Data-Intensive Document Clustering on GPU Clusters[☆,☆☆]

Yongpeng Zhang[a], Frank Mueller[a,*], Xiaohui Cui[b], Thomas Potok[b]

[a]*Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-7534*
[b]*Oak Ridge National Laboratory, Computational Sciences and Engineering Division, Oak Ridge, TN 37831*

Abstract

Document clustering is a central method to mine massive amounts of data. Due to the explosion of raw documents generated on the Internet and the necessity to analyze them efficiently in various intelligent information systems, clustering techniques have reached their limitations on single processors. Instead of single processors, general-purpose multi-core chips are increasingly deployed in response to diminishing returns in single processor speedup due to the frequency wall, but multi-core benefits only provide linear speedups while the number of documents in the Internet grows exponentially. Accelerating hardware devices represent a novel promise for improving the performance for data-intensive problems such as document clustering. They offer more radical designs with a higher level of parallelism but adaptation to novel programming environments.

In this paper, we assess the benefits of exploiting the computational power of Graphics Processing Units (GPUs) to study two fundamental problems in document mining, namely TF-IDF (Term Frequency-Inverse Document Frequency) and document clustering. We transform traditional algorithms into accelerated parallel counterparts that can be efficiently executed on many-core GPU architectures. We assess our implementations on various platforms ranging from stand-alone GPU desktops to Beowulf-like clusters equipped with contemporary GPU cards. We observe at least one order of magnitude speedups over CPU-only desktops and clusters. This demonstrates the potential of exploiting GPU clusters to efficiently solve massive document mining problems. Such speedups combined with the scalability potential and accelerator-based parallelization are unique in the domain of document-based data mining, to the best of our knowledge.

Keywords: High-Performance Computing, Accelerators, Data-Intensive Computing

[☆]This work was supported in part by NSF grant CCF-0429653, CCR-0237570 and a subcontract from ORNL. The research at ORNL was partially funded by Lockheed ShareVision research funds and Oak Ridge National Laboratory Seed Money funds.

[☆☆]An earlier version of this paper appeared at IPDPS'10 [1]. This journal version extends the earlier paper by a complete algorithmic study of the pre-processing step (the TF-IDF vector calculation), novel parallel algorithm re-design for GPU architecture, extra experimental results on single node desktop and extended related work.

[*]Principle corresponding author

Email addresses: mueller@cs.ncsu.edu (Frank Mueller), cuix@ornl.gov (Xiaohui Cui)

1. Introduction

Document clustering, or text clustering, is a sub-field of data clustering where a collection of documents are categorized into different subsets with respect to document similarity. Such clustering occurs without supervised information, *i.e.*, no prior knowledge of the number of resulting subsets or the size of each subset is required. Clustering analysis in general is motivated by the explosion of information accumulated in today's Internet, *i.e.*, accurate and efficient analysis of millions of documents is required within a reasonable amount of time. A recent flocking-based algorithm [2] implements the clustering process through the simulation of mixed-species birds in nature. In this algorithm, each document is represented as a point in a two-dimensional Cartesian space. Initially set at a random coordinate, each point interacts with its neighbors according to a clustering criterion, *i.e.*, typically the similarity metric between documents. This algorithm is particularly suitable for dynamical streaming data and is able to achieve global optima, much in contrast to our algorithmic solutions [3].

In this research, we first solve one of the fundamental problems in document mining, namely that of calculating TF-IDF vectors of documents. The TF-IDF vector is subsequently utilized to quantify document similarity in document clustering algorithms. In this work, we show how to re-design the traditional algorithm into a CPU-GPU co-processing framework and we demonstrate up to 10X speedup over a single-node CPU desktop.

In a second step, we aim at clustering at least one million documents at a time based on the TF-IDF-like similarity metric. In document clustering, the size of each document varies and can reach up to several kilo-bytes. Therefore, document clustering imposes an even higher pressure on memory usage than traditional data mining, where data set is of much smaller and constant size. Unfortunately, many accelerators, including GPUs, do not share memory with their host systems, nor do they provide virtual memory addressing. Hence, there is no means to automatically transfer data between GPU memory and host main memory. Instead, such memory transfers have to be invoked explicitly. The overhead of these memory transfers, even when supported by DMA, can nullify the performance benefits of execution on accelerators. Hence, a thorough design to assure well-balanced computation on accelerators and communication / memory transfer to and from the host computer is required, *i.e.*, overlap of data movement and computation is imperative for effective accelerator utilization. Moreover, the inherently quadratic computational complexity in the number of documents and the large memory footprints, however, make efficient implementation of flocking for document clustering a challenging task. Yet, the parallel nature of such a model bears the promise to exploit advances in data-parallel accelerators for distributed simulation of flocking.

As a result, we investigate the potential to purse our goal on a cluster of computers equipped with NVIDIA CUDA-enabled GPUs. We are able to cluster one million documents over sixteen NVIDIA GeForce GTX 280 cards with 1GB on-board memory each. Our implementation demonstrates its capability for weak scaling, *i.e.*, execution time remains constant as the amount of documents is increased at the same rate as GPUs are added to the processing cluster. We have also developed a functionally equivalent multi-threaded MPI application in C++ for performance comparison. The GPU cluster implementation shows dramatic speedups over the C++ implementation, ranging from 30X to more than 50X speedups.

The contributions of this work are the following:

- We design highly parallelized methods to build hash tables on GPU as a premise to calculate TF-IDF vectors for a given set of documents.
- We apply multiple-species flocking (MSF) simulation in the context of large-scale document clustering on *GPU clusters*. We show that the high I/O and computational throughput in such a cluster meets the demanding computational and I/O requirements.
- In contrast to previous work that targeted GPU clusters [4, 5], our work is one of the first to utilize CUDA-enabled GPU clusters to accelerate *massive data mining applications*, to the best of our knowledge.
- The solid speedups observed in our experiments are reported *over the entire application* (and not just by comparing kernels without considering data transfer overhead to/from accelerator). They clearly demonstrate the potential for this application domain to benefit from acceleration by GPU clusters.

The rest of the paper is organized as follows. We begin with the background description in Section 2. The design and implementation of TF-IDF calculation and document clustering are presented in Section 3 and 4, respectively. In Section 5, we show various speedups of GPU clusters against CPU clusters in different configurations. Related work is discussed in Section 6 and a summary is given in Section 7.

2. Background Description

In this section, we describe the algorithmic steps of TF-IDF and document clustering, and discuss details of the target programming environments.

2.1. TF-IDF

Term frequency (TF) is a measure of how important a term is to a document. The ith term's tf in document j is defined as:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{1}$$

where $n_{i,j}$ is the number of occurrences of the term in document d_j and the denominator is the number of occurrences of all terms in document d_j.

The inverse document frequency (IDF) measures the general importance of the term in a corpus of documents. This is done by dividing the number of all documents by the number of documents containing the term and then taking the logarithm.

$$idf_i = log\frac{|D|}{|\{d_j : t_i \in d_j\}|} \tag{2}$$

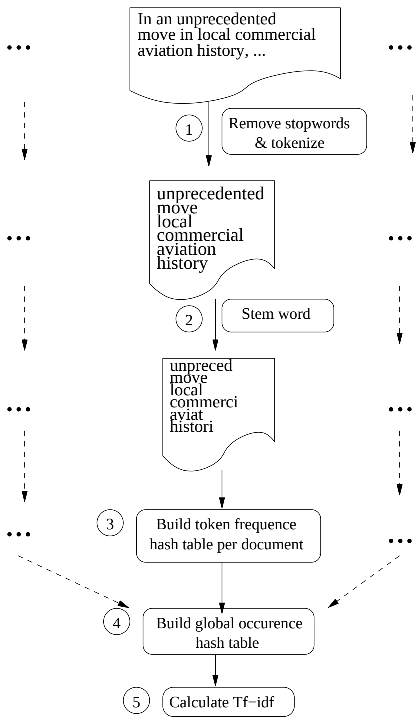

In an unprecedented move in local commercial aviation history, ...
1
Remove stopwords & tokenize
unprecedented
move
local
commercial
aviation
history
2
Stem word
unpreced
move
local
commerci
aviat
histori
3
Build token frequence hash table per document
4
Build global occurence hash table
5
Calculate Tf−idf

Figure 1: TF-IDF Workflow

where $|D|$ is the total number of documents in the corpus and $|\{d_j : t_i \in d_j\}|$ is the number of documents containing term t_i.

Then, the TF-IDF value of the ith term in document j is:

$$tfidf_{i,j} = tf_{i,j} * idf_i \tag{3}$$

The idea of TF-IDF can be extended to compare the similarities of two documents d_i and d_j. One of the simple way is to apply the similarity metric between any pair of documents i and j:

$$Sim_{i,j} = \sum_k |tfidf_{k,i} - tfidf_{k,j}|^2 \tag{4}$$

for k over all terms of both document i and j. Obviously, the smaller the value is, the more similar these two documents are considered.

There are many ways to calculate the TF-IDF given a corpus of documents. The most straightforward method, also used by us, is illustrated in Figure 1. The first step, which is part of the document preprocessing prior to the core TF-IDF calculation, excerpts and tokenizes each word of a document. It is also in this step that the stop words are removed. Stop words, also known as the noise words, are common words that do not contribute to the uniqueness of the document [6]. In the second step, some cognate words are transformed into one form by applying certain stemming patterns for each. This is necessary to obtain results with higher precision [7]. In step three, the document hash table is built for each document. The <key, value> pairs in the token hash table are the unique words that appear in the document and their occurrence frequencies, respectively. In step four, all of these token hash tables are reduced into one global occurrence table in which the keys remain the same, but values represent the number of documents that contain the associated key. The TF-IDF for each term can be easily calculated by looking up the corresponding values in the hash tables according to Equation 3 as seen in step five.

2.2. Flocking-based document clustering

The goal of document clustering is to form groups of individuals that share certain criteria. Document similarity derived from TF-IDF provides the foundation to determine such similarities. In flocking-based clustering, the behavior of a boid (individual) is based only on its neighbor flock mates within a certain range. Reynolds [8] describes this behavior in a set of three rules. Let $\vec{p_j}$ and $\vec{v_j}$ be the position and velocity of boid j. Given a boid noted as x, suppose we have determined N of its neighbors within radius r. The description and calculation of the force by each rule is summarized as follows:

- *Separation*: steer to avoid crowding local flock mates

$$\vec{f_{sep}} = -\sum_{i}^{N} \frac{\vec{p_x} - \vec{p_i}}{r_{i,x}^2} \tag{5}$$

  where $r_{i,x}$ is the distance between two boids i and x.

- *Alignment*: steer towards the average heading of local flock mates

$$\vec{f_{ali}} = \frac{\sum_i^N \vec{v_i}}{N} - \vec{v_x} \tag{6}$$

- *Cohesion*: steer to move toward the average position of local flock mates

$$\vec{f_{coh}} = \frac{\sum_i^N \vec{p_i}}{N} - \vec{p_x} \tag{7}$$

The three forces are combined to change the current velocity of the boid. In case of document clustering, we map each document as a boid that participates in flocking formation. For similar neighbor documents, all three forces are combined. For non-similar neighbor documents, only the *separation* force is applied.

2.3. GPU and CUDA

Graphics programming units (GPUs) differ from general-purpose microprocessors in their design for the single instruction multiple data (SIMD) paradigm. Due to the inherent parallelism of vertex shading, GPUs have adopted multi-core architectures long before regular microprocessors resort to such a design. While this decision is driven by increasing demands for faster and more realistic graphics effects in the former case, it is dictated by power and asymptotic single-core frequency limits for the latter. As a result, today's state-of-the-art GPUs consist of many small computation cores compared to few large cores in off-the-shelve CPUs, at the cost of devoting less die area for flow control and data caching in each core. Since graphics is a niche, albeit a very influential one, that drives the progress in GPU architectures, much attention has been paid to fast and independent vertex rendering. The computational rendering engines of GPUs can generally be utilized for other problem domains as well, but their effectiveness depends much on the suitability of numerical algorithms within the target domain for GPUs.

In recent years, GPUs have attracted more and more developers who strive to combine high performance, lower cost and reduced power consumption as an inexpensive means for solving complex problems. This trend is expedited by the emergence of increasingly user-friendly programming models, such as NVIDIA's CUDA, AMD's Stream SDK and OpenCL. Our focus lies on the former of these models.

CUDA is a C-like language that allows programmer to execute programs on NVIDIA GPUs by utilizing their streaming processors. The core difference between CUDA programming and general-purpose programming is the capability and necessity to spawn massive number of threads. Threads are grouped into *warps* as basic thread scheduling units [9]. The same code is executed by threads in the same *warp* on a given streaming processor. As these GPUs do not provide caches, memory latencies are hidden through several techniques: (a) Each streaming processor contains a small but fast on-chip shared memory that is exposed to programmers. (b) Large register files enable instant hardware context switch between *warps*. This facilitates the overlapping of data manipulation and memory access. (c) Off-chip global memory accesses issued simultaneously by multi-threads can be accelerated by coalesced memory access, which requires aligned access pattern for consecutive threads in *warps*.

In this work, the massive throughput offered by GPUs is the major source of speedup over conventional desktops.

2.4. MPI

The document flocking algorithm is not an embarrassingly parallel algorithm as it requires exchange of data between nodes. We utilize MPI as a means to exchange data between nodes. MPI is the dominant programming model in the high-performance computation domain. It provides message passing utilities with a transparent interface to communicate between distributed processes without considering the underlying network configurations. It is also the *de factor* industrial standard for message passing that offers maximal portability. In this work, we incorporate MPI as the basic means to communicate data between distributed computation nodes. We also combine MPI communication with data transfers between host memory and GPU memory to provide a unified distributed object interface that will be discussed later.

3. Design and Implementation of TF-IDF Calculation

One of the key challenges in algorithmic design for GPGPUs is to keep all processing elements busy. NVIDIA's philosophy to ensure high utilization is to oversubscribe,

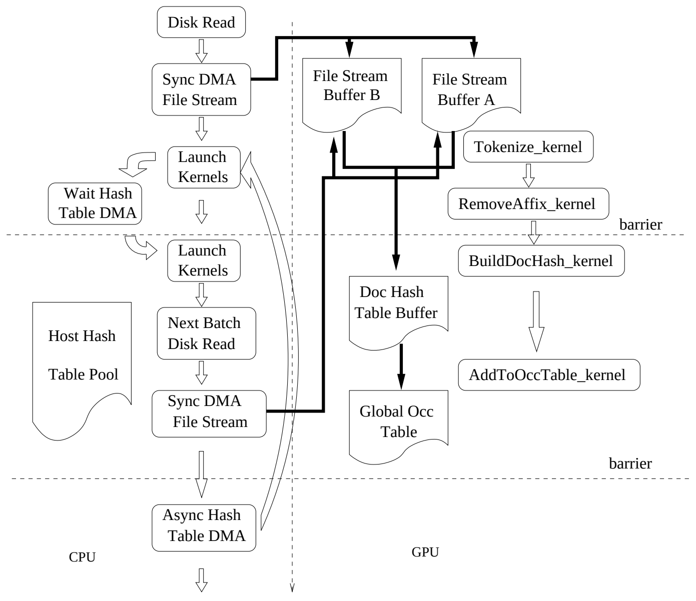


Figure 2: CPU/GPU Collaboration Framework

i.e., more parallel work is dispatched than there are physical stream processors available. Using latency-hiding techniques, a processor stalled on a memory reference can thus simply switch context to another dispatched work unit.

In order to fully utilize the large number of streaming processors in NVIDIA's GPUs, we process files in batches with the batch size chosen as 96. Several kernels are developed to implement the steps described in Section 2.1. Each batch process requires extensive data movement between host and GPU memories by DMA. First, to handle a large amount of documents/files, especially when total document size is larger than the GPU global memory, the document hash tables needs to be flushed out to host memory once they are completely constructed. Second, the raw data of a document is pushed from host memory to GPU global memory at the beginning of each batch process. To reduce the overhead of memory movement, we developed the CPU/GPU collaboration framework shown in Figure 2.

In each batch iteration, the CPU thread first launches the two preprocessing kernels (Tokenize_kernel and RemoveAffix_kernel) asynchronously. Before invoking the next kernels (BuildDocHash_kernel and AddToOccTable_kernel) that write to the document and global occurrence hash table buffers in the GPU's global memory, it waits for the completion signal of the previous batch's DMA that transfers the old batch table to host memory. When the GPU is busy generating the document hash tables and inserting tokens into the global occurrence table, the CPU can prefetch the next batch of files from disk and copy them to an alternate file stream buffer. At the end of the batch iteration, the CPU again asynchronously issues a memory copy of the document hash table to the host's memory. Only in the next batch's iteration will the completion of this DMA be

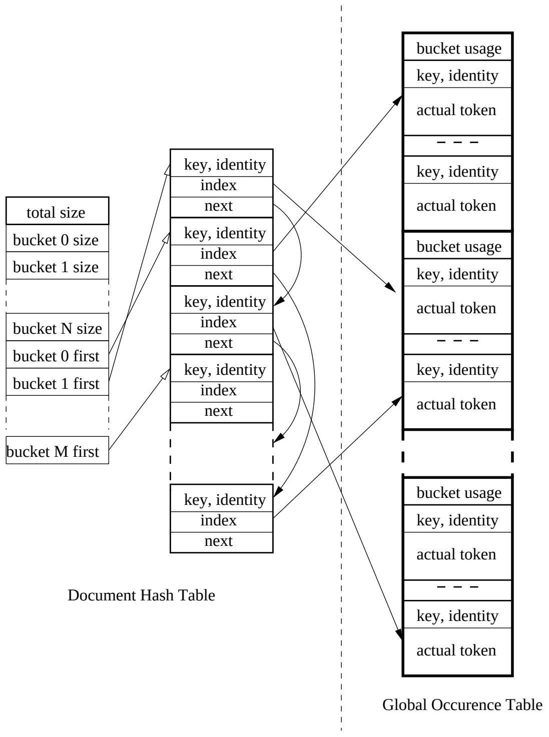


Figure 3: Hash Table Data Structures

synchronized. In this manner, part of the DMA time is overlapped with the GPU calculation by (a) double buffering the document raw data in GPU and (b) overlapping the hash table memory copy in the current batch with the stream preprocessing (tokenize and stem kernels) of the next batch [10].

To further reduce the DMA overhead, one may reduce the size of the document hash table. This table differs from the global occurrence table, which resides in GPU global memory but need not be copied to host until the end of execution. Therefore, the data structures of these tables differ slightly as shown in Figure 3. The document hash table contains a header and an array of entries, which are internally linked as a list if they belong to the same bucket. The header is used to determine the bucket size and to find the first entry in each bucket. In contrast, the global hash table consists of a big array of entries evenly divided into buckets. Because the number of unique terms is considered limited no matter how large the corpus size is, the number of buckets and the bucket size can be chosen sufficiently large to avoid possible bucket overflows.

Another effort to reduce the size of the document hash table avoids storing the actual term/word in the table. Instead, every entry simply maintains an index pointing to the corresponding entry in the global occurrence table where the actual term is saved. To reduce the number of hash key computations at hash insertion and during hash

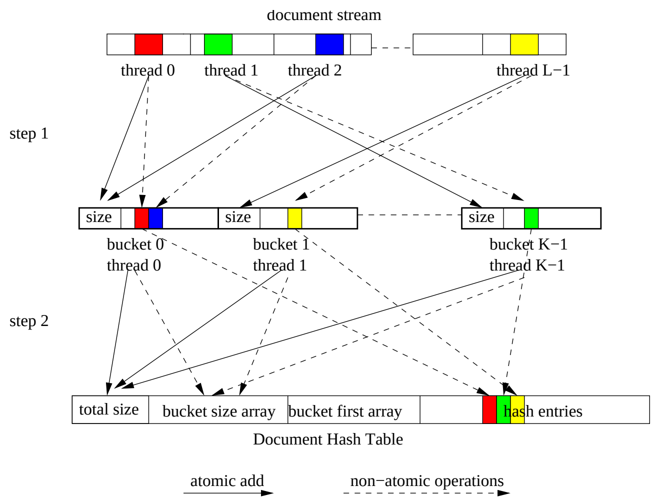


Figure 4: Building a Hash Table with Atomic Operations

searches, the key is saved as an "unsigned long" in both hash tables. To further reduce the probability of hash collisions (two terms sharing the same key), another field called identity is added as an "unsigned int" to help differentiate terms. The identity is then constructed as $(term\ length << 16)|(first\ char << 8)|(last\ char)$.

Upon investigation, we determined that atomic operations supported by certain GPUs via CUDA are facilitating the construction of a concise document hash table without adversely affecting the parallelism of the algorithm. We alternatively provide another method to generate the same hash table for GPUs without support for atomic operations. Even though the latter method is slower than the first, it is required for GPU devices that do not have atomic operation support (*i.e.*, devices with CUDA compute capability 1.0 or earlier).

3.1. Hash Table Updates using Atomic Operations

Access to hash table entries *via* atomic operations is realized in two steps as depicted in Figure 4. In the first step, the document stream is evenly distributed to a set of CUDA threads. The number of threads, L, is chosen explicitly to maximize GPU's utilization. A buffer storing the intermediate hash table, which is close to the structural layout of the global occurrence table, but with a smaller number of buckets K, is used to sort terms by their bucket IDs. Every time a thread encounters a new term in the stream and obtains its bucket ID, it issues an atomic increment (atomic-add-one) operation to affect the bucket size. (Notice that the objective of this algorithmic TF-IDF variant is not to identify identical terms. Instead, its chief objective is to compute a similarity metric.) If we assume that terms are distributed randomly, then contention during the atomic increment operation is the exception, *i.e.*, threads of the same warp are likely atomically incrementing disjoint bucket size entries.

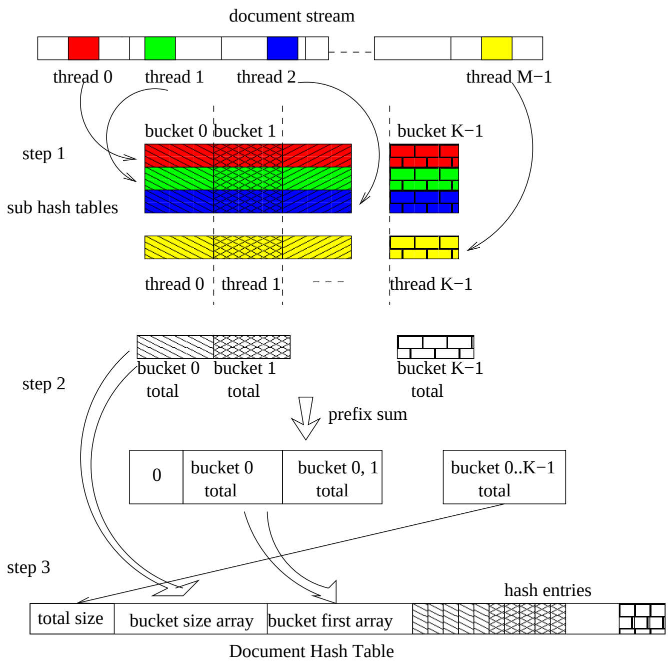


Figure 5: Building a Hash Table without Atomic Operation

In the next step, the intermediate hash table is reduced to the final, more concise document hash table shown in Figure 3. Each CUDA thread traverses one bucket in the intermediate hash table, detects duplicate terms, and, if finds a new term, reserves a place in the entry array by atomically incrementing the total size. It then pushes the new entry into the header of the linked bucket list. Since different threads operate on disjoint buckets, each linked list per bucket is accessed in mutual exclusion, which guarantees absence of write conflicts between threads.

3.2. Hash Table Updates without Atomic Operations

In GPUs without atomic instruction support, the document stream is first split into M packets, each of which is pushed into a different hash sub-table owned by one thread in a block, as shown in step 1 of Figure 5. By giving each thread a separate hash sub-table, we guarantee write protection (mutually exclusive writes of the values) between threads. In step 2, K threads are re-assigned to different buckets of the sub-table, identical terms are found in this step, and statistics for each bucket are generated. Because terms have been grouped by their keys in step 1, there will be no write conflicts between threads at this step either. The bucket size information is processed in step 3 to finally merge sub-tables to compose the final document hash table.

3.3. Discussions

The two procedures detailed above to handle hash tokens in a document do not require information from any other documents. Thus, each document can be processed simultaneously and independently in different GPU blocks. With a sufficiently large number of documents, we can fully utilize the GPU cores and exploit NVIDIA's latency hiding on memory references through oversubscription. However, in the first step of the second method, the number of packets M per document is delimited due to memory constraint and the efficiency of the following steps. We choose a value of $M = 16$ in our implementation. To compensate for this constraint, we can spawn more threads L in the first method, *e.g.*, by choosing $L = 512$. This constraint on parallelism results in a non-atomic approach that is slower than its atomic variant.

From the memory usage's perspective, the non-atomic approach consumes more global memory simply because the intermediate hash tables in the non-atomic approach are larger than that in the atomic approach. Both of the above methods cannot handle very large single documents that exceed the size of the global memory. Since our problem domain is that of Internet news articles, which typically do not exceed more than 10K words, documents fits in memory for our implementation. This framework is even suitable for arbitrarily large corpus sizes as we could reused without changes both intermediate hash tables and the document hash table, the latter of which is flushed to host memory for each batch of files.

4. Design and Implementation of Document Clustering

4.1. Programming Model for Data-parallel Clusters

We have developed a programming model targeted at message passing for CUDA-enabled nodes. The environment is motivated by two problems that surface when explicitly programming with MPI and CUDA abstraction in combination:

- Hierarchical memory allocation and management have to be performed manually, which often burdens programmers.
- Sharing one GPU card among multiple CPU threads can improve the GPU utilization rate. However, explicit multi-threaded programming not only complicates the code, but may also result in inflexible designs, increased complexity and potentially more programming pitfalls in terms of correctness and efficiency.

To address these problems, we have devised a programming model that abstracts from CPU/GPU co-processing and mitigates the burden of the programmer to explicitly program data movement across nodes, host memories and device memories. We next provide a brief summary of the key contributions of our programming model (see [11] for a more detailed assessment):

- We have designed a *distributed object interface* to unify CUDA memory management and explicit message passing routines. The interface enforces programmers to view the application from a data-centric perspective instead of a task-centric view. To fully exploit the performance potential of GPUs, the underlying run-time system can detect data sharing within the same GPU. Therefore, the network pressure can be reduced.

- Our model provides the means to spawn a flexible number of host threads for parallelization that may *exceed* the number of GPUs in the system. Multiple host threads can be automatically assigned to the same MPI process. They subsequently share one GPU device, which may result in higher utilization rate than single-threaded host control of a GPU. In applications where CPUs and GPUs co-process a task and a CPU cannot continuously feed enough work to a GPU, this sharing mechanism utilizes GPU resources more efficiently.

- An interface for advanced users to control thread scheduling in clusters is provided. This interface is motivated by the fact that the mapping of multiple threads to physical nodes affects performance depending on the application's communication patterns. Predefined communication patterns can simply be selected so that communication endpoints are automatically generated. More complex patterns can be supported through reusable plug-ins as an extensible means for communication.

We have designed and implemented the flocking-based document clustering algorithm in GPU clusters based on this GPU cluster programming model. In the following, we discuss several application-specific issues that arise in our design and implementation.

4.2. Preprocessing

The prerequisite of document clustering is to have a standard means to measure similarities between any two documents. While the TF-IDF concept exactly matches this need, there are two practical issues when targeting clusters:

- There is a reduce step (step 4 in Figure 1) to generate a single global occurrence hash table. This is a high payload all-to-all communication in clusters and thus is not scalable.

- The TF-IDF calculation cannot start until all documents have been processed and inserted in the global occurrence table. Therefore, it is not suited for stream processing.

A new term weighting scheme called term frequency-inverse corpus frequency (TF-ICF) has been proposed to solve the above problems at the scale of massive amounts of documents [12]. It does not require term frequency information from other documents within the processed document collections. Instead, it pre-builds the ICF table by sampling a large amount of existing literature off-line. Selection of corpus documents for this training set is critical as similarities between documents of a later test set are only reliable if both training and test sets share a common base dictionary of terms (words) with a similar frequency distribution of terms over documents. Once the ICF table is constructed, ICF values can be looked up very efficiently for each term in documents while TF-IDF would require dynamic calculation of these values. The TF-ICF approach enables us thus to generate document vectors in linear time.

4.3. Flocking Space Partition

The core of the flocking simulation is the task of neighborhood detection. A sequential implementation of the detection algorithm has $O(N^2)$ complexity due to pairwise checking of N documents. This simplistic design can be improved through space filtering, which prunes the search space for pairs of points whose distances exceed a threshold.

One way to split the work into different computational resource is to assign a fixed number of documents to each available node. Suppose there are N documents and P nodes. In every iteration of the neighborhood detection algorithm, the positions of local documents are broadcast to all other nodes. Such partitioning results in a lower communication overhead proportional to the number of nodes, and the detection complexity is reduced linearly by P per node for a resulting overhead of $O(N^2/P)$.

Instead of partitioning the documents in this manner, we break the virtual simulation space into row-wise slices. Each node handles just those documents located in the current slice. Broadcast messages that are previously required are replaced by point-to-point messages in this case. This partitioning is illustrated in Figure 6. After document positions are updated in each iteration, additional steps are performed to divide all documents into three categories. *Migrating documents* are those that have moved to a neighbor slice. *Neighbor documents* are those that are on the margin of the current slice. In other words, they are within the range of the radius r of neighbor slices. All other are *internal documents* in the sense that they do not have any effects on the documents in other nodes. Since the velocity of documents is capped by a maximal value, it is impossible for the migrating documents to cross an entire slice in one timestep. Both the migrating documents and neighbor documents are transferred to neighbor slices at the beginning of the next iteration. Since the neighborhood radius r is much smaller than the virtual space's dimension, the number of migrating documents and neighbor documents are expected to be much smaller than that of the internal documents.

Sliced space partitioning not only splits the work nearly evenly among computing nodes but also reduces the algorithmic complexity in sequential programs. Neighborhood checks across different nodes are only required for neighbor documents within the boundaries, not for internal documents. Therefore, on average, the detection complexity on each node reduces to $O(N^2/P^2)$ for slides partitioning, which is superior to traditional partitioning with $O(N^2/P)$.

4.4. Document Vectors

An additional benefit of MSF simulation is the similarity calculation between two neighbor documents. Similarities could be pre-calculated between all pairs and stored in a triangular matrix. However, this is infeasible for very large N because of a space complexity of $O(N^2/2)$, which dauntingly exceeds the address space of any node as N approaches a million. Furthermore, devising an efficient partition scheme to store the matrix among nodes is difficult due to the randomness of similarity look-ups between any pair of nearby documents. Therefore, we devote one kernel function to calculating similarities in each iteration. This results in some duplicated computations, but this method tends to minimize the memory pressure per node.

The data required to calculate similarities is a document vector consisting of an index of each unique word in the TF-ICF table and its associated TF-ICF values. To

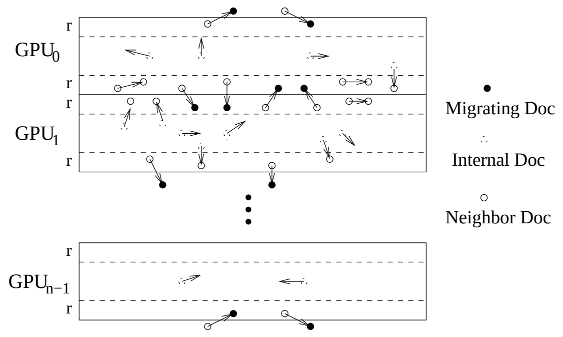


Figure 6: Simulation Space Partition

compute the similarity between two documents, as shown in Equation (4), we need a fast method to determine if a document contains a word given the word's TF-ICF index. Moreover, the fact that we need to move the document vector between neighbor nodes also requires that the size of the vector should be kept small.

The approach we take is to store document vectors in an array sorted by the index of each unique word in the TF-ICF table. This data structure combines the minimal memory usage with a fast parallel searching algorithm. Riech [13] describes an efficient algorithm to calculate the cosine similarities between any two sorted arrays. But this algorithm is iterative in nature and not suitable for parallel processing.

We develop an efficient CUDA kernel to calculate the similarity of two documents given their sorted document vectors as shown in Algorithm 1. The parallel granularity is set so that each block takes one pair of documents. Document vectors are split evenly by threads in the block. For each assigned TF-ICF value, each thread determines if the other document vector contains the entry with the same index. Since the vectors are sorted, a binary search is conducted to lower the algorithmic complexity logarithmic time. A reduction is performed at the end to accumulate differences.

4.5. *Message Data Structure*

In sliced space partitioning, each slice is responsible to generate two sets of messages for the slices above and below. The corresponding message data structures are illustrated in Figure 7. The document array contains a header that enumerates the number of neighbors and migrating documents in the current slice. Their global indexes, positions and velocities are stored in the following array for neighborhood detection in a different slice. Due to the various sizes of each document's TF-ICF vector and the necessity to minimize the message size, we concatenate all vectors in a vector array without any padding. The offset of each vector array is stored in a metadata offset array for fast access. This design offers efficient parallel access to each document's information.

4.6. *Optimizations*

The algorithmic complexity of sliced partitioning decreases quadratically with the number of partitions (see Section 4.3). For a system with a fixed number of nodes, a

Algo 1: Document Vector Similarity (CUDA Kernel)

```
// calculate the similarities between two DocVecs
__device__ void docVecSimilarity(DocVec* lhs, DocVec *rhs, float *output) {
  float sim(0.0f);
  float commonSim(0.0f);
  for (int i = 0; i < lhs→NumEntries; i += blockIdx.x) {
      float tficf = biSearch(entry, rhs→vectors);
      sum += pow(entry→tficf − tficf, 2);
      commonSim += pow(tficf, 2);
  }
  // ... reduce to threadIdx.x(0), store in sum
  __syncthreads();
  if (threadIdx.x == 0) {
      sum −= commonSim;
      sum = sqrtf(sum);
      // write to global memory
      *output = sum;
  }
}

__device__ float biSearch(VecEntry *entry, DocVector *vector) {
  int idx = entry→index;
  int leftIndex = 0;
  int rightIndex = vector→NumEntries;
  int midIndex = vector→NumEntries/2;
  while(true) {
      int docIdx;
      docIdx = vector→vectors[midIndex].index;
      if (docIdx < idx)
        leftIndex = midIndex + 1;
      else if (docIdx > idx)
        rightIndex = midIndex − 1;
      else
        break;

      if (leftIndex > rightIndex)
        return 0.0f;
      midIndex = (leftIndex + rightIndex)/2;
  }
  return vector→vectors[midIndex].tficf;
}
```

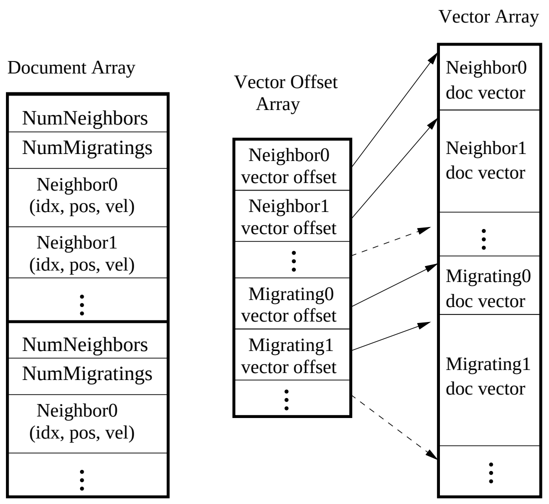


Figure 7: Message Data Structures

reduction in complexity could be achieved by exploiting multi-threading within each node. However, in practice, overhead increases as the number of partitions become larger. This is particularly this case for communication overhead. As we will see in Section 5, the effectiveness of such performance improvements differs from one system to another.

At the beginning of each iteration, each thread issues two non-blocking messages to its neighbors to obtain the neighboring and migrating documents' statuses (positions) and their vectors. This is followed by a neighbor detection function that searches its neighbor documents within a certain range for each internal document and migrated document. The search space includes every internal, neighbor and migrating document. We can split this function into three sub-functions: (a) internal-to-internal document detection; (b) internal-to-neighbor/migrating document detection and (c) migrating-to-all document detection. Sub-function (a) does not require information from other nodes. We can issue this kernel in parallel with communication. Since the number of internal documents is much larger than neighbor and migrated documents, we expect the execution time for sub-function (a) to be much larger than that of (b) or (c). From the system's point of view, either the communication or neighbor detection functions affects the overall performance.

One of the problems in simulating massive documents via the flocking-based algorithm is that as the virtual space size increases, the probability of flock formation diminishes as similar groups are less likely to meet each. In nature-inspired flocking, no explicit effort is made within simulations to combine similar species into a unique group. However, in document clustering, we need to make sure each cluster has formed only *one group* in the virtual space in the end without flock intersection. We found that an increase in the number of iterations helps in achieving this objective. We also dynamically reduce the size of the virtual space throughout the simulation. This increases

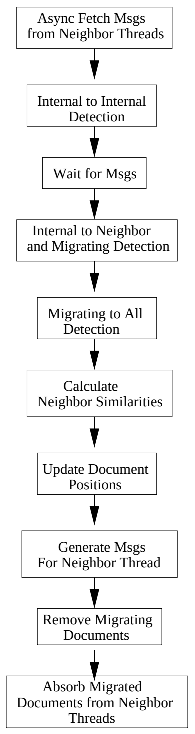

Async Fetch Msgs from Neighbor Threads
Internal to Internal Detection
Wait for Msgs
Internal to Neighbor and Migrating Detection
Migrating to All Detection
Calculate Neighbor Similarities
Update Document Positions
Generate Msgs For Neighbor Thread
Remove Migrating Documents
Absorb Migrated Documents from Neighbor Threads

Figure 8: Work Flow for a Thread in Each Iteration

the likelihood of similar groups to merge when they become neighbors.

4.7. Work Flow

The work flow for each space partition at an iteration is shown in Figure 8. Each thread starts by issuing asynchronous messages to fetch information from neighboring threads. Messages include data such as positions of the documents that have migrated to the current thread and documents at the margin of the neighbor slices. Those documents' TF-ICF vectors are encapsulated in the message for similarity calculation purposes, as discussed later.

Internal-to-internal document detection can be performed in parallel with message passing (see Section 4.6). The other two detection routines, in contrast, are serialized with respect to message exchanges. Once all neighborhoods are detected, we calculate the similarities between the documents belonging to the current thread and their detected neighbors. These similarity metrics are utilized to update the document positions in the next step where the flocking rules are applied.

Once the positions of all documents have been updated, some documents may have moved out the boundary of the current partition. These documents are removed from the current document array and form the messages for neighboring threads for the next iteration. Similarly, migrated documents received through messages from neighbors

	16 GPUs (NCSU)	16 CPUs (NCSU)	3 GPUs (ORNL)	3 CPUs (ORNL)
Nodes	16	16	4	4
CPU Cores	AMD Athlon Dual	AMD Athlon Dual	Intel Quad Q6700	Intel Quad Q6700
CPU Frequency	2.0 GHz	2.0 GHz	2.67 GHz	2.67 GHz
System Memory	1 GB	1 GB	4 GB	4 GB
GPU	16 GTX 280s	Disabled	3 Tesla C1060	Disabled
GPU Memory	1 GB	N/A	4 GB	N/A
Network	1 Gbps	1 Gbps	1 Gbps	1 Gbps

Table 1: Experiment Platforms

are appended to the current document array. This post-processing is performed in the last three steps in Figure 8.

5. Experimental Results

5.1. Experiment Setups

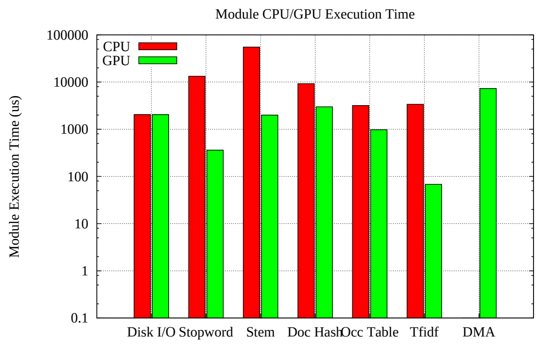
We conduct two independent sets of experiments to show the performance of our TF-IDF and document clustering results.

TF-IDF experiments are conducted on a stand-alone desktop in two configurations: with GPU enabled and disabled. When the GPU is disabled, we assess the performance of a functionally equivalent CPU baseline version (single-threaded in C/C++). The test platform utilizes Fedora 8 Core Linux with a dual-core AMD Athlon 2 GHz CPU with 2 GB of memory. The installation includes the CUDA 2.0 beta release and NVIDIA's Geforce GTX 280 as GPU devices. The test input data is selected from Internet news documents with variable sizes ranging from around 50 to 1000 English words (after stop-word removal). The average number of unique word in each article is about 400 words.

Similarly, the document clustering experiments are conducted on GPU-accelerated clusters with GPUs enabled and disabled. In the absence of GPUs, the performance of a multi-threaded CPU version of the clustering algorithm is assessed. In this version, internal document vectors are stored in STL hash containers instead of sorted document vectors used in GPU clusters. This combines benefits of fast serial similarity checking with ease of programming. The message structure is the same in both implementations. Hence, functions are provided to convert STL hashes to vector arrays and vice versa. In document clustering experiments, both GPU and CPU implementations incorporate the same MPI library (MPICH 1.2.7p1 release) for message passing and the C++ boost library (1.38.0 release) for multi-threading in a single MPI process. The GPU version uses the CUDA 2.1 release.

5.2. TF-IDF Experiments

In TF-IDF experiments, we first compare the execution time for one batch of 96 files. The individual module speedup and their percentages in total are shown in Figure 9 and Figure 10.



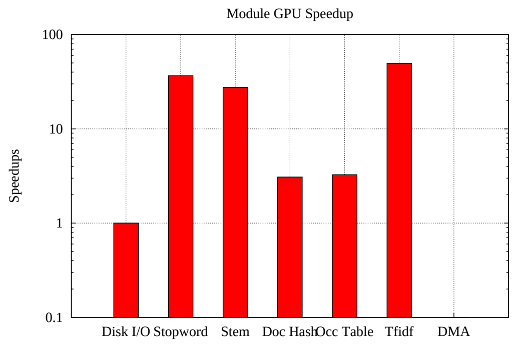


Figure 9: Per-Module Performance: CPU baseline vs. CUDA

Notice that the speedup on the y-axis of Figure 9 is depicted on a logarithmic scale. Compared to the CPU baseline implementation, we achieve more significant speedups for those modules engaged in the preprocessing phase (factor of 30 times faster in tokenize and 20 times faster in strip affixes kernels) than for those at the hash table construction phase (around 3 times faster in both document hash table and occurrence table insertion kernels). The limits in speedup during the latter are due to the multi-step hash table construction algorithms described in Section 3. The algorithm has certain overheads that the CPU benchmark does not contain. These overheads include (a) the construction of intermediate or hash sub-tables; (b) branching penalties suffered from the SIMD nature of GPU cores due to the imbalance in the distribution of tokens for a hash table's buckets; and (c) non-coalesced global memory access patterns as a result of the randomness of the hash key generation. Furthermore, the kernel for occurrence table insertion does not fully exploit all GPU cores because insertion is inherently serialized over files to avoid write conflicts within the same hash table bucket.

We also observe a reduction in the calculation time to the extent that the DMA overhead has become the largest contributor to overall time in a *single batch* scenario accounting for almost half of the total execution time. The combined time with disk I/O exceeds the total kernel execution time on GPU.

The observation above gives us the motivation to mitigate the memory overhead by

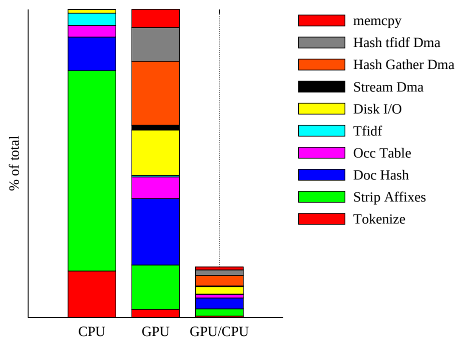


Figure 10: Per-Module Contribution to Overall Execution Time

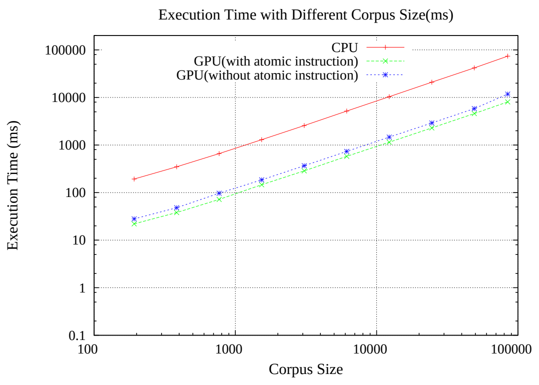


Figure 11: Execution Time with Different Corpus Size

double buffering the stream and hash tables when the corpus size gets larger. While we cannot hide the DMA overhead of a first batch, the DMA time of subsequent batches can be completely overlapped with the computational kernels in a *multi-batch* scenario. Figure 11 shows the execution time of CPU and CUDA with different corpus sizes.

The execution time of the two methods (both with and with the use of atomic instructions) are measured. With almost perfect parallelization between GPU calculation and data migration, we can hide almost all the kernel execution time in the DMA transfer and disk I/O time, which indicates a lower bound of the execution time. As a result the the asymptotic average batch processing time is almost half comparing to the single batch execution time, in which case the calculation and DMA cannot be overlapped. We also observe that the overall acceleration rates are 9.15 and 7.20 times faster than the CPU baseline.

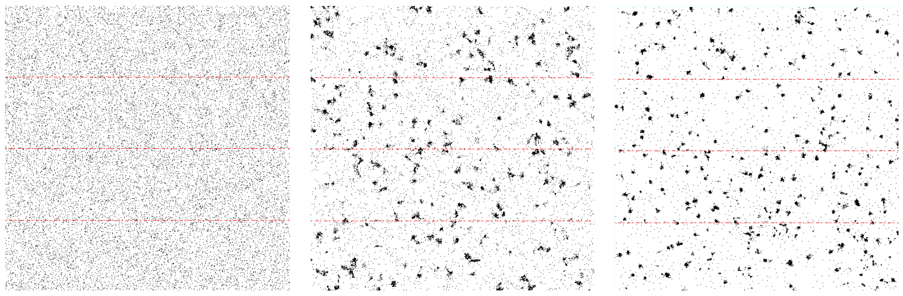

(a) Initial State (b) At Iteration 50 (c) At Iteration 500

Figure 12: Clustering 20K Documents in 4 GPUs

5.3. *Flocking Behavior Visualization*

We have implemented support to visualize the flocking behavior of our algorithm off-line once the positions of documents are saved after an iteration. The evolution of flocks can be seen in the three snapshots of the virtual plane in Figure 12, which shows a total of 20,000 documents clustered on four GPUs. Initially, documents are assigned at random coordinates in the virtual plane. After only 50 iterations, we observe an initial aggregation tendency. We also observe that the number of non-attached documents tends to decrease as the number of iterations increases. In our experiments, we observe that 500 iterations suffice to reach a stable state even for as many as a million documents. Therefore, we use 500 iterations throughout the rest of our experiments.

As Figure 12 shows, the final number of clusters in this example is quite large. This is because our input documents from the Internet cover widely divergent news topics. The resulting number is also a factor of the similarity threshold used throughout the simulation. The smaller the threshold is / the more strict the similarity check is, the more groups we will be formed through flocking.

5.4. *Document Clustering Performance*

We first compare the performance of individual kernels on an NVIDIA GTX 280 GPU hosted on a AMD Athlon 2 GHz Dual Core PC. We focus on two of the most time-consuming kernels: detecting neighbor documents (detection for short) and neighbor document similarity calculation (similarity for short). Only the GPU kernel is measured in this step. The execution time is averaged over 10 independent runs. Each run measures the first clustering step (first iteration in terms of Figure 12) to determine the speedup over the CPU version starting from the initial state. The speedup at different document sizes is shown in Figure 13. We can see that the similarity kernel on the GPU is about 45 times faster than on a CPU at almost all document sizes. For the detection kernel, the GPU is fully utilized once the document size exceeds 20,000, which gives a raw speedup of over 300X.

We next conducted experiments on two clusters located at NCSU and ORNL. On both clusters, we conducted test with and without GPUs enabled (see hardware configurations in Table 1). The NCSU cluster consists of sixteen nodes with CPUs and GPUs of lower RAM capacity for both CPU and GPU, while the ORNL cluster consists of

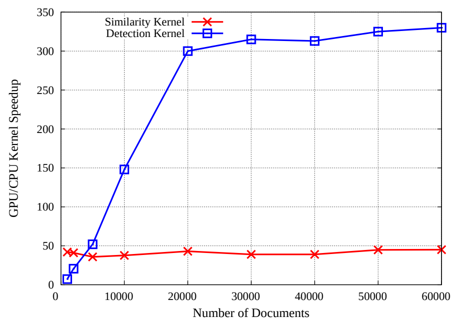


Figure 13: Speedups for Similarity and Detection Kernels

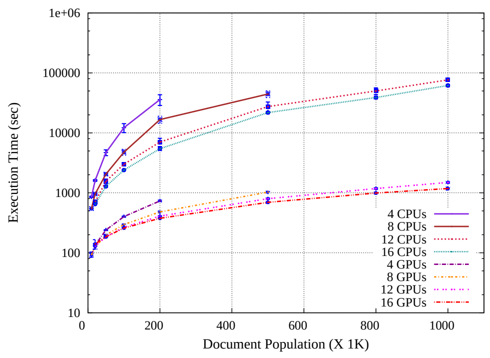


Figure 14: GTX 280 GPUs

fewer nodes with larger RAM capacity. As mentioned in Section 4.1, our programming model supports a flexible number of CPU threads that may exceed the number of GPUs on our platform. Thus, multiple CPU threads may share one GPU. In our experiments, we assessed the performance for both one and two CPU threads per GPU.

Figure 14 depicts the results for wall-clock time on the NCSU cluster. The curve is averaged over the execution for both one and two CPU threads per GPU. The error bar shows the actual execution time: the maximum/minimum represent one/two CPU threads per GPU, respectively. With increasing of number of nodes, execution time decreases and the maximal number of documents that can be processed at a time increases. With 16 GTX 280s, we are able to cluster one million documents within twelve minutes. The relative speedup of the GPU cluster over the CPU cluster ranges from 30X to 50X. As mentioned in Section 4.6, changing the number of threads sharing one GPU may cause a number of conflicts in resource. The benefit of multi-threading in this cluster is only moderate with only up to a 10% performance gain.

Though the ORNL cluster contains fewer nodes, its single-GPU memory size is four times larger than that of the NCSU GPUs. This enables us to cluster one million

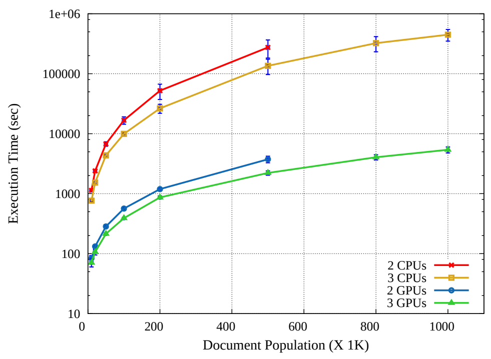


Figure 15: Tesla C1060 GPUs

documents with only three high-end GPUs. The execution time is shown in Figure 15. The performance improvement resulting for two CPU threads per GPU is more obvious in this case: at one million documents, three nodes with two CPU threads per GPU run 20% faster than the equivalent with just one CPU thread per GPU. This follows the intuition that faster CPUs can feed more work via DMA to GPUs.

Speedups on the GPU cluster for different number of nodes and documents are shown in the 3D surface graph Figure 16 for the NCSU cluster. At small document scale (up to 200k documents), 4 GPUs achieve the best speedup (over 40X). Due to the memory constraints in these GPUs, only 200k documents can be clustered on 4 GPUs. Therefore, speedups at 500k documents are not available for 4 GPUs. For 8 GPUs, clustering with 500k documents shows an increased performance. This surface graph illustrates the overall trends: For fewer nodes (and GPUs), speedups increase rapidly over for smaller number of documents. As the number of documents increases, speedups are initially on a plane with a lower gradient before increasing rapidly, *e.g.*, between 200k and 500k documents for 16 nodes (GPUs).

We next study the effect of utilizing point-to-point messages for our simulation algorithm. Because messages are exchanged in parallel with the neighborhood detection kernel for internal documents, the effect of communication is determined by the ratio between message passing time and kernel execution time: If the former is less than the latter, then communication is completely hidden (overlapped) by computation. In an experiment, we set the number of documents to 200k and vary the number of nodes from 4 to 16. We assess the execution time per iteration by averaging the communication time and kernel time among all nodes. The result is shown in Figure 17. For the GPU cluster, kernel execution time is always less than the message passing time. For the CPU cluster, the opposite is the case.

Notice that the communication time for the GPU cluster in this graph includes the DMA duration for data transfers between GPU memory and host memory. The DMA time is almost two orders of magnitude less than that of message passing. Thus, the GPU communication/DMA curve almost coincides with that of CPU cluster's communication time, even though the latter only covers pure network time as no host/device

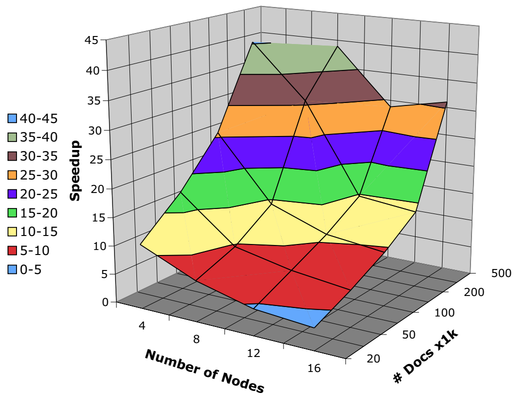


Figure 16: Speedups on NCSU cluster

Docs(k)	5	10	20	50	100	200	500	800	1000
4 nodes	74/9	67/8	64/5	58/3	52/1.5	49/0.9	NA	NA	NA
8 nodes	67/12	71/11	65/8	68/6	62/3.5	56/2	52/1.2	NA	NA
12 nodes	67/17	69/12	68/10	71/8	68/6	63/3	57/1.4	54/1.2	NA
16 nodes	63/18	63/13	71/12	69/9	65/7	66/4.2	59/1.9	60/1.5	55/1.1

Table 2: Fraction of Communication in GPU and CPU clusters (GPU/CPU) [in %]

DMA is required. This implies that internal PCI-E memory bus is not a bottleneck for GPU clusters in our experiments, which is important for performance tuning efforts. The causes for this finding are: (a) Network bandwidth is much lower than PCI-E memory bus bandwidth; and (b) messages are exchanges at roughly the same time on every node at each iteration, which may cause network congestion.

We further aggregate the time spent on message passing and divide the overall sum by the total execution time to yield the percentage of time spent on communication. For CPUs, the communication time consists of only the message passing time over the network. For GPUs, the communication time also includes the time to DMA messages to/from GPU global memory over the PCI-E memory bus. Table 2 shows the results for both GPU and CPU clusters. Generally speaking, in both cases, the ratio of communication to computation decreases as the number of documents per thread increases. The raw kernel speedup provided by GPU has dramatically increased the communication percentage. This analysis, indicating communication as a new key component for GPU clusters while CPUs are dominated by computation, implies disjoint optimization paths: faster network interconnects would significantly benefit GPU clusters while optimizing kernels even further would more significantly benefit CPU clusters.

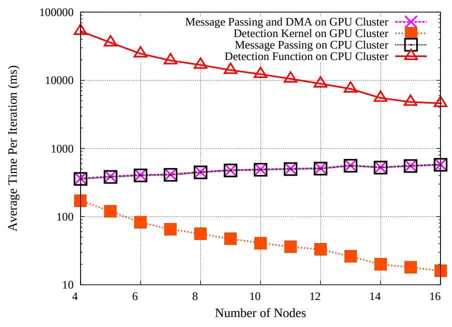


Figure 17: Communication and Computation in Parallel

6. Related Work

Our acceleration approach over CUDA to calculate document-level TF-IDF values uncovers yet another area of potential for GPUs where they outperform general-purpose CPUs. While it has been demonstrated that CUDA can significantly speedup many computationally intensive applications from domains such as scientific computation, physics and molecular dynamics simulation, imaging and the finance sector [14, 15, 16, 17, 18, 19], acceleration is less commonly used in other domains, especially those with integer-centric workloads, with few exceptions[20, 21]. This is partly due to the perception that fast (vector) floating-point calculation are the major contributor to performance benefits of GPUs. However, careful parallel algorithmic design may results in significant benefits as well. This is the premise of our work for text search workload deployment on GPUs.

Related research to document clustering can be divided into two categories: (1) fast simulation of group behavior and (2) GPU-accelerated implementations of document clustering. (1) The first basic flocking model was devised by Reynolds [22]. Here, each individual is referred as a “boid”. Three rules are quantified to aid the simulation of flocks: separation, alignment and cohesion. Since document clustering groups documents in different subsets, a multiple-species flocking (MSF) model is developed by Cui *et al.* [2]. This model adds a similarity check to apply only the separation rule to non-similar boids. A similar algorithm is found by Momen *et al.* [23] with many parameter tuning options. Computation time becomes a concern as the need to simulate large numbers of individuals prevails. Zhou *et al.* [24] describe a way to parallelize the simulation of group behavior. The simulation space is dynamically partitioned into P divisions, where P is the number of available computing nodes. A mapping of the flocking behavioral model onto streaming-based GPUs is presented by Erra *et al.* [25] with the objective of obstacle avoidance. This study predates the most recent language/run-time support for general-purpose GPU programming, such as CUDA, which allows simulations at much larger scale.

(2) Recently, data-parallel co-processors have been utilized to accelerate many computing problems, including some in the domain of massive data clustering. One successful acceleration platform is that of Graphic Processing Units (GPUs). Parallel

data mining on a GPU was assessed early on by Che *et al.* [26], Fang *et al.* [27] and Wu *et al.* [28]. These approaches rely on k-means to cluster a large space of data points. Since the size of a single point is small (*e.g.*, a constant-sized vector of floating point numbers to represent criteria such as similarity in our case), memory requirements are linear to the size of individuals (data points), which is constrained by the local memory of a single GPU in practice. Previous research has demonstrated more than five times speedups using a single GPU card over a single-node desktop for several thousands documents [29]. This testifies to the benefits of GPU architectures for highly parallel, distributed simulation of individual behavioral models. Nonetheless, such accelerator-based parallelization is constrained by the size of the physical memory of the accelerating hardware platform, *e.g.*, the GPU card.

7. Conclusion

In this paper, we present a complete application-level study of using GPUs to accelerate data-intensive document clustering algorithms.

We first propose a hardware-accelerated variant of the TF-IDF rank search algorithm exploiting GPU devices through NVIDIA's CUDA. We then develop two highly parallelized methods to build hash tables, one with and one without support of atomic instructions. Even though floating-point calculations are not dominating this text mining domain and its text processing characteristics limit the effectiveness of GPUs due to non-synchronized branching and diverging, data-dependent loop bounds, we achieve a significant speedup over the baseline algorithm on a general-purpose CPU. More specifically, we achieve up to a 30-fold speedup over CPU-based algorithms for selected phases of the problem solution on GPUs with overall wall-clock speedups ranging from six-fold to eight-fold depending on algorithmic parameters.

We further extend our work to a broader scope by implementing large-scale document clustering on GPU clusters. Our experiments show that GPU clusters outperform CPU clusters by a factor of 30X to 50X, reducing the execution time of massive document clustering from half a day to around ten minutes. Our results show that performance gains stem from three factors: (1) acceleration through GPU calculations, (2) parallelization over multiple nodes with GPUs in a cluster and (3) a well thought-out data-centric design that promotes data parallelism. Such speedups combined with the scalability potential and accelerator-based parallelization are unique in the domain of document-based data mining, to the best of our knowledge.

[1] Y. Zhang, F. Mueller, X. Cui, T. Potok, Large-scale multi-dimensional document clustering on gpu clusters, in: International Parallel and Distributed Processing Symposium, 2010, p. (accepted).

[2] X. Cui, J. Gao, T. E. Potok, A flocking based algorithm for document clustering analysis, J. Syst. Archit. 52 (8) (2006) 505–515.

[3] M. Steinbach, G. Karypis, V. Kumar, A comparison of document clustering techniques (2000).

[4] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU cluster for high performance computing, in: SC ’04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2004, p. 47.

[5] F. Chinchilla, T. Gamblin, M. Sommervoll, J. F. Prins, Parallel n-body simulation using GPUs, Tech. rep., University of North Carolina at Chapel Hill (2004).

[6] http://en.wikipedia.org/wiki/stop_words.

[7] M. Kantrowitz, B. Mohit, V. Mittal, Stemming and its effects on tfidf ranking (poster session), in: SIGIR ’00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval, ACM, New York, NY, USA, 2000, pp. 357–359. doi:10.1145/345508.345650.

[8] C. Reynolds, Steering behaviors for autonomous characters, in: Game Developers Conference, 1999.

[9] NVIDIA, NVIDIA CUDA Programming Guide(Version 2.0) (2008).

[10] T. Chen, Z. Sura, Optimizing the use of static buffers for dma on a cell chip, in: In The 19th International Workshop on Languages and Compilers for Parallel Computing, 2006.

[11] Y. Zhang, F. Mueller, X. Cui, T. Potok, Gpu-accelerated text minining, in: Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods, 2009.

[12] J. W. Reed, Y. Jiao, T. E. Potok, B. A. Klump, M. T. Elmore, A. R. Hurson, TF-ICF: A new term weighting scheme for clustering dynamic data streams, in: ICMLA ’06: Proceedings of the 5th International Conference on Machine Learning and Applications, IEEE Computer Society, Washington, DC, USA, 2006, pp. 258–263.

[13] K. Rieck, P. Laskov, Linear-time computation of similarity measures for sequential data, J. Mach. Learn. Res. 9 (2008) 23–48.

[14] M. Fatica, W.-K. Jeong, Accelerating matlab with cuda, in: HPEC, 2007.

[15] H. Nguyen(ed), GPU Gems 3, Addison-Wesley Professional, 2007.

[16] A. J. R. Ruiz, L. M. Ortega, Geometric algorithms on cuda, in: GRAPP, 2008, pp. 107–112.

[17] M. Curry, L. Ward, T. Skjellum, R. Brightwell, Accelerating reed-solomon coding in raid systems with gpus, in: International Parallel and Distributed Processing Symposium, 2008.

[18] http://www.nvidia.com/object/cuda_home.html.

[19] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the gpu using cuda, in: HiPC, 2007, pp. 197–208.

[20] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, Fast computation of database operations using graphics processors, in: SIGMOD ’04, ACM, New York, NY, USA, 2004, pp. 215–226.

[21] N. K. Govindaraju, N. Raghuvanshi, D. Manocha, Fast and approximate stream mining of quantiles and frequencies using graphics processors, in: SIGMOD ’05, ACM, New York, NY, USA, 2005, pp. 611–622.

[22] C. W. Reynolds, Flocks, herds, and schools: A distributed behavioral model, Computer Graphics 21 (4) (1987) 25–34.

[23] S. Momen, B. Amavasai, N. Siddique, Mixed species flocking for heterogeneous robotic swarms, in: EUROCON, 2007. The International Conference on ”Computer as a Tool”, 2007, pp. 2329–2336.

[24] B. Zhou, S. Zhou, Parallel simulation of group behaviors, in: WSC ’04: Proceedings of the 36th conference on Winter simulation, Winter Simulation Conference, 2004, pp. 364–370.

[25] U. Erra, R. De Chiara, V. Scarano, M. Tatafiore, Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance, in: Proceedings of Vision, Modeling and Visualization 2004 (VMV), 2004.

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, J. Parallel Distrib. Comput. 68 (10) (2008) 1370–1380.

[27] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sander, K. Yang, Parallel data mining on graphics processors, Tech. rep., The Hong Kong University of Science and Technology (October 2008).

[28] R. Wu, B. Zhang, M. Hsu, Clustering billions of data points using GPUs, in: UCHPC-MAW ’09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop, ACM, New York, NY, USA, 2009, pp. 1–6.

[29] J. S. Charles, T. E. Potok, R. M. Patton, X. Cui, Flocking-based document clustering on the graphics processing unit, NICSO (2007) 27–37.