that as K is increased the number of output clusters increases. This increase is nowhere near as pronounced as when T is varied and is part of the reason we achieve linear weak scaling, but it does gradually increase. This is due to the nature of moving to higher dimensional space [3]. As more terms are added to the term vectors, their normalized weights are reduced. This means that to maintain the same similarity value as when a smaller K was chose, more terms must be in common between the cluster and the document.

6. FUTURE WORK

As part of our ongoing work we plan to implement our system using a cluster of GPUs. GPUs between machines will be tied together via an interconnection network and programmed against using MPI. The goal of this work is to try and achieve our ideal speedup graphed in Figure 2.

In this scenario we use simple row-based partitioning to evenly distribute the rows of C_{all} amongst the various graphics cards. This partitioning can favor those cards with better memory bandwidth. The only modification that is required to Algorithm 2 is that the *global reduction* becomes a compute-node local reduction. A final reduction must be done between the MPI nodes to determine the maximum, and then this value needs to be scattered back out to notify

the compute nodes. In the case of a new cluster, the node next in line to store a cluster does so.

We can further increase throughput of this architecture by completely hiding the cost of the global reduction. If more than 1 document needs to be clustered, then with a slight relaxation of the algorithm, we can begin computing D_{i+1} immediately after D_i has had its thread-block local maxima copied to the CPU. The cost of the compute-node local maxima and global maxima can then be completely overlapped with the similarity computation of D_{i+1}. With the extension of the update buffer to an update queue the architecture is complete.

We also plan on investigating the application of our work to query processing. What makes this appealing is that while there is strict formatting imposed on the cluster matrix, the document term vector is free to be nearly unbounded in length.

7. CONCLUSION

We have shown a practical GPU-based implementation for online document clustering that is 43X faster than its CPU analog and exhibits weak linear scaling with respect to K, the maximum number of terms in a term vector. Our algorithm is based on an efficient similarity computation kernel that builds upon existing research in Sparse Matrix Vector multiplication. We provide an effective means to update a sparse matrix stored on a GPU and encoded in ELLPACK. Our performance for document collections less than 350K using just 1 GTX 280 is better than current research using between 4-16 GTX 280s. Our results for collections larger than 350K has a great potential to achieve strong linear scaling by increasing the effective memory bandwidth to the GPU kernel by increasing the number of GPUs. All while hiding any added communication costs. Our current implementation uses $O(m)$ memory on the GPU, where $m =$ the number of clusters, allowing us to cluster collections of 5.5M documents on a single GTX 480 GPU in 16.1 hours. The measured latency of our GPU implementation is .01 seconds per document when 1.61 million clusters exist in the system (100 documents per second).

8. REFERENCES

[1] *CUDA C Programming Guide 3.2*, Nov. 2010.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[3] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 217–235, London, UK, 1999. Springer-Verlag. ISBN 3-540-65452-6. URL `http://dl.acm.org/citation.cfm?id=645503.656271`.

[4] R. Bordawekar and M. M. Baskaran. Optimizing sparse matrix-vector multiplication on gpus. *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, (RC24704), 2008.

[5] J. M. Cavanagh, T. E. Potok, and X. Cui. Parallel latent semantic analysis using a graphics processing unit. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2505–2510, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-505-5. doi: http://doi.acm.org/10.1145/1570256.1570352. URL `http://doi.acm.org/10.1145/1570256.1570352`.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68:1370–1380, October 2008. ISSN 0743-7315. doi: 10.1016/j.jpdc.2008.05.014. URL `http://dl.acm.org/citation.cfm?id=1412749.1412827`.

[7] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *SIGPLAN Not.*, 45:115–126, Jan. 2010. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1837853.1693471. URL `http://doi.acm.org/10.1145/1837853.1693471`.

[8] X. Cui, J. Gao, and T. E. Potok. A flocking based algorithm for document clustering analysis. *J. Syst. Archit.*, 52:505–515, August 2006. ISSN 1383-7621. doi: 10.1016/j.sysarc.2006.02.003. URL `http://dl.acm.org/citation.cfm?id=1163824.1163829`.

[9] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sander, and K. Yang. Parallel data mining on graphics processors. Technical report, The Hong Kong University of Science and Technology, Oct. 2008.

[10] A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL `http://archive.ics.uci.edu/ml`.

[11] M. Harris. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2008. URL `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[12] S. A. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9 Suppl 2(Suppl 2):S10–9, Mar. 2008. ISSN 1471-2105. doi: 10.1186/1471-2105-9-S2-S10.

[13] J. W. Reed, Y. Jiao, T. E. Potok, B. A. Klump, M. T. Elmore, and A. 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*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2735-3. doi: 10.1109/ICMLA.2006.50. URL `http://dl.acm.org/citation.cfm?id=1193211.1193734`.

[14] K. Rieck and P. Laskov. Linear-time computation of similarity measures for sequential data. *J. Mach. Learn. Res.*, 9:23–48, June 2008. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1390681.1390683`.

[15] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*, 24:513–523, August 1988. ISSN 0306-4573. doi: 10.1016/0306-4573(88)90021-0. URL `http://dl.acm.org/citation.cfm?id=54259.54260`.

[16] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. Twitterstand: news in tweets. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 42–51, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-649-6. doi: 10.1145/1653771.1653781. URL `http://doi.acm.org/`

10.1145/1653771.1653781.

[17] B. E. Teitler, J. Sankaranarayanan, and H. Samet. Online document clustering using the GPU. Technical Report TR–4970, Computer Science Department, University of Maryland, College Park, MD, Aug. 2010. URL http://www.cs.umd.edu/~hjs/pubs/GPUClusteringFinalWithReferences.pdf.

[18] F. Vazquez, G. Ortega, J. J. Fernandez, and E. M. Garzon. Improving the performance of the sparse matrix vector product with gpus. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1146–1151, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4108-2. doi: http://dx.doi.org/10.1109/CIT.2010.208. URL http://dx.doi.org/10.1109/CIT.2010.208.

[19] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using gpus. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, UCHPC-MAW '09, pages 1–6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-557-4. doi: http://doi.acm.org/10.1145/1531666.1531668. URL http://doi.acm.org/10.1145/1531666.1531668.

[20] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. Efficient pagerank and spmv computation on amd gpus. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 81–89, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4156-3. doi: http://dx.doi.org/10.1109/ICPP.2010.17. URL http://dx.doi.org/10.1109/ICPP.2010.17.

[21] M. Zechner and M. Granitzer. Accelerating k-means on the graphics processor via cuda. In *Proceedings of the 2009 First International Conference on Intensive Applications and Services*, pages 7–15, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3585-2. doi: 10.1109/INTENSIVE.2009.19. URL http://dl.acm.org/citation.cfm?id=1547557.1548166.

[22] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Gpu-accelerated text mining. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, Mar. 2009.

[23] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Data-intensive document clustering on graphics processing unit (gpu) clusters. *Journal of Parallel and Distributed Computing*, 71(2):211 – 224, 2011. ISSN 0743-7315.