Online Document Clustering Using the GPU

Benjamin E. Teitler, Jagan Sankaranarayanan, Hanan Samet
Center for Automation Research
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, Maryland 20742 USA
{bteitler,jagan,hjs}@cs.umd.edu

August 2010

Online document clustering takes as its input a list of document vectors, ordered by time. A document vector consists of a list of *K* terms and their associated weights. The generation of terms and their weights from the document text may vary, but the TF-IDF (term frequency-inverse document frequency) method is popular for clustering applications [1]. The assumption is that the resulting document vector is a good overall representation of the original document. We note that the dimensionality of the document vectors is very high (potentially infinite), since a document could potentially contain any word (term). We also note that the vectors are sparse in the sense that most term weights have a zero value. We assume that each term not explicitly present in a particular document vector has a weight of zero. Document vectors are normalized.

Clusters are also represented as a list of weighted terms. At any given time, a cluster's term vector is equal to the average of all the document vector's contained by the cluster. Cluster term vectors are truncated to the top *K* terms (those containing the highest term weights). Cluster term vectors are kept normalized.

The objective of the algorithm is to partition the set of document vectors into a set of clusters, each cluster containing only those documents which are similar to each other with respect to some metric. For this paper, we consider the Euclidean dot product as the similarity metric, as it has been shown to provide good results with the TF-IDF metric [1]. The similarity between a cluster and a document is defined as the dot product between their term vectors.

We first present serial a algorithm for online clustering. We then describe a PRAM algorithm for parallel online clustering, assuming a CRCW model. Finally, we present a practical implementation of an approximate parallel online clustering algorithm, suitable for the CUDA parallel computing architecture [2].

1. Serial Clustering

The basic serial online clustering algorithm takes as input a list of *n* document vectors, as well as a clustering threshold *T* ranging between 0 and 1. Below is a high level overview of the algorithm.

Serial Clusterer 1:

```
For each document $D$ (ranging from 0 to $n - 1$)
        Choose the cluster $C$ most similar to $D$
        If similarity( $C$, $D$ ) > $T$
                Add document $D$ to cluster $C$, and recompute $C$'s term vector
        Else
                Create a new cluster consisting of only the document $D$
```

In the worst case, Serial Clusterer 1 takes **O(n^2)** dot products to cluster *n* documents. However, we can use our knowledge of the data to reduce the number of dot products needed. We observe that due to the high dimensionality of the vectors, spatial sorting data structures such as K-D trees, Quadtrees and their variants are inefficient as they degenerate to the linear case with high enough dimensions [3]. However, we can take advantage of the sparseness of document vectors to reduce the number of distances needed per document to only the number of clusters with a non-zero similarity. We assume that most document vectors have very few terms in common with other document vectors (otherwise, all documents would cluster together and little or no clustering would be necessary). Therefore, each term in document vector *D* will have a limited number of clusters whose term vector contains a non zero weight for that term.

We take advantage of this knowledge by keeping a list of these clusters for each unique term seen by the clustering algorithm so far. This enables us to reduce the number of dot products needed per document to only those dot products that will be non-zero.

Let *D*[*t*] represent the weight of term *t* for document *D* (the weight associated with *t* in *D*'s term vector). Similarly, let *C*[*t*] represent the weight of term *t* for cluster *C*. We can avoid unnecessary work within dot products by keeping the term weight in each term list with its corresponding cluster. For instance, the term list for term *t* is of the form:

TermList[*t*] = {(*C*1, *C*1[*t*]), (*C*2, *C*2[*t*]), (*C*3, *C*3[*t*]) … (*Cp*, C*p*[*t*])}

This indicates that cluster *Ci* contains a weight *Ci* for term *t*. Adding the weight information to the term list allows us to compute only the non-zero *partial* dot products between documents and clusters efficiently, since we have no need to look up *t*'s weight in *Ci*'s term vector. Below is the new serial clustering algorithm which makes use of the *TermList* data structure. Note that we use *D* both to refer to the document *D* (specifically its term vector), as well as the *D*'s priority in the list of documents to be clustered, ranging from 0 to *n – 1*.

Serial Clusterer 2:

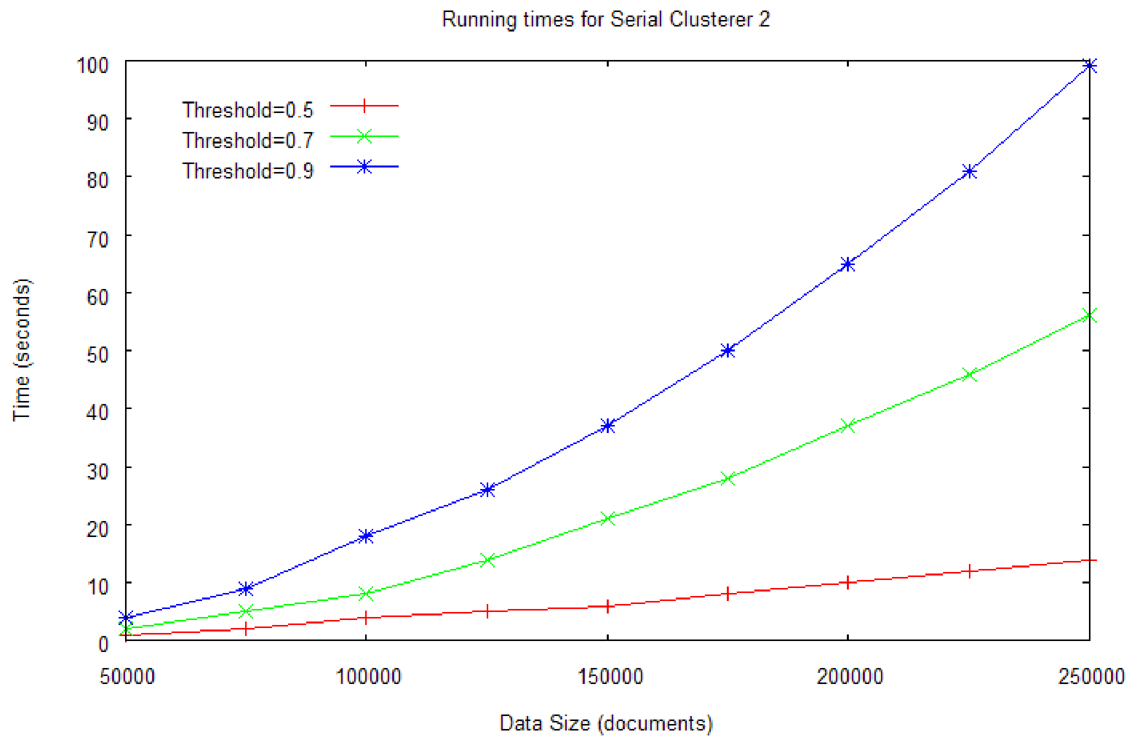
```
$TermList$ = Set of empty lists
For each document $D$ (ranging from 0 to $n - 1$)
        $Candidates$ ← Empty Set
```

```
Results ← Array of size D, initialized to all 0
For each term t in D's term vector
    For each (Ci, Ci[ t ]) in TermList[ t ]
        Results[ Ci ] = Results[ Ci ] + Ci[ t ] * D[ t ]
        If Candidates does not contain Ci
            Add Ci to Candidates

Choose the cluster C in Candidates with the max( Results[ C ] )
If similarity( C, D ) > T
    For each term t in C's term vector
        Remove C's entry (C, C[ t ]) from TermList[ t ]
    Add document D to cluster C and recompute C's term vector
    For each term t in C's term vector
        Insert (C, D[ t ]) into TermList[ t ]
Else
    Create a new cluster C consisting of only the document D
    For each term t in C's term vector
        Insert (C, C[ t ] ) into TermList[ t ]
```

We can approximate the running time cost of Serial Clusterer 2. Recall that *K* is the number of terms kept in each of the document and cluster term vectors. Let *L* represent the average number of clusters that contain any given term *t* at any specific time in the clustering algorithm. This indicates that to cluster any given document *D*, we have roughly $K * L$ partial dot product computations. We also have at most $K * L$ insertions into the *Candidates* set, each taking **O(1)** time using a hash set implementation. We finally have at most *K* deletion and *K* insertions from lists of size *L*, in order to update the *TermList* data structure. Assuming an array data structure for each *TermList*[*t*], we have **O(1)** insertion and **O(L)** deletion for each term. We have a total of $K * L + K * L + K + K * L = \mathbf{3 * K * (L + 1}$) time. This is **O(n * K * L)** for the entire algorithm. We note that although *L* is highly dependent on the dataset, it is expected to be far less than *n*. The memory require for Serial Cluster 2 is **O(m * K)**, where *m* is the total number of clusters at the end of the algorithm. This is because each cluster has *K* entries in the *TermList* structure.

We timed the Serial Clusterer 2 algorithm on a real world dataset, which consists of news documents from a span of 90 days taken from a wide variety of news sources. The documents are ordered by time of publication. Each news document contains 20 terms in its term vector ($K = 20$). Our implementation is written in C++ and compiled using g++ (GCC) version 4.1.2 with the –O3 optimization flag. We ran our implementation on an AMD 1.0 GHz processor.



2. Parallel Clustering (Single Document)

We first consider the case of parallelizing the work associated with clustering a single document, while still clustering each of the *n* documents in serial. This is an interesting case as it is guaranteed to produce near identical output to the serial clustering algorithm. Later we will discuss the case of processing multiple documents in parallel, and the effects on the clustering output.

Our goal is to parallelize as much of Serial Clusterer 2's document loop as possible. We first note that the dot product operations are highly parallelizable. All the partial dot product operations for a given document can be done in parallel. We can then run a parallel sorting operation with the cluster as the sorting key. Finally, we run a parallel summation operation to gather the completed dot products for each cluster, followed by a parallel maximum operation to choose the cluster with best similarity to *D*.

After the best cluster *C* has been chosen, we must update our *TermList* data structure to reflect the changes to *C*'s term vector. We first delete the old *TermList* entries of *C* by assigning a different processor to look at each entry of *TermList*[*t*], for every term *t* in *C*. Processors that find their entry (*Ci*, C*i*[*t*]) swap in the last value of the *TermList*[*t*] to compact that list (assuming an array implementation). Inserting the new (*Ci*, C*i*[*t*]) values can trivially be done by assigning *K* processors to add the new (*Ci*, C*i*[*t*]) to the end of their respective lists.

Below we present a high level parallel algorithm for clustering. We introduce the **pardo** keyword to indicate that the contents of a loop are done in parallel. We also introduce a value *ThreadID* which is available to each thread within a **pardo** loop. For *h* threads, the values of *ThreadID* range between 0 and *h* – 1 inclusively. Assume that each parallel thread is assigned a unique *ThreadID* value. We assume a PRAM architecture using the CRCW (Concurrent read – concurrent write) model [?].

Parallel Clusterer 1:

```
TermList = Set of empty lists
For each document D (ranging from 0 to n – 1)
    Partials ← Array initialized to all 0

    Let {t1, t2 … tK} be the terms in D's term vector.
    S ← { ({t1} x TermList[ t1 ])  ({t2} x TermList[ t2 ])  …  ({tK} x TermList[ tK ]) }
    For each (ti, Ci, C[ ti ]) in S pardo
        Partials[ ThreadID ] = (Ci, D[ ti ] * C[ ti ])

    Run parallel sort on Partials, sorting by Ci
    Run parallel summation on Partials (adding similar Ci)
    Run parallel max on Partials to produce best candidate cluster C

    If similarity( C, D ) > T
        Let {u1, u2 … uK} be the terms in C's term vector.
        R ← { ({u1} x TermList[ t1 ])  ({u2} x TermList[ t2 ])  …  ({uK} x TermList[ tK ]) }
        For each (ti, Ci, C[ ti ]) in R pardo
            If Ci == C //found a match
                Remove C's entry (C, C[ ti ]) from TermList[ ti ]
        Add document D to cluster C and recompute C's term vector
        For each term t in C's term vector pardo
            Insert (C, D[ t ]) into TermList[ t ]
    Else
        Create a new cluster C consisting of only the document D
        For each term t in C's term vector pardo
            Insert (C, C[ t ] ) into TermList[ t ]
```

We can estimate the running time of Parallel Clusterer 1. For each document, we can compute the partial dot products in **O(1)** time using **K * L** processors (we ignore the complication here of assigning *ThreadID*s to processors). Parallel sort is known to be logarithmic [?], and so we have **O(log(K * L))** time. Parallel summation of *Partials* can be done in **O(log(K * L))** time, and the parallel max operation is also **O(log(K * L))** time. Finally, the *TermList* maintenance operations are **O(1**) time each. The final running time for the algorithm is **O(n * log(K * L))**. The memory require for Parallel Cluster 1 is again **O(m * K)**.

We now discuss the process of assigning *ThreadID*s to processors for Parallel Clusterer 1's partial dot product computation. Recall that we have specified *L* as the average size of *TermList*[*t*] for any given term *t*. This is useful for analyzing running time, but the sizes *TermList*[*t*] will very greatly when clustering a *specific* document, which complicates the *ThreadID* assignment. Our goal is to decide on a specific (*ti*, *Ci*, *C*[*ti*]) to associate with every *ThreadID*. This requires deciding on one specific element of each *TermList*[*t*] for each *ThreadID*. One option is to compute the maximum *TermList*[*t*] size for all terms *t* for the given document *D*. Let *M* refer to this maximum. We can now use *M* * *K* threads for our partial dot product computation, some of which will have wasted work. Let *TermList*[*t*][*j*] refer to the *j*-th element of the term list for term *t*.

Let {*t1, t2 … tK*} be the terms in *D*'s term vector.
M ← maximum *TermList*[*ti*] size (*i* ranging from 1 to *K*)

ThreadID Assignment 1:

```
u ← ThreadID / M
if( size( TermList[ tu ] <= ThreadID % M )
        Partials[ ThreadID ] = NULL
else
        C = TermList[ tu ][ ThreadID % M ]
        Partials[ ThreadID ] = (C, D[ tu ] * C[ tu ])
```

The problem with this approach is that it can be very wasteful if the sizes of the term lists are severely lopsided (which is the common case in real data sets). We also have an additional problem of NULL data being written to the *Partial*s array to deal with. An approach that involves no wasted threads requires doing a little extra work to find out exactly which term list element *ThreadID* belongs to. Let size(*TermList*[*t*]) represent the number of elements currently in the term list for *t*.

Let {*t1, t2 … tK*} be the terms in *D*'s term vector.
TermSizes ← {size(*TermList*[*t1*]), size(*TermList*[*t2*]) … size(*TermList*[*tK*]) }
PrefixSums ← the prefix sums of *TermSizes*

ThreadID Assignment 2:

```
Run a binary search on PrefixSums to identify the smallest u such that ThreadID < PrefixSums[ u ]
C = TermList[ tu ][ PrefixSums[ u ] - ThreadID - 1 ]
Partials[ ThreadID ] = (C, D[ tu ] * C[ tu ])
```

We note that PrefixSums[*i* - 1] tells us how many threads should be assigned to term lists 1 upto *i* - 1. This means that the term *u* assigned to a given *ThreadID* is simply the first *u* such that *ThreadID* < *PrefixSums*[*u*]. The value *PrefixSums*[*u*] – *ThreadID* – 1 gives us the index into *TermList*[*u*] we are interested in.

Each binary search using ThreadID Assignment 2 takes **O(log(K))** time. We observe that this does not change the overall running time of Parallel Clusterer 1, since it is dominated by the cost of sorting.

Finally we note that the parallel deletion that occurs in Parallel Clusterer 1 requires an identical *ThreadID* configuration as the partial dot products upon implementation. Each deletion thread will receive a unique *ThreadID*, and must decide which *TermList* entry to examine. We can use ThreadID Assignment 2 where *t1, t2* … *tK* are the terms in *C*'s term vector, instead of *D*'s. Again, the overall running time is unchanged.

3. Extending To Multiple Documents in Parallel

In this section we examine an algorithm for clustering multiple documents in parallel. Assume we wish to cluster *Q* documents in parallel. We define the multiple document clustering algorithm below.

Multiple Document Clusterer 1:

```
While we have more documents to cluster
    Choose the next Q documents D1, D2, … DQ
    Choose clusters C1, C2, … CQ such that Ci is the most similar cluster to Di
    For i = 1 up to Q
        If similarity( Ci, Di ) > T
            Add document Di to cluster Ci and recompute Ci's term vector
```

The main difference between the multiple document algorithm and single document algorithm is that we assign best clusters to *Q* documents before updating the cluster term vectors and the index. This can lead to poor clustering in some cases, since document D*i* is never compared against the effects of D1, D2 … D*i*-1. This effect can possibly be mitigated by merging similar clusters at variance points in the algorithm. We assume the effects of this problem are minimal as long as *Q* is much less than *n.*

We wish to extend Parallel Clusterer 1 to cluster *Q* documents in parallel. We first note that computing the partial dot products for *Q* documents can be done using *Q* parallel instances of the single document version of the dot product computation. However, assigning *ThreadID*s for multiple documents now requires reasoning about which document a thread belongs to. This results in a binary search of a prefix sums array of size *K* * *Q* for each thread to assign work.

Assume *tij* refers to the *j-th* term of document *Di* (the *j-th* term of the *i-th* document we are clustering in parallel)

```
TermSizes ← {size(TermList[t11]), size(TermList[t12]) … size(TermList[ t1K ]),
             size(TermList[t21]), size(TermList[t22]) … size(TermList[ t2K ]),
             …
             size(TermList[tQ1]), size(TermList[tQ2]) … size(TermList[ tQK ])}
PrefixSums ← the prefix sums of TermSizes
```

ThreadID Assignment 3:

```
    Run a binary search on PrefixSums to identify the smallest u such that
        ThreadID < PrefixSums[ u ]

    q ← u / K
    r ← u % K
    C = TermList[ tqr ][ PrefixSums[ u ] - ThreadID - 1 ]
    Partials[ ThreadID ] = (Dq, C, D[ tqr ] * C[tqr ])
```

Note that each entry contained in *Partials* now contains an extra element which indicates which document the partial dot product belongs to. ThreadID Assignment 3 guarantees however that similar *Dq* values will be contiguous within *Partials*. This means that we can sort *Q* separate sub lists in parallel (each of size roughly *K* * *L*).

Once, we haven chosen the appropriate clusters *C1, C2 .. CQ*, we must now update the *TermList* data structure to reflect the changes of the *Q* cluster term vectors. We cannot simply do these operations in parallel for all *Q* documents as in the single document case, since different clusters may have terms in common. This means that they will update the same *TermList*[*t*] and interfere with each other. To deal with this issue, we assume the existence of an atomic addition operator (this is a reasonable assumption, since newer version of CUDA provide atomic operators). Below we define the parallel insertion operation given a term *t*, and an element to insert *x*.

```
AtomicInsertion( t, x ):
     size = atomicAdd( TermList[ t ].size, 1 )
    TermList[ t ][ size ] = x
```

Each thread that attempts to insert into a given *TermList*[*t*] will receive a unique slot to receive its element. After all insertions have been completed, the new size for *TermList*[*t*] is the new size of the list. The parallel deletion algorithm is defined below.

```
AtomicDeletion( t, x ):
    TermList[ t ].deleteNumber = 0
    TermList[ t ].deletePriority = 0
    TermList[ t ].newSize = TermList[ t ].size
    atomicAdd( TermList[ t ].deleteNumber, 1 )
    atomicAdd( TermList[ t ].deletePriority, 1 )
    atomicAdd( TermList[ t ].newSize, -1 )

    for i = 0 upto TermList[ t ].size – 1 pardo
        TermList[ t ][ i ].deleted = FALSE
        if( TermList[ t ][ i ] == x ) //Found the element we are searching for
            TermList[ t ][ i ].deleted = TRUE //mark we are deting
            if( i  >= TermList[ t ].size - TermList[ t ].deleteNumber )
                Return //nothing to do since end of list
            priority = atomicAdd( TermList[ t ][ i ].deletePriority, -1 )
            numSkip =  TermList[ t ].deleteNumber - TermList[ t ].priority
            j = The element numSkip elements from the end of TermList[ t ]
                such that TermList[ t ].deleted == FALSE
            TermList[ t ][ i ] = TermList[ t ][ x ] //compact list
    TermList[ t ].size = TermList[ t ].newSize //update the new size
```

The basic idea behind this algorithm is to assign a priority to each thread that finds an element to delete. Based on this priority, the thread picks the correct element near the end of the list to move into the hole created by the deleted element. This algorithm assumes each parallel call to AtomicDeletion has a unique (*t*, *x*) (no call has both the same *t* and *x* as another call). This is a valid assumption, since we can prune *Ci* values that are duplicates prior to running the AtomicDeletion, as the result of including them is the same as that without. The proof of correctness of this algorithm is outside the scope of this paper. A detailed algorithm for the parallel multiple document clusterer is outlined below.

Multiple Document Clusterer 2:

```
While we have more documents to cluster
    Choose the next Q documents D1, D2, … DQ

    Partials ← Array initialized to all 0

    TermSizes ← {size(TermList[t11]), size(TermList[t12]) … size(TermList[ t1K ]),
                 size(TermList[t21]), size(TermList[t22]) … size(TermList[ t2K ]),
                 …
                 size(TermList[tQ1]), size(TermList[tQ2]) … size(TermList[ tQK ])}
    PrefixSums ← the prefix sums of TermSizes

    For ThreadID = 0 upto PrefixSums[ Q * K – 1 ]- 1 pardo
        Run a binary search on PrefixSums to identify the smallest u such that
        ThreadID < PrefixSums[ u ]

        q ← u / K
        r ← u % K
        C = TermList[ tqr ][ PrefixSums[ u ] - ThreadID - 1 ]
        Partials[ ThreadID ] = (Dq, C, D[ tqr ] * C[ tqr ])

    Run parallel sort on Partials, sorting by Dq, Ci
    Run parallel summation on Partials (adding similar Dq, Ci)
    Run parallel max on Partials to produce best candidates C1 … CQ

    For i = 1 upto Q pardo
        For j = 1 upto K pardo
            AtomicDeletion( TermList[ Ci[ j ] ], Ci )

    Add documents to their correct clusters and recompute clusters' term vectors.

    For i = 1 upto Q pardo
        For j = 1 upto K pardo
            AtomicInsertion( TermList[ Ci[ j ] ], Ci )
```

Assume that *w* threads accessing the same memory using an atomicAdd operation take *w* time for all instances of atomicAdd to complete. Since at most *Q* clusters can access the same *TermList*[*t*] at a given time, both AtomicInsertion and AtomicDeletion are **O(Q)** time in the worst case.

The running time of Multiple Document Clusterer 2 is approximately **O((n / Q) * Max(log(K * L), Q))**. The memory requirement is on average **O(Max(m * K), K * L * Q)**, where *K* * *L* * *Q* is average size of the *Partials* array.

4. Approximate Nearest Clusters

Multiple Document Clusterer 2 requires a sort of *Partials* which is of size $Q * K * L$. We can prune this step by using an approximate nearest neighbor algorithm that also makes use of the atomicAdd operator. Assume C is the best cluster for a given document D (highest similarity). We wish to find a cluster Ci such that similarity(C, D) – similarity(Ci, D) <= *EPSILON*, for some *EPSILON*.

We can achieve this goal by creating B = (1.0 / *EPSILON*) buckets for each document D. Our new algorithm computes in parallel the distances between D and all other clusters Ci, using an atomicAdd on the partial dot products relevant to D and Ci. We then write Ci to the bucket for D corresponding to its range of similarity, namely b = similarity(Ci, D) / *EPSILON*. We choose the approximate nearest cluster Ci as the Ci in the highest non-zero bucket for D (the highest index bucket which had at least one write to it). The new partial dot product code for multiple document clustering is as follows:

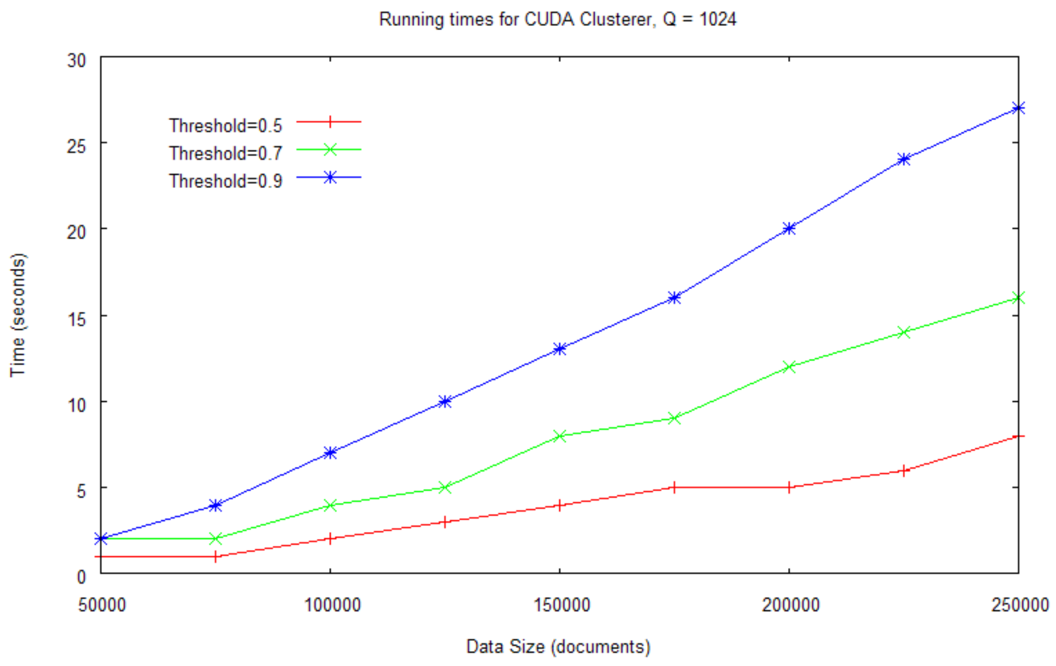
```
Let m be the current number of clusters
Sums ← Array initialized to 0
Buckets ← Array of size Q * B
For ThreadID = 0 upto PrefixSums[ Q * K – 1 ]- 1 pardo
        Run a binary search on PrefixSums to identify the smallest u such that ThreadID < PrefixSums[ u ]

        q ← u / K
        r ← u % K
        C = TermList[ tqr ][ PrefixSums[ u ] - ThreadID - 1 ]
        AtomicAdd( Sums[ q * m + C ], D[ tqr ] * C[ tqr ] )
        If Sums[ q * m + C ] > T //passes the threshold test
                Buckets[ q * B + Sums[ q * m + C ] / EPSILON ]
```

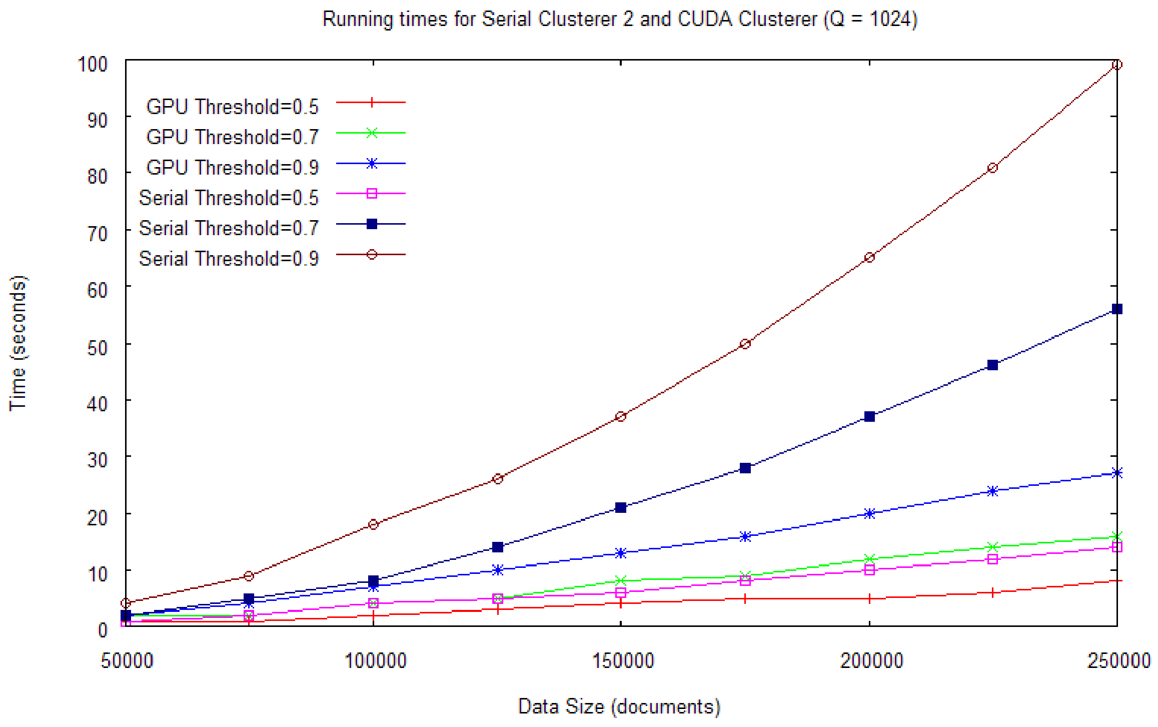
In the worse case of a document D having K terms in common with a cluster C, we have **O(*K*)** running time for this code block. The running time for a parallel document clusterer using the approximate nearest neighbor algorithm is approximately **O((n / Q) * Max(log(K * L), K))**. The memory requirement for any given iteration (clustering Q documents) is **O(m * Q)**, since the candidates clusters can range between 0 and m – 1, and we have Q documents in parallel.

5. Results

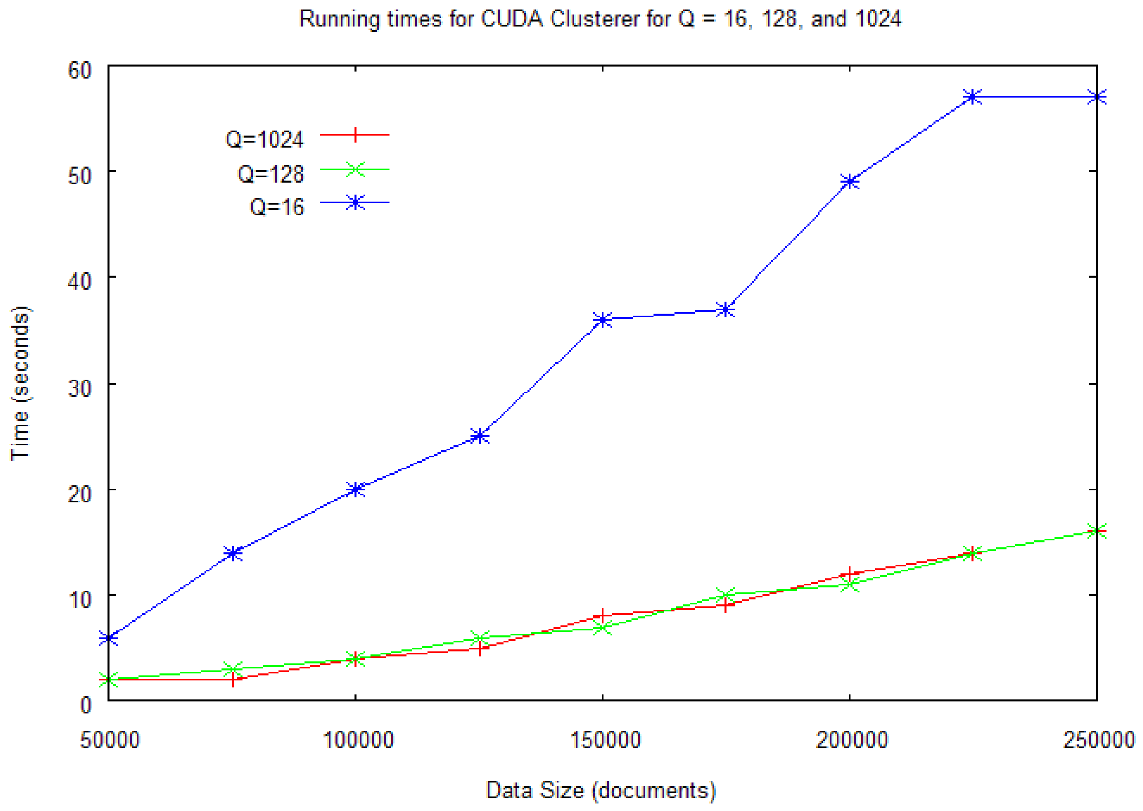
For our CUDA implementation, we selected an algorithm similar to Multiple Document Clusterer 2, using the approximate nearest cluster algorithm discussed in the previous section for. We use CUDA SDK version 2.2. We tested our implementation on a GeForce GTX 280, which has 240 cores and 1 GB of global memory. We tested our algorithm on the same real world data set as Serial Clusterer 2. Below is the results for Q = 1024 (1024 documents done in parallel) and B = 16.



The next graph directly compares the CUDA clusterer vs. Serial Clusterer 2 for the various threshold values. We note that the best speedup is achieved using the highest clustering threshold. This is expected as a higher clustering threshold means there will be more clusters, and therefore more cluster candidates per document (more non-zero partial dot products).



Finally we compare the running times of the CUDA Clusterer for three different values of *Q* (16, 128, and 1024) using a threshold of 0.7. We observe from this graph that there exists a threshold for *Q* such that no more speedup is possible. This tells us that the GPU is fully saturated at this threshold.



6. Conclusions

In this paper we have described a parallel algorithm for online document clustering. We have implemented a practical version of the algorithm in CUDA, and compared the running times with a serial online clustering algorithm. We have shown that significant speedups can be achieved for this problem on the GPU (3-4 times speedup for our biggest data size) for moderate to high clustering thresholds.

7. Acknowledgments

This work was supported in part by the National Science Foundation under Grants IIS-07-13501, IIS-09-48548, IIS-08-12377, and CCF-08-30618, and as well as NVIDIA Corporation.

8. References

1. G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. Information Processing & Management, 24(5):513-523, 1988.

2. J. Sanders and E. Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, Reading, MA, 2011.

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), C. Beeri and P. Buneman, eds., vol. 1540 of Springer Verlag Lecture Notes in Computer Science, pages 217-235, Berlin, Germany, January 1999.

4. R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high dimensional spaces. In Proceedings of the 24th International Conference on Very Large Data Bases (VLDB), A. Gupta, O. Shmueli, and J. Widom, eds., pages 194-205, New York, August 1998.