Shredder: GPU-Accelerated Incremental Storage and Computation

Pramod Bhatotia[†] *Rodrigo Rodrigues*[†] *Akshat Verma*[‡]

[†]*Max Planck Institute for Software Systems (MPI-SWS) and* [‡]*IBM Research – India*

Abstract

Redundancy elimination using data deduplication and incremental data processing has emerged as an important technique to minimize storage and computation requirements in data center computing. In this paper, we present the design, implementation and evaluation of Shredder, a high performance content-based chunking framework for supporting incremental storage and computation systems. Shredder exploits the massively parallel processing power of GPUs to overcome the CPU bottlenecks of content-based chunking in a cost-effective manner. Unlike previous uses of GPUs, which have focused on applications where computation costs are dominant, Shredder is designed to operate in both compute-and data-intensive environments. To allow this, Shredder provides several novel optimizations aimed at reducing the cost of transferring data between host (CPU) and GPU, fully utilizing the multicore architecture at the host, and reducing GPU memory access latencies. With our optimizations, Shredder achieves a speedup of over 5X for chunking bandwidth compared to our optimized parallel implementation without a GPU on the same host system. Furthermore, we present two real world applications of Shredder: an extension to HDFS, which serves as a basis for incremental MapReduce computations, and an incremental cloud backup system. In both contexts, Shredder detects redundancies in the input data across successive runs, leading to significant savings in storage, computation, and end-to-end completion times.

1 Introduction

With the growth in popularity of Internet services, online data stored in data centers is increasing at an ever-growing pace. In 2010 alone, mankind is estimated to have produced $1,200$ exabytes of data [1]. As a result of this "data deluge," managing storage and computation over this data has become one of the most challenging tasks in data center computing.

A key observation that allows us to address this challenge is that a large fraction of the data that is produced and the computations performed over this data are redundant; hence, *not* storing redundant data or performing redundant computation can lead to significant savings in terms of both storage and computational resources. To make use of redundancy elimination, there exist a series of research and product proposals (detailed in §8) for performing *data deduplication* and *incremental computations*, which avoid storing or computing tasks based on redundant data, respectively.

Both data deduplication schemes and incremental computations rely on storage systems to detect duplicate content. In particular, the most effective way to perform this detection is using *content-based chunking*, a technique that was pioneered in the context of the LBFS [35] file system, where chunk boundaries within a file are dictated by the presence of certain content instead of a fixed offset. Even though content-based chunking is useful, it is a computationally demanding task. Chunking methods need to scan the entire file contents, computing a fingerprint over a sliding window of the data. This high computational cost has caused some systems to simplify the fingerprinting scheme by employing sampling techniques, which can lead to missed opportunities for eliminating redundancies [9]. In other cases, systems skip content-based chunking entirely, thus forgoing the opportunity to reuse identical content in similar, but not identical files [24]. Therefore, as we get flooded with increasing amounts of data, addressing this computational bottleneck becomes a pressing issue in the design of storage systems for data center-scale systems.

To address this issue we propose Shredder, a system for performing efficient content-based chunking to support scalable incremental storage and computations. Shredder builds on the observation that neither the exclusive use of multicore CPUs nor the specialized hardware accelerators is sufficient to deal with large-scale data in a cost-effective manner: multicore CPUs alone cannot sustain a high throughput, whereas the specialized hardware accelerators lack programmability for other tasks

and are costly. As an alternative, we explore employing modern GPUs to meet these high computational requirements (while, as evidenced by prior research [25, 28], also allowing for a low operational cost). The application of GPUs in this setting, however, raises a significant challenge — while GPUs have shown to produce performance improvements for computation intensive applications, where CPU dominates the overall cost envelope [25, 26, 28, 45, 46], it still remains to be proven that GPUs are equally as effective for data intensive applications, which need to perform large data transfers for a significantly smaller amount of processing.

To make the use of GPUs effective in the context of storage systems, we designed several novel techniques, which we apply to two proof-of-concept applications. In particular, Shredder makes the following technical contributions:

GPU acceleration framework. We identified three key challenges in using GPUs for data intensive applications, and addressed them with the following techniques:

- **Asynchronous execution.** To minimize the cost of transferring data between host (CPU) and GPU, we use a double buffering scheme. This enables GPUs to perform computations while simultaneously data is transferred in the background. To support this background data transfer, we also introduce a ring buffer of pinned memory regions.
- **Streaming pipeline.** To fully utilize the availability of a multicore architecture at the host, we use a pipelined execution for the different stages of content-based chunking.
- **Memory coalescing.** Finally, because of the high degree of parallelism, memory latencies in the GPU will be high due to the presence of random access across multiple bank rows of GPU memory, which leads to a higher number of conflicts. We address this problem with a cooperative memory access scheme, which reduces the number of fetch requests and bank conflicts.

Use cases. We present two applications of Shredder to accelerate storage systems. The first case study is a system called Inc-HDFS, a file-system that is based on HDFS but is designed to support incremental computations for MapReduce jobs. Inc-HDFS leverages Shredder to provide a mechanism for identifying similarities in the input data of consecutive runs of the same MapReduce job. In this way Inc-HDFS enables efficient incremental computation, where only the tasks whose inputs have changed need to be recomputed. The second case study is a backup architecture for a cloud environment, where VMs are periodically backed up. We use Shredder on a backup server and use content-based chunking to perform efficient deduplication and significantly improve backup bandwidth.

We present experimental results that establish the effectiveness of the individual techniques we propose, as well as the ability of Shredder to improve the performance of the two real-world storage systems.

The rest of the paper is organized as follows. In Section 2, we provide background on content-based chunking, and discuss specific architectural features of GPUs. An overview of the GPU acceleration framework and its scalability challenges are covered in Section 3. Section 4 presents present a detailed system design, namely several performance optimizations for increasing Shredder's throughput. We present the implementation and evaluation of Shredder in Section 5. We cover the two case studies in Section 6 and Section 7. Finally, we discuss related work in Section 8, and conclude in Section 9.

2 Background

In this section, we first present background on content-based chunking, to explain its cost and potential for parallelization. We then provide a brief overview of the massively parallel compute architecture of GPUs, namely their memory subsystem and its limitations.

2.1 Content-based Chunking

Identification of duplicate data blocks has been used for deduplication systems in the context of both storage [35, 41] and incremental computation frameworks [15]. For storage systems, the duplicate data blocks need not to be stored and, in the case of incremental computations, a sub-computation based on the duplicate content may be reused. Duplicate identification essentially consists of:

1. Chunking: This is the process of dividing the data set into chunks in a way that aids in the detection of duplicate data.
2. Hashing: This is the process of computing a collision-resistant hash of the chunk.
3. Matching: This is the process of checking if the hash for a chunk already exists in the index. If it exists then there is a duplicate chunk, else the chunk is new and its hash is added to the index.

This paper focuses on the design of chunking schemes (step 1), since this can be, in practice, one of the main bottlenecks of a system that tries to perform this class of optimizations [9, 24]. Thus we begin by giving some background on how chunking is performed.

One of the most popular approaches for content-based chunking is to compute a Rabin fingerprint [42] over sliding windows of w contiguous bytes. The hash values produced by the fingerprinting scheme are used to create chunk boundaries by starting new chunks whenever the computed hash matches one of a set of markers (e.g., its

value *mod p* is lower or equal to a constant). In more detail, given a *w*-bit sequence, it is represented as a polynomial of degree $w-1$ over the finite field $GF(2)$:

$$f(x) = m_0 + m_1 x + \cdots + m_{w-1} x^{w-1} \quad (1)$$

Given this polynomial, an irreducible polynomial $div(x)$ of degree *k* is chosen. The fingerprint of the original bit sequence is the remainder $r(x)$ obtained by division of $f(x)$ using $div(x)$. Chunk boundary is defined when the fingerprint takes some pre-defined specific values called markers. In addition, practical schemes define a minimum *min* and maximum *max* chunk size, which implies that after finding a marker the fingerprint computation can skip *min* bytes, and that a marker is always set when a total of *max* bytes (including the skipped portion) have been scanned without finding a marker. The minimum size limits the metadata overhead for index management and the maximum size limits the size of the RAM buffers that are required. Throughout the rest of the paper, we will use $min = 0$ and $max = \infty$ unless otherwise noted.

Rabin fingerprinting is computationally very expensive. To minimize the computation cost, there has been work on reducing chunking time by using sampling techniques, where only a subset of bytes are used for chunk identification (e.g., SampleByte [9]). However, such approaches are limiting because they are suited only for small sized chunks, as skipping a large number of bytes leads to missed opportunities for deduplication. Thus, Rabin fingerprinting still remains one of the most popular chunking schemes, and reducing its computational cost presents a fundamental challenge for improving systems that make use of duplicate identification.

When minimum and maximum chunk sizes are not required, chunking can be parallelized in a way that different threads operate on different parts of the data completely independent of each other, with the exception of a small overlap of the size of the sliding window (*w* bytes) near partition boundaries. Using *min* and *max* chunk sizes complicates this task, though schemes exist to achieve efficient parallelization in this setting [31, 33].

2.2 General-Purpose Computing on GPUs

GPU architecture. GPUs are highly parallel, multithreaded, many-core processors with tremendous computational power and very high memory bandwidth. The high computational power is derived from the specialized design of GPUs, where more transistors are devoted to simple data processing units (ALUs) rather than used to integrate sophisticated pre-fetchers, control flows and data caches. Hence, GPUs are well-suited for data-parallel computations with high arithmetic intensity rather than data caching and flow control.

Figure 1 illustrates a simplified architecture of a GPU. A GPU can be modeled as a set of Streaming Multiprocessors (SMs), each consisting of a set of scalar processor cores (SPs). An SM works as SIMT (Single Instruction, Multiple Threads), where the SPs of a multiprocessor execute the same instruction simultaneously but on different data elements. The data memory in the GPU is organized as multiple hierarchical spaces for threads in execution. The GPU has a large high-bandwidth device memory with high latency. Each SM also contains a very fast, low latency on-chip shared memory to be shared among its SPs. Also, each thread has access to a private local memory.

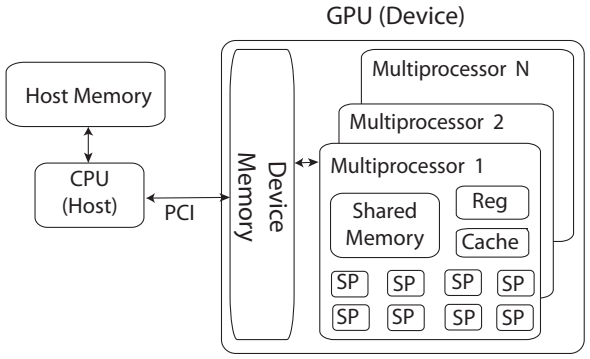

Figure 1: A simplified view of the GPU architecture.

Overall, a GPU architecture differs from a traditional processor architecture in the following ways: (*i*) an order of magnitude higher number of arithmetic units; (*ii*) minimal support for prefetching and buffers for outstanding instructions; (*iii*) high memory access latencies and higher memory bandwidth.

Programming model. The CUDA [6] programming model is amongst the most popular programming models to extract parallelism and scale applications on GPUs. In this programming model, a host program runs on the CPU and launches a kernel program to be executed on the GPU device in parallel. The kernel executes as a grid of one or more thread blocks, each of which is dynamically scheduled to be executed on a single SM. Each thread block consists of a group of threads that cooperate with each other by synchronizing their execution and sharing multiprocessor resources such as shared memory and registers. Threads within a thread block get executed on a multiprocessor in scheduling units of 32 threads, called a warp. A half-warp is either the first or second half of a warp.

2.3 SDRAM Access Model

Offloading chunking to the GPU requires a large amount of data to be transferred from the host to the GPU memory. Thus, we need to understand the performance of the memory subsystem in the GPU, since it is critical to chunking performance.

The global memory in the Nvidia C2050 GPU is GDDR5, which is based on the DDR3 memory architecture [2]. Memory is arranged into banks and banks are organized into rows. Every bank also has a sense amplifier, into which a row must be loaded before any data from the row can be read by the GPU. Whenever a memory location is accessed, an *ACT* command selects the corresponding bank and brings the row containing the memory location into a sense amplifier. The appropriate word is then transferred from the sense amplifier. When an access to a second memory location is performed within the same row, the data is transferred directly from the sense amplifier. On the other hand, if the data is accessed from a different row in the bank, a *PRE* (pre-charge) command writes the previous data back from the sense amplifier to the memory row. A second *ACT* command is performed to bring the row into the sense amplifier.

Note that both *ACT* and *PRE* commands are high latency operations that contribute significantly to overall memory latency. If multiple threads access data from different rows of the same bank in parallel, that sense amplifier is continually activated (*ACT*) and pre-charged (*PRE*) with different rows, leading to a phenomenon called bank conflict. In particular, a high degree of uncoordinated parallel access to the memory subsystem is likely to result in a large number of bank conflicts.

3 System Overview and Challenges

In this section, we first present the basic design of Shredder. Next, we explain the main challenges in scaling up our basic design.

3.1 Basic GPU-Accelerated Framework

Figure 2 depicts the workflow of the basic design for the Shredder chunking service. In this initial design, a multithreaded program running in user mode on the host (i.e., on the CPU) drives the GPU-based computations. The framework is composed of four major modules. First, the *Reader* thread on the host receives the data stream (e.g., from a SAN), and places it in the memory of the host for content-based chunking. After that, the *Transfer* thread allocates global memory on the GPU and uses the DMA controller to transfer input data from the host memory to the allocated GPU (device) memory. Once the data transfer from the CPU to the GPU is complete, the host launches the *Chunking* kernel for parallel sliding window computations on the GPU. Once the chunking kernel finds all resulting chunk boundaries for the input data, the *Store* thread transfers the resulting chunk boundaries from the device memory to the host memory. When minimum and maximum chunk sizes are set, the *Store* thread also adjusts the chunk set accordingly. Thereafter, the *Store* thread uses an upcall to notify the chunk boundaries to the application that is using the Shredder library.

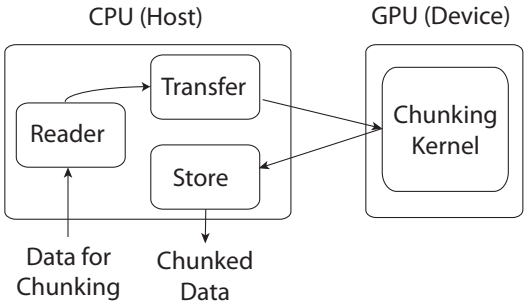

Figure 2: Basic workflow of Shredder.

The chunking kernel is responsible for performing parallel content-based chunking of the data present in the global memory of the GPU. Accesses to the data are performed by multiple threads that are created on the GPU by launching the chunking kernel. The data in the GPU memory is divided into equal sized sub-streams, as many as the number of threads. Each thread is responsible for handling one of these sub-streams. For each sub-stream, a thread computes a Rabin fingerprint in a sliding window manner. In particular, each thread examines a 48-byte region from its assigned sub-stream, and computes the Rabin fingerprint for the selected region. The thread compares the resulting low-order 13 bits of the region's fingerprint with a pre-defined marker. This leads to an expected chunk size of 4 KB. If the fingerprint matches the marker then the thread defines that particular region as the end of a chunk boundary. The thread continues to compute the Rabin fingerprint in a sliding window manner in search of new chunk boundaries by shifting a byte forward in the sub-stream, and repeating this process.

3.2 Scalability Challenges

The basic design for Shredder that we presented in the previous section corresponds to the traditional way in which GPU-assisted applications are implemented. This design has proven to be sufficient for computation-intensive applications, where the computation costs can dwarf the cost of transferring the data to the GPU memory and accessing that memory from the GPU's cores. However, it results in only modest performance gains for data intensive applications that perform single-pass processing over large amounts of data, with a computational cost that is significantly lower than traditional GPU-assisted applications.

To understand why this is the case, we present in Table 1 some key performance characteristics of a specific GPU architecture (NVidia Tesla C2050), which helps us explain some important bottlenecks for GPU-accelerated applications. In particular, and as we will demonstrate in subsequent sections, we identified the following bottle-

Parameter	Value
GPU Processing Capacity	1030 GFlops
Reader (I/O) Bandwidth	2 GBps
Host-to-Device Bandwidth	5.406 GBps
Device-to-Host Bandwidth	5.129 GBps
Device Memory Latency	400 - 600 cycles
Device Memory Bandwidth	144 GBps
Shared Memory Latency	L1 latency (a few cycles)

Table 1: Performance characteristics of the GPU (NVidia Tesla C2050)

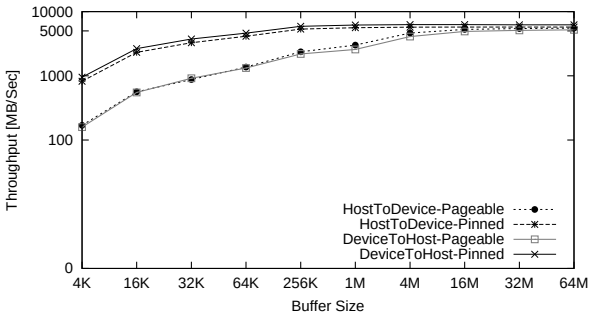

Figure 3: Bandwidth test between host and device.

necks in the basic design of Shredder.

GPU device memory bottleneck. The fact that data needs to be transferred to the GPU memory before being processed by the GPU represents a serial dependency: such processing only starts to execute after the corresponding transfer concludes.

Host bottleneck. The host machine performs three serialized steps (performed by the Reader, Transfer, and Store threads) in each iteration. Since these three steps are inherently dependent on each other for a given input buffer, this serial execution becomes a bottleneck at host. Also, given the availability of multicore architecture at the host, this serialized execution leads to an underutilization of resources at host.

High memory latencies and bank conflicts. The global device memory on the GPU has a high latency, of the order of 400 to 600 cycles. This works well for HPC algorithms, which are quadratic $O(N^2)$ or a higher degree polynomial in the input size N, since the computation time hides the memory access latencies. Chunking is also compute intensive, but it is only linear in the input size ($O(N)$, though the constants are high). Hence, even though the problem is compute intensive on traditional CPUs, on a GPU with an order of magnitude larger number of scalar cores, the problem becomes memory-intensive. In particular, the less sophisticated memory subsystem of the GPU (without prefetching or data caching support) is stressed by frequent memory access by a massive number of threads in parallel. Furthermore, a higher degree of parallelism causes memory to be accessed randomly across multiple bank rows, and leads to a very high number of bank conflicts. As a result, it becomes difficult to hide the latencies of accesses to the device memory.

4 Shredder Optimizations

In this section, we describe several novel optimizations that extend the basic design to overcome the challenges we highlighted in the previous section.

4.1 Device Memory Bottlenecks

4.1.1 Concurrent Copy and Execution

The main challenge we need to overcome is the fact that traditional GPU-assisted applications that follow the basic design were designed for a scenario where the cost of transferring data to the GPU is significantly outweighed by the actual computation cost. In particular, the basic design serializes the execution of copying data to the GPU memory and consuming the data from that memory by the Kernel thread. This serialized execution may not suit the needs of data intensive applications, where the cost of the data transfer step becomes a more significant fraction of the overall computation time.

To understand the magnitude of this problem, we measured the overhead of a DMA transfer of data between the host and the device memory over the PCIe link connected to GPU. Figure 3 summarizes the effective bandwidth between host memory and device memory for different buffer sizes. We measured the bandwidth both ways between the host and the device to gauge the DMA overhead for the Transfer and the Store thread. Note that the effective bandwidth is a property of the DMA controller and the PCI bus, and it is independent of the number of threads launched in the GPU. In this experiment, we also varied the buffer type allocated for the host memory region, which is allocated either as pageable or pinned memory regions. (The need for pinned memory will become apparent shortly.)

Highlights. Our measurements demonstrate the following: (i) small sized buffer transfers are more expensive than those using large sized buffers; (ii) the throughput saturates for buffer sizes larger than 32 MB (for pageable memory region) and 256 KB (for pinned memory region); (iii) for large sized buffers (greater than 32 MB), the throughput difference between pageable and pinned memory regions is not significant; and (iv) the effective bandwidth of the PCIe bus for data transfer is on the order of 5 GB/sec, whereas the global device memory access time by scalar processors in GPUs is on the order of

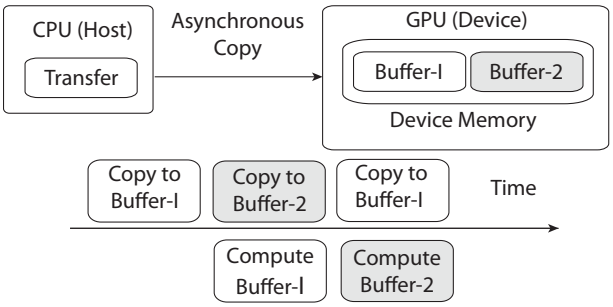

Figure 4: Concurrent copy and execution.

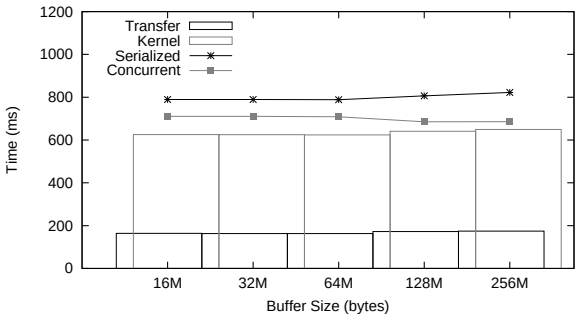

Figure 5: Normalized overlap time of communication with computation with varied buffer sizes for 1GB data.

144 GB/sec, an order of magnitude higher.

Implications. The time spent to chunk a given buffer is split between the memory transfer and the kernel computation. For a non-optimized implementation of the chunking computation, we spend approximately 25% of the time performing the transfer. Once we optimize the processing in the GPU, the host to GPU memory transfer may become an even greater burden on the overall performance.

Optimization. In order to avoid the serialized execution of the copy and data consumption steps, we propose to overlap the copy and the execution phases, thus allowing for the concurrent execution of data communication and the chunking kernel computations. To enable this, we designed a double buffering technique as shown in Figure 4, where we partition the device memory into twin buffers. These twin buffers will be alternatively used for communication and computation. In this scheme, the host asynchronously copies the data into the first buffer and, in the background, the device works on the previously filled second buffer. To be able to support asynchronous communication, the host buffer is allocated as a pinned memory region, which prevents the region from being swapped out by the pager.

Effectiveness. Figure 5 shows the effectiveness of the double buffering approach, where the histogram for transfer and kernel execution shows a 30% time overlap between the concurrent copy and computation. Even though the total time taken for concurrent copy and execution (`Concurrent`) is reduced by only 15% as compared to the serialized execution (`Serialized`), it is important to note that the total time is now dictated solely by the compute time. Hence, double buffering is able to remove the data copying time from the critical path, allowing us to focus only on optimizing the computation time in the GPU (which we address in § 4.3).

To support the concurrent copy and execution, however, requires us to pin memory at the host, which reduces the memory allocation performance at the host. We next present an optimization to handle this side effect and ensure that double buffering leads to an end-to-end increase in chunking bandwidth.

4.1.2 Circular Ring Pinned Memory Buffers

As explained above, the double buffering requires an asynchronous copy between host memory and device memory. To support this asynchronous data transfer, the host side buffer should be allocated as a pinned memory region. This locks the corresponding page so that accessing that region does not result in a page fault until the region is subsequently unpinned.

To quantify the allocation overheads of using a pinned memory region, we compared the time required for dynamic memory allocation (using `malloc`) and pinned memory allocation (using the CUDA memory allocator wrapper). Since Linux follows an optimistic memory allocation strategy, where the actual allocation is deferred until memory initialization, in our measurements we initialized the memory region (using `bzero`) to force the kernel to allocate the desired buffer size. Figure 6 compares the allocation overhead of pageable and pinned memory for different buffer sizes.

Highlights. The important take away points are the following: (i) pinned memory allocation is more expensive than the normal dynamic memory allocation; and (ii) an adverse side effect of having too many pinned memory pages is that it can increase paging activity for unpinned pages, which degrades performance.

Implications. The main implication for our system design is that we need to minimize the allocation of pinned memory region buffers, to avoid increased paging activity or even thrashing.

Optimization. To minimize the allocation of pinned memory region while restricting ourselves to using the CUDA architecture, we designed a circular ring buffer built from a pinned memory region, as shown in Figure 7, with the property that the number of buffers can be kept low (namely as low as the number of stages in the streaming pipeline, as described in §4.2). The pinned

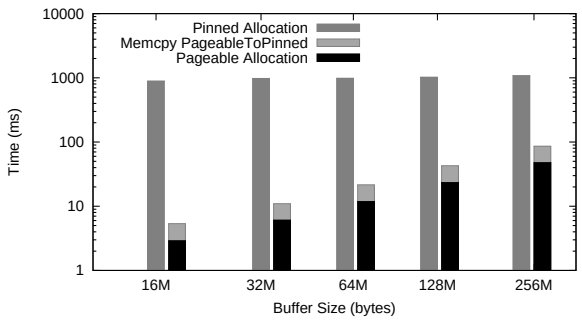

Figure 6: Comparison of allocation overhead of pageable with pinned memory region.

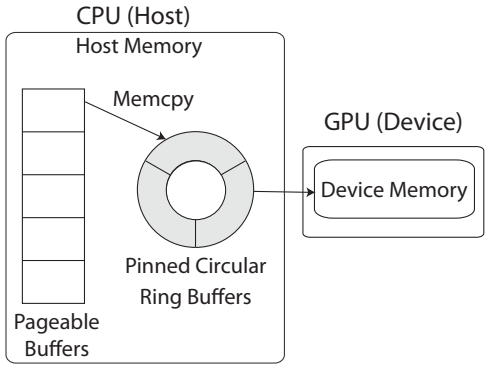

Figure 7: Ring buffer for the pinned memory region.

regions in the circular buffer are allocated only once during the system initialization, and thereafter are reused in a round-robin fashion after the transfer between the host and the device memory is complete. This allows us to keep the overhead of costly memory allocation negligible and have sufficient memory pages for other tasks.

Effectiveness. Figure 6 shows the effectiveness of our approach, where we compare the time for allocating pageable and pinned memory regions. Since we incur the additional cost of copying the data from pageable memory to the pinned memory region, we add this cost to the total cost of using pageable buffers. Overall, our approach is faster by an order of magnitude, which highlights the importance of this optimization.

4.2 Host Bottleneck

The previously stated optimizations alleviate the device memory bottleneck for DMA transfers, and allow the device to focus on performing the actual computation. However, the host side modules can still become a bottleneck due to the serialized execution of the following stages (`Reader`→`Transfer`→`Kernel`→`Store`). In this case, the fact that all four modules are serially executed leads to an underutilization of resources at the host side.

To quantify this underutilization at the host, we measured the number of idle spare cycles per core after the launch of the asynchronous execution of the kernel. Table 2 shows the number of RDTSC tick cycles for different buffer sizes. The RDTSC [8] (Read-Time Stamp Counter) instruction keeps an accurate count of every cycle that occurs on the processor for monitoring the performance. The device execution time captures the asynchronous copy and execution of the kernel, and the host kernel launch time measures the time for the host to launch the asynchronous copy and the chunking kernel.

Highlights. These measurements highlight the following: (i) the kernel launch time is negligible compared to the total execution time for the kernel; (ii) the host is idle during the device execution time; and (ii) the host has a large number of spare cycles per core, even with a small sized buffer.

Buffer size (bytes)	16M	32M	64M	128M	256M
Device execution time (ms)	11.39	22.74	42.85	85.7	171.4
Host kernel launch time (ms)	0.03	0.03	0.03	0.08	0.09
Total execution time (ms)	11.42	22.77	42.88	85.78	171.49
Host `RDTSC` ticks @ 2.67 GHz	3.0e7	6.1e7	1.1e8	2.7e8	5.3e8

Table 2: Host spare cycles per core due to asynchronous data-transfer and kernel launch.

Implications. Given the prevalence of host systems running on multicore architectures, the sequential execution of the various components leads to the underutilization of the host resources, and therefore these resources should be used to perform other operations.

Optimization. To utilize these spare cycles at the host, Shredder makes use of a multi-stage streaming pipeline as shown in Figure 8. The goal of this design is that once the Reader thread finishes writing the data in the host main memory, it immediately proceeds to handling a new window of data in the stream. Similarly, the other threads follow this pipelined execution without waiting for the next stage to finish.

To handle the specific characteristics of our pipeline stages, we use different design strategies for different modules. Since the Reader and Store modules deal with I/O, they are implemented as Asynchronous I/O (as described in §5.2.1), whereas the transfer and kernel threads are implemented using multi-buffering (a generalization of the double buffering scheme described in §4.1.1).

Effectiveness. Figure 9 shows the average speedup from using our streaming pipeline, measured as the ratio of time taken by a sequential execution to the time taken by our multi-stage pipeline. We varied the number of pipeline stages that can be executed simultaneously (by

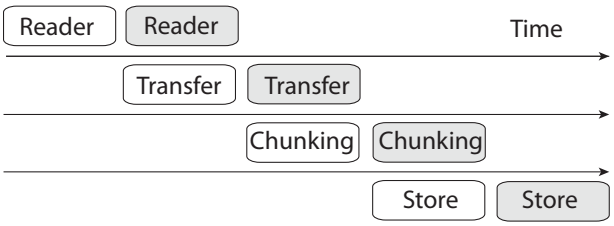

Figure 8: Multi-staged streaming pipeline.

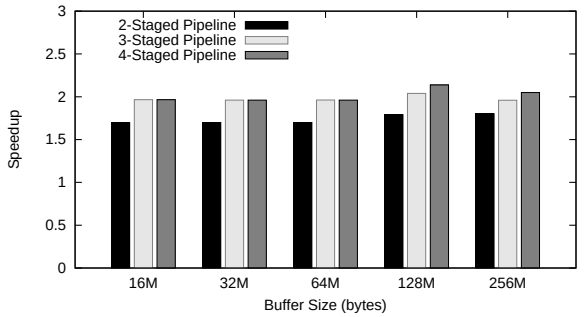

Figure 9: Speedup for streaming pipelined execution.

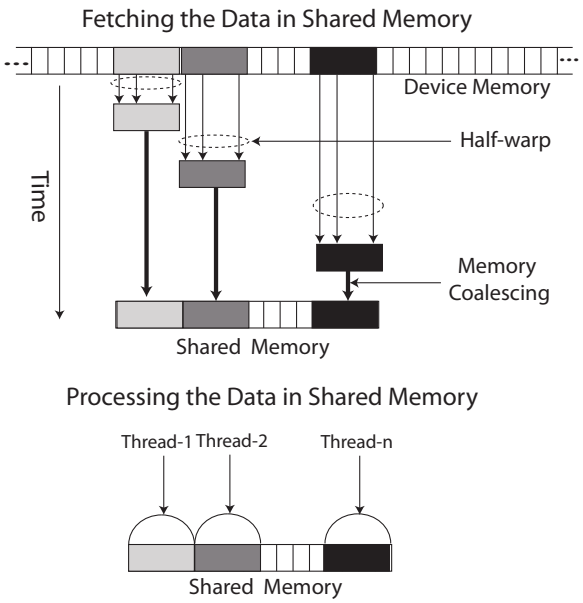

Figure 10: Memory coalescing to fetch data from global device memory to the shared memory.

restricting the number of buffers that are admitted to the pipeline) from 2 to 4. The results show that a full pipeline with all four stages being executed simultaneously achieves a speedup of 2; the reason why this is below the theoretical maximum of a 4X gain is that the various stages do not have equal cost.

4.3 Device Memory Conflicts

We have observed (in Figure 5) that the chunking kernel dominates the overall time spent by the GPU. In this context, it is crucial to try to minimize the contribution of the device memory access latency to the overall cost.

Highlights. The very high access latencies of the device memory (on the order of 400-600 cycles @ 1.15 GHz) and the lack of support for data caching and prefetching can imply a significant overhead in the overall execution time of the chunking kernel.

Implications. The hierarchical memory of GPUs provides us an opportunity to hide the latencies of the global device memory by instead making careful use of the low latency shared memory. (Recall from § 2.2 that the shared memory is a fast and low latency on-chip memory which is shared among a subset of the GPU's scalar processors.) However, fetching data from global to the shared memory requires us to be careful to avoid bank conflicts, which can negatively impact the performance of the GPU memory subsystem. This implies that we should try to improve the inter-thread coordination in fetching data from the device global memory to avoid these bank conflicts.

Optimization. We designed a thread cooperation mechanism to optimize the process of fetching data from the global memory to the shared memory, as shown in Figure 10. In this scheme, a single block that is needed by a given thread is fetched at a time, but each block is fetched with the cooperation of all the threads, and their coordination to avoid bank conflicts. The idea is to iterate over all data blocks for all threads in a thread block, fetch one data block at a time in a way that different threads request consecutive but non-conflicting parts of the data block, and then, after all data blocks are fetched, let each thread work on its respective blocks independently. This is feasible since threads in a warp (or half-warp) execute the same stream of instructions (SIMT). Figure 10 depicts how threads in a half-warp cooperate with each other to fetch different blocks sequentially in time.

In order to ensure that the requests made by different threads when fetching different parts of the same data block do not conflict, we followed the best practices suggested by the device manufacturer to ensure these requests correspond to a single access to one row in a bank [6, 7, 44]. In particular, Shredder lets multiple threads of a half-warp read a contiguous memory interval simultaneously, under following conditions: (i) the size of the memory element accessed by each thread is either 4, 8, or 16 bytes; (ii) the elements form a contiguous block of memory; i.e, the Nth element is accessed by the Nth thread in the half-warp; and (iii) the address of the first element is aligned at a boundary of a multiple of 16 bytes.

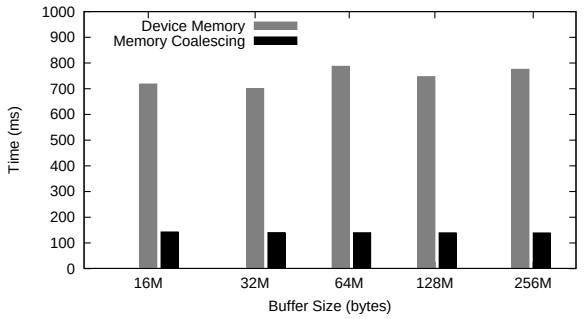

Figure 11: Normalized chunking kernel time with varied buffer-sizes for 1 GB data.

Effectiveness. Figure 11 shows the effectiveness of the memory coalescing optimization, where we compare the execution time for the chunking kernel using the normal device memory access and the optimized version. The results show that we improve performance by a factor of 8 by reducing bank conflicts. Since the granularity of memory coalescing is 48 KB (which is the size for the shared memory per thread block), we do not see any impact from varying buffer sizes (16 MB to 512 MB), and the benefits are consistent across different buffer sizes.

5 Implementation and Evaluation

We implemented Shredder in CUDA [6], and for an experimental comparison, we also implemented an optimized parallel host-only version of content-based chunking. This section describes these implementations and evaluates them.

5.1 Host-Only Chunking using `pthreads`

We implemented a library for parallel content-based chunking on SMPs using POSIX pthreads. We derived parallelism by creating pthreads that operate in different data regions using a Single Program Multiple Data (SPMD) strategy and communicate using a shared memory data structure. At a high level, the implementation works as follows: (1) divide the input data equally in fixed-size regions among N threads; (2) invoke the Rabin fingerprint-based chunking algorithm in parallel on N different regions; (3) synchronize neighboring threads in the end to merge the resulting chunk boundaries.

An issue that arises is that dynamic memory allocation can become a bottleneck due to the the serialization required to avoid race conditions. To address this, we used the Hoard memory allocator [13] instead of `malloc`.

5.2 Shredder Implementation

The Shredder library implementation comprises two main modules, the *host driver* and the *GPU kernel*. The host driver runs the control part of the system as a multi-threaded process on the host CPU running Linux. The GPU kernel uses one or more GPUs as co-processors for accelerating the SIMT code, and is implemented using the CUDA programming model from the NVidia GPGPU toolkit [6]. Next we explain key implementation details for both modules.

5.2.1 Host Driver

The host driver module is responsible for reading the input data either from the network or the disk and transferring the data to the GPU memory. Once the data is transferred then the host process dispatches the GPU kernel code in the form of RPCs supported by the CUDA toolkit. The host driver has two types of functionality: (1) the Reader/Store threads deal with reading and writing data from and to I/O channels; and (2) the Transfer thread is responsible for moving data between the host and the GPU memory. We implemented the Reader/Store threads using Asynchronous I/O and the Transfer thread using CUDA RPCs and page-pinned memory.

Asynchronous I/O (AIO). With asynchronous non-blocking I/O, it is possible to overlap processing and I/O by initiating multiple transfers at the same time. In AIO, the read request returns immediately, indicating that the read was successfully initiated. The application can then perform other processing while the background read operation completes. When the read response arrives, a signal registered with the read request is triggered to signal the completion of the I/O transaction.

Since the Reader/Store threads operate at the granularity of buffers, a single input file I/O may lead to issuing multiple `aio-read` system calls. To minimize the overhead of multiple context switches per buffer, we used `lio-listio` to initiate multiple transfers at the same time in the context of a single system call (meaning one kernel context switch).

5.2.2 GPU Kernel

The GPU kernel can be trivially derived from the C equivalent code by implementing a collection of functions in equivalent CUDA C with some assembly annotations, plus different access mechanisms for data layout in the GPU memory. However, an efficient implementation of the GPU kernel requires a bit more understanding of vector computations and the GPU architecture. We briefly describe some of these considerations.

Kernel optimizations. We have implemented minor kernel optimizations to exploit vector computation in GPUs. In particular, we used loop unrolling and instruction-level optimizations for the core Rabin fingerprint block. These changes are important because of the simplified GPU architecture, which lacks out-of-order execution, pipeline stalling in register usage, or instruction reordering to eliminate Read-after-Write (RAW) depen-

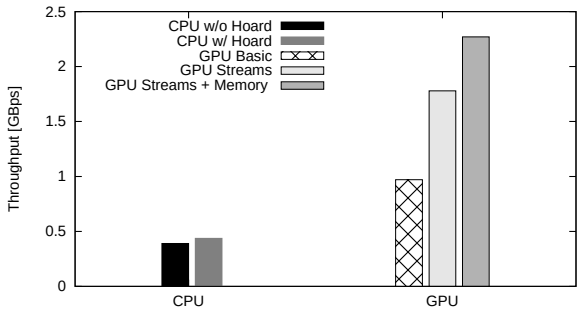

Figure 12: Throughput comparison of content-based chunking between CPU and GPU versions.

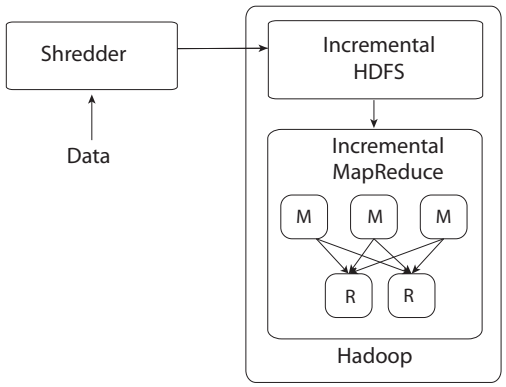

Figure 13: Incremental computations using Shredder.

dencies.

Warp divergence. Since the GPU architecture is Single Instruction Multiple Threads (SIMT), if threads in a warp diverge on a data-dependent conditional branch, then the warp is serially executed until all threads in it converge to the same execution path. To avoid a performance dip due to this divergence in warp execution, we carefully restructured the algorithm to have little code divergence within a warp, by minimizing the code path under data-dependent conditional branches.

5.3 Evaluation of Shredder

We now present our experimental evaluation of the performance of Shredder.

Experimental setup. We used a fermi-based GPU architecture, namely the Tesla C2050 GPU consisting of 448 processor cores (SPs). It is organized as a set of 14 SMs each consisting of 32 SPs running at 1.15 GHz. It has 2.6 GB of off-chip global GPU memory providing a peak memory bandwidth of 144 GB/s. Each SM has 32768 registers and 48 KB of local on-chip shared memory, shared between its scalar cores.

We also used an Intel Xeon processor based system as the host CPU machine. The host system consists of 12 Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz with 48 GB of main memory. The host machine is running Linux with kernel 2.6.38 in 64-bit mode, additionally patched with GPU direct technology [4] (for SAN devices). The GCC 4.3.2 compiler (with -O3) was used to compile the source code of the host library. The GPU code is compiled using the CUDA toolkit 4.0 with NVidia driver version 270.41.03. The posix implementation is run with 12 threads.

Results. We measure the effectiveness of GPU-accelerated content-based chunking by comparing the performance of different versions of the host-only and GPU based implementation, as shown in Figure 12. We compare the chunking throughput for the pthreads implementation with and without using the Hoard memory allocator. For the GPU implementation, we compared the performance of the system with different optimizations turned on, to gauge their effectiveness. In particular, GPU Basic represents a basic implementation without any optimizations. The GPU Streams version includes the optimization to remove host and device bottlenecks using double buffering and a 4-stage pipeline. Lastly GPU Streams + Memory represents a version with all optimizations, including memory coalescing.

Our results show that a naive GPU implementation can lead to a $2X$ improvement over a host-only optimized implementation. The observation clearly highlights the potential of GPUs to alleviate computational bottlenecks. However, this implementation does not remove chunking as a bottleneck since SAN bandwidths on typical data servers exceed 10 Gbps. Incorporating the optimizations lead to Shredder outperforming the host-only implementation by a factor of over $5X$.

6 Case Study I: Incremental Computations

This section presents a case study of applying Shredder in the context of incremental computations. First we review Incoop, a system for bulk incremental processing, and then describe how we used Shredder to improve it.

6.1 Background: Incremental MapReduce

Incoop [15] is a generic MapReduce framework for incremental computations. Incoop leverages the fact that data sets that are processed by bulk data processing frameworks like MapReduce evolve slowly, and often the same computation needs to be performed repeatedly on this changing data (such as computing PageRank on every new web crawl) [23, 34, 36]. Incoop aims at processing this data incrementally, by avoiding recomputing parts of the computation that did not change, and transparently, by being backwards-compatible with the interface used by MapReduce frameworks.

To achieve these goals, Incoop employs a fine-grained result reuse mechanism, which captures a dependence

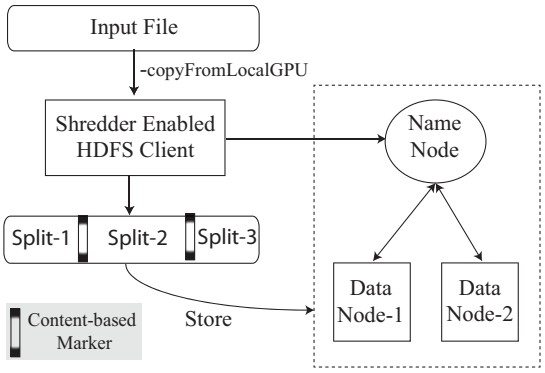

Figure 14: Shredder enabled chunking in Inc-HDFS.

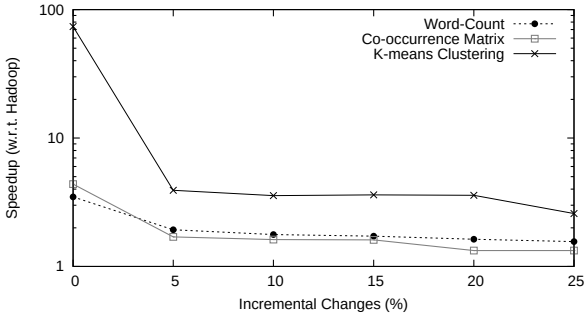

Figure 15: Speedup for incremental computation

graph among inputs and sub-computations, propagates changes along that graph so that only sub-computations that have changed need to be recomputed, and uses memoization to be able to reuse outputs from sub-computations whose inputs did not change. Incoop uses the Inc-HDFS file system (an extension to HDFS) to identify changes in the input and propagate them.

6.2 GPU-Accelerated Incremental HDFS

We use Shredder to support Incoop by designing a GPU-accelerated Incremental HDFS (Inc-HDFS), which is integrated with Incoop as shown in Figure 13. Inc-HDFS leverages Shredder to perform content-based chunking instead of using fixed-size chunking as in the original HDFS, thus ensuring that small changes to the input lead to small changes in the set of chunks that are provided as input to Map tasks. This enables the results of the computations performed by most Map tasks to be reused.

6.3 Implementation and Evaluation

We built our prototype GPU-accelerated Inc-HDFS on Hadoop-0.20.2. It is implemented as an extension to HDFS, where the computationally expensive chunking is offloaded to the Shredder-enabled HDFS client (as shown in Figure 14), before uploading chunks to the respective data nodes that will be storing them.

Inc-HDFS client. We integrated the Shredder library with Inc-HDFS client using a JAVA-CUDA interface. Once the data upload function is invoked, the Shredder library notifies the chunk boundaries to the Store thread, which in turn pushes the chunks from the memory of the client to the data nodes of HDFS.

Semantic chunking framework. The default behavior of the Shredder library is to split the input file into variable-length chunks based on the contents. However, since chunking is oblivious to the semantics of the input data, this could cause chunk boundaries to be placed anywhere, including, for instance, in the middle of a record that should not be broken. To address this, we leverage the fact that the MapReduce framework relies on the `InputFormat` class of the job to split up the input file(s) into logical InputSplits, each of which is then assigned to an individual Map task. We reuse this class to ensure that we respect the record boundaries in the chunking process.

HDFS shell. We extended the HDFS shell interface to invoke content-based chunking using the Shredder implementation. In particular, the shell interface offers new command (in addition to `copyFromLocal`) for uploading data in Inc-HDFS: `copyFromLocalGPU`.

Evaluation. We evaluated the effectiveness of incremental computations by measuring the speedups w.r.t. Hadoop for varying percentages of changes in the input data. Figure 15 shows the performance gains on a 20-node cluster, where all three MapReduce applications (K-means, Word-Count, Co-occurrence Matrix) show significant improvement in run-time for incremental runs. The effectiveness of the incremental approach degrades as the percentage of changes in the data set increases. Note that this experiment is not meant to highlight the speedup enabled by GPU acceleration, but instead shows how, after the data is chunked using Shredder, detecting duplicates at the storage level can imply savings in computation time.

7 Case Study II: Incremental Storage

In this section, we present our second case study where we use Shredder in the context of a consolidated incremental backup system.

7.1 Background: Cloud Backup

Figure 16 describes our target architecture, which is typical of cloud back-ends. Applications are deployed on virtual machines hosted on physical servers. The file system images of the virtual machines are hosted in a virtual machine image repository stored in a SAN volume. In this scenario, the backup process works in the following manner. Periodically, full image snapshots are taken for all the VM images that need to be backed up.

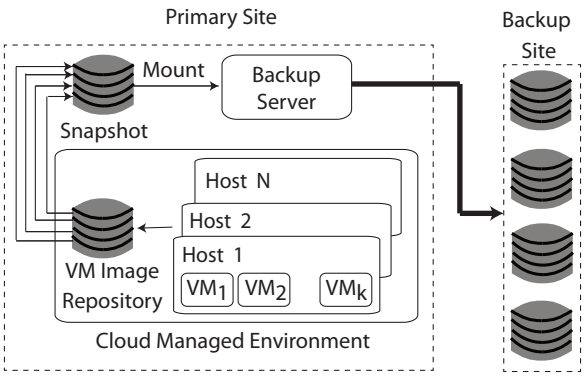

Figure 16: A typical cloud backup architecture

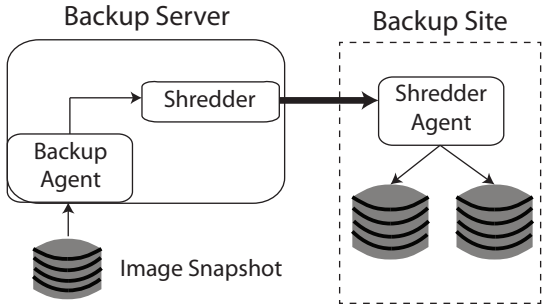

Figure 17: GPU-accelerated consolidated backup setup

The core of the backup process is a backup server and a backup agent running inside the backup server. The image snapshots are mounted by the backup agent. The backup server performs the actual backup of the image snapshots onto disks or tapes. The consolidated or centralized data backup process ensures compliance of all virtual machines with the agreed upon backup policy. Backup servers typically have very high I/O bandwidth since, in enterprise environments, all operations are typically performed on a SAN [30]. Furthermore, the use of physical servers allows multiple dedicated ports to be employed solely for the backup process.

7.2 GPU-Accelerated Data Deduplication

The centralized backup process is eminently suitable for deduplication via content-based chunking, as most images in a data-center environment are standardized. Hence, virtual machines share a large number of files and a typical backup process would unnecessarily copy the same content multiple times. To exploit this fact, we integrate Shredder with the backup server, thus enabling data to be pushed to the backup site at a high rate while simultaneously exploiting opportunities for savings.

The Reader thread on the backup server reads the incoming data and pushes that into Shredder to form chunks. Once the chunks are formed, the Store thread computes a hash for the overall chunk, and pushes the chunks in the backup setup as a separate pipeline stage. Thereafter, these hashes collected for the chunks are batched together to enqueue in an index lookup queue. Finally, a lookup thread picks up the enqueued chunk fingerprints and looks up in the index whether a particular chunk needs to be backed up or is already present in the backup site. If a chunk already exists, a pointer to the original chunk is transferred instead of the chunk data. We deploy an additional Shredder agent residing on the backup site, which receives all the chunks and pointers and recreates the original uncompressed data. The overall architecture for integrating Shredder in a cloud backup system is described in Figure 17.

7.3 Implementation and Evaluation

Since high bandwidth fibre channel adapters are fairly expensive, we could not recreate the high I/O rate of modern backup servers in our testbed. Hence, we used a memory-driven emulation environment to experimentally validate the performance of Shredder. On our backup agent, we keep a master image in memory using memcached [5]. The backup agent creates new file system images from the master image by replacing part of the content from the master image using a predefined similarity table. The master image is divided into segments. The image similarity table contains a probability of each segment being replaced by a different content. The agent uses these probabilities to decide which segments in the master image will be replaced. The image generation rate is kept at 10 Gbps to closely simulate the I/O processing rate of modern X-series employed for I/O processing applications [30].

In this experiment, we also enable the requirement of a minimum and maximum chunk size, as used in practice by many commercial backup systems. As mentioned in Section 3, our current implementation of Shredder is not optimized for including a minimum and maximum chunk size, since the data that is skipped after a chunk boundary is still scanned for computing a Rabin fingerprint on the GPU, and only after all the chunk boundaries are collected will the Store thread discard all chunk boundaries within the minimum chunk size limit. As future work, we intend to address this limitation using more efficient techniques that were proposed in the literature [31, 33].

As a result of this limitation, we observe in Figure 18 that we are able to achieve a speedup of only 2.5X in backup bandwidth compared to the pthread implementation, but still we manage to keep the backup bandwidth close to the target 10 Gbps. The results also show that even though the chunking process operates independently of the degree of similarity in input data, the backup bandwidth decreases when the similarity between the data decreases. This is not a limitation

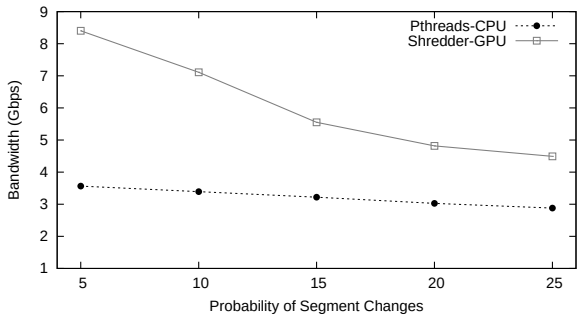

Figure 18: Backup bandwidth improvement due to Shredder with varying image similarity ratios

of our chunking scheme but of the unoptimized index lookup and network access, which reduces the backup bandwidth. Combined with optimized index maintenance (e.g., [18]), Shredder is likely to achieve the target backup bandwidth for the entire spectrum of content similarity.

8 Related Work

Our work builds on contributions from several different areas, which we briefly survey.

GPU-accelerated systems. GPUs were initially designed for graphics rendering, but, because of their cost-effectiveness, they were quickly adopted by the HPC community for scientific computations [3, 37]. Recently, the systems research community has leveraged GPUs for building other systems. In particular, PacketShader [25] is a software router for general packet processing, and SSLShader [28] uses GPUs in web servers to efficiently perform cryptographic operations. GPUs have also been used to accelerate functions such as pattern matching [46], network coding [45], and complex cryptographic operations [26]. In our work, we explored the potential of GPUs for large scale data, which raises challenges due to the overheads of data transfer.

Recently, GPUs have been shown to accelerate storage systems. In particular, Gibraltar RAID [17], uses GPUs in software-based RAID controllers for performing high-performance calculations of error correcting codes. However, this work does not propose optimizations for efficient data transfer.

The most closely related work to Shredder proposes a framework for using GPUs to accelerate computationally expensive MD-based hashing primitives in storage systems [10, 22]. Our work focuses on large-scale data systems where the relative weight of data transfer can be even more significant. In particular, our chunking service uses Rabin fingerprinting, which is less computationally demanding than MD5, and is impacted more significantly by the serialization and memory latency issues. In addition, while the two papers address similar bottlenecks in GPU-based systems, our techniques go beyond the ones proposed in [10, 22]. In particular, this prior work proposes memory management optimizations to avoid memory bank conflicts when accessing the shared memory, whereas in contrast we address the issue of bank conflicts in the global device memory of the GPU, which required us to introduce a novel thread cooperation mechanism using memory coalescing. In comparison to the remaining two optimizations we propose in Shredder, this prior work does not mention an execution pipeline that uses multicores at the host, and support for asynchronous data transfer is mentioned but not described in detail in their paper. We also present two real life end-to-end case studies of incremental MapReduce and cloud backup that benefit from Shredder.

Incremental Computations. Since modifying the output of a computation incrementally is asymptotically more efficient than recomputing everything from scratch, researchers and practitioners have built a wide range of systems and algorithms for incremental computations [15, 23, 27, 34, 36, 38, 39]. Our proposal speeds up the process of change identification in the input and is complementary to these systems.

Incremental Storage. Data deduplication is commonly used in storage systems. In particular, there is a large body of research on efficient index management [14, 18, 32, 48, 49]. In this paper, we focus on the complementary problem of content-based chunking [21, 29, 35]. High throughput content-based chunking is particularly relevant in environments that use SANs, where chunking can become a bottleneck. To overcome this bottleneck, systems have compromised the deduplication efficiency with sampling techniques or fixed-size chunking, or they have tried to scale chunking by deploying multi-node systems [16, 19, 20, 47]. A recent proposal shows that multi-node systems not only incur a high cost but also increase the reference management burden [24]. As a result, building a high throughput, cost-effective, single node systems becomes more important. Our system can be seen as an important step in this direction.

Network Redundancy Elimination. Content-based chunking has also been proposed in the context of redundancy elimination for content distribution networks (CDNs), to reduce the bandwidth consumption of ISPs [9, 11, 12, 40]. Also, many commercial vendors (such as Riverbed, Juniper, Cisco) offer middleboxes to improve bandwidth usage in multi-site enterprises, data centers and ISP links. Our proposal is complementary to this work, since it can be used to improve the throughput of redundancy elimination in such solutions.

9 Conclusions and Future Work

In this paper we have presented Shredder, a novel framework for content-based chunking using GPU acceleration. We applied Shredder to two incremental storage and computation applications, and our experimental results show the effectiveness of the novel optimizations that are included in the design of Shredder.

There are several interesting avenues for future work. First, we would like to incorporate into the library several optimizations for parallel content-based chunking [31, 33]. Second, our proposed techniques need to continuously adapt to changes in the technologies that are used by GPUs, such as the use of high-speed InfiniBand networking, which enables further optimizations in the packet I/O engine using GPU-direct [4]. Third, we would like explore new applications like middleboxes for bandwidth reduction using network redundancy elimination [11]. Finally, we would like to incorporate Shredder as an extension to recent proposals to devise new operating system abstractions to manage GPUs [43].

Acknowledgments

We thank Remzi Arpaci-Dusseau, the sysnets group members at MPI-SWS, Rose Hoberman, the anonymous reviewers, and our shepherd, Mark Lillibridge, for comments and suggestions on this paper. We thank Krishna Gummadi for granting access to the GPU hardware.

References

[1] The data deluge. http://www.economist.com/node/15579717, Feb. 2010.

[2] Calculating Memory System Power for DDR3. http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3%20Power.pdf, Jan. 2012.

[3] General Purpose computation on GPUs. http://www.gpgpu.org, Jan. 2012.

[4] GPUDirect. http://developer.nvidia.com/gpudirect, Jan. 2012.

[5] Memcached. http://memcached.org/, Jan. 2012.

[6] NVidia CUDA. http://developer.nvidia.com/cuda-downloads, Jan. 2012.

[7] NVidia CUDA Tutorial. http://people.maths.ox.ac.uk/ğilesm/-hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf, Jan. 2012.

[8] Using the RDTSC Instruction for Performance Monitoring - Intel Developers Application Notes . http://www.ccsl.carleton.ca/ jamuir/rdtscpm1.pdf, Jan. 2012.

[9] AGGARWAL, B., AKELLA, A., ANAND, A., BALACHANDRAN, A., CHITNIS, P., MUTHUKRISHNAN, C., RAMJEE, R., AND VARGHESE, G. EndRE: an end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX conference on networked systems design and implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 28–28.

[10] AL-KISWANY, S., GHARAIBEH, A., SANTOS-NETO, E., YUAN, G., AND RIPEANU, M. Storegpu: exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th international symposium on High performance distributed computing* (New York, NY, USA, 2008), HPDC '08, ACM, pp. 165–174.

[11] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 219–230.

[12] ANAND, A., SEKAR, V., AND AKELLA, A. Smartre: an architecture for coordinated network-wide redundancy elimination. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (New York, NY, USA, 2009), SIGCOMM '09, ACM, pp. 87–98.

[13] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ASPLOS'00, ACM, pp. 117–128.

[14] BHAGWAT, D., ESHGHI, K., LONG, D. D. E., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS'09* (2009).

[15] BHATOTIA, P., WIEDER, A., RODRIGUES, R., ACAR, U. A., AND PASQUIN, R. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 7:1–7:14.

[16] CLEMENTS, A. T., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized deduplication in san cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 8–8.

[17] CURRY, M. L., WARD, H. L., SKJELLUM, A., AND BRIGHTWELL, R. A lightweight, gpu-based software raid system. In *Proceedings of the 2010 39th International Conference on Parallel Processing* (Washington, DC, USA, 2010), ICPP '10, IEEE Computer Society, pp. 565–572.

[18] DEBNATH, B., SENGUPTA, S., AND LI, J. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX'10, USENIX Association, pp. 16–16.

[19] DONG, W., DOUGLIS, F., LI, K., PATTERSON, H., REDDY, S., AND SHILANE, P. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX conference on File and stroage technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 2–2.

[20] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. Hydrastor: a scalable secondary storage. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), FAST'09, USENIX Association, pp. 197–210.

[21] ESHGHI, K., AND TANG, H. K. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Tech. Rep. HPL-2005-30R1, HP Technical Report, 2005.

[22] GHARAIBEH, A., AL-KISWANY, S., GOPALAKRISHNAN, S., AND RIPEANU, M. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (New York, NY, USA, 2010), HPDC '10, ACM, pp. 167–178.

[23] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: automatic management of data

and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[24] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIX'11, USENIX Association, pp. 25–25.

[25] HAN, S., JANG, K., PARK, K., AND MOON, S. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 195–206.

[26] HARRISON, O., AND WALDRON, J. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Security'08, USENIX Association, pp. 195–209.

[27] HE, B., YANG, M., GUO, Z., CHEN, R., SU, B., LIN, W., AND ZHOU, L. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 63–74.

[28] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. Sslshader: cheap ssl acceleration with commodity processors. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 1–1.

[29] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 18–18.

[30] KULKARNI, V. Delivering on the i/o bandwidth promise: over 10gb/s large sequential bandwidth on ibm x3850/x3950 x5. Tech. rep., IBM, 2010.

[31] LILLIBRIDGE, M. Parallel processing of input data to locate landmarks for chunks, 16 August 2011. U.S. Patent No. 8,001,273.

[32] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), FAST'09, USENIX Association, pp. 111–123.

[33] LILLIBRIDGE, M., ESHGHI, K., AND PERRY, G. Producing chunks from input data using a plurality of processing elements, 12 July 2011. U.S. Patent No. 7,979,491.

[34] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 51–62.

[35] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 174–187.

[36] OLSTON, C., CHIOU, G., CHITNIS, L., LIU, F., HAN, Y., LARSSON, M., NEUMANN, A., RAO, V. B., SANKARASUBRAMANIAN, V., SETH, S., TIAN, C., ZICORNELL, T., AND WANG, X. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 international conference on Management of data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 1081–1090.

[37] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A., AND PURCELL, T. J. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[38] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–15.

[39] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '09)* (June 15 2009).

[40] PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 2–2.

[41] QUINLAN, S., AND DORWARD, S. Awarded best paper! - venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), FAST '02, USENIX Association.

[42] RABIN, M. O. Fingerprinting by random polynomials. Tech. rep., 1981.

[43] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 233–248.

[44] SAXENA, V., SABHARWAL, Y., AND BHATOTIA, P. Performance evaluation and optimization of random memory access on multicores with high productivity. In *International Conference on High Performance Computing (HiPC)* (2010), IEEE.

[45] SHOJANIA, H., LI, B., AND WANG, X. Nuclei: Gpu-accelerated many-core network coding. In *Proc. of IEEE Infocom , Rio de Janeiro* (2009), INFOCOM'09, pp. 459–467.

[46] SMITH, R., GOYAL, N., ORMONT, J., SANKARALINGAM, K., AND ESTAN, C. Evaluating gpus for network packet signature matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'09* (2009).

[47] UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CALKOWSKI, G., DUBNICKI, C., AND BOHRA, A. Hydrafs: a high-throughput file system for the hydrastor content-addressable storage system. In *Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 17–17.

[48] XIA, W., JIANG, H., FENG, D., AND HUA, Y. Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIX'11, USENIX Association, pp. 26–28.

[49] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), FAST'08, USENIX Association, pp. 18:1–18:14.