To appear in the *ADBIS workshop on GPUs In Databases*, Sep 2013, Genoa, Italy

Online Document Clustering Using GPUs*

Benjamin E. Teitler, Jagan Sankaranarayanan
Hanan Samet, and Marco D. Adelfio

Department of Computer Science
University of Maryland
College Park, Maryland 20742 USA
{bteitler,jagan,hjs,marco}@cs.umd.edu

Abstract. An algorithm for performing online clustering on the GPU is proposed which makes heavy use of the atomic operations available on the GPU. The algorithm can cluster multiple documents in parallel in way that can saturate all the parallel threads on the GPU. The algorithm takes advantage of atomic operations available on the GPU in order to cluster multiple documents at the same time. The algorithm results in up to 3X speedup using a real time news document data set as well as on randomly generated data compared to a baseline algorithm on the GPU that clusters only one document at a time.

Keywords: Document Clustering, GPU, NewsStand, TwitterStand

1 Introduction

Our work on indexing spatial and temporal data [1, 4–6] and similarity searching [3, 7, 9] in the serial domain as well as in a distributed domain [10] and on GPUs [2] has led us to work on indexing textual representations of spatial data found in documents such as news articles [11] and tweets [8] to be accessed using a map query interface. A key piece of technology that makes all these systems work is an online clustering algorithm that takes news articles and noisy tweets as input streams and aggregates them into news topics. As news articles and Tweets enter our system as an input stream, we assign them to news clusters, which is a one-shot process, meaning that once an article is added to a cluster, it remains there forever. We will never revisit or recluster the news article, which is desirable because articles and Tweets are coming at a high throughput rate, and we need a fast and efficient clustering system that maintains good quality clustering output. In other words, our clustering algorithm is an online algorithm, and the additional constraints imposed on this problem add new complexity. Our clustering algorithm is different from traditional document clustering algorithms (such as the ones used by Google News) as we do not have access to the entire

* This work was supported in part by the National Science Foundation under Grants IIS-08-12377, IIS-09-48548, IIS-10-18475, and IIS-12-19023; and by Google and NVIDIA. J. Sankaranarayanan is currently at NEC Labs

data set at the start of the algorithm. In particular, our online clustering algorithm is a leader-follower type algorithm [8, 11], which means that our similarity function takes into account both content as well as the publishing time.

The focus of this paper is on developing clustering methods that are both online in nature as well as being able to take advantage of the parallelism and computational intensity afforded by Graphical Processing units in order to keep pace with the rate of arrival of these news articles. To cope with the high rate of our input stream and the need to process the input quickly in one shot, requires the mapping of the online clustering algorithm to the GPU in order to achieve a reasonable speed-up versus a CPU only implementation. Our clustering implementation uses the vector space model to represent documents, and makes use of the popular TF-IDF (term frequency inverse document frequency) method for computing term weights. We use the Euclidean dot product as our similarly metric between document vectors and cluster vectors. Online clustering is a challenging problem for the GPU because it is bandwidth intensive as opposed to being only computationally intensive. Fast document clustering requires maintaining an index on the clusters associated with every term in the document corpus. This allows for fast pruning of clusters that have no terms in common with a given document. This index is large and must be stored in GPU memory. The index is highly dynamic, and new parallel algorithms must be developed to update the index in an efficient manner. Another challenge that we face is how to evenly assign computations to each thread, as the work associated with each document to be clustered is extremely variable. Finally, other challenges emerge when the entire corpus cannot fit into GPU main memory.

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 [12]. 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 any term not explicitly present in a particular document vector has a weight of zero. Document vectors are normalized. In addition, clusters are 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) and then are 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 [12]. The similarity between a cluster and a document is defined as the dot product between their term vectors.

The rest of the paper is organized as follows: Section 2 presents a sequential algorithm for online clustering. Section 3 describes a PRAM algorithm for par-allel online clustering one document at a time using a CRCW programming model. Section 4 presents a practical implementation of a parallel online clustering algorithm, which clusters mutiple documents at a time suitable for the CUDA parallel computing architecture [13]. Experimental results are presented in Section 5. Concluding remarks are provided in Section 6.

2 Sequential Clustering on the GPU

We first present a simple algorithm to cluster documents on the GPU one document a time. This algorithm also serves as a baseline for our main algorithm that will be presented later that can cluster multiple documents at the same time. The basic sequential 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.

For each document D (ranging from 0 to $n-1$)
Choose the cluster C most similar to D
if *similarity(C, D)* $> T$ **then**
  Add document D to cluster C
  Recompute C's term vector
else
  Create a new cluster consisting of only the document D
end

Algorithm 1: Sequential Clusterer 1 on the GPU

In the worst case, Algorithm 1 takes $O(n^2)$ dot products to cluster n documents as each document could end up forming its own cluster. However, the sparseness of document vectors means that very few number of distance computations are needed per document [2, 14]. Most document vectors have very few terms in common with other document vectors. Therefore, for each term in document vector D, we will have a limited number of clusters whose term vector contains a non-zero weight for that term. By keeping a list of clusters for each unique term seen by the clustering algorithm so far, we can 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: TermList$[t] = (C_1, C_1[t]), (C_2, C_2[t]), \cdots (C_p, C_p[t])$.

This indicates that cluster C_i contains a non-zero weight 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 C_i's term vector. We describe a sequential algorithm on the GPU in Algorithm 2 which makes use of the TermList data structure.

```
TermList ← Set of empty lists
for each document D (ranging from 0 to n − 1) do
    Candidates ← Empty Set
    Results ← Array of size D, initialized to all 0
    for each term t in D's term vector do
        for each (C_i, C_i[t]) in TermList[ t ] do
            Results[ C_i ] = Results[ C_i ] + C_i[t] * D[t]
            if Candidates does not contain C_i then
                Add C_i to Candidates
            end
        end
    end
    Choose the cluster C in Candidates with the max(Results[C] )
    if similarity(C, D) > T then
        for each term t in C's term vector do
            Remove C's entry (C, C[t]) from TermList[t]
        end
        Add document D to cluster C and recompute C's term vector
    else
        Create a new cluster C consisting of only the document D
    end
end
```

Algorithm 2: Sequential Clusterer 2 on the GPU

Note that we use D both to refer to the document and its term vector.

We calculate the approximate running time cost of Algorithm 2 as follows. 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 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, and the run-time of the algorithm is given by $O(n * K * L)$. We note that although L is highly dependent on the dataset, it is expected to be far less than n. The memory required for Algorithm 2 is $O(m * K)$, where m is the total number of clusters at the end of the algorithm.

3 Parallel Clustering of a 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 sequentially. Later we will discuss the case of processing multiple documents in parallel, and its effects on the clustering output.

Our goal is to parallelize as much of sequential clusterer's document loop as

```
TermList ← Set of empty lists
for each document D (ranging from 0 to n − 1) do
    Partials ← Array initialized to all 0
    Let t_1, t_2, ··· t_K be the terms in D's term vector
    S ← (t_1 × TermList[t_1]) ∪ ··· (t_K × TermList[t_K])
    for each (t_i, C_i, C[t_i]) in S parallel do
        Partials[ThreadID] = (C_i, D[t_i] ∗ C[t_i])
    end
    Run parallel sort on Partials, sorting by C_i
    Run parallel summation on Partials (adding similar C_i)
    Run parallel max on Partials to produce best candidate cluster C
    if similarity(C, D) > T then
        for each term t in C's term vector do
            Remove C's entry (C, C[t]) from TermList[t]
        end
        Add document D to cluster C and recompute C's term vector
    else
        Create a new cluster C consisting of only the document D
    end
    Add document terms in C to TermList
end
```

Algorithm 3: Parallel Clusterer Algorithm 1 on the GPU

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 (C_i, $C_i[t]$) swap in the last value of the TermList[t] to compact that list (assuming an array implementation). Inserting the new $(C_i, C_i[t])$ values can be done trivially by assigning K processors to add the new $(C_i, C_i[t])$ to the end of their respective lists.

We now present a high level parallel algorithm for clustering in Algorithm 3. We introduce the parallel keyword to indicate that the contents of a loop are performed in parallel. We also introduce a value ThreadID which is available to each thread within a parallel 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 use a PRAM architecture using the CRCW (Concurrent read-concurrent write) model [15] to analyze the run-time of parallel algorithms even though the GPU has a less restrictive computation model.

We can estimate the running time of Algorithm 3 as follows. For each document, we can compute the partial dot products in $O(1)$ time by using $K * L$

processors (we ignore the complication here of assigning ThreadIDs to processors). Parallel sort is known to be logarithmic [14], and so this takes $O(log(K*L))$ time. Parallel summation of Partials can be done in $O(log(K*L))$ time, and the parallel max operation also takes $O(log(K*L))$ time. Finally, the TermList maintenance operations take $O(1)$ time each. The running time for the algorithm is $O(n*log(K*L))$. The memory require for Algorithm 3 is again $O(m*K)$.

We now discuss the process of assigning ThreadIDs to processors for Algorithm 3'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 of TermList[t] will vary greatly when clustering a specific document, which complicates the ThreadID assignment. Our goal is to decide on a specific $(t_i, C_i, C[t_i])$ to associate with every ThreadID. This requires deciding on one specific element of each TermList[t] for each ThreadID. Let TermList[t][j] refer to the j-th element of term list for term t. Let size (TermList[t]) represent the number of elements currently in the term list for t.

Let $t_1, t_2 \cdots t_K$ be the terms in $D's$ term vector
TermSizes $\leftarrow size(TermList[t_1]) \cdots size(TermList[t_K])$
PrefixSums $\leftarrow$ the prefix sums of TermSizes
Binary search on PrefixSums to find the smallest u s.t., ThreadID $<$ PrefixSums[u]
C = TermList[t_u][PrefixSums[u] - ThreadID - 1]
Partials[ThreadID] = (C, D[t_u] * C[t_u])

Algorithm 4: Thread assignment

We note that PrefixSums[$i-1$] indicates how many threads should be assigned to term lists 1 up to $i-1$. 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] in which we are interested. Each binary search using Algorithm 4 takes $O(log(K))$ time. Binary searches over global memory arrays can be inefficient. The performance can be improved by using an additional temporary array and another PrefixSum, which has much better locality and therefore processes data faster. While the complexity of PrefixSum is O(log n), it does not change the overall running time of Algorithm 3, since it is dominated by the cost of sorting. Finally, we note that the parallel deletion that occurs in Algorithm 3 requires an identical ThreadID configuration as the partial dot products. Each deletion thread will receive a unique ThreadID, and must decide which TermList entry to examine. We can use Algorithm 4 where $t_1, t_2 \cdots t_K$ are the terms in C's term vector, instead of D's. Again, the overall running time is unchanged.

4 Clustering Multiple Documents in Parallel

In this section we examine an algorithm for clustering multiple documents in parallel. Assume that we wish to cluster Q documents in parallel. We define the multiple document clustering algorithm below as Algorithm 5.

The main difference between the multiple document and single document

```
while we have more documents to cluster do
    Choose the next Q documents D_1, D_2, ··· D_Q
    Choose clusters C_1, C_2, ··· C_Q such that C_i is the most similar cluster to D_i
    for i = 1 to Q do
        if similarity(C_i, D_i ) > T then
            Add document D_i to cluster C_i and recompute C_i's term vector
        end
    end
end
```

Algorithm 5: Multiple Document Clusterer 1

algorithms is that we assign the best clusters to Q documents before updating the cluster term vectors and the index. This can lead to poor clustering in some cases, since, for example, document D_i is never compared against the effects of $D_1, D_2 \cdots D_{i-1}$. Merging similar clusters at varying points in the algorithm can possibly mitigate this effect. We assume that the effects of this problem are minimal as long as Q is much less than n. We wish to extend Algorithm 3 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 ThreadIDs for multiple documents now requires reasoning about to which document a thread belongs. This results in a binary search of a prefix sums array of size $K * Q$ for each thread to assign work. Assume that t_{ij} refers to the j-th term of document D_i (the j-th term of the i-th document that we are clustering in parallel). Note that each entry contained in Partials now contains an extra element, which indicates the document to which the partial dot product belongs. Algorithm 6 guarantees however that similar D_q values will be contiguous within Partials. This means that we can sort Q separate sub-lists in parallel (each of size roughly $K * L$).

```
TermSizes ← size(TermList[t_11]) ··· size(TermList[t_1K]),
            size(TermList[t_Q1]) ··· size(TermList[t_QK])
PrefixSums ← prefix sums of TermSizes
Binary search to identify smallest u s.t. PrefixSums ← prefix sums of TermSizes
Binary search to identify smallest u s.t. ThreadID < PrefixSums[u]
q ← u/K
r ← u % K
C = TermList[tqr][PrefixSums[u] - ThreadID - 1 ]
Partials[ThreadID] = (D_q, C, D[tqr] * C[tqr])
```

Algorithm 6: ThreadID Assignment 3

Once, we haven chosen the appropriate clusters $C_1, C_2 \cdots C_Q$, we must update the Term-List data structure to reflect the changes of the Q cluster term vectors. We cannot simply perform 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 use the atomic addition operator available in CUDA, while acknowledging that their frequent use can result in

performance degradation. Parallel insertion of a term t, and an element to insert x is given by: size = atomicAdd(TermList[t].size, 1) followed by 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 a little more involved and is given below as Algorithm 7.

```
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 parallel do
    TermList[t][i].deleted = FALSE
    if TermList[t][i] == x then
        TermList[t][i].deleted = TRUE
        if i ≥ TermList[t].size - TermList[t].deleteNumber then
            Return
        end
        priority = atomicAdd(TermList[t][i].deletePriority, -1 )
        numSkip = TermList[t].deleteNumber - TermList[t].priority
        j = elements from end of TermList[t] s.t., TermList[t].deleted is FALSE
        TermList[t][i] = TermList[t][x]
    end
    TermList[t].size = TermList[t].newSize
end
```

Algorithm 7: AtomicDeletion(t, x)

The basic idea behind Algorithm 7 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 C_i values that are duplicates prior to running the AtomicDeletion, as the result of including them is the same as that when we don't include them.

Due to space limitations, the details of the Multiple Document Clusterer 2 are provided in [16]. The running time of this algorithm is given by $O((n/Q) * \max(log(K*L), Q))$, while the memory requirement is $O(\max(m*K), K*L*Q)$.

5 Experimental Results

First, we evaluated the performance of Algorithm 2 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 result is shown in Figure 1a. The documents are ordered by the 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 tested our implementation on a GeForce GTX 280, which has 240 cores and 1 GB of global memory. The CPU was an AMD 3GHz processor with 4 cores. It can be seen that the algorithm takes about 50 seconds to cluster 250k documents.

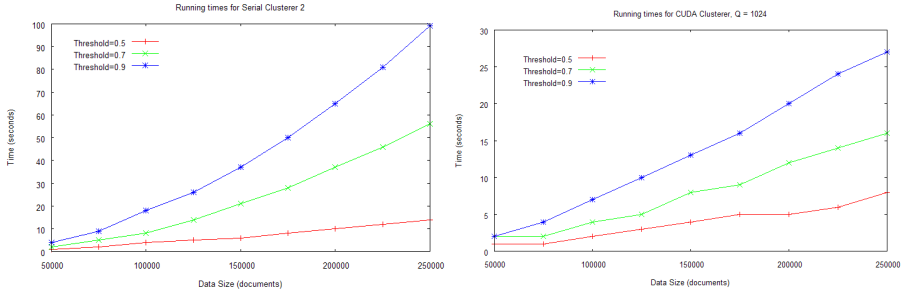

Fig. 1. Running time of a) Sequential Algorithm 2 b) Multiple Clusterer 2 for different thresholds and data sizes

Next, we performed clustering of more than one document at a time using the Multiple Document Clusterer 2 algorithm. Figure 1b is the result for $Q = 1024$ (1024 documents done in parallel). It can be seen that the algorithm takes only 15 seconds to cluster 250k documents. In contrast the Sequential Clusterer on the GPU takes only 50 seconds denoting a 3X speed up by performing clustering in parallel. Furthermore, 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).

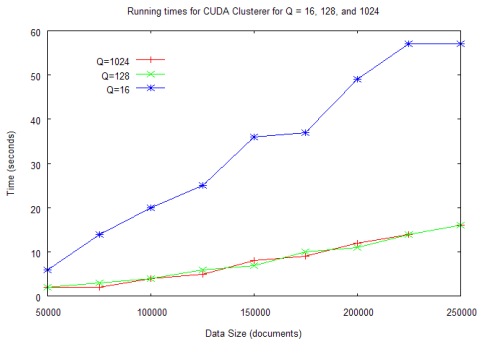

Fig. 2. Running time of Multiple Clusterer 2 for different values of Q

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 Figure 2 that there is a significant performance improvement in increasing the value of Q from 16 to 128. However, increasing the value of Q more does not result in significant reduction of running time. This indicates that the GPU's

threads have saturated when the number of documents is above 16. Note however that setting a large value of Q does not seem to have a detrimental effect on the running time of the algorithm.

6 Concluding Remarks

In this paper we have described a parallel algorithm for online document clustering. We have shown that 3X speedups can be achieved when clustering multiple documents at the same time instead of one at a time. Future work will focus on incorporating the algorithm into our NewsStand and TwitterStand production systems and developing a variant of the algorithm that makes limited use of atomic operations.

References

1. Hjaltason, G.R., Samet, H.: Speeding up construction of PMR quadtree-based spatial indexes. VLDB Journal 11(2), 109–137 (Oct 2002).
2. Lieberman, M.D., Sankaranarayanan, J., Samet, H.: A fast similarity join algorithm using graphics processing units. In: IEEE ICDE, 1111–1120, Apr 2008.
3. Samet, H.: K-nearest neighbor finding using MaxNearestDist. IEEE TPAMI 30(2):243–252 (Feb 2008).
4. Samet, H., Alborzi, H., Brabec, F., Esperança, C., Hjaltason, G.R., Morgan, F., Tanin, E.: Use of the SAND spatial browser for digital government applications. CACM 46(1):63–66 (Jan 2003).
5. Samet, H., Rosenfeld, A., Shaffer, C.A., Webber, R.E.: A geographic information system using quadtrees. Pattern Recognition 17(6):647–656 (Nov 1984)
6. Samet, H., Tamminen, M.: Bintrees, CSG trees, and time. Computer Graphics 19(3):121–130 (Jul 1985), also in *SIGGRAPH*, 1985
7. Sankaranarayanan, J., Alborzi, H., Samet, H.: Efficient query processing on spatial networks. In: ACM GIS, 200–209, Nov 2005.
8. Sankaranarayanan J., Samet H., Teitler B. E., Lieberman M. D. and Sperling J.: Twitterstand: News in tweets. In ACM GIS, 42–51, Nov 2009.
9. Sankaranarayanan, J., Samet, H., Varshney, A.: A fast all nearest neighbor algorithm for applications involving large point-clouds. Computers & Graphics 31(2):157–174 (Apr 2007)
10. Tanin, E., Harwood, A., Samet, H.: A distributed quadtree index for peer-to-peer settings. In: IEEE ICDE, 254–255, Apr 2005.
11. Teitler B.E., Lieberman M.D., Panozzo D., Sankaranarayanan J., Samet H., Sperling J., NewsStand: A new view on news. In ACM GIS, 144–153, Nov 2008.
12. Salton G. and Buckley C.: Term-weighting approaches in automatic text retrieval. Information Processing & Management, 24(5):513-523 (1988).
13. Sanders J. and Kandrot E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, Reading, MA, 2011.
14. Weber R., Schek H.-J., and Blott S.: A quantitative analysis and performance study for similarity-search methods in high dimensional spaces. In VLDB, 194-205, August 1998.
15. Vishkin, U.: Thinking in parallel: Some basic data-parallel algorithms and techniques. College Park, MD (2007).
16. Teitler B. E, Sankaranarayanan J., and Samet H.: Online document clustering using the GPU. CS-TR 4970, UMD, College Park, MD, August 2010.