Indian Journal of Science and Technology, *Vol 9(20), DOI: 10.17485/ijst/2016/v9i20/94694, May 2016*

ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Design and Implementation of GPU-based File Similarity Evaluation System

Yeong-Dae Kim[1], Byung-Kwan Kim[1], Sung-Bong Jang[2], Saang-Yong Uhmn[1] and Young Woong Ko[1*]

[1]Department of Computer Engineering, College of Information and Electronic Engineering, Hallym University, Chuncheon, Gangwon, 200-702, Republic of Korea; alexkim@hallym.ac.kr, kwani@hallym.ac.kr, suhmn@hallym.ac.kr, yuko @hallym.ac.kr

[2]Department of Computer Software Engineering, Kumoh National Institute of Technology, 61 Daehak-ro, Gumi, Kyoung-Buk, 730-701, Republic of Korea; sungbong.jang@kumoh.ac.kr

Abstract

Background/Objectives: Recently, storage systems and backup systems are popularly used and the number of duplicated data is increased drastically. To minimize data storage size and efficient use of network bandwidth, we proposed de-duplication systems and file similarity measurement schemes with GPGPU scheme. The GPGPUs are applied to file similarity measurement for computation speedup. **Methods/Statistical Analysis:** To cope with the problem accompanying the parallelization of the measurement, we compare two implementations with shared memory and preprocessing. In addition, we propose an alternative to Rabin fingerprinting algorithm to lessen the computational burden of the algorithm to the GPUs. We compare the performance of the systems in time elapsed for several files. **Findings:** First, we found through experiments that the preprocessing was slightly faster than the shared memory scheme for the overlapped region of consecutive data segments which were assigned to different cores. This region should be shared by two cores for fingerprinting. By adapting GPGPU parallelization with the preprocessing technique for file similarity measurement, the proposed system outperformed the systems with a multi-core CPU. Also, it gets faster for the bigger file. In addition, we made the system three times faster by adapting an alternative to Rabin fingerprinting algorithm. It eliminates the computational burden of the algorithm and provides comparable results to the system with the latter. **Improvements:** The procedure will be beneficial to de-duplication system in determining file similarity and finding duplicated regions of two files. We achieved speedup in the measurement of file similarity by parallelization on GP-GPUs with two methods for overlaps of consecutive data segments and an alternative fingerprinting algorithm.

Keywords: File Similarity, Fingerprinting, Parallelization on GPUs

1. Introduction

Thanks to recent advancements in hardware technology, the price of storage systems required per gigabyte of data has decreased. However, the rapid growth of the volume of digital data has resulted in an increase in the cost for investment in a large number of systems. Recently, there are much attention to data duplication technology to reduce the cost required for data storage and management. The key concept of data duplication is to remove the duplicate part of data by comparing a file with other existing files stored in a server in order to store only unique part of the data, rather than simply store the entire data in a storage system. File similarity evaluation schemes are underlying scheme in data deduplication[1–4]. In data de-duplication, file similarity scheme is very effective tools to find and eliminate duplicated data blocks[5,6]. We can find similar files using similarity evaluation and get rid of duplicated data blocks by using de-duplication algorithm. Through data de-duplication techniques, we can achieve the improvement of storage utilization and minimize network bandwidth. File similarity also can be used for digital forensics that compares suspected files to malicious software by using their hash values[7].

* *Author for correspondence*

The meaning of file similarity is a numerical value expressed in the percentage that how many chunks are the same to each other. For this reason, for the measurement of file similarity, the target files should be divided into a number of chunks and these chunks are compared to each other. Thus, the required time will be increased with the size of target files. To reduce the operation time, a new method of the measurement of file similarity based on GPU is suggested in the recent. GPU has a lot of cores roughly several hundreds of cores and these cores are connected in parallel to each other. GPU cores manipulates massive arithmetic operations within very short time, therefore we can reduce overall time for parallel computation.

There are several well-known research results for file similarity[8,9] are widely known file similarity evaluation algorithms. Both algorithms generate block-unit hash information for original files and complete a file digest based on the generated hash information. Sdhash is a tool that implements the similarity digest hashing algorithm that selects features with probabilistic methods using Shannon entropy instead of Rabin fingerprint[10]. In sdhash, a fixed-size, 256-byte Bloom filter was created. A maximum of 128 features are allocated per Bloom filter. All created Bloom filters are compared independently with the Bloom filters of other files, thereby evaluating file similarity through an average of entire compared values. Ssdeep is a tool that implements the Context Triggered Piecewise Hash algorithm that performs file similarity evaluation using two hash methods. Ssdeep creates a 170-byte file digest regardless of the original file size, which is an advantage because of the reduction in storage space. However, it has a limitation of large error in the file similarity evaluation results.

In this paper, we suggest a measurement method for file similarity using GPGPU parallel system. The proposed system adapt variable-length chunking scheme and accelerates hashing computation using GPU cores. In this paper, we provide efficient GPU computation scheme avoiding gray-area problem which occurs on multi-thread computation. By comparing the proposed method to CPU-based parallel algorithm, we can provide the usefulness of GPU parallel computation scheme. The rest of this paper is organized as follows. In Section 3, we explain the design principle of proposed GPU-based file similarity system and implementation details. In Section 4, we show performance evaluation result of the proposed system and we conclude and discuss future research plan.

2. GPU-based File similarity Evaluation System

The key idea of the proposed system is to minimize computation time of file similarity evaluation system using GPGPU scheme. To accomplish computation speedup, we divide the similarity computation module into several piece of GPU computation. This technique is much faster than traditional file similarity evaluation systems which assign computation on CPU cores. In the proposed system, we divide a file stream into variable-sized chunks, calculates a hash value for each chunk and finally computes similarity by comparing how many chunks exist within files. In this system, we accelerates chunking and hashing stage using GPGPU scheme. The chunking stage computes Rabin fingerprints for every offsets of a file to find regions with the target pattern and creates chunks. The hashing stage computes hash values for all chunks and puts them in a hash list. The final stage, the comparison, compares the list with the other of a file to find same values. The chunks with the same hash value are considered being identical. Figure 1 depicts the process of three stages for similarity measurement that the system utilizes for similarity between files. The similarity index is computed based on the differences between the lists and used as a file similarity measurement.

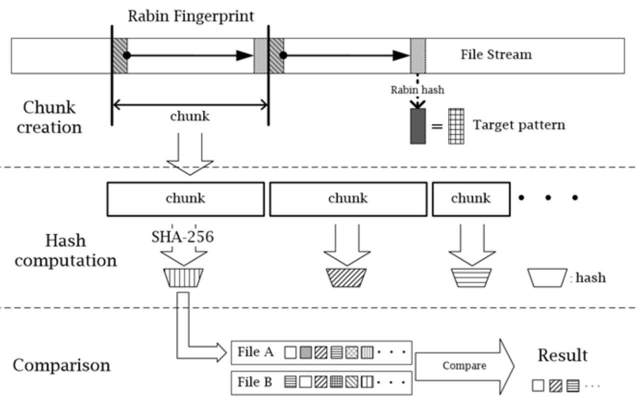

Figure 1. The process of similarity measurement.

2.1 System Architecture

Figure 2 shows the components of the system and workflow between them. Compared to traditional systems, the chunking stage is divided into two tasks for a gray area problem: Find anchors and marshal them. The problem occurs when a window for proper fingerprinting spans two segments for two threads. To cope with the problem, the results from the anchor finding stage are collected and arranged in the marshalling. Actually, the anchor finding determines all candidate locations for anchors without considering the maximum and minimum for the chunk size.

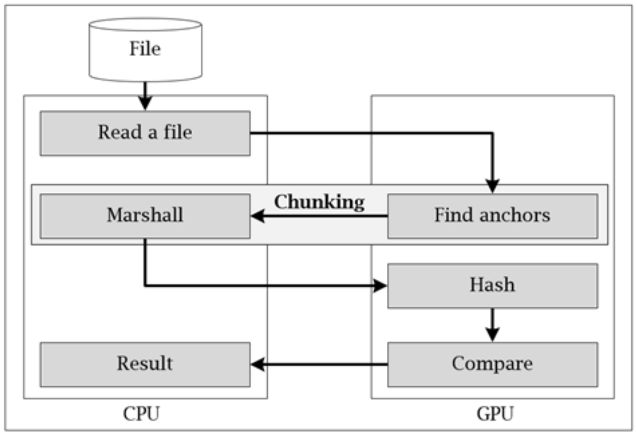

Figure 2. The flow of the proposed system.

An anchor is nothing but a boundary between consecutive chunks. Traditional systems have the minimum and the maximum sizes for a chunk. If all candidates are anchors, the accuracy of the measurement may improve at the costs of many small chunks that result in overheads. The system imposes the minimum chunk size to avoid this phenomenon. On the contrary, the maximum size is to limit the chunks when it does not find an anchor. With these values, the unavailability of previous anchors in the multi-thread system causes difference in the sets of chunks compared to a single-thread system. From this observation, the proposed system does not impose these values for the threads except the first one at the anchor finding. Because Rabin fingerprint computes a hash for data in a window shifting by byte and a file is divided into segments for threads, a window spanning consecutive segments that we call a gray area should be noticed. Our system adopted a shared memory and preprocessing and compared their performance by experiments.

2.2 Minimum and Maximum Sizes for a Chunk

In multi-thread systems, each thread manipulates a segment of a predefined size to create a list of non-overlapping chunks. It adopts Rabin fingerprint algorithm with the pattern of a marker. For the segments of the file except the first, it finds all occurrences of the marker.

Figure 3(a) depicts the segment S_2 with all occurrences of the marker. Several possible sets of chunks in the segment are shown in Figures 3(b), 3(c) and 3(d) depending on the last anchor in segment S_1. Figure 3(b) shows a case in which the first location of the marker becomes an anchor. Figure 3(c) is a case in which the second location becomes an anchor. The maximum chunk size is imposed for chunks in the case of Figure 3(d) in which there is an anchor before the first location. For this difference, the system finds all occurrences of the marker in all segments without considering the limits of the chunk size except the first. The marshalling stage handles the assignment of anchors to the occurrences of the marker. The CPU performs it from the segment S_1 with the limits so that the result of the assignments is identical to that by the single-thread system.

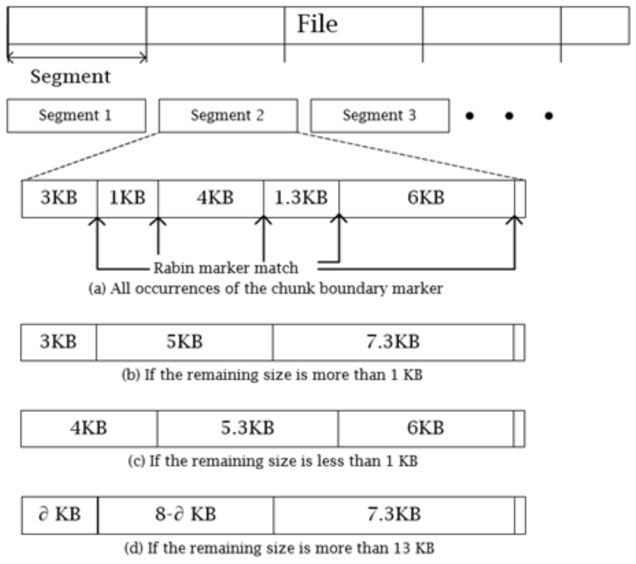
File
Segment
Segment 1
Segment 2
Segment 3
3KB
1KB
4KB
1.3KB
6KB
Rabin marker match
(a) All occurrences of the chunk boundary marker
3KB
5KB
7.3KB
(b) If the remaining size is more than 1 KB
4KB
5.3KB
6KB
(c) If the remaining size is less than 1 KB
∂ KB
8-∂ KB
7.3KB
(d) If the remaining size is more than 13 KB

Figure 3. Anchor position changes according to the parallel processing.

2.3 Gray Area

It was above-mentioned that there is another cause of the different assignments on the multi-thread system: The gray area problem. A file is divided into a set of non-overlapping segments that are assigned to the threads. The gray is the area spanning consecutive segments that should be considered for the proper fingerprinting. In Figure 4(a), all occurrences of the marker are indicated. Figure 4(b) depicts the result by the single-thread system with the limits. An anchor is not created at the location C in 4(b) because the size of the chunk between the location B and C is smaller than the minimum, 4 KB. Figure 4(c) shows a case in which an anchor B is not found because of the gray area problem: The shaded regions at the location B is not included in the segment S_2. These two systems generate different results even with identical algorithm, which should be minimized to improve the reliability of the result.

To cope with the problem, the last part of the previous segment, S_{i-1}, is included for fingerprinting the segment S_i. However, a same portion of the memory allocated among threads may cause performance degradation. To deal with this situation, we first adopted two methods, a shared memory and preprocessing gray areas and compared their performance.

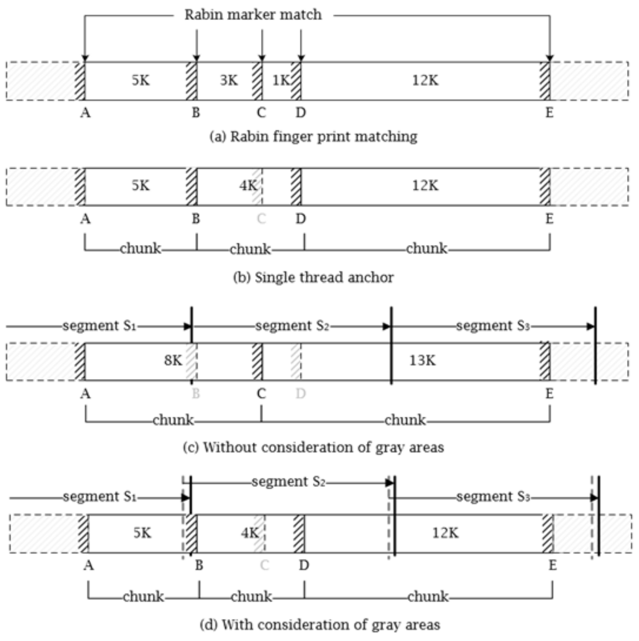

Figure 4. Gray area problem.

2.4 Lightweight Rabin-Fingerprint

The lightweight Rabin-fingerprint algorithm is a method for implementing fingerprints using polynomials over a finite field.

The algorithm is presented in Figure 5. It accepts three items: A segment of the input file and its size and the size of the window for fingerprinting. It generates a list of anchors and the number of them. Each thread finds anchors in the given segment by comparing the hashes with that of the patter by shifting a window by byte. When two hashes are identical, an anchor is assigned to that location.

```
Input:
  Segment File : File Segment per core
  Size : Segment File Size
  W : Window Size
Output:
  Anchor : Chunk match location
  $k$ : number of Anchor
begin
  Chunk (InputFile)
    for $I \leftarrow 0$ to Size do
      if $I < W$ then
        $Value_I = Value_{I-1} * 256 + File[I]$
      else
        $Value_I = Value_{I-1}$-$(256^W * File[I-W]) + File[I]$
      end
      if $Value_I == Patten$ then
        $Anchor_k = I + 1$
      end
    end
  end
end
```

Figure 5. Lightweight Rabin-fingerprint algorithm.

3. Experiment Results

As stated above, the proposed system generates anchors which are identical to that by the single-thread system. We carried out the experiments on a system with an i7-4770K CPU, 16 GB RAM and a GTX-980 with 4 GB memory running Windows 7 64 bit edition. The experiments were carried out in three scenarios. First, we tried to compare the effects of two methods for the gray area problem in execution time: A shared memory and preprocessing of the area. Second, we compared the performance of the system on a CPU and GPUs. Third, we compared the proposed algorithm with traditional Rabin fingerprint. We generated eight files randomly of 10, 20, 50, 100, 200, 300, 400 and 500 MB respectively. For each size of the file, we generated three files: The original and two others with the similarity of 10% and 60%, respectively.

3.1 The Gray Area Problem Solution

Figure 6 shows the results by the shared memory and the preprocessing for the gray area. For the files of size less than 300 MB, two methods performed similarly and showed difference for the files larger than 400 MB. Because the preprocessing technique outstripped the other, the former was utilized in the following experiments:

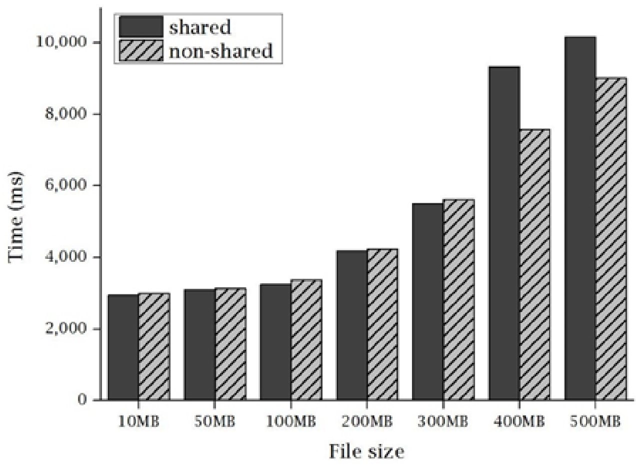

Figure 6. Performance comparison of two methods for the gray area problem.

3.2 n-Threads vs. GPUs

Figure 7 shows the difference in performance obtained by different number of threads. Although the single-thread system outperformed others for the file of 10 MB, the more the threads are used, the faster the job finished in general. The difference increases drastically as the size grows.

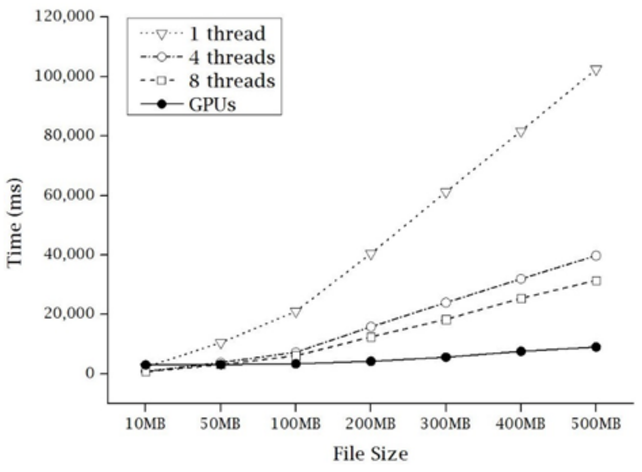

Figure 7. Similarity measurement time per threads.

3.3 Rabin-Fingerprint vs. Lightweight Rabin-Fingerprint

We applied lightweight Rabin-fingerprint to make the kernel lighter and suitable to GPUs and compared its performance with one with Rabin fingerprint by experiments with files of different sizes.

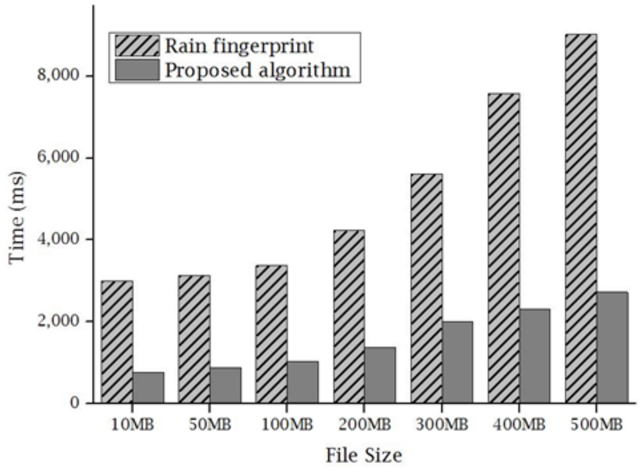

Figure 8. Comparison of file similarity measurement time.

Figure 8 shows the elapsed time by two kernels. The lightweight Rabin-fingerprint algorithm was three times faster in time for all the cases. Although it found different positions for anchors, it provided similar errors in similarity measurements.

4. Conclusion

In this paper, we analyze the performance of a file similarity measurement system based on VLC scheme through parallel processing using GPU. We applied GPU to file similarity system and obtained performance improvement compared to the system on conventional CPUs. To cope with the gray area problem, the data is preprocessed. The proposed algorithm for GPUs resulted in performance improvement with comparable errors. The future works include improvement in query capability of the system.

5. Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and future Planning (2014R1A2A1A11054160).

6. References

1. Meyer DT, Bolosky WJ. A study of practical deduplication. ACM Transactions on Storage (TOS). 2012 Jan; 7(4):1–14.

2. Lillibridge M, Eshghi K, Bhagwat D, Deolalikar V, Trezis G, Camble P. Sparse indexing: Large scale, inline deduplication using sampling and locality. FAST; 2009 Feb. p. 111–23.
3. Venish A, Sankar KS. Framework of data de-duplication: A survey. Indian Journal of Science and Technology. 2015 Oct; 8(26):1–7.
4. Anand SK, Karthigha M. A survey on removal of duplicate records in database. Indian Journal of Science and Technology. 2013 Apr; 6(4):1–6 .
5. Kulkarni P, Douglis F, LaVoie JD, Tracey JM. Redundancy elimination within large collections of files. USENIX Annual Technical Conference, General Track; 2004 Jun. p. 59–72.
6. Jung HM, Park SY, Lee JG, Ko YW. Efficient data de-duplication system considering file modification pattern. International Journal of Security and its Applications. 2012 Apr; 6(2):1–6.
7. Roussev V. An evaluation of forensic similarity hashes. Digital Investigation. 2011 Aug; 8:S34–41.
8. Roussev V. Data fingerprinting with similarity digests. Advances in Digital Forensics VI; 2010 Jan.p. 207–26.
9. Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digital Investigation. 2006 Sep; 3:91–7.
10. Muthitacharoen A, Chen B, Mazieres D. A low-bandwidth network file system. ACM SIGOPS Operating Systems Review. 2001 Oct; 35(5):174–87.