# Redundant Network Traffic Elimination with GPU Accelerated Rabin Fingerprinting

Jianhua Sun, Hao Chen, *Member, IEEE*, Ligang He, *Member, IEEE*, and Huailiang Tan

**Abstract**—Recently, redundant network traffic elimination has attracted a lot of attention from both the academia and the industry. A core challenge and enabling technique in implementing redundancy elimination is to perform content-based chunking, which typically involves the computationally heavy Rabin fingerprinting algorithm. In this paper, we propose a GPU-based implementation of Rabin fingerprinting to address this issue. To maximize performance gains, a diverse set of optimization strategies, such as efficient buffer management, GPU memory hierarchy optimization, and balanced load distribution, is proposed by either exploiting the intrinsic hardware features or addressing domain-specific challenges. Extensive evaluations on both the overall and microscopic performance reveal the effectiveness of the GPU-accelerated Rabin fingerprinting algorithm, and we can achieve up to 40 Gpbs throughput on a GTX 780 card. The throughput shows 1.87× speedup against the state-of-the-art using comparable hardware. In addition, although some optimization designs are specific for the problem, techniques proposed in this work including the indexed compact buffer scheme and approximate sorting would also be beneficial and applicable to other network applications leveraging GPU acceleration.

**Index Terms**—Rabin fingerprinting, GPU, redundancy elimination, data deduplication, approximate sorting

## 1 Introduction

With the ever-growing popularity of Internet services, we have seen steady and continuous growth in Internet traffic in the past decade. Eliminating the transfer of redundant data can lead to significant savings in terms of network resources. As a result, redundant network traffic elimination has attracted a lot of attention from both the academic and the industry [2], [3], [4]. Research proposals and products from major commercial vendors (such as Riverbed, Juniper, Cisco) typically use protocol-independent redundancy elimination (RE) [19] to perform data deduplication. Such RE systems are typically applied at the TCP/IP layer by using middle-boxes placed at two communicating ends, which cooperatively cache payloads from network flows and reconstruct original content by fingerprinting traffic payloads and identifying matches in the cached data. This process is usually independent of applications/protocols and transparent to end hosts. Thus, bandwidth savings can be achieved for all the traffic between the two middle-boxes.

The aforementioned middle-boxes are always advertised as "WAN optimizers", which has been increasingly deployed in large scale production systems. The effectiveness of WAN optimization relies on the detection of duplicate content. The most effective and widely used technique for similarity detection is called *content-based chunking*, which was pioneered by the work of LBFS file system [11] and proposed to identify repeated byte ranges between packets transferred in the network [19]. Content chunking is the process of partitioning a byte-string into non-overlapping sub-strings called chunks, whose boundaries are determined by the content of the data instead of fixed offsets. Rabin fingerprinting first introduced in [12] perhaps is the most popular method for content-based chunking. Although the Rabin fingerprinting algorithm plays a critical role in content chunking, it is computationally demanding. The chunking operation needs to scan every byte of the input string to compute a fingerprint over a sliding window of the input data. Therefore, addressing this computational bottleneck to balance the detection precision and throughput becomes an inevitable design issue in real systems, including WAN optimizers and storage deduplication systems.

Some previous studies [7], [10], [16], [20] have shown that GPUs can significantly improve the performance of network applications including software routers [7], SSL cryptographic operations in web servers [10], pattern matching in intrusion detection systems [20], and network coding [16]. Due to the massively data-parallel computing model of modern GPUs, offloading computation-intensive operations to GPUs is a natural choice to address the performance bottleneck for many applications. Motivated by the success of these prior studies and the performance issues faced by current RE systems, in this paper, we will reveal the potential of GPUs in improving the performance of the chunking process in RE systems. In particular, we focus on parallelizing the Rabin fingerprinting algorithm with the GPU. To the best of our knowledge, this work is the first to quantify the GPU-based acceleration of Rabin fingerprinting with detailed algorithmic design and architecture-specific optimizations in the context of redundant network traffic elimination.

Based on the CUDA framework, our implementation of the GPU-based Rabin fingerprinting shows significant improvement as compared to the CPU-based counterparts,

- *J. Sun, H. Chen, and H. Tan are with the College of Computer Science and Electronic Engineering, Hunan University, ChangSha 410082, China. E-mail: {jhsun, haochen}@aimlab.org, tanhuailiang@hnu.edu.cn.*
- *L. He is with the Department of Computer Science, University of Warwick, Coventry CV47AL, United Kingdom, and the College of Computer Science and Electronic Engineering, Hunan University, ChangSha 410082, China. E-mail: liganghe@dcs.warwick.ac.uk.*

*Manuscript received 10 Nov. 2014; revised 18 June 2015; accepted 5 Aug. 2015. Date of publication 25 Aug. 2015; date of current version 15 June 2016. Recommended for acceptance by J. L. Träff.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TPDS.2015.2473166*

enabling traffic payload chunking at the rate of 40 gigabit per second. Concretely, our fully-optimized implementation are up to 40× faster than a single-threaded CPU version, and 4.2× faster than a baseline implementation on GPU, reaching a maximum throughput of about 40 Gbps. In addition, our prototype achieves a speedup of about 10 × compared to the SampleByte fingerprinting scheme in [2], which selects the chunks based on the most redundant bytes that is determined by training the fingerprinting algorithm on sample traffic and recording the predefined values in a lookup table.

We make the following contributions in this work:

- Overall, we reveal the benefit and potential of employing GPU to accelerate the computation of Rabin fingerprints in eliminating redundant network traffic using specifically designed data structures and algorithms.
- In particular, we extensively explore the ways of optimizing every aspects of the fingerprinting algorithm, such as using indexed compact buffer to mitigate data transfer overhead, taking full advantage of various types of memory to achieve maximum memory throughput, and relying on a fast approximate sorting operation to balance load among compute units on the GPU.
- We conduct extensive experiments to validate the effectiveness of our approach.

The source code of our implementation is available at https://github.com/aimlab/rabinGPU.

## 2 Background

In this section, we present an overview of the GPU architecture and the Rabin fingerprint algorithm with its application to content-based chunking.

### 2.1 General-Purpose Computing on GPUs

The current generation of GPUs have thousands of processing cores that can be used for general-purpose computing. For example, the Kepler GPU GTX780 consists of 12 Streaming Multiprocessors (SMXs), each equipped with up to 192 Stream Processors (SPs). Each SMX has 64 KB of on-chip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache, or can be evenly split between L1 cache and shared memory. The L2 cache (1,536 KB) is shared by all SMXs. In addition to the L1 cache, Kepler introduces a 48 KB read-only data cache. Each SMX has 64 KB 32bit registers equally split to the threads running in one block. In contrast, the off-chip global memory has a much larger size (typically in the GB range) and longer access latency.

The schedulable execution unit on the GPU is called a *warp* formed by a group of 32 threads. Warps are grouped together into *cooperative thread arrays* (CTAs), which are correspondingly structured as a *grid*. Typically, the threads in a warp follow the same execution path and operate on distinct data in single instruction multiple threads (SIMT) fashion in order to achieve maximal parallelism. Warp divergence may occur when there are conditional branches taken on the execution path. Launching a large number of threads concurrently is a recommended way to hide the latency of global memory access and to better utilize the computational resources on the GPU.

### 2.2 Content-Based Chunking and Rabin Fingerprinting

Data deduplication has been a hot topic in both storage and WAN optimization systems in recent years. The basic idea underlying data deduplication systems is that of *content-based chunking*, which essentially consists of three steps: 1) Chunking that partitions the data into different chunks based on data content instead of fixed offsets. 2) Hashing that calculates a collision resistant hash for each chunk. 3) Matching that checks if the hash of a new chunk already exists by searching the hash store. In this paper, our focus is on parallelizing the data chunking (particularly with Rabin fingerprinting) using GPUs, since it is practically one of the main bottlenecks in these systems.

In the following, we show how Rabin fingerprinting works and introduce some notation to help clarify several concepts of the chunking approach. Given a data block (e.g., the payload of a packet) of size $S$ bytes, The Rabin fingerprinting algorithm calculates the fingerprints over a sliding window of size $\omega$ ($S \geq \omega$), which advances one byte at a time. The size of the sliding window is configurable and typically ranges from 12 to 64 bytes in network traffic RE systems. Storing all possible fingerprints for identifying duplicate content is obviously impractical, considering the potentially large amount of fingerprints for each data block ($S - \omega + 1$ and $S \gg \omega$). As a result, only a small fraction of representative fingerprints that contain specific values (e.g., the least significant 8 bits are all 0) are recorded. The window position (the first byte of the window) of the corresponding representative fingerprint is considered as the boundary (*marker*) of a chunk. The *chunk* is the byte string between two markers. The *fingerprint* is a pseudo-random hash of the content in the sliding window beginning at the marker.

## 3 Design and Implementation

In this section, we first present a baseline implementation of the Rabin fingerprinting on the GPU. Then, we show the potential bottlenecks in this basic design by analyzing the memory access patterns and control flow divergence with respect to the specifics of the GPU architecture. At last, we describe in detail our novel optimizations to address the performance issues in the baseline implementation.

### 3.1 Baseline Implementation

Listing 1 shows the baseline implementation of the Rabin fingerprinting algorithm written in CUDA. To efficiently utilize the computing resources on the GPU and avoid extra state management and synchronization, we follow a simple approach adopted by most previous work, in which each packet is processed by one thread independently. In this naive implementation, there are five notable components: (1) The input buffer '*uchar *in*' that is a large continuous memory block for organizing network packets, each of which is stored in a fixed-size bucket. (2) The output buffer '*uint64_t *out*' that is used for the storage of fingerprints found. (3) The data structure '*struct rabinpoly window *rw*'

that contains two pre-computed lookup tables U and T, which aid the computation of fingerprints. (4) The auxiliary data buffer '*uchar *rabin_buf*' maintained for each packet as it is processed (at lines 14, 25, 27). (5) The core computation loop that reads packet input in byte stream (line 26), performs read/write accesses to the auxiliary buffer (at lines 25, 27), updates the fingerprint value (at lines 29, 30), and records the fingerprint in the buffer '*out*' if it is a desired one (at line 33).

Listing 1: Baseline implementation of Rabin fingerprinting.

```
1  #define BUCKET_SZ 1500
2  #define RABIN_WINDOW_SZ 32
3  #define RABIN_REPR_BITS 8
4
5  __global__ void rabin_kernel_naive (uchar *in, int *size_array, int total_threads,
6            struct rabinpoly_window *rw, uchar *rabin_buf, uint64_t *out)
7  {
8    int tid = threadIdx.x + blockDim.x * blockIdx.x;
9    if (tid >= total_threads) return;
10
11   /* get the size of each packet and its offset in the input string 'in' */
12   int size = size_array[tid];
13   uchar *input = in + tid * BUCKET_SZ;
14   uchar *rbuf = rabin_buf + tid * RABIN_WINDOW_SZ;
15
16   /* parameters for rabin fingerprinting */
17   uint64_t p, fingerprint = 0, mask = (1UL << RABIN_REPR_BITS) - 1;
18   int bufpos = -1, shift = rw->rp.shift;
19   uchar old_char, new_char;
20
21   /* main loop calculating and storing fingerprints*/
22   for (int i = 0; i < size; i++) {
23     if (++bufpos >= RABIN_WINDOW_SZ) bufpos = 0;
24
25     old_char = rbuf[threadIdx.x + bufpos * blockDim.x];
26     new_char = input[i];
27     rbuf[threadIdx.x + bufpos * blockDim.x] = new_char;
28
29     p = fingerprint ^ rw->U[old_char];
30     fingerprint = ((p << 8) | new_char) ^ rw->rp.T[p >> shift];
31
32     if (i < RABIN_WINDOW_SZ - 1) continue;
33     if (!(fingerprint & mask)) { /* storing fingerprints to 'out' */ }
34   }
35 }
```

It is straightforward to derive the GPU implementation of Rabin fingerprinting based on its CPU equivalent. However, an efficient GPU implementation requires a deep understanding of the GPU architecture, such as the memory subsystem, the branch divergence, and the load imbalance in warps or CTAs. We briefly describe some of these considerations to motivate the need for our optimizations.

*Memory analysis*. The non-optimized device memory access is one of the main reasons that make the baseline implementation suffer a severe performance penalty. First, all the memory operations are performed directly on the global memory (see lines 25, 26, 27, 29, 30, and 33). Although accesses to the global memory go through L2 cache, its limited size and the sequential access to the packet stream in each thread render the L2 cache almost useless. In particular, the core processing loop involves frequent accesses to the auxiliary buffer and the pre-computed lookup tables, which may incur non-trivial overhead. Thus, we should first focus attention on optimizing the memory access for the key data structures.

Second, most existing works use fixed-size bucket to buffer network packets before transferring them to the GPU. As compared to this intuitive design that may result in unnecessary host-to-device data transfer overhead due to irregular distribution of packet size, we choose a more complex buffering scheme that stores packets one by one in a consecutive buffer, and maintains indexes for the packet buffer to facilitate packet processing on the GPU.

Third, besides the optimizations for the key data structures, specific features exposed by different GPU memory spaces should also be leveraged to improve performance. For example, byte-wise access to the input stream that resides in the global memory may lead to underutilization of the memory subsystem, given the abundant memory bandwidth of GPUs. Accessing the global memory with long words is a promising way to alleviate this issue, as we will show in this paper. Combining the optimizations mentioned above with the appropriate exploitation of distinct memory spaces, such as the on-chip shared memory and register, we can expect further significant improvement in performance.

*Branch divergence analysis*. When the GPU threads execute different instructions in a warp, branch divergence can occur. *if-then-else* and *loop* statements are common sources of divergence. In Listing 1, execution path divergences at lines 22, 23, 32, and 33 cause performance degradation. Some divergences are inevitable because of the choices of algorithms and data structures, such as the ones at lines 22, 23, and 33, while others may be avoidable like the one at line 32 by moving the *if* statement out of the loop. Seemingly inevitable divergence can also be alleviated if the right trade-off can be identified, such as allowing redundant computation to circumvent explicit branches as discussed in this paper. To address these issues, we will present optimization strategies to amortize or eliminate performance overhead caused by the divergence in later sections.

*Load imbalance analysis*. Due to the one-packet-per-thread processing model and the sequential access pattern for each packet, if the network traffic exhibits abnormal distribution of packet size, unbalanced loads imposed on GPU threads would be detrimental to the overall performance. For example, threads processing packets with large payload will iterate the *for* loop at line 22 many more times than other threads, thus stalling other threads in the same warp. Arranging the packets in certain order may be a viable solution to this issue, if minimal overhead incurred by additional operations can be guaranteed. GPU-accelerated sorting has been demonstrated as an effective solution in many applications. However, some constrains made in conventional sorting algorithms can be relaxed in our case, which would lead to further improvement in performance as detained in this paper.

## 3.2 Optimizations

In this section, we present our novel optimizations that extend the baseline implementation to overcome the bottlenecks briefed in the previous section.

### 3.2.1 Indexed Compact Packet Buffer

Transferring data from the host memory (CPU) to the device memory (GPU) and vice versa incurs non-trivial overhead, because the GPU is usually connected to the system via the PCIe bus whose bandwidth is still a limiting factor, although there has been significant improvement in recent revisions of the PCIe standard. For example, in our test machine (a GTX780 GPU connected to motherboard via PCIe 2.0 $\times 16$ link), the data transfer rate peaks at 5.3 GB/s for the host-to-device transfer and 5.9 GB/s for the device-to-host transfer, whereas the device memory access bandwidth on the GPU is

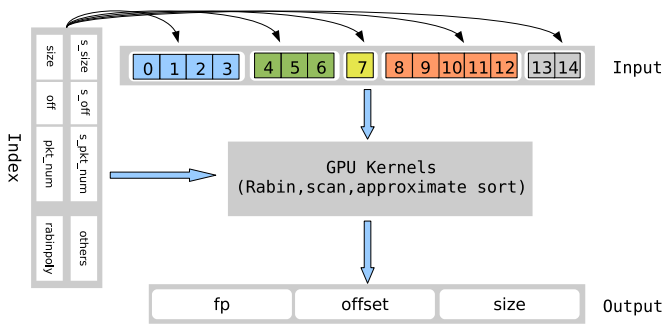



Fig. 1. Illustration of the indexed compact packet buffer.

144 GB/sec, an order of magnitude higher. Worse, the PCIe bus suffers much more overhead for small-sized data transfers. As a result, most existing work exploiting GPU acceleration adopts a simple approach in which a large buffer is maintained to accumulate network packets and all the packets collected during a given period of time are moved to the GPU in batches. The buffer is typically divided into fixed-size buckets to contain individual packets. This design performs well for certain workload, but it has important implications when considering the performance generally.

*Implications.* (1) Using fixed-size bucket may lead to reduced effective bandwidth in data transfer, because the bucket size is usually conservatively chosen to guarantee that it is large enough to queue variable-length packets, and the transfer of unused bytes would cause a huge waste of bus bandwidth in most cases. (2) The design of fixed-size bucket may also result in potential underutilization of parallelism of the GPU. For example, suppose we have a large buffer of size 15 MB and the bucket size is 1,500 Bytes, so the maximum number of packets the GPU can process concurrently is 10,000. However, in most cases, the spare space in the buffer can be leveraged to store more packets to increase the degree of parallelism of the GPU. (3) The time spent on chunking a specific buffer is split between the data transfer and the kernel computation. The above two aspects indicate that it is highly possible that the data transfer may dominate the overall execution time, a situation that GPU developers should try to avoid.

*Approaches.* In order to address these performance issues, we propose an indexed compact buffer scheme similar to one of the optimizations addressed in [21]. Instead of allocating fixed-size bucket for each packet, we store packets in the buffer one after another with no space in-between. An index structure is thus maintained to facilitate the management of the packet buffer. We believe this scheme is general and applicable to many other scenarios using GPU as the accelerator, such as software routers and variable-length data deduplication in storage systems.

Fig. 1 depicts an overview of the indexed buffer scheme. We first present the usage of several mostly-related indexes, and others will be detailed gradually in the following. The arrays *size* and *off* are two indexes used to locate the element belonging to each packet, and they are both generated on the CPU when accumulating packets into the buffer. The *size* array records the real length of each packet. However, in order to improve performance by exploiting the GPU memory hierarchy, we partition each packet into fixed-size data unit aligned to a 16 byte boundary. Therefore, the actual packet size measured with the 16-byte-long unit needs to be recalculated on the GPU as shown at lines 39-40 of Listing 2. We can maintain the *size* array according to data alignment, but the sorting optimization for load imbalance (discussed later) requires the real packet size as input. So, we retain the current design for simplicity. The array *off* contains the values representing the offset of the first unit of each packet in the *raw input* array. The offset is calculated based on the aligned size. Also initialized on the CPU, the array *pkt_num* is one of the inputs to an approximate sorting operation that can generate a reordered array to aid in mapping GPU threads to right packets. The three arrays prefixed by 's_' are the outcome of the sorting operation. The *rabinpoly* is a fingerprinting specific structure that contains precalculated parameters and is shared by all GPU threads.

Listing 2: Optimized implementation of Rabin fingerprinting.

```
1  #define THREADS_PER_BLOCK 128
2  #define ELEM_SZ 16
3  #define FIXED_FP 12
4  __global__ void rabin_kernel_opt (uchar *in, uint *size_array, uint *offset_array,
5                                    uint total_threads, uint *packet_number_array,
6                                    struct rabinpoly_window *rw, uint64_t *out)
7  {
8    __shared__ uint64_t U[256], T[256];
9    __shared__ uchar buf[THREADS_PER_BLOCK * RABIN_WINDOW_SZ];
10
11   int tid = threadIdx.x + blockDim.x * blockIdx.x;
12
13   for (int i = 0; i < 256 / blockDim.x; i++) {
14     U[threadIdx.x + blockDim.x * i] = rw→U[threadIdx.x + blockDim.x * i];
15     T[threadIdx.x + blockDim.x * i] = rw→rp.T[threadIdx.x + blockDim.x * i];
16   }
17
18   /* mapping from thread id to location in shared memory 'buf' */
19   int idx = threadIdx.x / 32 + (threadIdx.x % 32) * 4;
20   /* reset shared memory. */
21   for (int i = 0; i < RABIN_WINDOW_SZ; i++)
22     buf[idx + i * blockDim.x] = 0;
23
24   if (tid >= total_threads) return;
25
26   uint real_size = size_array[tid];
27   uint offset = offset_array[tid];
28   int4 result, *input = (int4 *)in + offset;
29
30   uint packet_number = packet_number_array[tid];
31   uint64_t *out_idx = out + packet_number * FIXED_FP;
32   uint16_t *out_offset = (uint16_t *)(out + NUM_PACKETS * FIXED_FP);
33   out_offset = out_offset + packet_number * FIXED_FP;
34   /* parameters for rabin fingerprinting */
35   int bufpos = -1, shift = rw→rp.shift, aligned_size = real_size;
36   uint64_t p, fingerprint = 0, mask = (1UL << RABIN_REPR_BITS) - 1;
37   uchar old_char, new_char, tmp[ELEM_SZ];
38
39   if (real_size % ELEM_SZ != 0)
40     aligned_size = real_size + (ELEM_SZ - real_size % ELEM_SZ);
41
42   ... /* performs computation on the first RABIN_WINDOW_SZ bytes */
43
44   int index = 0, i = RABIN_WINDOW_SZ / ELEM_SZ;
45   for (; i < aligned_size / ELEM_SZ; i++) {
46     if (++bufpos >= RABIN_WINDOW_SZ) bufpos = 0;
47
48     result = *(input + packet_number + i * total_threads);
49     fill_buffer(tmp, result);
50
51     for (int j = 0; j < ELEM_SZ; j++, bufpos++) {
52       old_char = buf[idx + bufpos * blockDim.x];
53       new_char = tmp[j];
54       buf[idx + bufpos * blockDim.x] = new_char;
55
56       p = fingerprint ^ U[old_char];
57       fingerprint = ((p << 8) | new_char) ^ T[p >> shift];
58
59       if (!(fingerprint & mask) && (index < FIXED_FP))
60         out_idx[index++] = fingerprint;
61         out_offset[index] = (i << 8) | j;
62     }
63     bufpos--;
64   }
65   __syncthreads();
66   ... /* write fingerprints and metadata to global memory */
67 }
```

Although depicted separately in Fig. 1, memory space allocated for the indexes *size*, *off*, and *pkt_num* is actually a part of the packet buffer, located at the beginning of the buffer. In this way, we can either copy the index to the GPU separately or as a whole with the packet payload according to different optimization strategies as discussed in later sections. Besides the three arrays that have counterparts on both the CPU and GPU, other indexes only require allocating memory on the GPU side and the memory can be reused by subsequent computations. The space cost of maintaining these indexes is negligible given the large mount of memory equipped with current GPUs and our problem scale. Although not a integral part of the index, the output is similarly organized into a contiguous memory space that is split into three components each with different data types. The array *fp* stores the 64-bit fingerprints found, the arrays *offset* (16-bit) and *size* (8-bit) record the precise position and the number of fingerprints discovered for each packet respectively. This design (compact memory layout and minimum data types) can reduce the overhead of device-to-host data transfer.

### 3.2.2 Managing the GPU Memory Hierarchy

In our baseline implementation, each GPU thread scans an assigned packet one byte at a time from the global memory, implying quite substantial underutilization of the GPU's massive memory bandwidth. An optional solution is to assign multiple threads to each packet in order to improve the utilization of memory subsystem. However, this would make the implementation complicated, if not possible, because of the philosophy underlying the Rabin fingerprinting computation that requires strictly sequential processing order for bytes in a specified window. In addition, the high access latency of device memory (on the order of hundreds of cycles) motivates the use of faster on-chip memories for frequently accessed data like the lookup tables and auxiliary buffer.

*Implications*. In order to achieve optimal performance, we should exploit the distinct characteristics of various memories in the GPU memory hierarchy.

*Approaches*. In Fermi and Kepler architecture, the hardware always issues memory transactions of 128-byte if the L1 cache is enabled (corresponding to the cache line size); otherwise, 32-byte transactions are performed. This means that it is always preferable to use longer words for better utilization of the memory bandwidth [21], [22]. Our optimized implementation fetches 16 bytes at a time by using the largest built-in vector data type *int4*, as shown at lines 28 and 48 in Listing 2. In accordance with this design, on the host side, we need to align each packet to 16-byte boundary when collecting packets into the buffer. Although packet alignment implies the reservation of extra unused space, the performance benefit from aligning packets far outweighs the penalty incurred by unnecessary data transfers. In addition, we will present an approach to avoiding complex processing logic in fingerprints computation due to data alignment in Section 3.2.3. Furthermore, in order to avoid maintaining another aligned-size array and thus the transfer cost, we generate the aligned size dynamically on the GPU (see lines 39 and 40 in Listing 2).

We leverage shared memory to optimize the data access to the lookup tables and per-thread auxiliary buffer (defined at lines 8 and 9 in Listing 2). Since each thread performs both read and write access to its own auxiliary buffer, careful partitioning of the shared memory is essential to prevent bank conflicts and ensure maximized performance. In particular, on Fermi architecture GPUs, shared memory is divided into 32 banks, each of which is 4-byte wide, and successive 4-byte words is assigned to successive banks. Accesses to shared memory are issued per warp, so the data layout in shared memory is crucial for performance. As shown in Fig. 2, to eliminate bank conflicts, we manage the shared memory in such a way that fits exactly to the hardware architecture. The sequentially increased numbers in squares reflect how successive 4-byte words are assigned to banks. Accesses to the shared memory by the threads in a warp are evenly distributed to all the banks. The auxiliary buffer for a specific thread is allocated in the same bank and organized in a strided layout. Using the formula $threadIdx.x/32 + (threadIdx.x\%32) * 4$ to calculate the base address for each buffer and setting the stride to 128 (32 banks * 4-byte), we can perfectly map the threads in a CTA (of size 128) to the shared memory. Although other partitions may also be possible, our current design significantly outperforms a flat partitioning scheme, in which the shared memory is split into equally-sized and contiguous blocks that are allotted to threads sequentially.

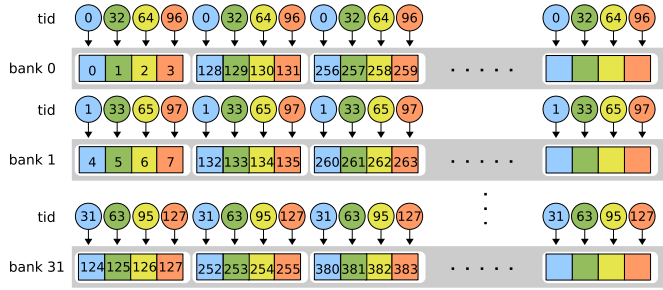



Fig. 2. Bank-conflict-free access to shared memory.

Because the lookup tables are shared by all the threads in a CTA and exhibit a read-only random access pattern, they are implemented as-is with no further layout optimizations. In addition, for Kepler GPUs, the number of banks is still 32, but the bank width has doubled to 8-byte (both 8-byte and 4-byte access mode are supported). When we explicitly set the access mode by calling *cudaDeviceSharedMemConfig* with *cudaSharedMemBankSizeEightByte* as the argument, we observed immediate effect on performance for about 8 percent gains because of the improved bandwidth utilization of accessing the 64-bit arrays U and T. Allocating too much shared memory for each thread may have negative impact on performance due to low theoretical CUDA occupancy. To address this issue, we tried to achieve higher occupancy by offloading the lookup tables storage from the shared memory to the 48 KB read-only data cache introduced to Kepler GPUs (using the *__ldg*() intrinsic). This cache supports full speed unaligned memory access patterns, and has the potential to reduce bandwidth contention in the shared memory. Unfortunately, we observed non-trivial performance degradation by leveraging the data cache, which is radically different from our initial expectations.

Fermi GPU has 32 K 32-bit registers per SM, and the maximum theoretical register allocation per thread is 63 32-bit

registers. Kepler GK110 GPU has doubled the number of registers per SM. Given the abundance and access speed of registers, we can even allocate storage for arrays that are indexed with constants (guarantee no spills to local memory) using registers. For example, the input stream read from the global memory to a local variable (of type *int4*) is copied to the array *tmp* by byte-wise shifting the *.x*, *.y*, *.z* and *.w* fields (performed by the function *fill_buffer*). For example, for a value $v$ of type *int4*, we can extract the bytes of its field *.x* using the code sequence: $\mathrm{buf}[0] = (\textbf{uchar})\mathbf{v}.\mathbf{x}$; $\mathrm{buf}[1] = (\textbf{uchar})(\mathrm{v.x} \gg 8); \mathrm{buf}[1] = (\textbf{uchar})(\mathrm{v.x} \gg 16); \mathrm{buf}[1] = (\textbf{uchar})(\mathrm{v.x} \gg 24)$. In this way, we can not only guarantee maximized data access performance, but also make the implementation straightforward as compared to alternatives that explicitly access the fields of the *int4* structure. Another interesting observation is that when the size of the sliding window is less than 16, the auxiliary buffer can reside totally in the register. However, this scheme does not scale well to large-sized sliding window even the performance benefit is non-trivial. The implication is that there would be room for further improvement in performance if we adopt a mixed-storage model for the auxiliary buffer by combining the shared memory and register. We leave this as our future work. In order to avoid excessive register usage that may also lower the CUDA occupancy like the shared memory, variables that remain same to all the threads in a warp are stored in the constant memory that has the property of delivering a value in a single cycle if the value is requested by all the threads. For example, the variables *mask* and *shift* in Listing 2 are excellent candidates for the constant memory. We do not use constant memory for these variables in our current implementation because it does not incur excessive usage of registers.

#### 3.2.3 Alleviating Divergence

GPU's SIMT architecture enforces the threads in a warp to execute in a stepwise fashion in order to achieve optimal performance. When a data dependent conditional branch occurs, serialized execution is imposed to the warp until all threads in it converge to the same execution path. In our case, both the inherent algorithmic choice and optimization strategy proposed earlier may result in divergence as discussed below.

*Implications.* Since it is often inevitable to introduce divergence in real applications, to avoid performance issues arose from divergence, care must be taken in constructing the algorithm in order to eliminate or minimize the effect of conditional branches.

*Approaches.* First, the fingerprinting algorithm works by scanning the content over a sliding window (we denote its size by $s$), so each thread firstly needs to read the beginning $s-1$ bytes before performing real fingerprint calculation, which requires a precondition check. But placing the check inside the main calculation loop would make it wasteful and unnecessary to perform checks for the remaining bytes. Thus, we separate the fingerprinting of the initial $s$ bytes and the remained into two loops (see the comments at line 42 of Listing 2 and the initialization expression of the *for* loop at lines 44-45). In addition, we observed no further improvement by enabling the compiler's loop unroll (#*pragma unroll*) optimization.

Second, the code shown at line 59 of Listing 2 is a conditional branch that determines if a desired fingerprint is found. At the first look, in addition to divergence, saving the fingerprints directly into the global memory would stall the corresponding threads for much longer, and the randomness of fingerprint distribution among different threads may exacerbate the problem. However, in practice, the cost of storing the full set of fingerprints for a data stream is prohibitively high in terms of storage space, thus only a small proportion of fingerprints whose least significant $n$ bits are 0 are recorded. In our case, the maximum segment size for TCP payload is 1,460 and we set $n$ to 8 and $s$ to 32, so the average number of fingerprints found in a packet would be about six according to $(1460 - s + 1)/2^n$ (11 for $n = 7$, and 22 for $n = 6$). Hence, the performance impact due to writing fingerprints to the global memory would be much smaller than expected, and additional optimizations may not be worthwhile. We have attempted to optimize this issue by temporarily storing the fingerprints in registers and flushing them to the global memory at the end of the kernel. But this endeavor brings no satisfactory results, coinciding with our assumption and analysis.

Third, accessing the global memory based on packet alignment and built-in vector types like *int4* has the advantage of increased memory bandwidth utilization. However, the content padded due to alignment should be considered in computing the fingerprint. For example, we need to trace how many bytes are left to perform fingerprinting, and know if the remaining bytes are within or cross the alignment boundary. Taking into account all these corner cases would apparently complicate the implementation and hurt performance consequently. As a result, we adopt a divergence-free strategy, which intentionally allows the unnecessary fingerprinting of the padding bytes, but can distinguish alignment-incurred fingerprints by examining the position of fingerprints recorded in an offset array as shown at line 61 of Listing 2. With the real length of packets, simple calculation based on the recorded information can filter out the false positives. For a packet aligned on a 16-byte boundary, at most 15 superfluous bytes are fingerprinted, and on average less than one false positive (according to the analysis in the above paragraph) will be generated. This overhead is negligible as compared to that incurred by implementing complex conditional logics in the main loop to avoid fingerprinting the padding bytes.

#### 3.2.4 Optimizing Load Imbalance with Approximate Sorting

Inherent irregularity in some applications may cause unbalanced workload distribution among threads on the GPU, resulting in ineffective utilization of compute resource. In our case, processing packets with a high length variance will lead to workload imbalance within warps and thread blocks. For example, a single thread/warp processing a comparatively large packet would cause resource waste as every packet will take as many cycles as the largest one to process in the warp/thread block.

*Implications.* Because current GPUs provide no support for fine-grained thread-level task scheduling, tackling the inefficiency incurred by workload irregularity imposes the responsibility of fine-tuning the algorithm on developers.

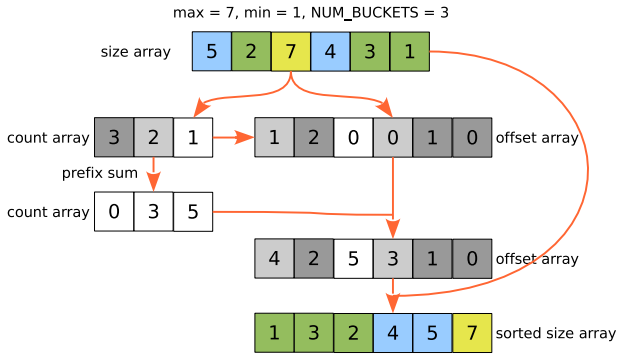



Fig. 3. Illustration of approximate sorting.

An ideal solution to this problem should work well for a wide range of workloads. For example, it should achieve highest possible performance for highly irregular workloads, and at the same time guarantee lowest possible cost for regular workloads.

*Approaches.* To address this issue, we initially explored the idea of identifying outliers in packet streams to defer their processing to subsequent kernels [9]. Without any success due to the considerable overhead, we resorted to other solutions instead. Although existing work [18] indicates that it is not worthwhile to group packets with identical or similar size by performing sorting on GPUs. On the contrary, we found that pre-sorting the packets on CPU can result in a significant performance increase for the fingerprinting kernel as demonstrated in GASPP [23]. Moreover, balancing the workload within warps or thread blocks does not necessarily require a strict ordering of packets. These investigations are the direct motivation for inventing a GPU-based approximate sorting algorithm as detailed below. Approximate sorting may also be used to alleviate load imbalance in GPU-based graph algorithms [9], and to improve the performance of real-time graphics applications [17].

As shown in Fig. 3, our algorithm operates in three steps. First, each element in the *size* array is mapped into a bucket (the number of buckets is a pre-defined parameter and typically much less than the input size). In this step, we maintain an ordering among all elements that are mapped into the same buckets and a counter array that records the size of each bucket. Second, an exclusive *prefix sum* operation is performed on the counter array. In the third step, the results of the above two steps are combined to produce the final coordinates that are then used to transform the input vector into the approximately-sorted form.

*Step 1.* Similar to many parallel sorting algorithms that subdivide the input into equally-sized buckets and then sort each bucket in parallel, we first map each element of the *size* array into a bucket. As shown in Listing 3, the number of buckets is a fixed value *NUM_BUCKETS*, and the mapping procedure is a linear projection of each element in the input vector onto one of the *NUM_BUCKETS* buckets. The linear projection is demonstrated at lines 10 and 11 in Listing 3, where the variables of *min* and *max* represent the minimum and maximum value in the input respectively, which can be obtained when accumulating packets into the packet buffer on the CPU. In this way, each bucket represents a partition of the interval $[min, max]$, and all buckets have the same width of $(max - min)/NUM\_BUCKETS$. The elements in the input vector are assigned to the target bucket whose value range contains the corresponding element. In addition, another array *bucket_count* is maintained to record the number of elements assigned to each bucket. As shown at line 13, the counting is based on an atomic function provided by CUDA, *atomicInc*, to avoid the potential conflicts incurred by concurrent writes. The function *atomicInc* returns the old value located at the address presented by its first parameter, which can be leveraged to indicate the local ordering among all the elements assigned to the same bucket. The Kepler GK110 GPU has substantially improved the throughput of global memory atomic operations as compared to Fermi GPUs, which also has been validated in our implementation.

Listing 3: Assigning elements to buckets.

```
1  __global__ void assign_bucket (uint *input, uint length, uint max, uint min,
2                                 uint *offset, uint *bucket_count, uint *bucket_index)
3  {
4    int idx = threadIdx.x + blockDim.x * blockIdx.x;
5    uint bucket_idx;
6    if (idx >= length) return;
7
8    uint value = input[idx];
9
10   bucket_idx = (value - min) * (NUM_BUCKETS - 1) / (max - min);
11   bucket_index[idx] = bucket_idx;
12
13   offset[idx] = atomicInc(&bucket_count[bucket_idx], length);
14 }
```

*Step 2.* Having obtained the counters for each bucket and the local ordering within a specific bucket, we perform a *prefix sum* operation on the counters to determine the address at which each bucket's data would start. Given an input array, the *prefix sum*, also known as *scan*, is to generate a new array in which each element *i* is the sum of all elements up to and including/excluding *i* (corresponding to *inclusive* and *exclusive prefix sum* respectively). Because the length of the count array (*NUM_BUCKETS*) is typically less than that of the longest packet, performing the scan operation on CPU is much faster than on GPU. However, due to the data transfer overhead (in our case, two transfers), and the fact that we observed devastating performance degradation when mixing the execution of the CPU-based *scan* with other GPU kernels in a CUDA stream, the parallel *prefix sum* is performed on GPU using the CUDPP library [8].

*Step 3.* By combining the atomically-incremented offsets generated in step 1 and the bucket data locations produced by the prefix sum (as shown at lines 13-16 in Listing 4), it is straightforward to scatter the *size-offset* pairs (notice the two different offsets, one for packet metadata and the other for sorting) to proper locations (see lines 19-20). With the sorted offset array, threads in the same warp or block is able to process packets that are similar in size, leading to balanced workload distribution. A side effect of the sorting operation is that we can not directly save fingerprints at locations indexed by thread ids, because of the remapping of threads to packets. Therefore, we maintain another index array to number each packet in increasing order (the same as how thread ids are sequentially assigned by the GPU), and this array can also be sorted in the approximate sorting (see line 21 in Listing 4). Relying on it, each thread can be connected to its designated position (see lines 30-33 in Listing 2).

Choosing a suitable value for the number of buckets may have important implications for the efficiency and

effectiveness of our sorting algorithm. As the number of buckets increases, for inputs exhibiting non-constant distribution of packet size, our algorithm would approximate more closely to the ideal sorting, while the overhead of performing the *prefix sum* may increase accordingly. When decreasing the number of buckets, besides the effect of getting a coarse-grained approximation for the input vector, time variations for the kernel *assign-bucket* may occur as a result of using the atomic operation to resolve conflicts when multiple elements are assigned to the same bucket concurrently.

Listing 4: Approximate sorting.

```
__global__ void appr_sort (uint *key, uint *key_sorted, uint *value,
        uint *value_sorted, uint *packet_number, uint *packet_number_sorted,
        uint length, uint *offset, uint *bucket_count, uint *bucket_index)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    uint count = 0;
    if (idx >= length) return;

    uint key = key[idx];
    uint value = value[idx];
    uint number = packet_number[idx];

    uint bucket_index = bucket_index[idx];
    count = bucket_count[bucket_index];
    uint off = offset[idx];
    off = off + count;

    __syncthreads();
    key_sorted[off] = key;
    value_sorted[off] = value;
    packet_number_sorted[off] = number;
}
```

#### 3.2.5 Concurrent Copy and Execution with Streams

GPU-assisted applications may suffer from the prominent overhead of transferring data between the host and device over the PCIe link. In particular, for scenarios where the data transfer overhead significantly outweigh the actual computation cost, the overall performance would be severely constrained.

*Implications.* The time-consuming data transfer in chunking a given packet buffer may dominate significantly over the computation time. Worse, the optimizations proposed to accelerate the kernel execution would become irrelevant because optimizing computation may make the data transfer an even greater burden on the overall performance.

*Approaches.* We leverage the GPU hardware support to address this issue. In particular, we use GPU streams to achieve concurrent copy and execution with each stream encapsulating the task of chunking a specific packet buffer. In a single stream, the execution is serialized because of the interrelated dependency among distinct kernels. However, streams processing different packet buffers can be scheduled by the GPU independently to enable not only concurrent copy and execution, but also concurrent kernel launches from separate streams. In particular, the index metadata is treated differently in two scenarios. On one hand, although the metadata is located at the header of the packet buffer, for high-end GPUs, in order to achieve high throughput, we often maintain a relatively large buffer that is at least one order of magnitude larger than the metadata. Due to the serialized execution within streams, performing sorting on the indexes after the whole buffer is transferred to the GPU can not fully exploit the concurrent copy engine. To this end, we intentionally break up the transfer of the metadata and payload data and invoke the sorting kernel after the metadata copy (from CPU to GPU), instead of performing sorting when both metadata and payload are loaded to GPU. With this simple optimization, we see a constant performance boost about %10. On the other hand, in environments where the kernel computations dominate in the overall cost, it would be preferable to transfer the whole buffer to the GPU in one transaction instead of two.

#### 3.2.6 Other Optimizations Considered

For certain applications (like ours), the difference of favored data layout between the CPU and GPU may result in suboptimal performance. On CPUs, applications may prefer row-major data layout because of the extensive support for cache locality, prefetching, et al. in hardware. In contrast, GPU applications benefit from coalesced memory access, which requires a column-major data layout so that threads in a warp can access contiguous data in the global memory. This discrepancy motivated us to develop a data remapping approach, which is independent of the fingerprinting algorithm by manipulating the coordinate of each data element with a series of kernel invocations. We indeed observed nontrivial improvement in performance with the column-major layout, but the cost of the data remapping process itself is relatively high. So we excluded this optimization. We hope that the hardware, driver, or runtime could provide mechanisms to automate the transformation of data layout, saving additional cost but without compromising transparency.

#### 3.2.7 Comparison with Shredder

Shredder [5] is a high performance content-based chunking system for storage systems, and it also implements a GPU-based Rabin fingerprinting. Although Our work shares some similarities with Shredder, major differences exist between the two systems as discussed below.

First, Both Shredder and our system use the optimization of concurrent copy and execution to overcome the bottleneck of DMA transfer between CPU and GPU. The difference is that Shredder proposes a design of double buffering, while our system separates the metadata transfer from the payload transfer, which makes the concurrent execution of approximate sorting possible. In addition, Shredder has two unique designs on the host side. One is the circular ring buffers to minimize the penalty of allocating pinned memory; the other is the multi-stage streaming pipeline to obtain better resource utilization at the host.

Second, because optimizing the access to shared memory to avoid bank conflicts has significant impact on performance, Shredder fetches data from global memory to shared memory that is evenly split among threads. However, our system only uses the shared memory to optimize the access to key data structures such as the lookup table and auxiliary buffer, and a strided layout of shared memory for the auxiliary buffer is adopted instead of a flat partitioning. We rely on large word to optimize global memory access, which is also presented in Shredder. Moreover, more algorithmic details and design options are exposed in this paper as shown in Section 3.2.2.

Third, Shredder only qualitatively discusses warp divergence and its impact on performance, while this paper

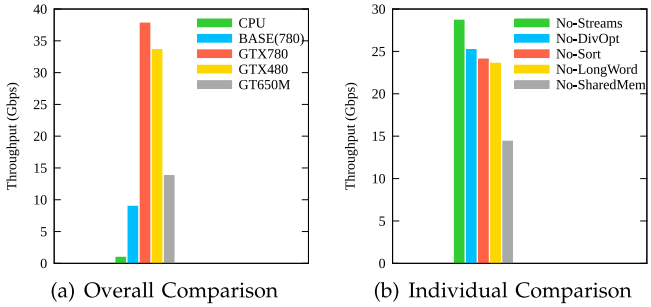



Fig. 4. Performance comparison of indexed and fixed buffer schemes.

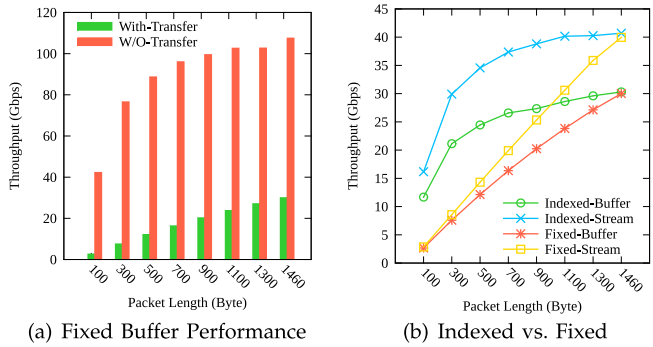



Fig. 5. Performance comparison of indexed and fixed buffer schemes.

reveals more implementation-level considerations on alleviating divergence, such as separating the fingerprinting of the initial byte sequence and intentionally performing unnecessary fingerprint computation due to memory alignment. In addition, we observe no improvement with loop unroll, which is contrary to the results made in Shredder.

Fourth, as compared to Shredder that only concerns about fixed-size data blocks in storage systems, two distinguishing features, indexed compact packet buffer and approximate sorting index, are proposed to tackle specific needs in network systems. In particular, for variable-length network packets, indexed compact buffer can be used to optimize GPU resource utilization, and approximate sorting is effective on alleviating load imbalance. However, these are not the main issues for fixed-size data processing as in Shredder.

## 4 Performance Evaluation

In this section, we evaluate the performance of the GPU-based Rabin fingerprinting algorithm. The experiments were conducted on a machine equipped with an AMD phenomII X6 1055T CPU (hexa-core 2.8 GHz), 8 GB main memory, and an Nvidia GTX 780 (Kepler) GPU card. The GPU has 12 SMXs, each containing 192 cores (2304 cores in total), and 3 GB device memory. The maximum amount of shared memory for each SM is 48 KB. The operating system was 64-bit Ubuntu 12.04 with CUDA 5.5 and NVIDIA driver of version 331.67 installed.

### 4.1 Overall Performance

We first present the comparison of overall performance using different hardware configurations or software implementations, and evaluate how each optimization method contributes to the overall performance.

In Fig. 4a, all the evaluations on GPUs use the same configuration parameters (16 K packets, four streams, and uniform packet size distribution). The baseline GPU implementation on GTX780 achieves 8.98 Gbps throughput, which is 9.5× better than the result (0.95 Gpbs) of the single-threaded CPU version. The throughput on GTX780 with all optimization enabled is 37.79 Gpbs. Our simple multi-threaded implementation on the hexa-core CPU shows linear scalability, indicating that a machine with 40 CPU cores would be the basic requirement to obtain results comparable to the GTX780 GPU; Of course, with different packet size distributions we may obtain different performance results. In addition, we observed 33.6 and 13.8 Gbps throughput on GTX480 and GT650M (on a Macbook pro) respectively, which indicates the cost-effectiveness of our approach because of the decent performance results on low-end GPUs. The result on GTX480 shows a 1.87× (33.6 bps versus 18 bps) speedup of throughput compared to Shredder under comparable hardware (Telsa C2050 for Shredder). Both the GPUs belong to the Fermi GF100 architecture, and have the same number of cores (448) and the same memory interface width (384 bit).

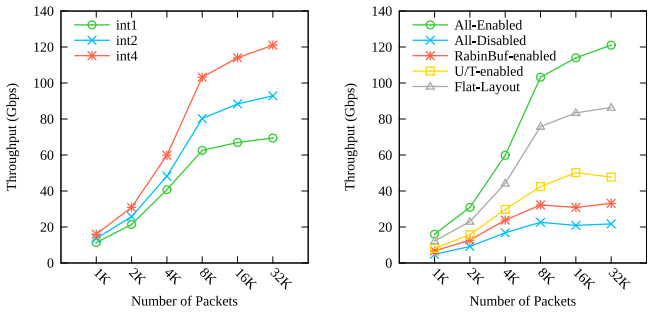
In order to quantitatively evaluate the contribution of individual optimization strategy to the overall performance, we conducted experiments in which we separately disable one specific optimization with other optimizations enabled. As shown in Fig. 4b (results were collected using the GTX780 GPU), by intentionally plotting the data in the histogram in descending order, we can comparatively observe the impact of each optimization method on the performance. Because it is difficult to accurately measure performance metrics across CUDA streams, we use the result obtained from disabling streams as the baseline. From Figs. 4a and 4b, we can see that CUDA streams improve the throughput by %32. For a single stream, optimizations including the divergence alleviation, approximate sorting, long word access, and shared memory can improve the throughput by %14, %19, %21, and %99 respectively. The above analysis reveals that shared memory and streams are the top 2 most important contributors to the performance, while other optimizations also lead to non-trivial improvement.

### 4.2 Optimization Analysis

In this section, we investigate the impact of each optimization strategy proposed in this paper on performance in detail. The results were obtained using the GTX780 GPU.

#### 4.2.1 *Indexed Buffer versus Fixed Buffer*

Fig. 5a shows the throughput of Rabin fingerprinting using the fixed-size buffer scheme that is adopted by most previous studies. In this experiment, we set the number of packets to be processed to 16 K. We compare the performance difference when the data transfer overhead is or is not considered. In both cases, the gradual increase of the throughput as the packet size increases indicates that larger payload can always make more efficient utilization of the GPU. When not taking into account the data transfer, the throughput peaks at 107 Gbps for the maximum packet size, and drops to 42 Gbps for packet size of 100 bytes, corresponding to a 2.5× degradation in performance. We can




(a) Long Word Performance (b) Shared Memory Performance

Fig. 6. Evaluation of the memory hierarchy optimizations.

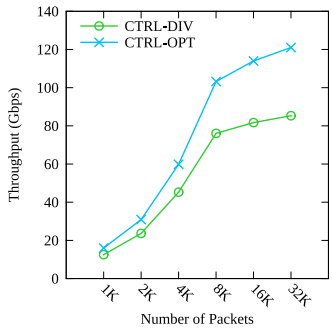



Fig. 7. Control divergence optimization.

see only marginal improvement when the packet size becomes larger than 700, which implies near-saturated GPU utilization. In comparison, the data transfer incurs significant overhead to the throughput. For example, packets with full payload achieve 30 Gpbs throughput, but the throughput is only 2.7 Gbps for 100-byte packet, indicating a 11× performance discrepancy. Comparing the performance deviations (e.g., 2.5× versus 11×) and observing the linear scalability of throughput in the later case, we can conclude that the packet size has more important implications for performance when the data transfer overhead is considered.

Fig. 5b compares the performance of two schemes, i.e., the indexed buffer and fixed-size buffer. We only present the results involving data transfer overhead, and also include the performance data obtained from using CUDA streams to show some interesting findings. Undoubtedly, the indexed scheme outperforms the fixed buffer scheme significantly in both stream and non-stream based scenarios. The throughput (without streams) increases from 2.7 and 11.7 Gbps (100-byte) to 30 and 30.3 Gbps (full payload) for the two buffering schemes respectively. The performance curves of using streams exhibit similar trend, but much higher throughput especially for the indexed buffer. For example, the throughput raises from 2.9/16.2 Gpbs (smallest packet) to 40/40.7 Gbps (full payload) respectively. Obviously, we obtain approximately the same throughput for full-payload packets (e.g., 30 versus 30.3 and 40 versus 40.7). In addition, two findings are worth further exploration. First, the fixed buffer scheme scales linearly with respect to the packet size, and the performance gains from using streams are marginal for small packet. However, the improvement of the stream-optimized indexed buffer is much more significant for the full range of packet sizes (see the large gap between the blue and green curve). Second, we can learn that the indexed buffer without streams still outperforms the fixed buffer using streams when the packet size is smaller than 1,100 bytes, by observing the intersection of the green and yellow curve.

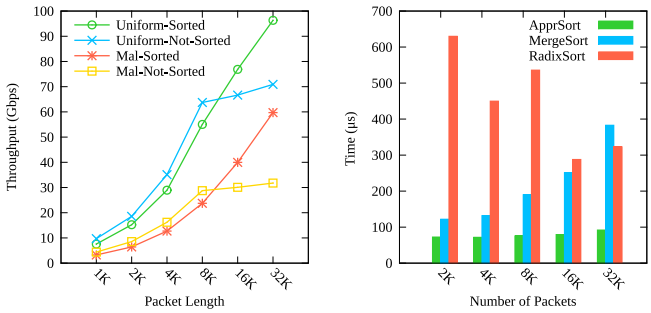
### 4.2.2 Memory Hierarchy Optimizations

Fig. 6a presents the performance comparison using different word sizes. The $x$ axis shows the number of packets offloaded to the GPU in a batch, which is equal to the number of threads due to the one-packet-per-thread model. As the number of threads increases, the throughput climbs up accordingly for all word sizes as a result of the massive multi-thread capability of the GPU that can effectively hide the memory access latency. We can observe the wide performance disparities among different word sizes when the number of threads exceeds 8 K. Obviously, accessing the global memory using longer word constantly results in greater performance. When using *int4* and 32 K threads, the throughput achieves 121 Gbps that is 1.3 and 1.75× better than using *int2* and *int1* respectively. One side effect of using large word size is that more memory (on both the GPU and CPU side) may be required due to the data alignment, which consequently incurs additional CPU-GPU transfer overhead. However, we believe that these costs are acceptable as compared to the potential benefits. For example, suppose we have 16 K packets with uniform size distribution, typically the alignment-induced extra space is in the order of hundreds of Kilobytes, which incurs tens of microseconds of transfer overhead over a PCIe 2.0 link.

In Fig. 6b, we demonstrate the impact of shared memory on performance under a variety of configurations. When the shared memory optimization is completely disabled, the maximum achievable throughput is less than 22 Gbps. In stark contrast to this, the peak throughput is up to 6× higher with the fully-enabled shared memory optimization. By separately enabling the optimization for the auxiliary buffer (RabinBuf) and lookup tables (U/T), it is clear to observe that the latter has greater positive influence on the throughput, and the plateau of the former occurs much earlier. However, standalone exploitation of either of the options produces non-optimal results (see the large gaps against the 'All-enabled' solution). In addition, Fig. 6b plots the performance of a flat partitioning scheme of the auxiliary buffer, in which the whole buffer is split into equal-sized and contiguous blocks that are then assigned to each thread sequentially. The substantial variability in throughput reveals that a holistic optimization using the shared memory is critical for achieving optimum performance. On Kepler GPUs, enabling 64-bit access to the shared memory of the lookup tables with *cudaSharedMemBankSizeEightByte* further improves the run time of the fingerprinting kernel by about 7.5 percent.

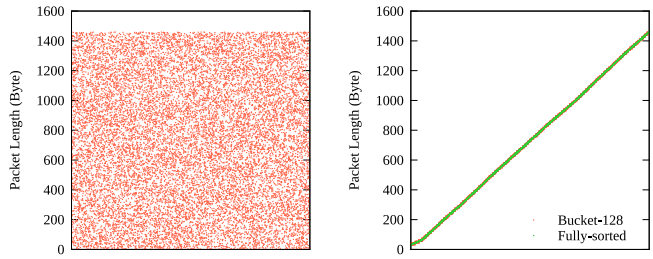
### 4.2.3 Control Divergence Optimizations

In evaluating the impact of the optimization against control divergence, we compare its performance with an unoptimized version that introduces several additional branches into the processing loops (one for the inner loop and three for the outer loop) in order to handle the corner cases, such as counting how many bytes are left, and determining when the outer loop should exit. In this experiment, all optimizations are enabled by default except the divergence optimization. As shown in Fig. 7, alleviating divergence can speed




(a) Sorted vs. Non-sorted (b) Algorithm Comparison

Fig. 8. Performance evaluation of sorting.




(a) Uniform Distribution (b) Distribution After Sorting

Fig. 9. Comparison between full-sort and approximate-sort.

up performance significantly especially when the number of threads is greater than 8 K. The throughput at 32 K threads reaches 121 Gbps that is 1.4× better than that (85.3 Gbps) of the unoptimized implementation.

We evaluate the performance impact of the amount fingerprints generated at runtime by varying the $n$ least significant bits as discussed in Section 3.2.3. $n$ is defaulted to 8 (a typical value in network scenarios), which yields six fingerprints for each packet on average. When we set $n$ to 6 to generate 22 fingerprints averagely, a performance degradation of 5.6 percent of the Rabin kernel was observed.

#### 4.2.4 Approximate Sorting

Although sorting can be used to alleviate unbalanced workload distribution, more attentions should be paid on the accuracy and associated overhead in order to achieve satisfactory gains in performance. In this section, we evaluate the effectiveness of the approximate sorting algorithm by answering three questions: To what extent can we improve the performance by sorting? How fast is the approximate sorting as compared to existing algorithms? How close does our approach approximate the fully sorted scenario?

Fig. 8a demonstrates the impact of sorting on performance in two cases. In the first experiment, we investigate how the throughput varies as a function of the number of packets whose sizes follow an uniform distribution. We can see the marginal drop of performance due to the inherent overhead of sorting, and insufficient packets fed to the GPU, which causes low resource utilization. However, by increasing the packet volume, the throughput scales almost linearly and eventually outperforms the non-sorted counterpart with increasingly large strides. In the second case, we conduct testing on a synthetic workload exhibiting malformed size distribution, with which we intentionally arrange the packets in a CUDA block with the same size (200) but the last one that is exceptionally large (1,300). The performance curves reveals that sorting performs equally well and is also beneficial for uncommon distributions of packet size. Inevitably, the approximate sorting incurs overhead that would hurt performance especially for scenarios where the data volume is small. However, typical GPU-assisted network applications often batch as much packets as possible to the device to amortize the PCI-e overhead, so we believe the approximate sorting is a viable solution for such applications.

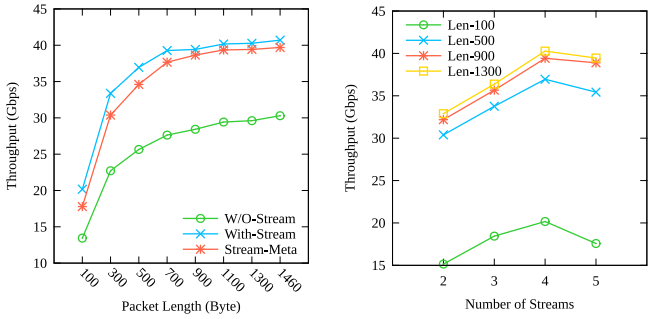
Fig. 8b presents the answers for the second question. We compare approximate sorting with two commonly-used and high-performance GPU sorting algorithms, radix sort from the thrust library [1] and merge sort from CUDA SDK that is based on [13]. We assume a uniform size distribution for all the experiments. As shown in the figure, our approach outperforms the competitors significantly. The approximate sorting consumes less than 100 microseconds that are the sum of the execution time of three kernels, reflecting 8.7× in the best case and 3-4× in average improvement of execution time. It is worth noting that the overhead of approximate sorting remains nearly invariant for different workloads, while merge sort's cost exhibits gradual increase as the number of packets increases, and radix sort shows no distinguishable characteristics in behavior and performs poorly under light workloads.

The effectiveness of approximate sorting in part relies on the accuracy of the algorithm, which is conceptually similar to counting-based sorting by mapping data items into a fixed number of buckets instead of performing pairwise comparisons. The number of buckets is the key parameter to tune the accuracy. Fig. 9a plots the size distribution of 16 K packets before sorting, and Fig. 9b illustrates the closeness of the results to the fully-sorted counterpart, where we only consider a small bucket size (128) that gives a conservative estimation about accuracy (large bucket size would produce more accurate sorting results). The two almost perfectly overlapped curves indicate that the approximate sorting not only is superior to existing algorithms in performance, but also retains high degree of accuracy. The axis $X$ in both figures represents the index of an array where each element is a packet of different length.

At last, we measure the grouping overhead for full-payload packets, and an overhead of 4.4 percent was observed when the approximate sorting was enabled. However, this overhead is avoidable for cases where all packets have the same length, because we invoke the sorting kernel only on conditions when the maximum and minimum length of collected packets are not equal.

#### 4.2.5 Concurrent Execution with Streams

Previous micro-benchmarks mainly focus on the evaluation of individual optimization, which only reports results excluding the CPU-GPU data transfer overhead. As a common practice, in this section, we investigate how to improve the overall performance by leveraging CUDA streams. As shown in Fig. 10a, the achieved throughputs with streams (four streams) are consistently much better than the counterpart with streams disabled across the full range of packet size. We can also observe the performance degradation (marked by 'Stream-Meta') where the metadata and payload




(a) Overall Performance (b) Scalability

Fig. 10. Performance evaluation with streams.

are transferred to the GPU together. The reason is that splitting the transfer of the metadata and real payload, and performing sorting on the metadata right after its transfer, can potentially improve intra-stream concurrency. Fig. 10b shows how the throughput scales with the number of streams, exhibiting similar variation over four different workloads. The performance peaks with four streams.

## 5 Related Work

*Traffic Redundancy Elimination.* Traffic redundancy elimination has been becoming increasingly popular in industry and academia recently. Since the pioneering work of Spring and Wetherall [19], protocol-independent redundancy elimination have demonstrated its usefulness in many scenarios. EndRE [2] is a system proposed to deploy TRE at the end host to maximize bandwidth savings. It proposed the SampleByte mechanism that identifies chunk boundary using a lookup table to accelerate the computation of fingerprints. Studies in [3], [4] further exploited the potential of deploying TRE techniques in network-wide environment. Redundancy elimination has also been explored in wireless environment [14]. Furthermore, PACK [25] is the first to propose a predictive approach (as compared to traditional full-synchronized approach) to eliminating redundant traffic for Cloud applications. Given the broad interests and approved benefits of TRE, we believe that the proposed GPU-assisted fingerprinting can be leveraged in these TRE systems to alleviate performance bottlenecks. For example, running our fingerprinting algorithm on a low end GPU can significant outperform the SampleByte approach.

*Offloading:* GPUs have recently been demonstrated to show substantial performance improvement for many network workloads. PacketShader [7] achieves 40 Gbps IPv4 and IPv6 forwarding. SSLShader [10] shows compelling performance enhancement for a secure sockets layer server by offloading RSA, AES and SHA-1 cryptographic primitives to GPUs. In [24], a GPU-based name lookup system for content-centric networks is presented, and the experimental results show that large scale name lookup can be performed on GPU at high speed and low latency. In [20], the authors reported that the GPU can accelerate regular expression matching by up to 48× over CPU implementations. Nuclei [16], a GPU-based network coding system, shows that a NVidia 8800 GT GPU can outperform an 8-core Intel Xeon server. Besides the network workloads, GPU acceleration has been successfully applied to many other areas such as virtual machines [15] and buffer cache in operating system [6]. Our work shares some similarities with these existing studies, such as batching packets to the GPU and accelerating GPU computation with streams. However, the indexed compact buffer scheme and the approximate sorting algorithm are unique in our system, and we believe both are applicable to other GPU-based network packet processing systems.

The most closely related work is Shredder [5], which is a system designed for performing efficient content-based chunking to support scalable incremental storage and computations. In Shredder, GPU is employed to meet the high computational requirements. Although both our work and Shredder use GPU to accelerate computing fingerprints, there are major differences in two aspects. First, Shredder focuses on the issues from a system architecture perspective, and lacks some algorithmic details. Second, application specific challenges (storage versus network) also lead to unique designs in our work, such as the indexed compact buffer scheme and approximate sorting. In addition, due to the extensive optimizations proposed, our experimental results show significant performance improvement against Shredder (about 1.87×) with comparable hardware, and 2.2× speedup was observed with newer GPUs (GTX 780).

## 6 Conclusion

In this paper, we present the design of a GPU-based Rabin fingerprinting algorithm for efficient redundancy elimination of network traffic. To maximize performance gains, a set of optimization strategies is proposed to address the algorithm-specific needs and domain-specific challenges such as the buffer scheme and sorting algorithm for network traffic analysis. Our evaluation shows that GPU-assisted Rabin fingerprinting can achieve up to 40 Gpbs throughput, greatly advancing the state of the art. In addition, we believe that the proposed indexed compact buffer scheme and approximate sorting can also be applied to other network applications leveraging GPU computation.

## Acknowledgments

This research was supported in part by the National Science Foundation of China under grants 61272190, 61572179 and 61173166, the Program for New Century Excellent Talents in University, and the Fundamental Research Funds for the Central Universities of China.

## References

[1] Thrust. [Online]. Available: https://developer.nvidia.com/Thrust, 2015.

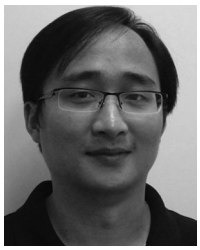
[2] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "EndRE: An end-system redundancy elimination service for enterprises," in *Proc. USENIX Conf. Netw. Syst. Design Implementation*, San Jose, CA, USA, Apr. 2010, pp. 419–432.

[3] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: The implications of universal redundant traffic elimination," in *Proc. ACM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Seattle, WA, USA, Aug. 2008 pp. 219–230.

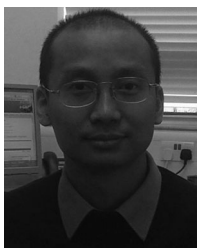
[4] A. Anand, V. Sekar, and A. Akella, "SmartRE: An architecture for coordinated network-wide redundancy elimination," in *Proc. ACM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Barcelona, Spain, Sep. 2009, pp. 87–98.

[5] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation," in *Proc. 10th USENIX Conf. File Storage Technol.*, San Jose, CA, USA, Feb. 2012, pp. 171–185.
[6] H. Chen, J. Sun, L. He, K. Li, and H. Tan, "BAG: Managing GPU as buffer cache in operating systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1393–1402, Jun. 2014.
[7] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proc. ACM SIGCOMM Conf.*, New Delhi, India, Sep. 2010 pp. 195–206.
[8] M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, and A. Davidson. (2009). CUDPP: CUDA data parallel primitives library. [Online]. Available: https://github.com/cudpp/cudpp
[9] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph algorithms at maximum warp," in *Proc. 16th ACM Symp. Principles Practice Parallel Programm.*, 2011, pp. 267–276.
[10] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Conf. Netw. Syst. Design Implementation*, 2011, pp. 1–14.
[11] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. 18th ACM Symp. Operating Syst. Principles*, Lake Louise, AB, Canada, Oct. 2001 pp. 174–187.
[12] M. Rabin, "Fingerprinting by random polynomials," Center Res. Comput. Technol., Harvard Univ., Cambridge, MA, USA, Tech. Rep. TR-CSE-03-01, 1981.
[13] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–10.
[14] S. H. Shen, A. Gember, A. Anand, and A. Akella, "REfactoring content overhearing to improve wireless performance," in *Proc. Annu. Int. Conf. Mobile Comput. Netw.*, Las Vegas, NV, USA, 2011 pp. 217–228.
[15] L. Shi, H. Chen, and J. H. Sun, "vCUDA: GPU-accelerated high-performance computing in virtual machine," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–11.
[16] H. Shojania, B. Li, and X. Wang, "Nuclei: GPU-accelerated many-core network coding," in *Proc. IEEE INFOCOM*, 2009, pp. 459–467.
[17] E. Sintorn and U. Assarsson, "Real-time approximate sorting for self shadowing and transparency in hair rendering," in *Proc. Symp. Interactive 3D Graph. Games*, 2008, pp. 157–162.
[18] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for network packet signature matching," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 175–184.
[19] N. Spring, and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *Proc. ACM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Stockholm, Sweden, Aug. 2000, pp. 87–95
[20] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *Proc. 12th Int. Symp. Recent Adv. Intrusion Detection*, 2009, pp. 265–283.
[21] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Parallelization and characterization of pattern matching using GPUs,"in *Proc. IEEE Int. Symp. Workload Characterization*, Austin, TX, USA, Nov. 2011 pp. 216–225.
[22] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: A multi-parallel intrusion detection architecture," in *Proc. 18th ACM Conf. Comput. Commun. Security*, Chicago, IL, USA, Oct. 2011 pp. 297–308.
[23] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-accelerated stateful packet processing framework," in *Proc. USENIX Annu. Tech. Conf.*, Philadelphia, PA, USA, Jun. 2014, pp. 321–332.
[24] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang, "Wire speed name lookup: A GPU-based approach," in *Proc. 10th USENIX Conf. Netw. Syst. Design Implementation*, Lombard, IL, USA, 2013 pp. 199–212.
[25] E. Zohar, I. Cidon, and O. O. Mokryn, "The power of prediction: Cloud bandwidth and cost reduction," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Toronto, ON, Canada, 2011, pp. 86–97.

**Jianhua Sun** received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2005. She is an associate professor at the College of Computer Science and Electronic Engineering, Hunan University, China. Her research interests are in security and operating systems. She has published more than 50 papers in journals and conferences, such as *IEEE Transactions on Parallel and Distributed Systems* and *IEEE Transactions on Computers*.



**Hao Chen** received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2005. He is currently a professor at the College of Computer Science and Electronic Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He has published more than 60 papers in journals and conferences, such as *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, IPDPS, IWQoS, HiPC, and ICPP. He is a member of the IEEE and the ACM.



**Ligang He** received the bachelor's and master's degrees from the Huazhong University of Science and Technology, Wuhan, China, and the PhD degree in computer science from the University of Warwick, United Kingdom. He was also a Postdoctoral researcher at the University of Cambridge, United Kingdom. In 2006, he joined the Department of Computer Science at the University of Warwick as an assistant professor, and then became an associate professor. His areas of interest are parallel and distributed computing, grid computing and cloud computing. He has published more than 70 papers in international conferences and journals, such as *IEEE TPDS*, IPDPS, Cluster, CCGrid, MASCOTS. He also served as a member of the program committee for many international conferences, and was the reviewer for a number of international journals, including *IEEE TPDS*, *IEEE TC*, *IEEE TASE*, etc. He is a member of the IEEE.

**Huailiang Tan** received the BS degree from Central South University, China, in 1992, and the MS degree from Hunan University, China, in 1995, and the PhD degree from Central South University, China, in 2001. He has more than eight years of industrial R&D experience in the field of information technology. He was a visiting scholar at Virginia Commonwealth University from 2010 to 2011. He is currently an associate professor at the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include embedded systems and GPU architectures.