HARENS: Hardware Accelerated Redundancy Elimination in Network Systems

Kelu Diao*, Ioannis Papapanagiotou† and Thomas J. Hacker*
* Computer and Information Technology, Purdue University, West Lafayette, IN 47907, USA
† Platform Engineering, Netflix, Los Gatos, CA 95032, USA
Emails: diaokelu@gmail.com, ipapapa@ncsu.edu, tjhacker@purdue.edu

***Abstract*—With the increasing growth in the amount of information stored on remote locations and cloud systems, many service providers are seeking ways to reduce the amount of redundant information. Data deduplication can reduce the network traffic without loss of information, and consequently increase the available network bandwidth. However, due to the heavy computation overhead for detecting and reducing the redundant data, deduplication itself can become a bottleneck in high capacity links. In this paper, we propose a method named Hardware Accelerated Redundancy Elimination in Network Systems (HARENS). HARENS can significantly improve the performance of redundancy elimination in a network system by leveraging General Purpose Graphic Processing Unit (GPGPU) optimizations, as well as other optimizations such as the use of a hierarchical multi-threaded pipeline, Hash-Match, and memory efficiency techniques. Our results indicate that throughput can be increased by a factor of 14 compared to a native implementation of a network deduplication algorithm, providing a net transmission increase of up to 10.7 Gigabits per second (Gbps).**

***Index Terms*—Redundancy Elimination; Deduplication; GPU acceleration.**

I. Introduction

As more people engage in activities related to online services, network performance is becoming a big concern for service providers. Network performance is affected by factors such as available bandwidth, the type of the network traffic, and the distance of the data transfer. The issue is more prevailing in remote locations where fiber optic cables are not yet deployed. Hence, network redundancy elimination techniques can improve the throughput of the co-located traffic by removing duplicate chunks of data. At the same time, when caching is applied at the higher levels of the stack, it can offer benefits such as reducing the response time and requests to the server farm. On the other hand, a more fine grained redundancy elimination approach at the packet level can provide higher network throughput.

Generally, there are two methods for reducing network data redundancy, proxy-caches and Data Redundancy Elimination (*DRE*). *Proxy-cache* is a system in which an Internet object is cached in a proxy server, and a reference to the same object would result in the data being served by the cache and not from the server. Proxy caches are highly effective in HTTP traffic and in cases in which the content is not dynamic [1]. Another method, *DRE*, is a system comprised of two middleboxes, one at the backend and one at the aggregation part of the network.

9781-5090-1445-3/16$31.00

Both middleboxes perform the same operation based on the direction the traffic flows. For example, if the traffic flows from the server to the clients, then the backend middlebox stores chunks of data, and the corresponding hashes. Duplicate hashes are then represented with a representative pointer. When the mixed message of data and pointers arrive at the aggregation middleware, it performs the reconstruction of the data. Since not all pointers can fit in the memory of system, an LRU (or similar) replacement algorithm can potentially be used [2].

We implemented a generic deduplication method that could be deployed within proxy caches or DRE middleboxes. Although our approach would perform better using DRE because DRE middleboxes are more powerful and flexible than proxy caches. The redundancy elimination procedure we deployed is based on the following steps: 1) the data are fetched from a network socket, 2) the data are split into equal sized chunks (performing the proper queueing at the lower level), 3) a unique hash is computed for each chunk, and 4) the hash is checked against a hash table. A hit means that the data are redundant.

In practice, we found that the data fetching step is fast enough, with the only limitation being transferring the data from the socket to the user space. Hence, it does not require acceleration. There are two approaches for the object/packet chunking step: the legacy sliding window approach, and SampleBytes, a window sampling approach proposed by Aggarwal [3]. In the sliding window approach, a sliding window is put at the beginning of the data stream and slid to the end of the stream one byte a time. For each step, the sliding window algorithm computes the Rabin hash value of the window, and applies a certain sampling function (e.g. MODP, MAXP) on the hash values to choose a subset of the windows. The first bytes of the windows are denoted as the fingerprints, which divides the objects/packets into chunks. Another approach is *SampleBytes*, which picks up some sample windows and computes their Rabin hash values, and then applies a sampling function to choose fingerprints from the sample windows. We use the sliding window approach in our work because the SampleBytes approach has the potential to lose redundancy detection opportunities. In our work, the length of the sliding window is 32 bytes, and the sampling function we use is MODP. For the chunk hashing step, a general redundancy elimination algorithm computes the hash values of each chunk.

For the chunk matching step, the hash values computed in the previous step are used to detect redundant chunks, and to replace the redundant chunks with metadata that indicates the location of these chunks in the cache.

Most of the research in the area of DRE has leveraged Rabin fingerprinting. Although Rabin fingerprinting can be effective for slower legacy network connections, modern network interfaces are faster than 1Gbps, hence becoming a processing bottleneck. More specifically, through application profiling, we identified the time required to compute the hashes and the time spent for memory management of the algorithm is significant. EndRE [3] uses a sampling technique to bypass this issue, i.e. a subset of all windows are analyzed to avoid the computation overhead to analyze all the data.

Our work is inspired by the data storage deduplication research performed by Shredder [4]. More specifically, Shredder uses GPU acceleration with a multi-threaded pipeline to improve the disk deduplication, hence saving space on the disk and accelerating the performance of the disk interface. Shredder has demonstrated good performance improvements for content chunking, but is still not fast enough for Gigabit networks. We applied acceleration techniques, including GPU acceleration, a multi-threaded pipeline, and our Hash-Match algorithm. We improved the algorithm throughput up to 10.7 Gbps. While we investigate network deduplication, most of the proposed optimizations can also be used for disk-based deduplication.

To make network traffic redundancy elimination efficient, we divided the redundancy elimination algorithm into three steps, overlapping the execution of each step by using a multi-threaded pipeline technique, and applying optimization techniques for each step. The three main challenges addressed by our approach and our novel contributions are:

- **Multi-threaded Pipeline With Async Memory Transfer:** When the DRE algorithm is run serially, the system resources are stalled waiting for CPU, Memory I/O, Disk I/O, backplane etc. In our work, we used a multi-threaded pipeling along with asynchronized memory transfers. Hence, data can be transferred between the host memory and the CUDA device memory, without waiting for the kernel execution to complete. With this optimization, we could achieve significantly better performance.
- **CUDA Acceleration & Optimization:** As mentioned above, Rabin hash is the most common technique for object and packet chunking. It is computationally intensive to perform this operation at Gbps speed. We optimized the chunking algorithm by a factor of 14 by leveraging internal CUDA optimizations. We achieved that by utilizing most of the Stream Multiprocessors (SMs) of the GPU and the availabile threads per core. To achieve maximum efficiency, we applied the asynchronized memory transfer technique and balanced the resource allocation to obtain 93.89% actual occupancy, and 100% theoretical occupancy of the GPU.
- **Hash-Match:** In DRE systems, a hash table is used to store the hash value of each chunk. However, at high throughput even thread-safe hash-tables can become a bottleneck due to thread-locking. Thread-locking in hash tables can be highly affected by the access patterns. Hence, we developed a *Hash-Match architecture* for chunk hashing and matching. Hash-Match is inspired by Hadoop Map-Reduce. The Hashers compute a hash value for each chunk, and shuffle it to the Matchers. Then the Matchers match the hash values to discover redundancy that can be exploited.

The remainder of this paper is organized as follows. Section II discusses related work and the limitations of these efforts. Section III present the details of our algorithm and optimization techniques. Our evaluation approach and experimental results are covered in Section IV. We also summarized the results and impacts of our experiments in this section. In Section V, we conclude and discuss our future work.

II. Related Work

A. Content-Based Data DeDuplication

Fingerprinting is a method that uses a small signature to represent a larger data object. Prior studies have proposed two approaches related to the fingerprinting method: the Fingerprint Expansion (FPE) method and the Fingerprint Partition (FPP) method. The FPE method is applied in the work of Manber [5], Spring [2], and Schleimer [6]. This method treats the fingerprints as anchors in the data series, matches the anchors between two objects/packets, and expands the anchors to find the maximum chunk match when two anchors match. The FPP method is applied in the work of Muthitacharoen [7], Rhea [8], Tolia [9], Pucha [10], Anand [11], Aggarwal [3], Bhatotia [4], and Papapanagiotou [12]. The FPP method treats the fingerprints as break points in the data series, divides the data series into chunks from the break points, and matches the chunks within and across objects/packets based on the non-collision hash value (normally SHA) of each chunk. The FPE method would find a longer match than FPP once it finds an anchor match, but it is a slow technique because the program needs to compare every byte before and after the two matching anchors until it reaches different bytes. We applied the FPP method, because it is much faster than FPE, and speed is the main concern in our work.

There are also different methods that can be used for selecting fingerprints. A simple approach would be to choose a fingerprint every several bytes. However, a slight change in the data would make a significant change in the fingerprint set. For example, if there is a byte inserted in the data series, the fingerprints after that insertion would all be shifted left by one byte. This problem could be solved by using content-based chunking, which is categorized as MODP (MOD p) and MAXP (local MAX in p bytes) by Anand [11]. Besides MODP and MAXP, Aggarwal [3] proposed a method named SampleByte in the work EndRE. MODP is a method that chooses a subset of the fingerprints, in which every fingerprint is selected based on whether the fingerprint modulo p (a predetermined number) is equal to another predetermined

number (0 as in our work). Manber [5] applied this technique in his work focused on finding similar files in a large file system. Spring and Wetherall [2] first proposed applying this technique for redundancy elimination in network systems. Bhatotia [4], Muthitacharoen [7], Papapanagiotou [12], Pucha [10], Rhea [8], and Tolia [9] also adopted the MODP method or used the tools that adopted the MODP method in their work. The MAXP method selects a local maximum (or minimum) in every continuous block with length p. Anand [11], and Schleimer [6] adopted the MAXP method in their work. The SampleByte method proposed by Aggarwal [3] adopted the ideas from both MODP and MAXP. This method skips $p/2$ bytes every time when a fingerprint is chosen. Because MODP uses a pre-determined value p to filter fingerprints, and the network data could be clustered, it could miss some redundancy detection opportunities. SampleByte would lose even more redundancy detection opportunities than MODP. However, our work adopted the MODP method because it can provide approximately the same detected redundancy rate as MAXP, according to the experiment conducted by Aggarwal [3], and partitions data into fewer chunks, which would make the critical steps faster.

In other related work [13], Anand described SmartRE, a software based system that seeks to eliminate redundancy at a packet level across a network of systems using middleboxes. The goals of our work is similar to Anand, but differs in two ways: 1) we use GPUs to speed analysis of the data; and 2) we do not restrict the data chunk size to packets. The benefits of our approach are that it seeks to exploit the computational power of widely available GPUs to help reduce network traffic, and it also could be applied to data chunks beyond the size of a packet.

B. Efforts of Acceleration

Previous studies focused on accelerating the redundancy elimination algorithm to increase the speed of deduplication. Anand [11] and Schleimer [6] applied MAXP, which is faster than MODP. Anand [11] applied a Bloom Filter to accelerate chunk matching, but the Bloom Filter has a relatively high false positive probability, does not have the information about chunk location, and has difficulty in the case of deletion. Aggarwal [3] proposed the SampleByte method, which loses redundancy detection opportunities. Bhatotia [4] used a GPU to accelerate the Rabin fingerprint process, but it is mainly targeted for redundancy elimination in incremental storage, and it did not address accelerating chunk hashing and matching procedures. Papapanagiotou [12] and [14] proposed a hybrid method using both proxy caches and a redundancy elimination module. In our work, we mainly focus on accelerating the whole process of redundancy elimination, which is complementary to the work of Papapanagiotou [12].

Bianco [15], and Kolb [16] proposed a data de-redundancy approach using Map-Reduce. But their work applied a brute force approach for object chunking and duplicate chunk matching, which is inefficient, and used existing Map-Reduce tools, which may not be perfectly suited for the redundancy elimination task. Our work uses a Rabin fingerprinting algorithm for object/packet chunking, which is much faster. We also implemented a Hash-Match approach, which is similar to this Map-Reduce algorithm, to eliminate the key-value pair process in Map-Reduce, because the value also performs as the key in our algorithm. It also eliminates the merge/sort step, and passes the output of a Hasher to Matchers immediately instead of passing a list after the Hasher is done. This real-time Hash-Match communication approach is the key of acceleration.

III. Methodology

In this section, we first introduce an overview of the data deduplication algorithm we used for our acceleration techniques. Next, we describe the optimization techniques we applied to accelerate the algorithm.

A. Algorithm Overview

The common procedure for a redundancy elimination algorithm typically consists of three steps: object/packet chunking, chunk hashing, and chunk matching.

1) Packet/Object Chunking:
In this step, the program reads in the traffic stream either online or off-line, applies a sliding window to scan through the whole input stream, and marks the beginning of a window as a fingerprint based on the MODP rule. The fingerprints divide the stream into chunks.

Algorithm 1 MODP Rabin Fingerprinting Algorithm

```
window ← 0
fingerprints ← empty set
while window + δ < stream length do
  hashValue = ComputeRabinHash(stream[window], δ)
  if (hashValue mod σ) = 0 then
    fingerprints.Add(window)
  end if
  window ← window + 1
end while
```

The procedure of the MODP Rabin fingerprinting algorithm for object/packet chunking used in this step is briefly shown in Algorithm 1. During initialization, $window$ is placed at the beginning of the stream and $fingerprints$ is an empty set. Then this $window$ is slid to the end of the stream, moving right by one byte in each step. In each step, the algorithm also computes the Rabin hash value of the stream that is covered by the $window$ (we use δ as the size of the $window$ for the rest of this paper). If the hash value modulo a given number σ equals 0, the Rabin fingerprinting algorithm marks the beginning of this $window$ as a $fingerprint$ and then adds it into the set. We adopted an optimized form of the Rabin hash algorithm introduced by Broder [17], which takes a reference table along with the data stream as the input, and provides better performance than the original algorithm.

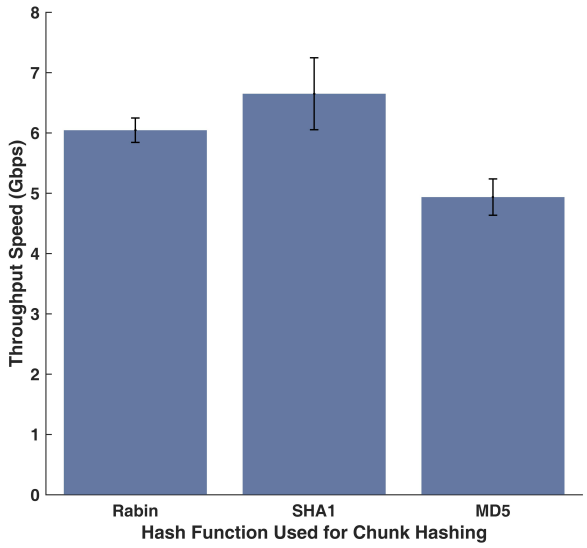

Fig. 1. Chunk hashing throughput speed using Rabin, SHA1, and MD5 hashes (single threaded)

2) Chunk Hashing:

Having partitioned the stream into chunks, a hash value is then computed for each chunk. The hash function used for chunk hashing needs to be effective and efficient. A hash function that can be used in this case should satisfy Equation 1, which means the number of objects and number of hash values are within the same order of magnitude, so that the hash collision probability should be close to 0. The Rabin, SHA1, and MD5 hashes are all good choices for chunk hashing, because they are all light-weight and have a low hash collision probability.

$$\frac{\big||Hash(S)| - |S|\big|}{|S|} \approx 0 \tag{1}$$

To assess the throughput and collision probability of the Rabin, SHA1, and MD5 algorithms, we tested the throughput (shown in Fig. 1) and the hash collision rate of three hashing methods on YouTube video data we generated using the trace file collected by Zink [18], which is publicly available in the UMass trace repository [19]. The experiment was conducted on 6 traces each contained 2 GB data for each hashing method. We found that the Rabin hash had a 0.1% – 0.2% collision rate and observed no hash collision using SHA1 and MD5. From our results, we found that the SHA1 hash had much better throughput performance than the other two hashing methods. We found that the Rabin hash could occasionally encounter hash conflicts, while SHA1 and MD5 did not demonstrate this problem. Therefore, we chose SHA1 as our chunk hashing method.

3) Chunk Matching:

In this step, our algorithm stores the hash values computed in the previous step in a hash table to represent the data chunks stored in cache. It reports a duplicate chunk when it discovers a hash value that already exists in the hash table. Hash conflict should not be a problem because the hash conflict rate of SHA1 is small enough to be safely ignored according to Spring [2], and the least recently used hash values in the table are replaced as the cache is managed.

A cache chunk replacement management method was necessary, given the limited amount of available memory. We chose Least Recently Used (LRU) as our chunk replacement algorithm because we believe that repetitive patterns are demonstrated, which means the redundant chunks are most likely to be redundant copies of recently accessed data locally.

B. CUDA Acceleration

In the object/packet chunking step, we compute the Rabin hash for each window, which is (stream length - δ + 1) windows, which makes this step very computationally intensive. But it also has a great feature in that the Rabin hash function computes the hash value using exactly δ bytes as input, and will follow exactly the same mathematical procedure, which would take about the same amount of time. The features of the MODP Rabin fingerprinting algorithm in this step makes it a perfect candidate for CUDA acceleration. Besides, we also applied CUDA optimization techniques to make the best advantage of CUDA.

Shared memory access is much faster than global memory access in the CUDA architecture, which makes it beneficial to transfer data to shared memory prior to computation. Besides, we have multiple threads reading from the same memory location because of the overlap of windows, which causes access conflicts and hence only one of the threads that access the same memory slot would run while the others are waiting. So we made two copies of some input data in shared memory and aligned the data to avoid half of the access conflict as well as improve the memory bandwidth. It is impossible to make more copies of data in shared memory because of the limited size of shared memory.

The CUDA kernel instructions could be stalled for many reasons. We ran an un-optimized CUDA algorithm in Visual Studio and generated Figure 2, which shows the potential reasons that could slow down CUDA kernel execution. As shown in Figure 2, the issues that could affect the performance includes: the speed of instruction fetch and constant miss, which requires well arranged code order to take full advantage of the cache; execution dependency and memory dependency, which requires avoidance or reduction of the scenario that the threads in GPU depends on one another; and memory throttle and pipe busy, which requires avoidance or reduction of the scenario that multiple threads access the same slot of data.

Moreover, in CUDA compute capability 3.5, the maximum threads per multiprocessor is 2048; the maximum shared memory per multiprocessor is 49152 bytes; and the maximum register file size per multiprocessor is 65536 bytes [20]. To achieve the most threads per multiprocessor, we need to balance the size of shared memory and registers allocated per multiprocessor. In this method, we transfer data from global memory in the GPU to two shared memory slots in each block, which stores two shifted copies of input data, and used the

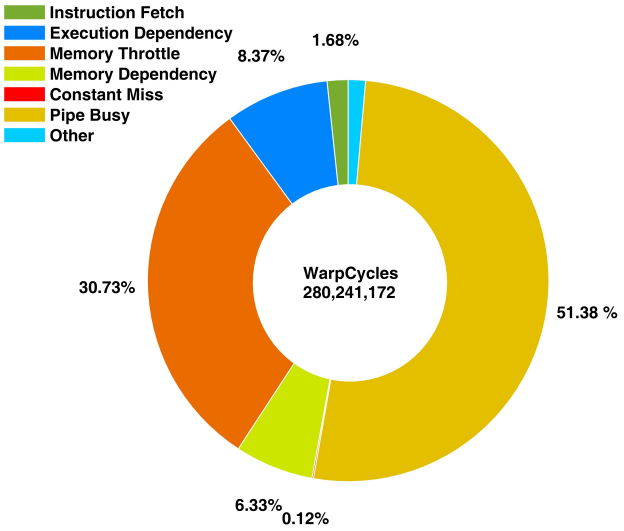

Fig. 2. Distribution of stalled warps cycles of each issue stall reason before optimization

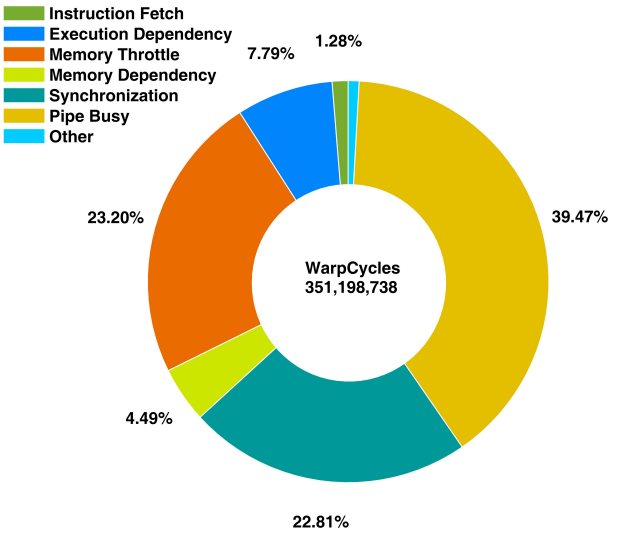

Fig. 3. Distribution of stalled warps cycles of each issue stall reason after optimization

CUDA kernel registers for the other memory usage. The GPU threads are synchronized after transferring data into shared memory, so that a large part of stall reasons for the threads are switched from memory throttle, memory dependency, and pipe busy to synchronization, as shown in Figure 3. Although more warp cycles were stalled in the optimized code, overall the execution time was reduced by around 10% because a) the average stall time was reduced, and b) a large part of issue stalls (synchronization) were under control. A GPU with CUDA architecture consists of multiple Streaming Multiprocessors (SM). Each SM has its independent instruction unit,

TABLE I
GPU OCCUPANCY

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks	-	4	16
Active Warps	60.09	64	64
Active Threads	-	2048	2048
Occupancy	93.89%	100%	100%
Warps			
Threads/Block	-	512	1024
Warps/Block	-	16	32
Block Limit	-	4	16
Registers			
Registers/Thread	-	17	255
Registers/Block	-	12288	65536
Registers/SM	-	49152	65536
Block Limit	-	5	16
Shared Memory			
Shared Memory/Block	-	1034	49152
Shared Memory/SM	-	4136	49152
Block Limit	-	38	15

constant cache, texture cache, shared memory, and multiple Streaming Processors (SP), which have independent registers and share the "shared memory". To fully utilize the GPU, we expect the occupancy of each SM to be as high as possible. Table I shows the GPU occupancy of our program. According to the NVIDIA developers' guide [21], occupancy is "the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU", and theoretical occupancy is "the maximum occupancy given the execution configuration". Our work balanced the usage of registers and shared memory to increase the number of active warps and obtained 93.89% actual occupancy and 100% theoretical occupancy.

C. Hash-Match

In the chunk hashing and chunk matching step. A simple approach is to compute the hash value of each chunk, store the hash values in a hash table, and then detect duplicate chunks by referring to the hash table. This approach, however, requires a large number of almost-random memory accesses from repeated reads of the hash table, which requires an excessive amount of time. Another idea is to launch multiple threads to execute chunk matching tasks, which would provide much better performance. But there exists an upper boundary of throughput in the approach of simply launching multiple threads to execute chunk matching tasks, because there would be a massive amount of time spent waiting to acquire locks on the hash table, as well as the data structure that stores chunk hashing results to prevent data race conditions. We developed our Hash-Match algorithm inspired by Hadoop Map-Reduce with the goal of parallelizing data chunk hashing (using Hasher) and data chunk hash matching (using Matcher). We used our Hash-Match architecture, which significantly improved the performance of these two steps.

In the Hash-Match architecture, we launched μ threads of chunk hashing as the Hashers, and γ threads of chunk matching as the Matchers. Each Matcher i maintains its own

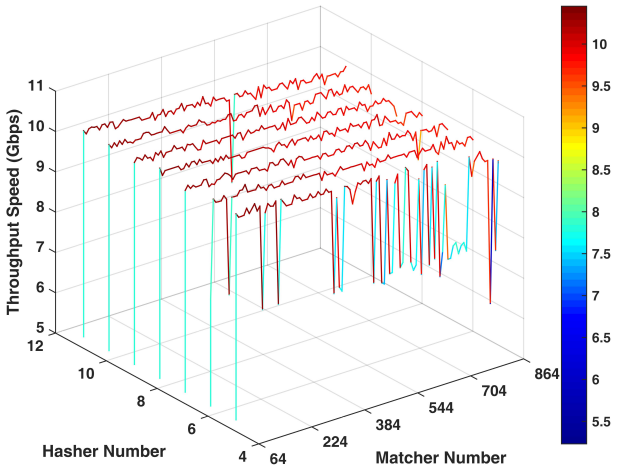

Fig. 4. Influence of the number of Hashers and Matchers on performance

hash table, which only contains hash values h such that $(h \mod \gamma) = i$. So that the hash tables perform as a implicit shuffle step between Hashers and Matchers. We conducted an experiment using a varying number of Hashers and Matchers to measure the influence of Hashers and Matchers on the throughput speed of our algorithm. As shown in Fig. 4, the throughput speed fluctuates when there are four Hashers, but stabilized when there are six or more Hashers. The throughput speed fluctuates in a small range when the number of Matchers is less than 200, and decreases slowly when the number of Matchers exceeds 200. An explanation for this is that the benefit of having more Matchers cannot make up the cost for the operating system to switch among the threads when there are too many Matchers running simultaneously. Our experimental results showed that the algorithm performed best with $\mu \in [6, 16]$ and $\gamma \in [64, 200]$. We used $(\mu, \gamma) = (8, 64)$ in our experiments.

D. Multi-threaded Pipeline

Pipelining is the work mode in which threads labor simultaneously, and pass their output to the next threads as input. To increase the throughput of our algorithm, we used a multi-threaded pipeline technique to minimize the idle time of each device.

Furthermore, we used asynchronous memory transfer, which allows the program to simultaneously execute CUDA kernel functions and transmit data between the CUDA kernel and host memory during the CUDA accelerated object/packet chunking step. This CUDA stream technique allows the kernel to process data in the order it is sent by the program. The program would be transmitting results from the CUDA kernel to the host memory, and input data from host memory to the CUDA kernel while it is executing a CUDA kernel function. We use this technique as the second layer pipeline within the pipelining of the whole process.

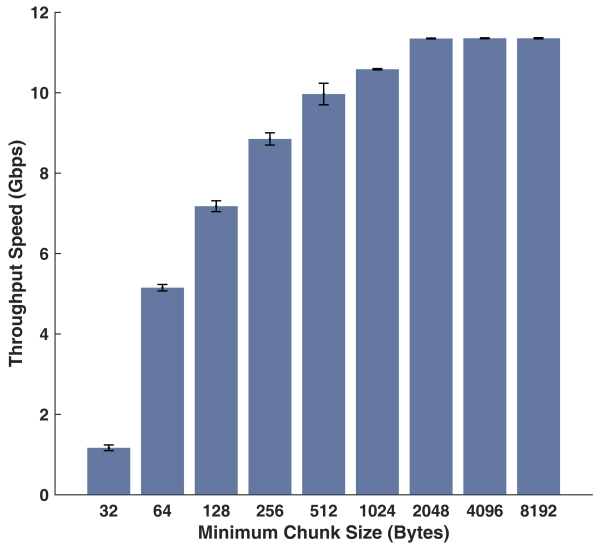

Fig. 5. Throughput speed of HARENS with varying chunk size

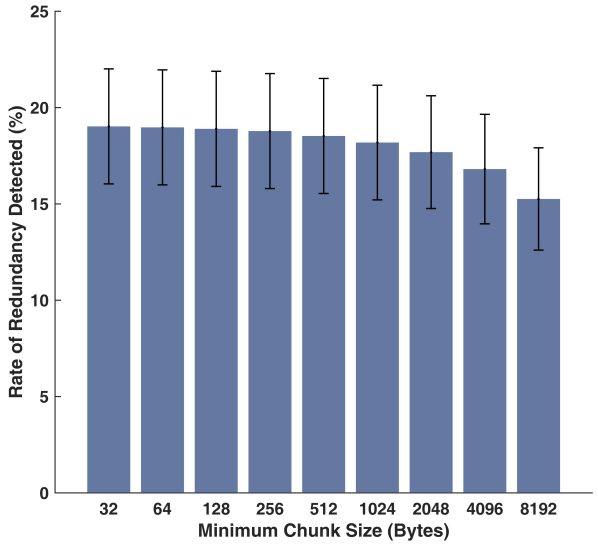

Fig. 6. Redundancy rate detected with different restrains on chunk size

E. Optimal Chunk Size

The chunk size in our algorithm is controlled by two parameters: σ and the minimum chunk size. σ is used to select fingerprints on the packet/object chunking step. If the Rabin hash value modulo σ equals 0, the Rabin fingerprinting algorithm marks the beginning of current $window$ as a $fingerprint$. We skip one fingerprint if its distance from the previous partition is less than minimum chunk size. In this research, we set σ equal to minimum chunk size for simplicity of analysis. As is shown in Figure 5 and Figure 6, the throughput peaked at a chunk size of 2048 bytes, and there is not a statistically significant difference in the reduction rate

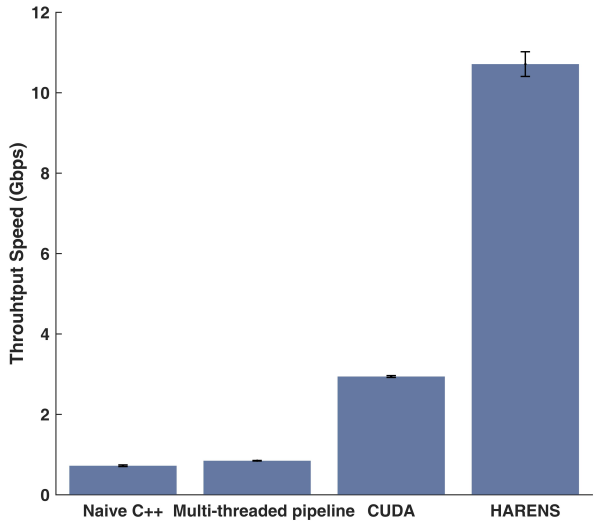

Fig. 7. Throughput speed of the four methods

going from a chunk size of 1024 bytes to 2048 bytes.

IV. Evaluation

We evaluated our method by comparing the throughput and accuracy of our program with three other benchmark programs [1]. The algorithms we evaluated are:

- A naive C++ implementation of the Rabin fingerprint data de-duplication algorithm
- A multi-threaded pipeline accelerated algorithm
- A CUDA accelerated algorithm
- Our approach, HARENS

We evaluated these methods on a computer with an Intel(R) Core(TM) i7-5930K CPU @ 3.50 GHz, with 12 cores and 32.0 GB RAM, the Operating system was Windows 8 64-bit. The GPU installed in this machine was an NVIDIA Tesla K40c.

We generated our experimental data based on YouTube traces collected by Zink [18], which is publicly available in the UMass trace repository [19]. We wrote a batch downloading script of YouTube video[2], inspired by Ficano [22], and concatenated the video files as our input in the same order the video files appeared in the YouTube trace collected by Zink (with repeats when they are present in the trace). We did not intend to simulate the downloading, where the videos might be downloaded simultaneously and overlap with each other. As the size of video files (normally 10-20 MB) is relatively small in terms of size of the cache (8-16 GB), the video files would not compete for cache space. Therefore, the order of packets of the videos would not affect the experimental results.

In Fig. 7 we show the throughput speed of the four methods we evaluated. According to this graph, the multi-threaded pipeline accelerated algorithm demonstrated about a 17% throughput improvement over the naive C++ implementation, because it overlapped the execution time of each step. The improvement was not significant because each step was still time consuming. The CUDA accelerated algorithm demonstrated about a 3 times throughput improvement over the naive C++ implementation, because it shortened the time of object/packet chunking. Our method HARENS demonstrated about a 14 times throughput improvement over the naive C++ implementation, because it not only shortened the execution time of object/packet chunking, chunk hashing, and chunk matching, but also overlapped the execution time of each step.

[1]The source code can be found in https://github.com/ipapapa/HARENS

[2]The code can be found in https://github.com/KeluDiao/gotube

TABLE II
Redundancy Rate Detected

	Mean	Standard Deviation
Naive C++	18.50	3.04
Multi-threaded pipeline	18.50	3.04
CUDA	18.50	3.03
HARENS	17.92	2.96

In Table II, we show the detected redundancy rate of the trace file using the four methods. The redundancy rates detected by the naive C++ implementation, the multi-threaded pipeline accelerated approach, and the CUDA accelerated approach are similar. Our method detected slightly less redundancy than the other three methods, but it is a side effect that we can tolerate because of the huge throughput speed increment.

Our HARENS approach provided a significant improvement in throughput speed compared with a singular approach of using a CPU, multi-threaded pipeline, CUDA alone. Our hybrid approach simultaneously exploits the technologies available on the system, and represents a novel integrated technique that can be used for increasing data throughput and reducing network data redundancy.

V. Conclusions

In this paper we presented HARENS, an efficient approach we developed for redundancy elimination in network systems. We divided the process of redundancy elimination into four steps: fetching data, partitioning packets/objects chunks, computing SHA1 hash values for chunks, and matching chunks by comparing hash values. Each step was treated as a Finite State Machine (FSM), which ran separately but shared data buffers to synchronize with each other. These FSMs performed the role of workers in a pipeline, which can keep the tasks running simultaneously. For the object/packet chunking step, we used a GPU to accelerate the Rabin fingerprinting algorithm. In this step, we made use of shared memory to improve memory bandwidth, applied asynchronized memory transfer to minimize the blocking time of the kernel instructions, and balanced the usage of GPU registers and shared memory to activate the largest possible number of threads in execution. Hence we achieved the highest possible theoretical GPU occupation. For the chunk hashing and chunk matching step, we applied our Hash-Match architecture which distributed and scheduled the

work load of these two steps in thousands of threads which improved the overall performance by about a factor of 14 times.

There are several interesting avenues for future work. We could explore adopting the hybrid method introduced by Papapanagiotou [12], making our redundancy elimination module work with a proxy cache module, to see if it can improve performance. We could also explore making better use of hardware and reduce the time consumption of synchronizing the FSMs. Besides, we found that there is significant time spent in lock acquisition and release. A lock-free method would be another good topic for further investigation.

ACKNOWLEDGMENT

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

REFERENCES

[1] I. Papapanagiotou, E. M. Nahum, and V. Pappas, "Smartphones vs. laptops: Comparing web browsing behavior and the implications for caching," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 2012, pp. 423–424.

[2] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 87–95, 2000.

[3] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "EndRE: An End-system Redundancy Elimination Service for Enterprises," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 28–28.

[4] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: Gpu-accelerated incremental storage and computation," in *File and Storage Technologies*, 2012, p. 14.

[5] U. Manber, "Finding Similar Files in a Large File System," in *USENIX WINTER 1994 TECHNICAL CONFERENCE*, 1994, pp. 1–10.

[6] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 76–85.

[7] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-bandwidth Network File System," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, Oct. 2001.

[8] S. C. Rhea, K. Liang, and E. Brewer, "Value-based Web Caching," in *Proceedings of the 12th International Conference on World Wide Web*, ser. WWW '03. New York, NY, USA: ACM, 2003, pp. 619–628.

[9] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil, "An Architecture for Internet Data Transfer," in *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 19–19.

[10] H. Pucha, D. G. Andersen, and M. Kaminsky, "Exploiting Similarity for Multi-source Downloads Using File Handprints," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 2–2.

[11] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in Network Traffic: Findings and Implications," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 37–48, Jun. 2009.

[12] I. Papapanagiotou, R. D. Callaway, and M. Devetsikiotis, "Chunk and object level deduplication for web optimization: A hybrid approach," in *Communications (ICC), 2012 IEEE International Conference on*. IEEE, 2012, pp. 1393–1398.

[13] A. Anand, V. Sekar, and A. Akella, "SmartRE: An architecture for coordinated network-wide redundancy elimination," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, ser. SIGCOMM '09. New York, NY, USA: ACM, 2009, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/1592568.1592580

[14] R. Callaway and I. Papapanagiotou, "Dynamic caching module selection for optimized data deduplication," Sep. 18 2014, US Patent App. 14/059,959. [Online]. Available: http://www.google.com/patents/US20140281262

[15] G. Dal Bianco, R. Galante, and C. A. Heuser, "A fast approach for parallel deduplication on multicore processors," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1027–1032. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982411

[16] L. Kolb, A. Thor, and E. Rahm, "Dedoop: Efficient deduplication with hadoop," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1878–1881, Aug. 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367527

[17] A. Z. Broder, "Some applications of Rabins fingerprinting method," in *Sequences II*. Springer, 1993, pp. 143–152.

[18] M. Zink, K. Suh, Y. Gu, and J. Kurose, "Watch global, cache local: Youtube network traffic at a campus network: measurements and implications," in *Electronic Imaging 2008*. International Society for Optics and Photonics, 2008, pp. 681 805–681 805.

[19] UMass Trace Repository. http://traces.cs.umass.edu/index.php/ Network/Network. Accessed:2015-10-09.

[20] "CUDA Toolkit Documentation," https://docs.nvidia.com/cuda/cuda-c-programming-guide/#kernels, accessed:2015-09-24.

[21] "NVIDIA Nsight Development Platform, Visual Studio Edition User Guide," http://http.developer.nvidia.com/NsightVisualStudio/2.2/Documentation/UserGuide/HTML/Content/Profile_CUDA_Settings.htm, accessed:2016-05-24.

[22] N. Ficano, "pytube," https://github.com/nficano/pytube.git, 2015.