Fast Document Similarity Computations using GPGPU

Parijat Shukla[1] and Arun K. Somani[2]
[1]*Xillinx, Inc., HITEC City, Hyderbad, India*
[2]*Dept. of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, U.S.A.*

Keywords: Deduplication, Semi-structured Data, NoSQL, Big Data, Parallel Processing, GPGPU, Data Shaping.

Abstract: Several Big Data problems involve computing similarities between entities, such as records, documents, etc., in timely manner. Recent studies point that similarity-based deduplication techniques are efficient for document databases. Delta encoding-like techniques are commonly leveraged to compute similarities between documents. Operational requirements dictate low latency constraints. The previous researches do not consider parallel computing to deliver low latency delta encoding solutions. This paper makes two-fold contribution in context of delta encoding problem occurring in document databases: (1) develop a parallel processing-based technique to compute similarities between documents, and (2) design a GPU-based document cache framework to accelerate the performance of delta encoding pipeline. We experiment with real datasets. We achieve throughput of more than 500 similarity computations per millisecond. And the similarity compuatation framework achieves a throughput in the range of 237-312 MB per second which is up to 10X higher throughput when compared to the hashing-based approaches.

1 INTRODUCTION

Semi-structured data is a prominent component of Big Data and is becoming more important day-by-day. Increasing importance of semi-structured data has generated vast interest in document-oriented databases (Apache Cassandra,); (MongoDB,); (Apache HBase,). The document databases store the semi-structured data, which is hierarchical in nature, in formats such as JSON, Avro, XML, etc., see references (XML,); (YAML: YAML Ain't Markup Language,); (JSON,); (Binary JSON,). Semi-structured data is also a part of scientific workflows. Scientific meta data associated with experiments, measurements, etc. are oftentimes in xml or other semi-structured format.

Oftentimes, these document databases are plagued with volume-borne challenges when dealing with data storage and data movement requirements. The scientific workflows require large scale data transfers across geographical locations. Deduplication helps in reducing the amount of data transferred and reduces the bandwidth requirement during transfers. Deduplication techniques such as similarity-based deduplication are deployed to overcome storage- and bandwidth-related issues in the document databases.

Recent studies conclude that delta encoding (compression) based deduplication offers advantages when compared to other data deduplication approaches such as (Xu et al., 2015). This is due to the fact that (1) regions of similarity are small, and (2) such similar regions are scattered in the deduplication stream.

The Research Problem. We address the following research problem: How can we leverage General Purpose Graphics Processing Units(GPGPUs) effectively to accelerate the the delta encoding pipeline? This work has two components: (1) How can we tackle the volume-related challenges associated with processing of Big Data workloads, and (2) How can we design a GPGPU-based solution which alleviates the performance bottlenecks of existing delta encoding solutions?

Our Response. Our response to first component is: (1) Design output aware techniques. For instance, computations involving favorable set of inputs must incur lesser time complexity when compared to computing with unfavorable set of inputs. Our response to second component is: (1) Design a parallel processing based solution which circumvents the pathologies of existing solutions? For example, trade the compute power of GPGPUs to avoid auxiliary data structures?

This work makes the following contributions:

- Develop a novel technique to compute degree of similarity in tree-structured data via identifying the similarity patterns.

Shukla, P. and Somani, A.
Fast Document Similarity Computations using GPGPU.
DOI: 10.5220/0006960303230331
In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018) - Volume 1: KDIR*, pages 323-331
ISBN: 978-989-758-330-8

- Develop an novel technique to compute similarity between two objects in context of delta encoding problem.
- Design a GPU-based document cache to accelerate the delta encoding pipeline in context of document databases.

The paper is organized as follows. Section 2 covers the background and related work. Section 3 describes our data shaping technique, and Section 4 describes our similarity computation technique. Section 5 discusses our GPU-based document cache framework for delta compression. Section 6 describe the experimental methodology and results obtained. Section 7 concludes the paper.

2 BACKGROUND

GPGPU. GPGPUs are advancing the high performance and high throughput computing. GPGPUs are basically Single Instruction Multiple Thread (SIMT) machines. In a GPGPU, a large number of threads execute a single instruction concurrently. The GPGPUs comprise hundreds of streaming processor (SP) cores, which operate in a parallel fashion. Streaming processors (SPs) are arranged in groups and each group of SPs is referred as Streaming Multiprocessor (SM). All the SPs within one SM execute instructions from the same memory block in lock-step fashion. GPGPUs are equipped with large number of registers. Main memory in GPUs is referred as device global memory and incurs a high access latency. Communication within SPs of one SM is through a low latency shared memory structure.

Unlike mainstream CPUs, the GPGPUs lack a rich memory hierarchy. Typically, only a single level limited size cache memory is available. A portion of available cache memory is designated as read only cache, which is used for caching a read-only portion of the device global memory. In this paper, we refer GPGPUs and Single Instruction Multiple Thread (SIMT) interchangeably.

Related Work. A study proposes algorithms for delta compression and remote file synchronization(Suel and Memon, 2002). Mogul et al. studies benefits of delta encoding and data compression for HTTP(Mogul et al., 1997). File system support for delta compression(MacDonald, 2000).

Zdelta, a tool for delta compression (Trendafilov et al., 2002a); (Trendafilov et al., 2002b). A cluster-based delta compression of a collection of files is studied in (Ouyang et al., 2002). A pre-cache similarity-based delta compression for use in a data storage system is studied in (Yang and Ren, 2012).

Recently, Shilane et al. focussed on preferential selection of candidates for delta compression (Shilane et al., 2016). Zhang et al. focussed on reducing solid-state storage device write stress through opportunistic in-place delta compression(Zhang et al., 2016b).

Xia et al. proposed DARE which is a deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads(Xia et al., 2016).

Wen et al. proposed edelta which is a word-enlarging based fast delta compression approach(Xia et al., 2015). Li et al. explored use of hardware accelerator for similarity based data deduplication(Li et al., 2015). Zhang et al. explored application of delta compression for energy efficient STT-RAM register file on GPGPUs(Zhang et al., 2016a).

3 DATA SHAPING FOR GPGPUS

We consider semi-structured form data represented in XML, JSON, etc. Figure 1(a) depicts one such semi-structured document represented in XML notation. The document is basically an excerpt of *revision* metadata from Mediawiki dataset (Wikimedia,). Precisely, this revision document relates to one of the revisions made by some wiki user RoseParks hose user id is 99, as shown under the *contributor*. The document contains several other information such as timestamp, comment, model, format, text id, sha1 hash of that update.

A given document is organized as a sequence of objects, wherein an object is marked by the start of a node label. For example, in Figure 1(a), XML label *revision* shown as $< revision >$ marks the beginning of object *revision*.

Figure 1(b) depicts the resulting encoding and its memory representation. The object *revision* starts at byte 384, shown as B'384, in memory.

4 PARALLEL DOCUMENT SIMILARITY COMPUTATION

In this section, we describe the application of our data shaping technique (described above) to compute document similarity metric in a parallel manner.

Specifically, we first describe the similarity computation problem being considered in this paper in Section 4. Next, we present an outline of the solution to similarity computation problem in Section 4. Remaining parts of this section covers details.

Figure 1: (a) Example of history metadata document. (b) Memory representation of document.

The Similarity Computation Problem. We consider the following problem: Given two rooted, ordered tree objects O1, O2. Also given is a similarity threshold, θ. Compute difference between O1, O2.

We make two key observations: (1) Closer are the objects, lesser is the similarity (dissimilarity) computation time. (2) Contiguous dissimilarity is better than disjoint dis-similarity.

Basic idea is that since the pattern of similarity is directly related to the final overhead incurred after delta encoding, it makes sense to track the patterns of similarity between the objects being considered for delta encoding. For example, if the two objects (to be encoded) are completely similar (identical) than the delta encoding overhead in insignificant.

Similarity Computation: An Overview. Apply data shaping technique described in Section 3 to obtain T1 and T2 which are linearized memory representation of O1 and O2. Compare linearized tree objects in parallel and record element-wise outcome. Perform data compaction and record dissimilarity pattern P. This is realized through the pseudocode as Listing 1. Classify P as a positive or negative pattern by comparing it with a set of template patterns. Specifically, P is classified as a positive pattern when P is closer to pattern(s) favorable to attain the desired δ. If P is classified as a positive pattern, then compute α_i, the similarity metric corresponding to i^{th} orientation pair. Else, P is classified as a negative pattern.

Similarity Patterns. Figure 2(a) depicts four patterns of similarity distribution. A shaded box denotes corresponding entries in linearized version of trees are dissimilar. First category of similarity pattern is when both source and sink are unshaded, and no migration is needed then.

Second category of similarity pattern denotes case when source is shaded while sink is unshaded. In this case, migration takes place from source to sink.

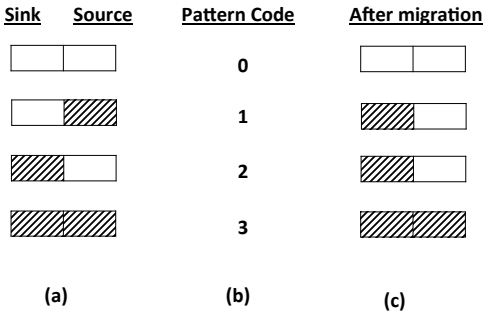

Figure 2: (a) Similarity patterns. Dissimilarity migration takes place from source to sink. A shaded box indicates corresponding entries in linearized trees are not similar. (b) Pattern codes, and (c) Post-migration similarity distribution.

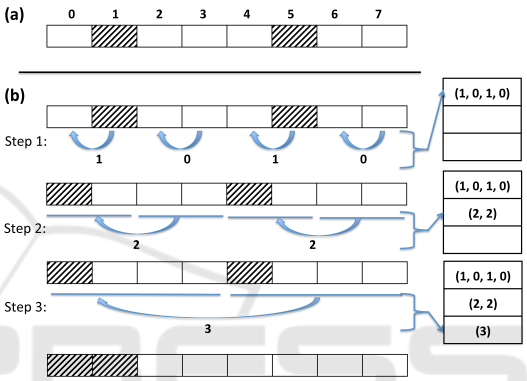

Figure 3: (a) Similarity distribution after comparing O1 and O2. (b) Steps to register similarity pattern. Number of steps is $\log_2(\max(O1, O2))$, which is 3 in this example. Similarity pattern code is indicated along the curved arrows. (1) In first step, cell size is 1. (2) In second step, size of cell is 2. (3) In third and final step size of 4 cells is considered. The similarity pattern is denoted as {(1, 0, 1, 0), (2, 2), (3)}. [Comment: add O1 and O2 and show the differences at index 1 and 5. And show how XORMatch is obtained.].

Third category of similarity pattern denotes case when source is unshaded while sink is shaded. No migration takes place in such cases. Fourth category of similarity pattern represents scenario of no similarity. No migration can happen in such cases. Figure 2(b) depicts the pattern code associated with these four categories of similarity distribution patterns. Figure 2(c) depicts the distribution after migration.

Registering Similarity Pattern. Figure 3(a) depicts distribution of similarity obtained after comparing O1 and O2. Note that two entries are shaded. The shaded entries signify the the corresponding elements in O1 and O2 are dissimilar. Specifically, when O1 and O2 are compared element-wise, elements 1 and 5 are dissimilar.

Figure 3(b) depicts how we record similarity pattern for O1 and O2. The similarity registration pro-

cess operates in binary reduction fashion. Total number of steps involved in registration process are bounded by $\log_2(\max(O1,O2))$, where $\max(O1,O2)$ denotes the maximum of O1 and O2. In this case, number of steps in reduction process are $\log_2(8) = 3$.

The figure depicts a three step reduction operation. In Step 1, four sets of operations occur in parallel (refer Figure 3(b)). Specifically, entries 0 and 1 participate in first set of comparison. Similarity pattern corresponds to type 1, which is marked on the arch from source to sink.

Second set of operation comprises entries 2 and 3. Note that both of the entries are unshaded which indicates a similarity pattern corresponds of type 0, as marked on the arch from source to sink. Third and fourth set of operations comprises entries 4 and 5, and 6 and 7, respectively. And similarity patterns corresponds to type 1 and type 0, respectively. After Step 1, pattern table is updated with similarity pattern (1, 0, 1, 0).

In Step 2, two set of operations occur in parallel. In first set of operation, entries 0-1 form the sink and entries 2-3 form source. Similarity pattern corresponds to type 2, as marked on the arch from source to sink. In second set, the participating entries are 4-7 wherein entries 4-5 form the sink and entries 6-7 form source. The similarity pattern corresponds to type 2 as marked on the arch. After Step 2, pattern table is updated with similarity pattern (2, 2).

In Step 3, one set of operation takes place wherein all the entries participate. Entries 0-3 form the sink and entries 4-7 form source. Similarity pattern corresponds to type 3, which is marked on the arch from source to sink. After Step 3, pattern table is updated with similarity pattern (3). With this step the final pattern table is as follows: {(1, 0, 1, 0), (2, 2), (3) }.

Listing 1 depicts the methodology to register the similarity distribution pattern.

Classification of Similarity Patterns. Next, we discuss some of the common similarity patterns from delta encoding perspective.

Intuitively, if O1 and O2 are identical, then the similarity pattern comprises of all 0's, and denoted as {(0, 0, 0, 0), (0, 0), (0)}.

If O1 and O2 are completely different, then the similarity pattern comprises all 3's, and denoted as {(3, 3, 3, 3), (3, 3), (3)}.

Consider another similarity pattern {(0, 0, X, X), (0, X), (X)}, where X=(1, 2, 3). This is similarity pattern represents a scenario where first half of O1 is identical to the first half of O2. Note that this is based on the observing similarity pattern emerging out of the first step which is {(0, 0, X, X) }.

Likewise, another similarity pattern {(X, X, 0, 0), (X, 0), (X)}, where X=(1, 2, 3), indicates that the bottom halves of O1 and O2 are identical.

Such patterns indicate that there exist a significant amount of contiguity in the similarity. Contiguous similarity pattern represent scenarios which are highly favorable for delta encoding. Reason being that the overhead resulting from delta encoding would be limited due to the contiguous nature of the similarity (or dissimilarity) in the objects being considered. Such favorable patterns are also referred as positive patterns.

Listing 1: Register-Pattern-GPU.

```
{ int step;
  int maxSteps = log(max(O1, O2));
  int maxThreads = max(O1, O2)/2;
  int i;  int Start;
  if ( blockIdx.x  < num_BLOCKS )
  { if ( threadIdx.x  < maxThreads )
    { for ( i = 0; i < maxSteps; i++)
        {if(threadIdx.x<maxThreads)
        { TID = threadIdx.x;
          if ( XORMatch[TID + 1] == 0 )
          {if ( XORMatch[TID] == 0 )
           {Sim_Pattern[i][TID] = 0; }
           else Sim_Pattern[i][TID]=2;}
            else{ Start = 0;
             if(XORMatch[TID] == 0)
             {Sim_Pattern[i][TID]=1;
              Start = 0;}
              else{Sim_Pattern[i][TID]=3;
               Start = sizeAT[TID];
            for(j=0;j<sizeAT[TID+1];j++)
               {XORMatch[TID+j]=
               XORMatch[(TID+1)+j];}
          sizeAT[TID]+=sizeAT[TID+1];
        }}}maxThreads = maxThreads/2;
        __threadfence_system();
 } } } } return;
}
```

Consider a pattern such as {(Y, Y, Y, Y), (3, 3), (3)}, where Y=(1, 2) indicates that similarity is non-contiguous. Such patterns represent worst case scenarios and are unfavorable for delta encoding from overhead aspect. Such unfavorable patterns are also referred as negative patterns.

We argue that negative patterns or unfavorable patterns are not likely to benefit the cause of delta encoding. This is due to the diverging nature of the overhead accruing. The task of finding delta encoding for objects, which exhibit such negative pattern similarity, should be abandoned in favor of other more meaningful computations.

Algorithm 1: ClassifySImilarityPattern( ).

```
1:  while (Level < Thresh) do
2:     Check all the nodes at this level:
3:     if ( NodeValue == 0 )) then
4:        case = BestCase, Exit
5:     end if
6:     if ( NodeValue == 1 )) then
7:        case = HalfIdentical
8:     end if
9:     if ( NodeValue == 2 )) then
10:       case = HalfIdentical
11:    end if
12:    if ( NodeValue == 3 )) then
13:       case = Dissimilar
14:    end if
15: end while
```

5 GPU-BASED DOCUMENT CACHE

In this section, we describe our proposed GPU-based In-memory Document Cache for Fast Delta Encoding delta encoding framework. The proposed delta encoding framework comprises (1) similarity computation substrate proposed in previous section and (2) a delta encoding system.

We maintain the documents in their native form. In other words, we do not hash the documents instead we represent them as-they-are. This approach offers several advantages: (1) Representing the documents preserves their structural details whereas those crucial structural details are lost after hashing. (2) Representing documents in their native form also facilitates obtaining their similar segments or the similarity pattern. Note that this similarity pattern information is very critical for determining if the similarity is good enough to produce effective delta compression.

Figure 4(a) depicts our GPGPU-based document cache and delta encoding framework. The framework depicts the a set of documents as input, GPU-based document cache, a delta encoder, and output. We consider a First-In-First-Out (FIFO).

Now, we describe operational aspects of the framework. The framework operates in batch mode. Input dataset is parsed and documents are organized their id-wise. A set of k input documents are presented to the document cache to obtain their corresponding similar documents. We refer to this set of k documents as query documents. The query documents are searched in document cache concurrently. If the document cache contains document(s) similar to the query document, the identifiers of the similar document(s) are passed to the delta encoder. The identifier comprises ID(s) of the similar document(s) and the similarity pattern.

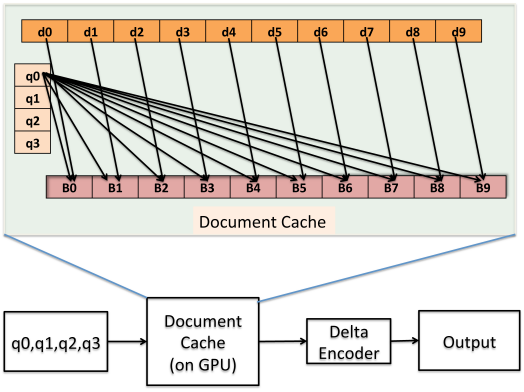

Figure 4: Block diagram of the framework comprising GPU-based document cache and delta encoder. The document cache is shown in detail (TOP).

Mapping of Similarity Computations on GPU. Let N be the total number of entries in the document cache and let N_Q be the total number of entries in the query set. Computations are organized into blocks in a 3D manner as follows:
(a) blockIdx.x = N;
(b) blockIdx.y = *max_ydim_threadblocks*;
(c) blockIdx.z = *max_ydim_threadblocks* / N_Q, where *max_ydim_threadblocks* is the maximum y- or z-dimension of a grid of thread blocks.

Note that the value of *max_xdim_threadblocks*, the maximum x-dimension of a grid of thread blocks, for Nvidia GPUs with compute capability higher than 3.0 is (2^{31}-1), which is a relatively very high number w.r.t. the size of document cache being considered in this work.

Figure 4(TOP) describes the mapping of document similarity computations on to the blocks of GPU using a document cache comprising 10 entries and a four query documents. The figure shows the mapping process of only one query, $q0$, for the sake of clarity. From the figure, we can see that each GPU block numbered as B0, B1, $\cdots$, B9 is assigned a copy of query document $q0$ and entries $d0$, $d1$, $\cdots$, $d9$, respectively. This set of computation forms the blockIdx.y=0. Similarly, blockIdx.y=1 handles the set of ten computations corresponding to query document $q1$. And computations due to query documents $q2$ and $q3$ are handled by blockIdx.y=2 and blockIdx.y=3, respectively.

6 EVALUATION

We used Kepler K20 GPGPUs in our experiments. The K20 device comprises 2496 Cuda cores or streaming processors (SPs) @706 MHz, and is equipped with 5 GB GDDR5 on-board memory. The compute platform comes with CPU having following specifications: Intel (R) Xeon (R) CPU E5-2650 @ 2.00 GHz machine running GNU/Linux. Algorithms proposed in this paper are implemented in C/Cuda7.5.

We used dumps of Mediawiki revision metadata for our experiments. The dataset is in XML format. We consider each revision metadata as one document. Each document is marked within $< revision >$ and $< /revision >$ tags. Each revision document comprises *contributor* which in turn is composed of *id* and *user name*. Then, each revision document contains several other information such as timestamp, comment, model, format, text id, sha1 hash of the update made under that revision.

Table 1 lists details of the datasets. *Wiki-57MB* dataset is 57 MB in size and contains $\sim$ 127000 documents. Similarly, *Wiki-107MB* and *Wiki-1.1GB* datasets are 107 MB and 1.1 GB in size, respectively; and each of them comprise 236,000 and 2,364,000 documents, respectively.

Table 1: Datasets.

Name	Size	Num. of documents
Wiki-57MB	57 MB	126,828
Wiki-107MB	107 MB	236,086
Wiki-1.1GB	1.1 GB	2,363,912
Wiki-11GB	11 GB	23,678,264

6.1 Data Shaping Overhead

Input dataset comprising documents is parsed on CPU. Table 2 depicts the parsing overhead. From this table we observe that wall-clock time elapsed in data shaping of 107 MB input document is 1.927 seconds. From this table we also observe that for a 10.2X increase in input size, the corresponding increase in parsing time is 10.16X indicating a nearly linear relation between input size and parsing time. The overall data shaping throughput is $\sim$55 MB per second.

Table 2: Data Shaping overhead (on CPU).

Name	Parsing (on CPU)
Wiki-107MB	1.927 seconds
Wiki-1.1GB	19.596 seconds

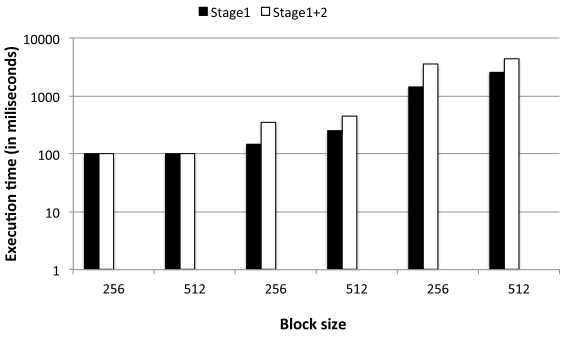

Figure 5: Throughput performance of similarity pattern algorithm on GPU.

6.2 Computation Throughput

To measure raw compute throughput of our similarity computation technique described in Section 4 on K20 GPU, we compute the similarity of one document from that dataset with all other documents. All of the resulting similarity computations run concurrently on GPU. Consider the 107 MB dataset Wiki-107MB which has $\sim$236,000 documents. As result of comparing one document against all other documents in that dataset, a total of $\sim$236,000 concurrent computations are generated.

Figure 5 plots the compute throughput of the similarity pattern algorithm for three wiki datasets. Note that the performance reported in the figure is from execution on K-20 GPU having 2496 Cuda cores. X-axis represents three Wiki datasets of 57 MB, 107 MB, and 1.1 GB in sizes. For each dataset, we experiments with GPU block sizes of 256 and 512 threads. Y-axis is log scale and represents the execution time elapsed on GPU as milliseconds.

From Figure 5, we observe that for block sizes of 256, time elapsed in Stage 1 for concurrent similarity computation of $\sim$236,000 is $\sim$148 milliseconds. And the time elapsed in Stage 1 and Stage 2 combined is $\sim$354 milliseconds. This results in a throughput of 302 MB per second. In terms of similarity computations, this amounts to an overall raw compute throughput of 666 similarity computations per millisecond. When using a GPU block of 512 threads, the time elapsed on Stage 1 is 257 milliseconds and that elapsed in Stage 1 and Stage 2 combined is $\sim$450 milliseconds, resulting in a throughput of 237 MB per second. These execution times yield a raw compute throughput of $\sim$524 similarity computations per millisecond.

Similarly, for 1.1 GB dataset, the time elapsed on Stage 1 and Stages 1, 2 combined is $\sim$1462 milliseconds and $\sim$3524 milliseconds, respectively when using block of 256 threads. This results in throughput of 312 MB per second. This yields a raw com-

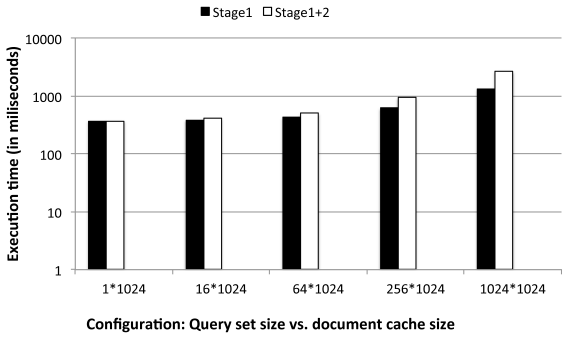

Figure 6: Performance of small document cache on varying size of query set.

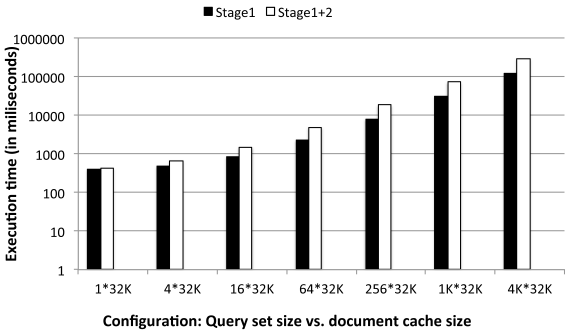

Figure 7: Performance of medium document cache on varying size of query set.

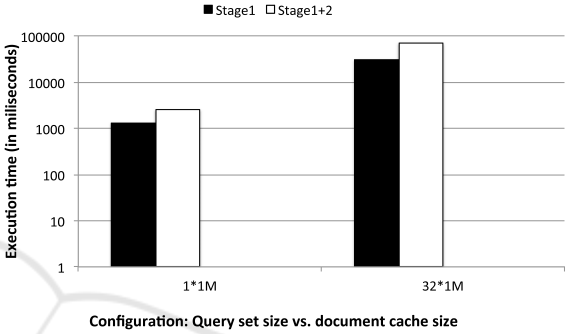

Figure 8: Performance of large document cache on varying size of query set.

pute throughput of ∼670 similarity computations per millisecond. And when using block size of 512, time elapsed on Stages 1 and 2 combined is ∼4478 milliseconds yielding in a throughput of 245 MB per second. And the raw compute throughput is ∼527 similarity computations per millisecond.

6.3 Scalability

We determine the size of query documents and document cache empirically. To this end we conduct the following three set of experiments: (1) Small document cache having 1024 entries, (2) Medium document cache with 32K entries, and (3) Large document cache having 1M document entries. For these set of experiments we set the GPU block sizes as 512 threads.

Figure 6 plots performance of small document cache on varying query set size from 1 to 1024 in steps of 4X. X-axis represents the configuration of similarity computation. For example, 16*1024 denotes a configuration when query set size is 16 and document cache size is 1024. Note that the configuration also indicates the total number of document similarity computations involved for the document cache and query set being considered. Y-axis is log scale and represents the execution time elapsed on GPU as milliseconds.

From Figure 6, we observe that for a document cache of 1024 entries the time elapsed in Stage 1 varies from ∼372 milliseconds to ∼1346 milliseconds when the size of query set is varied from 1 to 1024. From the figure, we also observe that for the same document cache the time elapsed in stages 1 and 2 combined varies from ∼374 milliseconds to ∼2649 milliseconds. These execution time measurements reveal that for our small document cache, an increase in query set size by 1024X results in ∼7X rise in execution time. This indicates that medium document cache is highly scalable.

Figure 7 plots performance of medium document cache on varying query set size from 1 to 16*1024 in steps of 4X. X-axis represents the configuration of similarity computation. Y-axis is log scale and represents the execution time elapsed on GPU in milliseconds. From Figure 7, we observe that for a document cache of 32K entries, where K=1024, the time elapsed in Stage 1 varies from ∼391 milliseconds to ∼125 seconds when the size of query set is varied from 1 to 4K. From the figure, we also observe that for the same document cache the time elapsed in stages 1 and 2 combined varies from ∼430 milliseconds to ∼293 seconds. These measurements reveal that for our medium document cache, an increase in query set size by 4096X results in ∼682X rise in execution time. This indicates that medium document cache is scalable.

Similarly, Figure 8 plots performance of our large document cache. The query set of size 1 and 32 are used. X-axis represents the configuration of similarity computation and log-scale Y-axis represents the time in milliseconds. From this figure, we observe that for our large document cache, the time elapsed in execution is ∼2.6 seconds and ∼71.85 seconds for query set of size 1 and 32, respectively.

6.4 Comparison with Hashing

Now, we analyse the performance of hashing-based approach. First, we measure performance of hashing-based approach. To this end, we use different values of window size, average chunk size, minimum chunk size, and maximum chunk size to understand the effect of these parameters on the performance. These measurements are carried out Intel CPU. We use four set of configurations which are as follows:

Config ID 1: Window size = 24 bytes; Average chunk size = 32 bytes; Minimum chunk size = 16 bytes; Maximum chunk size = 48 bytes;

Config ID 2: Window size = 24 bytes; Average chunk size = 64 bytes; Minimum chunk size = 32 bytes; Maximum chunk size = 96 bytes;

Config ID 3: Window size = 24 bytes; Average chunk size = 128 bytes; Minimum chunk size = 64 bytes; Maximum chunk size = 192 bytes;

Config ID 4: Window size = 48 bytes; Average chunk size = 32 bytes; Minimum chunk size = 16 bytes; Maximum chunk size = 48 bytes;

Table 3 depicts the performance outcome of the hashing-based approach under these configurations. From this table, we note that the performance of hashing-based approach is independent of the values of window sizes, average chunk sizes, minimum and maximum chunk sizes. This observation holds true when size of the input is increased by 10X. The throughput of the hashing-based approach is in the range of 28-29 MB/sec. Specifically, when using smaller dataset of 107 MB, the throughput observed is 28.1 MB/sec. And when the larger dataset of 1.1 GB is used, the throughput of hashing-based approach is 29.4 MB/sec.

The similarity compuatation framework proposed in this paper achieves a throughput in the range of 237-312 MB per second (refer to 6.2). Alternatively, our novel approach result in up to 10X higher throughput.

Table 3: Chunking and Hashing Time (on CPU).

Dataset	ID	Parsing + Chunking (Rabin)	Parsing + Chunking (Rabin) + Hashing (Murmur3)
Wiki-107MB	1	3.768 sec	3.803 sec
Wiki-107MB	2	3.756 sec	3.808 sec
Wiki-107MB	3	3.764 sec	3.804 sec
Wiki-107MB	4	3.761 sec	3.804 sec
Wiki-1.1GB	1	36.951 sec	37.375 sec

7 CONCLUSION

Several Big Data problems involve computing similarities between entities, such as records, documents, etc., in timely manner. Recent studies point that similarity-based deduplication techniques are efficient for document databases. Delta encoding-like techniques are commonly leveraged to compute similarities between documents. Operational requirements dictate low latency constraints. The previous researches do not consider parallel computing to deliver low latency delta encoding solutions.

Through this paper, we made a two-fold contribution in context of delta encoding problem occurring in document databases: (1) developed a parallel processing-based technique to compute similarities between documents, and (2) designed a GPU-based document cache to accelerate the performance of delta encoding pipeline. We experiment with real datasets. Our experiments demonstrate the effectiveness of GPUs in similarity computations. Specifically, we achieve throughput of more than 500 similarity computations per millisecond. And the similarity compuatation framework proposed in this paper achieves a throughput in the range of 237-312 MB per second which is up to 10X higher throughput when compared to the hashing-based approaches.

REFERENCES

Apache Cassandra. http://cassandra.apache.org/.

Apache HBase. http://hbase.apache.org/.

Binary JSON. http://bsonspec.org/.

JSON. http://www.json.org/.

Li, D., Wang, Q., Guyot, C., Narasimha, A., Vucinic, D., Bandic, Z., and Yang, Q. (2015). Hardware accelerator for similarity based data dedupe. In *NAS*, pages 224–232.

MacDonald, J. (2000). *File system support for delta compression.* PhD thesis, Masters thesis. Dep. of EECS, University of California at Berkeley.

Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B. (1997). Potential benefits of delta encoding and data compression for http. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 181–194.

MongoDB. http://www.mongodb.org/.

Ouyang, Z., Memon, N., Suel, T., and Trendafilov, D. (2002). Cluster-based delta compression of a collection of files. In *WISE*, pages 257–266.

Shilane, P. N., Wallace, G. R., and Huang, M. L. (2016). Preferential selection of candidates for delta compression. US Patent 9,262,434.

Suel, T. and Memon, N. (2002). Algorithms for delta compression and remote file synchronization.

Trendafilov, D., Memon, N., and Suel, T. (2002a). zdelta: An efficient delta compression tool.

Trendafilov, D., Trendafilov, D., Memon, N., Memon, N., Suel, T., and Suel, T. (2002b). zdelta: An efficient delta compression tool. Technical report.

Wikimedia. https://dumps.wikimedia.org.

Xia, W., Jiang, H., Feng, D., and Tian, L. (2016). Dare: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE Trans. on Computers*, 65(6):1692–1705.

Xia, W., Li, C., Jiang, H., Feng, D., Hua, Y., Qin, L., and Zhang, Y. (2015). Edelta: A word-enlarging based fast delta compression approach. In *HotStorage 2015*, Santa Clara, CA.

XML. http://www.w3.org/XML/.

Xu, L., Pavlo, A., Sengupta, S., Li, J., and Ganger, G. R. (2015). Reducing replication bandwidth for distributed document databases. SoCC, pages 222–235. ACM.

YAML: YAML Ain't Markup Language. http://www.yaml.org/.

Yang, Q. and Ren, J. (2012). Pre-cache similarity-based delta compression for use in a data storage system. US Patent App. 13/366,846.

Zhang, H., Chen, X., Xiao, N., and Liu, F. (2016a). Architecting energy-efficient stt-ram based register file on gpgpus via delta compression. In *Proc. of 53rd Annual Design Automation Conference*, page 119. ACM.

Zhang, X., Li, J., Wang, H., Zhao, K., and Zhang, T. (2016b). Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *FAST 2016*, pages 111–124.