Efficient Near-Duplicate Document Detection using FPGAs

Xi Luo Walid Najjar Vagelis Hristidis
Computer Science and Engineering
UC Riverside
Riverside, CA, USA
{luox,najjar,vagelis}@cs.ucr.edu

***Abstract*—Detecting duplicate and near-duplicate documents is critical in applications like Web crawling since it helps save document processing resources. Simhash is a state-of-art method to assign a bit-string fingerprint to a document, such that similar documents have similar fingerprints. Finding the near-duplicates in a large collection of documents consists of two stages: (a) compute the simhash fingerprint of each document, and (b) find pairs of similar fingerprints by computing their Hamming distance.**

Previous work has focused on optimizing the second stage, i.e., avoiding the quadratic number of comparisons to compute the all to all Hamming distance. However, our experiments show that the total time is dominated by the first stage (the fingerprints computation), which is the focus of this paper. We propose an implementation of simhash on Field Programmable Gate Arrays (FPGAs), by implementing a customized fingerprint computing engine in hardware that exploits parallelization and pipelining opportunities. We present a comprehensive experimental evaluation on large diverse real document datasets. Our experiments show a speedup of 362x in the simhash computation, and savings of up to 98% in overall near-duplicate detection execution time compared to using multi-core CPUs.

Keywords-duplicate detection; FPGA; hardware; document similarity; hashing

I. INTRODUCTION

Detecting duplicate and near-duplicate documents is important in many applications. In Web crawling, it avoids spending resources on parsing and indexing near duplicate document, which have little value to users. In Web or news search, it allows grouping together results with near-identical content, in order to avoid overwhelming the user.

Simhash [3] is a popular hash-based method to detect near-duplicates in document collections [9, 12, 17]. The key idea is that each document is represented by a short, e.g., 64-bit, fingerprint that summarizes its terms. These fingerprints are then used for comparing documents, which leads to great time savings.

In particular, the process consists of two stages: *(a)* the simhash calculation stage computes the fingerprint value of every document in the collection; *(b)* the matching stage finds pairs of near-duplicates documents by comparing their simhash fingerprint values. Note that for matching fingerprints, the actual documents must be compared since the process may generate false positive matches.

Previous work [12, 17] has focused on how to optimize the second stage of the process, i.e., the matching stage, to avoid performing a quadratic number of simhash fingerprint value comparisons. For instance, the key contribution of [12] is a redundant organization of the simhash signatures of a collection of documents that is used for quickly testing if a new document is a near-duplicate of one of the documents in the collection. We provide more details in Section II.B.

However, our experiments show that the fingerprint calculation stage dominates the overall execution time by a factor of up to 75 to 1 (see Section IV). In this paper we propose a novel approach to optimize the first stage of simhash-based near-duplicate detection, by leveraging FPGAs to quickly read through large numbers of documents and generating their simhash fingerprints.

The key idea is to configure multiple customized fingerprint-computing engines in hardware through which documents are streamed one character (byte) every cycle. Eight such engines can be configured on each of four FPGAs, on the Convey Computers HC-2ex [4] allowing up to 32 documents to be processed concurrently. The engines operate at 150 MHz resulting in a processing capacity of 4.8 GB/s.

The contributions of this paper are:

1) We show how the simhash signature of a document can be efficiently computed using FPGAs (Section III).
2) We perform comprehensive experiments on large real data sets. The experiments show a speedup of 362x in the simhash computation, and savings of up to 98% in overall near-duplicate detection execution time compared to using multi-core CPUs (Section IV).

We present background on FPGAs and simhash in Section II. Related work is discussed in Section V. We conclude and present a discussion on the significance of our results in Section VI.

II. BACKGROUND

A. Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are integrated circuits on which digital logic structures can be dynamically configured under software control. They can be viewed as a very large collection of logic gates, flip-flops and other hardwired logic devices whose connections, set in software by a configuration file, make it possible to instantiate any logic circuit on the chip. At the heart of FPGAs is a programmable N-input Look-Up Table (LUT) that can implement any Boolean function of N inputs (N=4 in Figure 2). Typical values of N range from four to six. Multiple LUTs can be combined to implement large and complex logic functions. A typical FPGA architecture consists of a rectangular array of logic blocks, called Common Logic Block (CLB), shown in Figure 1, each consisting of a small number of LUTs, two to four, with some flip-flops and multiplexers. Signals are routed between CLBs over a grid of routing channels, itself programmable in software.

To program an FPGA, the logic design is described

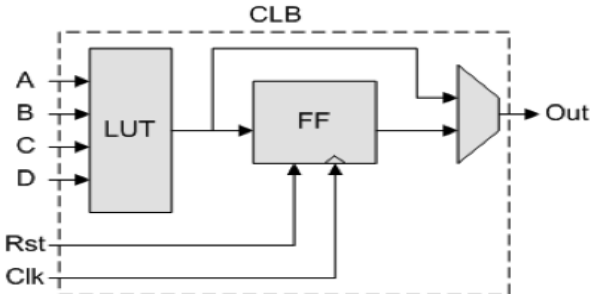

Figure 2: Example of a Common Logic Block (CLB) of an FPGA.

using a Hardware Description Language (HDL, e.g. VHDL, Verilog or SystemC). This program is then processed through a complex tool chain (synthesis, technology mapping, place and route) to generate the bit file that would configure the FPGA.

The performance advantages of FPGA-based heterogeneous platforms arise from their ability to map highly repetitive computations to the FPGA as a hardware circuit through which data is streamed. This model allows the large-scale exploitation of parallelism at various levels of granularity. Furthermore, in recent years FPGAs have witnessed a tremendous increase in size, speed and I/O bandwidth making this model even more attractive. While these advantages are shared with Application Specific Integrated Circuits (ASIC), FPGAs however can also be reconfigured, are more adaptable to changes in applications and specifications, and exhibit a faster time to market. In summary, FPGA accelerators combine the re-programmability of software with the inherent speed of hardware, albeit slower than ASICs. A survey of FPGAs and ASICs can be found at [7].

B. Simhash and Near-Duplicate Document Detection

The process of discovering near-duplicate documents consists of two stages: *(a)* the simhash fingerprint calculation stage computes the simhash fingerprint value of all documents in the collection; *(b)* the matching stage finds pairs of near-duplicates documents by comparing their fingerprint values.

a) Simhash Calculation Stage

Simhash [3] is a hashing scheme that maps an object (document) that consists of features (terms) to a fingerprint bit-string. The key desirable property of simhash is that the number of common bits in the simhash fingerprint of two documents is positively correlated to the cosine similarity of the documents. (Cosine similarity is a popular distance function in Information Retrieval.)

To compute the f-bits simhash of a document, we first have to compute the f-bits signature of each term, as we explain below. Further, we have to count the number of occurrences of each term t in the document, which is the weight of t. Then, we create a vector V of f integers. To compute V[i] we add the weight of each term whose signature has a 1 in the i-th position, and subtract the weight of each term whose signature has a 0 in the i-th position. Finally, the simhash vector has 1 in positions i with V[i]>0, and 0 in the other positions.

1) *Term Hash Function*: To generate a hash signature for a term (word) we can use various hashing schemes whose goal is that very different documents should have very different signatures [8, 19]. We use the sdbm hashing function in our implementation and experiments, which is the most popular function for simhash implementations, and is also used in other popular text handling systems like Berkeley DB due to its effectiveness. sdbm generates a signature for a term (word) using the following function that iterates over the characters of the word:

```
hash(i) = hash(i-1)*65599 + str[i]   (1)
```

Variable i goes over the characters of the word, 0 to s, where s-1 is the length of the word. The signature of the word is

```
Unsigned long sdbm (unsigned char *str)
{
  Unsigned long h = 0;
  int c;
  while (c = *str++)
    h = c + (h << 6) + (h << 16) - h;
  return h;
}
```

Figure 1: Implementation of sdbm hash function in C.

Consider document: "A school is a school if it has students and teachers"

The terms, with their weight and their 64-bit term hash signatures are:

Term	*Weight*	*sdbm signature*
school	*2*	*0001100010100100001000101000010101011000001101010000111011110100*
students	*1*	*0110001001010100000110011101001010001000110100111001101100111000*
teachers	*1*	*1010011000101110111000111100110100100111001000010100000110110001*

Hence, the vector V is: -2,-2,0,0,0,-2,0,-4,0,-2,2,-2,-2,4,-2,-4,-2,-2,2,-2,-2,-4,2,0,4,0,-4,-2,-2,2,-2,2,-2,0,-2,0,2,-2,-2,-2,-2,-2,2,2,2,-4,0,-2,4,-2,-2,-4,-2,2,0,2,0,2,0,4,4,-2,0,-4,-2

and the simhash of the document is: 0011101010100100001000111100010101011000001101010000111111110100

Figure 3: Simhash calculation example using sdbm as word hashing function.

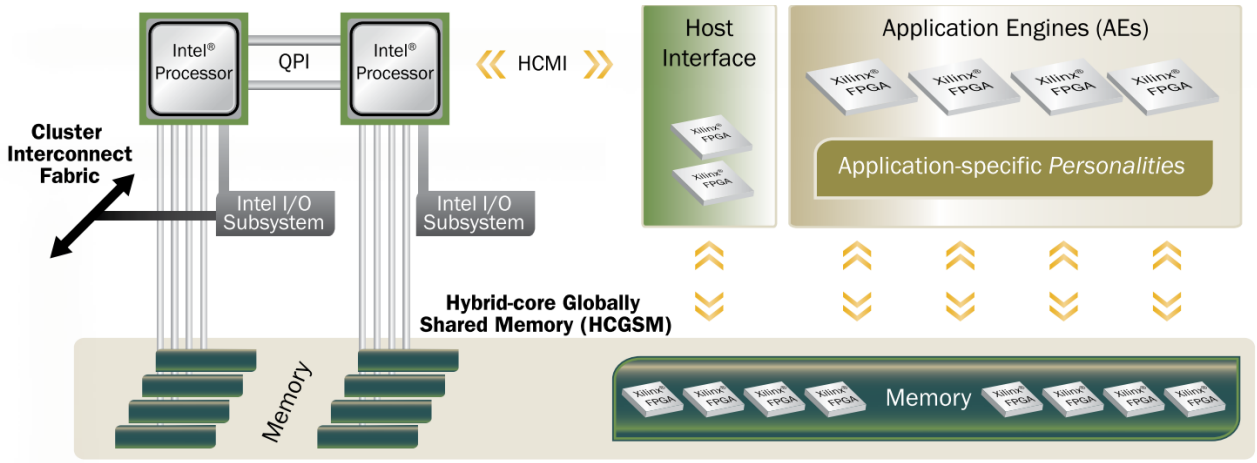

Figure 4 - Architecture of the Convey HC-2ex heterogeneous multiprocessor:

hash(s). To initialize, we set hash(-1)=0. The constant 65599 was picked after experimenting with different constants, and is a prime number. An efficient implementation of Equation 1 in C is shown in Figure 1. Figure 3 shows an example of the simhash computation for a small document.

b) Matching Stage

After the simhash fingerprints are calculated, the problem is how to efficiently detect near-duplicate pairs of documents, that is, documents whose fingerprint values have at most k different bits. For 64-bit fingerprints, k=3 is a typical value used [13]. Researchers have proposed ways to optimize this stage by avoiding the naïve quadratic cost of fingerprint comparisons [13, 18]. In particular, the key problem is how to decide if a new document is near duplicate to any of the existing documents in the collection.

The main idea in [13], which we have also implemented in our experiments (Section IV), is to create several copies of the table of fingerprints of the collections, where in each copy we permutate the positions of the bits. The key intuition is that the prefix of a new simhash will match the prefix of at least one of the copies of each near-duplicate simhash in the collection. Hence, we can do a binary search on each of the table copies using the prefix bits to find all near-duplicate documents. The number of copies and the length of the matching prefix are determined based on the value of k and space vs. time tradeoff considerations. More details are available at [12].

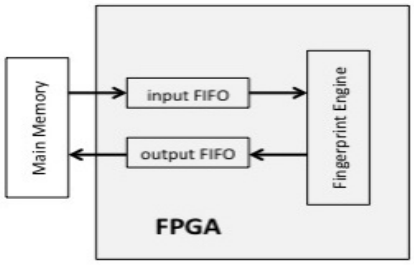

Figure 5: FPGA interface to memory, all channels are 64-bit wide.

III. Computing Simhash Signatures Using FPGAs

A. The Convey Computers Heterogeneous Architecture

The Convey Computers [4, 5, 6], shown in Figure 4, are the first heterogeneous machines with a cache coherent virtual

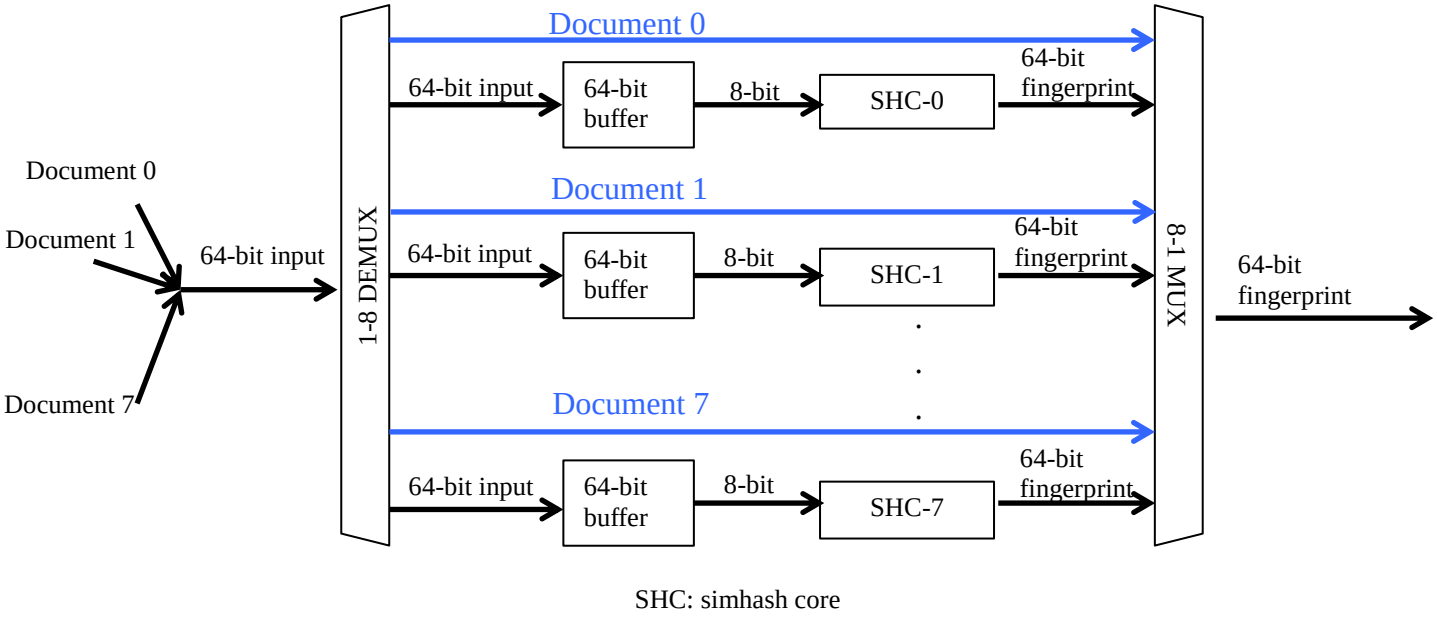

Figure 6: The Fingerprint Engine, implements the simhash algorithm on eight 8-bit consecutive values.

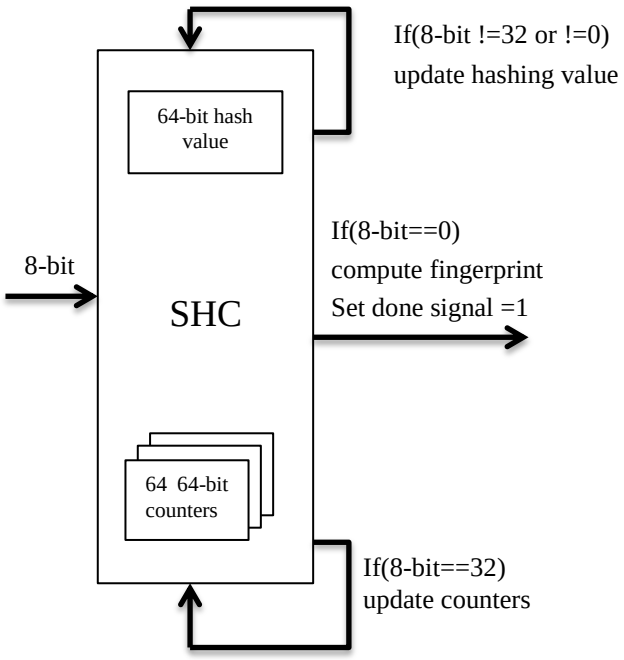

Figure 7: The SimHash Core computes the 64-bit signature.

memory space that is shared between the software (CPU execution) and hardware (FPGA execution). This allows an application to switch its execution between software and hardware without needing to offload data, and thus it could be done with little overhead. Furthermore, the user need not worry about virtual to physical address translation. It should be noted that this memory space is divided into two regions: host and co-processor memory spaces. While both spaces are shared, improperly assigned data can hinder performance due to inefficient accesses across these spaces [1]. The Coprocessor consists of four large FPGAs, called Application Engines (AEs). Each AE interfaces to eight Memory Controllers (MC) via a full crossbar supporting memory requests reordering. Each MC supports two ports, even/odd, each capable of eight bytes per cycle at 150 MHz [4]. A total of 16 memory ports deliver a peak bandwidth of 19.2 GB/sec per AE.

Each FPGA on the Convey HC-2ex has 16 memory channels; to fully use its bandwidth, eight will be used as input channels and eight as output channels. As shown in Figure 5, the input and output channels are connected to a Fingerprint Engine (FE). Eight such engines are configured on each of the four FPGAs.

The structure of the Fingerprint Engine is shown in Figure 6: inside each FE are eight SimHash Cores (SHC) feeding an eight-to-one multiplexer. Each SHC processes the data from one document, which is represented in ASCII. A 64-bit word is read from each of eight documents, in a round robin fashion each cycle, and fed to one of the SHC on the FE. The eight characters in the 64-bit word are processed sequentially by each SHC, as shown in Figure 7.

Once a character is read, the SHC executes the pseudo-code shown Figure 8: either the hash value or the summing weight is updated based on the character's value. When a

space is found, (ASCII value 32), a word has been processed and the summing weights are updated, otherwise the hash value is updated. The end of document is marked by NULL (ASCII zero), which triggers the generation of that fingerprint based on the current summing weights (by setting *done=1* in the code of Figure 8).

Since there are eight SHC, it is possible that eight fingerprints are generated during the same cycle, where one character is read from each document at each cycle. The multiplexer serves also as a priority encoder and latch to allow only one fingerprint output every cycle. We assume that any useful document is at least eight characters long, so no fingerprint will be lost. When two or more fingerprints are generated in the same cycle, one is sent to the output and the others are latched and returned in subsequent cycles.

```
uint_64 [64] summing_weight;
uint_64 hash_value, fingerprint;
char letter;
bool done;

while (done! =1)
{
  if (letter!=32 || letter!=0)
    hash_value = letter  +  (hash_value << 6) + (hash_value <<
    16) - hash_value;
  else if (letter == 32)
    for (int i =0; i<64; i++)
    {
        if (hash[i] ==0)
            summing_weight [i]++;
        else
            summing_weight [i]--;
    }
}

while (done ==1)
{
    for (int i =0; i<64; i++)
    {
        if (summing_weight [i] >=0)
            fingerprint [i]==0;
        else
            fingerprint [i]==1;
    }
}
```

Figure 8: Code executed at SimHash Cores (SHC).

There are four FPGAs on Convey. They can process data in parallel. To achieve best performance, the documents in a collection are split into four roughly equal sets, one for each FPGSA. Each FPGA will then process its own data set at the same time. That is, this configuration can process up to 4x8=32 document concurrently, one character from each document during each cycle. This means that we can process up to 32 bytes per cycle, 150 MHz.

IV. Experiments

A. *Hardware Configurations and Experimental Setup*

In our experiments we used the HC-2ex described in Section II.A for both the software (CPU based) as well as the hardware (FPGA based) implementations. The HC-2ex has four Virtex-6 LX760 FPGAs in the AE, two Intel Xeon processors E5-2643 4-core 3.3GHz (used in the software implementation), and 96 GB shared memory (used by both implementations). Both the software and hardware implementations were evaluated on the same machine. The software implementation was run in multithreaded mode (eight threads on four cores).

Both experiments were run with the data (documents) resident in memory to factor out the effects of disk I/O. All software results are measured using eight way multi-threaded implementations on the multi-core CPUs. All FPGAs are running at 150 MHz clocks.

We use 64-bit simhash signatures, which is the standard length used in simhash implementations [12]. As shown in Figure 1, sdbm hashing is chosen as the term hash function, which has been proven to be both efficient and with low collision rates. It is also hardware friendly, as we show in Section III.

B. *Data Sets Used*

The data sets used are Wikipedia pages[1], TREC[2] documents and Twitter posts. All Wikipedia pages were downloaded. The TREC data are English documents from TREC Volumes 1 to 5[3]. The Twitter data are one month of the sample of about 1% of the Twitter data, provided by the Twitter Streaming API[4], from May 11 to June 10 2013.

All data was preprocessed by applying standard information retrieval techniques: stop words removal, html tag removal, and punctuation removal. Some key statistics of the data sets are shown in Table I.

TABLE I – DATA SET STATISTICS

	Size before preprocessing (GB)	**Size after preprocessing (GB)**	**Number of documents**	**Average number of words per document**
Wikipedia	41	17	4,017,414	4,431
TREC	8.9	3.6	1,634,249	2,346
Twitter	11.3	5.1	114,333,421	48

[1] Downloaded from http://en.wikipedia.org/wiki/Wikipedia:Database_download
[2] Text REtrieval Conference (TREC), trec.nist.gov/
[3] as described in http://trec.nist.gov/data/docs_eng.html
[4] https://dev.twitter.com/docs/streaming-apis

One key difference between Twitter and other two data sets is that Twitter contains documents with very small number of words. Since simhash is a dimensionality reduction technique, low dimensional data limits its effectiveness. Nevertheless, it is interesting to see how our methods perform for collections of many short documents.

C. Results

1) *Execution Time of First Stage of Simhash:* Table II shows the execution times for computing the simhash signatures of all documents in each data set for both the software and FPGA-accelerated implementations.

TABLE II- SIMHASH EXECUTION TIME

	CPU (s)	FPGA (s)	Speed-up
Wikipedia	605	1.67	362
TREC	130	0.36	361
Twitter	192	0.53	362

It is interesting to note that the speed-up is independent of the data sets used and is around 362X. This can be explained by the fact that the FPGA implementation is a collection of statically configured concurrent pipelines that together can achieve a throughput of 4.8 GB/s.

2) *Execution Time of Second Stage of Simhash:* Next, we study the performance relationship between the two stages of simhash near-duplicate detection. In particular, we measure the execution time of the second stage, which is the matching phase. We adopt the matching implementation described in [13], summarized in Section II.B. We use software (CPU) to execute this stage, since using FPGAs is not a natural choice for this, given the frequent memory access patterns and the large memory required. We define near-duplicates to be documents with up to k=3 different bits in their simhash, as was also selected in [12].

In particular, the 64 simhash bits are divided into 4 groups. Each group contains 16 bits. We must consider all permutations of these four groups. There are four ways to choose one group from the four groups. Therefore, four tables will be created in memory, i.e., four copies of the simhash fingerprints.

For each fingerprint, we look up the four tables using binary search on the first 16 bits, and probe all matching fingerprints. The numbers of fingerprints (smallest power of 2 larger than the number of documents in data set) being generated, and the average number of probes per match[5] are shown in Table III, along with the total time to perform this process for all fingerprints in the data set (the last column). We see that the ratio of time for the first and second stage of simhash is about 75 to 1 (605 to 8.11) for software execution for the Wikipedia data set.

TABLE III- SIMHASH MATCHING TIMES USING SOFTWARE EXECUTION

	# fingerprints	Avg # probes	Time consumed (s)
Wikipedia	2^{23}	2^{7}	8.11
TREC	2^{25}	2^{9}	3.12
Twitter	2^{28}	2^{12}	276.00

Note that the matching time for Twitter is much larger because of the much larger number of documents, which leads to much more probes per match as shown in Table III. This time can be easily reduced by considering more tables, that is, more copies of the data, e.g., using 5 or 6 tables instead of four. However, the goal of Table III is to show that the second phase of simhash is generally much faster than the first phase (Table II), which is the focus of this paper.

3) *Analysis of overall Simhash Execution Time*: Table IV summarizes the execution time of the two stages of simhash, and shows that using FPGAs for the first stage leads to dramatic overall time savings. We see a 98% time reduction for the Wikipedia and TREC datasets. Note that the overall savings for the Twitter data set would be much larger if we used more tables in the second stage as discussed above.

Table IV shows the total execution time for the second stage of simhash, using CPU implementation.

TABLE IV- TOTAL TIMES (SIMHASH CALCULATION AND MATCHING)

	CPU (s)	FPGA (s)	Time Savings
Wikipedia	613	9.78	98.4%
TREC	133	3.48	97.4%
Twitter	468	276.53	40.9%

4) *Analysis of overall Simhash Execution Time with Different Multi-thread Strategy and Different Number of CPU Cores*: In the above software implementation, we used document-level multithreading strategy. That means each thread will calculate one document's fingerprint at a time. For completeness, we also implemented a word-level multithreading strategy, where each thread calculates one word's sdbm signature at a time. The results show that word-level multithreading is slower. Using Wikipedia, TREC, and Twitter data to test word-level multithreading implementation, the execution times are 1153 s, 206 s, and 1016 s, respectively.

[5] The second and third columns are not important for this paper, and require understanding the details of the method in [13]; we provide them for completeness.

We also ran our software implementation on another cluster with more CPU cores to see how the number of CPU cores affects our result. The test cluster has four AMD Opteron™ processors 6272 8-core 1.4GHz. While testing with Wikipedia, TREC and Twitter data, the execution times are 440 s, 100 s, 380 s. As expected, the execution times decrease with CPU core number and clock frequency.

V. Related Work

Simhash Documents Matching: [17] proposed a way to improve the performance of the matching stage of simhash-based similarity detection by using a probabilistic approach that considers which bits in simhash are more volatile. This allows them to improve on the time and space complexity over [12] if a small amount of recall can be sacrificed. However, they do not propose any optimization of the simhash calculation stage, which is the focus of this paper.

FPGAs in Information Retrieval: [20] propose an FPGA-based Web search query execution, where FPGAs are used to read inverted indexes, and rank documents. In contrast, our work focuses on the crawling phase of Web search, where near-duplicate elimination is critical. Vanderbauwhede et al. [18] propose an FPGA-based approach to match documents to profiles described by keywords and weights. They split the profiles into bins of fixed length and use a hash signature for each bin. Then, each document term is compared against these hash signatures. In contrast, we map documents to simhash fingerprints, and also all documents have a fingerprint of the same length. These differences allow us to have a higher speedup than [8], even though the problems addressed are not the same. [16] proposed methods for computing the similarities between vectors using FPGAs, to be used in data mining applications.

FPGAs in Database Queries: [10] propose an FPGA-based query execution approach, where FPGAs are used to find XML projections to improve query performance. The acceleration of XML queries on XML documents, using FPGAs, has been proposed in several papers [13, 14, 15].

VI. Significance and Conclusions

In this paper we showed how FPGAs can be used to dramatically speedup the process of detecting near-duplicates in large document collections. Our experiments show that a machine with four FPGAs can achieve a speedup of up to 362x compared to a four core CPU.

In addition to the significance of detecting near-duplicates (or exact duplicates which can also be handled similarly), our work shows the great potential of using FPGAs for various document processing applications.

Our techniques can be applied to applications that need to read documents and for each document the computation requires a relatively small size of intermediate storage (in simhash we need to store the intermediate value of simhash) and output for each document (in simhash we output the simhash value of each document).

Other applications can benefit from this configuration. For instance, if we have a high-throughput stream of documents like news or social posts, and we want to perform document filtering [2] or build a publish-subscribe system [11]. That is, we want to match the documents of the stream against continuous queries. A query could be a user-specified bag of keywords of interest, or a structured query that has a condition on the location, user, and text of a social post (e.g., tweet).

However, not all document applications can directly benefit from our FPGAs setting. For instance, to build an inverted index, one has to count the frequency of each word in a document; the number of unique words can be large and hence an FPGA does not offer a natural solution without relying on a large external storage.

Acknowledgment

Work partially supported by NSF Awards CCF-1219180, IIS-1161997, IIS-1216032 and IIS-1216007.

References

[1] T. Brewer. Instruction set innovations for the Convey HC-1 computer. Micro, IEEE, 30(2):70 –79, March-April 2010.

[2] J. Callan. (1996, August). Document filtering with inference networks. In Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval (pp. 262-269). ACM.

[3] M. S. Charikar, "Similarity Estimation Techniques from Rounding Algorithms," in Proc. ACM STOC, May 2002, pp. 380–388.

[4] Convey Computer. Convey Computer Reference Manual, www.conveycomputer.com/resources/, 2013.

[5] Convey Computer. Convey Computer Programmer's Guide, www.conveycomputer.com/resources/, 2013.

[6] Convey Computer. Convey Computer PDK Reference Manual, www.conveycomputer.com/resources/, 2013.

[7] K. Compton, & S. Hauck. (2002). Reconfigurable computing: a survey of systems and software. ACM Computing Surveys (csuR), 34(2), 171-210.

[8] R. J. Enbody and H. C. Du, "Dynamic Hashing Schemes", ACM Computing Surveys, vol. 20, no. 2, 85-113, 1988.

[9] M. R. Henzinger, "Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms," in Proc. ACM SIGIR, Aug. 2006, pp. 284–291

[10] T. Jens, W. Louis, N. Chongling, (2012). Skeleton automata for FPGAs: Reconfiguring without reconstructing. In SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 229-240

[11] H. Liu, V. Ramasubramanian, and E. G. Sirer. (2005, October). Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement (pp. 3-3). USENIX Association.

[12] G. S. Manku, A. Jain, & A. Das Sarma, (2007, May). Detecting near-duplicates for web crawling. In Proceedings of the 16th international conference on World Wide Web (pp. 141-150). ACM.

[13] R. Moussalli, M. Salloum, W. A. Najjar and V. J. Tsotras. Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs in Proc. Int. Conf. on Data Engineering 2011 (ICDE), Hanover, Germany.

[14] R. Moussalli, M. Salloum, W. A. Najjar and V. Tsotras. Accelerating XML Query Matching Through Custom Stack Generation on FPGAs, Proc. Int. Conf. on High-Performance Embedded Architectures and Compilers, January 25-27, 2010, Pisa, Italy.

[15] R. Moussalli, M. Vieira, W. Najjar and V. Tsotras. Stream-Mode FPGA Acceleration of Complex Pattern Trajectory Querying, in Proceedings 13th International Symposium on Spatial and Temporal Databases (SSTD), Munich, Germany, August 21-23, 2013.

[16] D.G. Perera, L. Kinfun, (2008) Parallel Computation of Similarity Measures Using an FPGA-Based Processor Array. In Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference, pp. 955-962

[17] S. Sood, & D. Loguinov. (2011, October). Probabilistic near-duplicate detection using simhash. In Proceedings of the 20th ACM international conference on Information and knowledge management (pp. 1117-1126). ACM.

[18] W. Vanderbauwhede, L. Azzopardi, M. Moadeli. (2009) FPGA-accelerated Information Retrieval: High-efficiency document filtering. In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference, pp. 417-422

[19] Ozan Yigit. Hash Functions. Available at http://www.cse.yorku.ca/~oz/hash.html, Accessed 6/15/2013

[20] J. Yan, Z. Zhao, N. Xu, X. Zhang, F. Hsu: (2012, May). Efficient query processing for web search engine with FPGA. In 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines (pp. 97-100).