

A GPU Accelerated Storage System

Abdullah Gharaibeh, Samer Al-Kiswany, Sathish Gopalakrishnan, Matei Ripeanu

Electrical and Computer Engineering Department

The University of British Columbia

Vancouver, Canada

{abdullah, samera, sathish, matei}@ece.ubc.ca

ABSTRACT

Massively multicore processors, like, for example, Graphics Processing Units (GPUs), provide, at a comparable price, a one order of magnitude higher peak performance than traditional CPUs. This drop in the cost of computation, as any order-of-magnitude drop in the cost per unit of performance for a class of system components, triggers the opportunity to redesign systems and to explore new ways to engineer them to recalibrate the cost-to-performance relation.

In this context, we focus on data storage: We explore the feasibility of harnessing the GPUs' computational power to improve the performance, reliability, or security of distributed storage systems. In this context we present the design of a storage system prototype that uses GPU offloading to accelerate a number of computationally intensive primitives based on hashing.

We evaluate the performance of this prototype under two configurations: as a content addressable storage system that facilitates online similarity detection between successive versions of the same file and as a traditional system that uses hashing to preserve data integrity. Further, we evaluate the impact of offloading to the GPU on competing applications' performance. Our results show that this technique can bring tangible performance gains without negatively impacting the performance of concurrently running applications.

Further, this work sheds light on the use of heterogeneous multicore processors for enhancing low-level system primitives, and introduces techniques to efficiently leverage the processing power of GPUs.

Categories and Subject Descriptors

D.4.3 [Operating Systems]: File Systems Management - Distributed file systems. D.4.8 [Operating Systems]: Performance - Measurements, Modeling and Prediction. I.3.1 [Computer Graphics]: Hardware Architecture - Graphics processors, Parallel Processing.

General Terms

Performance, Design, Experimentation.

Keywords

Storage system design, massively-parallel processors, graphics processing units (GPUs), content addressable storage.

1. INTRODUCTION

The development of massively multicore processors has led to a rapid increase in the amount of computational power available on a single die. There are two potential approaches

to make the best use of this computational capacity and greater hardware support for concurrency: one is to design applications that are inherently parallel, while the other is to enhance the functionality of applications via ancillary tasks that improve an application's behavior along dimensions such as reliability and security.

While it is possible to increase the degree of parallelism of existing applications, a significant investment is needed to refactor them. Moreover, not all applications offer sufficient scope for parallelism. Therefore, we look into the second approach and investigate techniques to enhance applications along non-functional dimensions. Specifically, we start from the observation that a number of techniques that enhance the reliability, scalability and/or performance of distributed storage systems (e.g., erasure coding, content addressability [1, 2], online data similarity detection [3], integrity checks, digital signatures) generate computational overheads that often hinder their use on today's commodity hardware. We consider the use of Graphics Processing Units (GPUs) to accelerate these tasks, in effect using a heterogeneous massively multicore system that integrates different execution models (MIMD and SIMD) and memory management techniques (hardware vs. application-managed caches) as our experimental platform.

We have previously demonstrated that GPUs can accelerate the computation of hash-based primitives [4]. This paper investigates the system-level challenges and quantifies the benefits of integrating GPU-offloading in a complete storage system. To this end, we have prototyped (Section 3) a distributed storage system which integrates our *HashGPU* library with the *MosaStore* content-addressable storage system. Most of the integration challenges are addressed by *CrystalGPU* a newly developed generic runtime layer that optimizes task execution on the GPU. Our experimental evaluation (Section 4) demonstrates that the proposed architecture enables significant performance improvements compared to a traditional architecture that does not offload compute-intensive primitives.

The contribution of this work is fourfold:

- First, we demonstrate the viability of employing massively multicore processors, GPUs in particular, to support storage system services. To this end, we evaluate, in the context of a content-addressable distributed storage system, the throughput gains enabled by offloading hash-based primitives to GPUs. We provide a set of data-points that inform the storage system designers' decision whether exploiting massively multi-core processors to accelerate the storage system operations is a viable approach for particular workloads and deployment environment characteristics.

- Second, we empirically demonstrate, under the set of workloads and configurations we consider, that exploiting the GPU computational power enables close to optimal system performance; that is, additional compute power will only enable minimal performance gains.
- Third, we shed light on the impact of GPU offloading on competing applications running on the same node as the distributed storage middleware. We focus on two issues in particular: First, while employing a GPU holds the potential to accelerate computationally intensive operations, the need to transfer the data back and forth to the device adds a significant load on a shared critical system resource, the I/O subsystem. Our experiments demonstrate that this added load does not introduce a new bottleneck in the system. Second, we quantitatively evaluate the performance impact on concurrently running compute- and IO-intensive applications.
- Finally, we introduce techniques to efficiently leverage the processing power of GPUs. Recent support for general-purpose programming (e.g., NVIDIA’s CUDA [5]) reduces the effort needed to develop applications that benefit from the massive parallelism GPUs offer. Significant challenges to efficiently use GPUs, however, remain. To address these challenges, on the one side, we have designed and integrated *CrystalGPU*, an independent runtime layer that efficiently manages the interaction between the hosted application and the GPU. *CrystalGPU* transparently enables the reuse of GPU buffers to avoid memory allocation overheads, overlaps the data transfer with computation on the GPU, and enables the transparent use of multiple GPU devices.

Context. Although this work focuses on hash-based primitives, we argue that the proposed approach can be extended to other routines that support today’s distributed systems like erasure coding [6], compact dataset representation using Bloom filters or Merkel trees, as well as data compression, deduplication [7], and encryption/decryption. To this end, parallel algorithms for these primitives exist, and can benefit from GPU support (e.g., parallel Reed-Solomon coding [8], parallel compression algorithm, and parallel security checking [9]). Additionally, the offloading approach can be used in an active-storage design to support specialized storage systems that focus on, for example, content-based image retrieval [10]. In this context, data-parallel processing can be offloaded to the GPU to enable significant performance improvements, provided that the computation performed per byte of input data is sufficiently high to amortize the additional GPU overheads.

It is worth noting that multiple computational cores have been used to improve the performance of some security checks such as on-access virus scanning, sensitive data analysis, and taint propagation [9]. In fact, without parallel execution, the overhead of executing some of these tasks is prohibitive. The spirit of this work is similar: without support for parallel execution, many storage system optimizations cannot be deployed due to their associated overheads.

The decision to use GPUs is motivated by several factors. Recent commodity GPUs like the NVIDIA’s 9800 GX2 (256 cores @ 1.5GHz) offer a ten-fold higher peak computational rate than Intel processors in the same price range. *This drop in the cost of computation, as any order-of-magnitude drop in*

the cost per unit of performance for a class of system components, triggers the opportunity to redesign systems and to explore new ways to engineer them to recalibrate the cost-to-performance relation. We believe that GPUs can be a cost-effective enhancement to high-end server systems.

Further, using GPUs aligns with current technological trends: First, while multi-core CPUs currently do not offer this level of parallelism, they are expected to provide similar parallelism over the next decade and will be able to use some of the techniques prototyped in a GPU context. Second, future massively multicore processors will likely extend the scope of heterogeneity already present in existing processors (e.g., IBM’s Cell BE); they will likely integrate heterogeneous cores, multiple execution models (e.g., SIMD/MIMD core blocks), and non-uniform memory architectures [11, 12]. Section 5 further discusses this issue.

An alternative is the use of application-specific integrated circuits (ASICs) as hardware accelerators. ASICs are an acceptable solution if every storage system were to use exactly the same schemes for verifying data integrity, compression, etc., and if these schemes never change. Improvements in algorithms and feature additions would require new ASICs that would, in turn, make other ASICs obsolete. The use of GPUs promotes flexibility and limits hardware waste.

The relationship with our prior work: In the past we have surveyed the use of hash functions in existing storage systems and developed *HashGPU* [4], a library that accelerates these hash computations using GPUs. This work extends the earlier effort in many dimensions: First, we have continued to improve *HashGPU* and we implemented a new set of optimizations that harness the CUDA 2.0 runtime environment. Second, and more importantly, this work explores the end-to-end opportunities to harness massively multicore platforms to support distributed storage systems by integrating *HashGPU* with a storage system prototype. Our current work is a *full-system integration effort* that goes beyond other related work (discussed in Section 6) on hashing and on using GPUs for general-purpose computation. Our experiences in developing this system suggest lessons that are likely to be important even under different circumstances or technology trends (as discussed in Sections 5 and 7). We note that, to make this paper self-contained, there is a minimal amount of overlap with our previous publication.

2. BACKGROUND

A number of primitives commonly used in distributed storage systems generate computational overheads that set them apart as potential bottlenecks in today’s systems, which increasingly employ multi-Gbps links and/or are built on top of high throughput storage devices (e.g., SSDs [13]).

For instance, on-the-fly data compression offers the possibility to significantly reduce storage footprint and network effort at the expense of performing additional computations. However, the compression throughput can be prohibitively low (4MBps for *bzip2* and 16MBps for *gzip* in our experiments on a 2.4Ghz Intel Core2 quad core processor) not only for high-performance computing scenarios but also for desktop deployments. Similarly, while encryption enables clients to securely store their data at an untrusted server, its throughput (between 38MBps and 157MBps in our experiments depending on the encryption algorithm used) can

be lower than the client’s network/disk throughput. Similarly, erasure codes’ encoding throughput is limited by their computational complexity (for example, RAID6 systems using Reed Solomon codes achieve lower than 60MBps coding throughput using all cores of an Intel Core2 Quad processor [14]).

We aim to understand the viability of offloading these data-processing intensive operations to a GPU to dramatically reduce the load on the source CPU and enhance overall system performance. In this context, we focus on the use of hashing-based primitives in storage systems. This section highlights their computational overheads (2.1), and presents a brief overview of NVIDIA GPU’s architecture (2.2)

2.1 Use of Hashing in Storage Systems

Hash-based primitives are commonly used in storage systems as building blocks that help provide content addressability, data integrity checks, data similarity detection, and compact set representation. In this context, there are two main uses of hashing:

Direct Hashing. This technique computes the hash of an entire data block. In systems that support content addressability [1, 2, 7], data blocks are identified based on their content: data-block identifiers are simply the hash value of the data. This approach provides a number of attractive features. First, it provides a flat namespace for data-block identifiers and a naming system which simplifies the separation between file- and data-block metadata. Second, it enables *probabilistic* detection of similar data-blocks by comparing their hash value [1, 3]. Finally, it provides implicit integrity checks. The hash computation, however, imposes overheads that may limit performance, especially in two common scenarios. First, for workloads with frequent updates, changing a few bytes in the data may require rehashing multiple data blocks. Second, in systems that use large blocks (e.g., GoogleFS [15] uses 64MB blocks), the hashing computational overhead becomes a non-negligible component (possibly the largest) of a data access operation.

Sliding Window Hashing. The techniques mentioned above require dividing large files into multiple blocks. To this end, two approaches are possible: fixed-size blocks, and detecting block boundaries (i.e., the markers for blocks’ start and end) based on content. This is done by hashing a large number of overlapping data windows inside the data-block and declaring a block boundary when the hash value matches a predefined value. Low-Bandwidth File System (LBFS) [3] and JumboStore [16] both adopt this latter technique, also dubbed *sliding window hashing*.

When used for similarity detection, these two techniques offer a tradeoff between computational overheads and the similarity ratio detected: unlike fixed-size blocks, detecting block boundaries based on content is stable to data deletions/insertions, yet significantly more computationally intensive, hence leading to lower throughput. In fact, the limited throughput of sliding window hashing is the main reason its proponents [3] recommend its use in storage systems supported by low-bandwidth networks. On an Intel Core2 Duo 6600 2.40GHz processor, and using data from a checkpointing application, our sliding window hashing implementation offered 7 to 51MBps throughput, and similarity detection rates between 82% and 37% respectively,

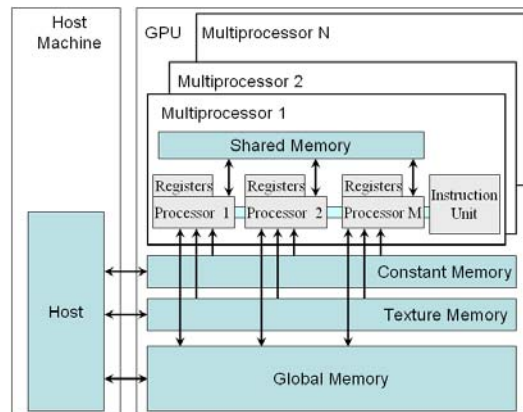


Figure 1. GPU Architecture.

depending on the configuration. Throughput in this range is clearly a bottleneck when using Gbps network links or fast disks.

2.2 GPU-Related Background

This section presents a brief overview of GPU architecture, programming model, and typical application structure. While we focus on NVIDIA’s architecture and programming environment (the Compute Unified Device Architecture (CUDA) [17]) similar issues emerge with other GPU vendors or programming environments.

At a high-level view (Figure 1), NVIDIA’s GPUs are composed of a number of SIMD multiprocessors. Each multiprocessor incorporates a small but fast *shared memory* (16KB in all GeForce 9x and 8x GPUs). All processors in the multiprocessor have direct access to this memory. Additionally, all multiprocessors have access to three other device-level memory modules: the *global*, *texture*, and *constant* memory modules, which are also accessible from the host. The global memory supports read and write operations and it is the largest (with size ranging from 256 to 1024MB). The texture and constant memories are much smaller and offer only read access to GPU threads. Apart from size, the critical characteristic differentiating the various memory modules is their access latency. While accessing the shared memory takes up to four cycles, it takes 400 to 600 cycles to access global memory. Consequently, to achieve maximum performance, applications should maximize the use of shared memory and processor registers.

The CUDA programming model extends the C language

Table 1: The processing stages of a GPU task

Stage	Operations performed
(1) Preprocessing	Device initialization; memory allocation on the host and device; task-specific data preprocessing on the host
(2) Data Transfer In	Data transfer from host’s memory to device global memory.
(3) Processing	Looping through: (3.1) Data transfer from global GPU memory to shared memories [optional]; (3.2) Task’s ‘kernel’ processing; and (3.3) Result transfer back to global memory.
(4) Data Transfer Out	Output transfer from device global memory to the host memory.
(5) Post-processing	Task-specific post processing on the host; resource release [optional].

with directives that expose the GPU’s execution model and memory layout. Additionally the development environment provides a runtime library to manage task execution, multi-GPU devices, timers, memory, and per-multiprocessor thread synchronization.

A GPU task is composed of five stages: preprocessing; host-to-device data transfer; kernel processing; device-to-host results transfer and post-processing. As Table 1 indicates, for a data-parallel application, the *processing* step is usually repeated until all input data is processed.

3. SYSTEM DESIGN

This section first discusses the design challenges of a GPU accelerated storage system, and then details our storage system design.

3.1 Design and Integration Challenges

Efficient offloading to the GPU implies extracting the target application’s parallelism, and, equally importantly, employing efficient memory, data transfer, and thread management techniques. Improper task decomposition, memory allocation and management, or data transfer scheduling can lead to dramatic performance degradation [18]. Apart from the above performance related issues there are two additional areas of concern: minimizing integration effort and preserving the separation of concerns between application-related issues, accelerator logic, and resource allocation issues related to the GPU. Overall, six areas of concern exist:

- *Minimizing the integration effort.* The volume of coding required to enable GPU-offloading should not dwarf the potential performance gains. Ideally, the primitives offloaded are stateless and offloading is simplified preserving the API and the semantic of the original implementation. Even in this case, however, there are two factors for additional complexity, as we shall argue: first, memory allocation which must be done at the device level, and, second, the asynchronous nature of task execution management on the GPU as well as other device-management operations. Our design hides these complexities under simple, intuitive interfaces.
- *Separation of concerns.* The integrated design should preserve the separation between the logic of the accelerated primitive(s) and the runtime management layer required for efficient resource management at the device. The main driver for this requirement is facilitating the addition of new primitives that will benefit from offloading through the same runtime layer. Our layered design (Figure 2) and the generic task-management functionality offered by the *CrystalGPU* layer, address this concern. This separation, however, required changes in our *HashGPU* library.
- *Hiding data transfer overheads.* For streaming applications, that is, for applications that send multiple GPU-tasks back to back, overlapping the transfer of input data to the GPU with the computation step of a previous task can hide the data transfer overhead. We explore these optimizations through the *CrystalGPU* layer (Section 3.3).
- *One-copy host to device data transfers.* Data transfers to and from the device require DMA transfers from (respectively to) non-pageable host memory. If an application presents data residing in pageable memory for transfer to the GPU then the CUDA runtime library first allocates a new buffer in non-pageable memory, copies the

data to this new buffer, and finally launches the DMA transfer. Thus, to avoid these additional overheads (data-copy and new buffer allocation), the application should present data in buffers allocated in non-pageable memory.

- *Hiding memory allocation overheads.* Additionally, since allocating memory in non-pageable memory is an expensive operation, the application should reuse buffers to the extent possible. To this end we have added a memory management module to the *CrystalGPU* layer (Section 3.3). This module offers a *CrystalGPU*-specific implementation of `malloc` and `free` function calls and manages non-pageable memory buffers used through the application run (these buffers are allocated at the application initialization).
- *Simplifying the use of multiple GPUs in multi-GPU systems.* CUDA provides only a low-level API for using multiple GPUs by the same application. Managing and balancing the load between multiple GPUs significantly increases application complexity. To circumvent this limitation, and benefit from multi-GPU systems, *CrystalGPU* provides an API that enables transparent and balanced use of multiple GPUs. (Section 3.3).
- *Efficient use of shared memory,* the fastest, yet the smallest, GPU memory module. Three reasons make the efficient use of the shared memory challenging. First, shared memory is often small compared to the volume of data being processed. Second, the shared memory is divided into banks and all read and write operations that access the same bank are serialized (i.e., generate bank conflicts), hence, reducing concurrency. Consequently, to maximize performance, the concurrent threads should access data on different banks. The fact that a single bank does not represent a contiguous memory space increases the complexity of efficient memory utilization. Finally, while increasing the number of threads per multiprocessor helps hiding global memory access latency, this does not directly lead to a linear performance gain. The reason is that increasing the number of threads decreases the amount of shared memory available per thread.

To efficiently use the shared memory without increasing programming complexity, we have implemented a shared memory management mechanism (Section 3.2.2).

3.2 A GPU-accelerated Storage System Prototype

The prototyped distributed storage system integrates three main components: *MosaStore* storage system, *HashGPU* library, and *CrystalGPU* (Figure 2). We integrated *MosaStore* with *HashGPU* and *CrystalGPU* such that the compute intensive hash-based processing required to support content addressability is outsourced to a GPU. The rest of this section details the design of each of these three components (Sections 3.2.1 to 3.2.3) and the role of each of them in the final system prototype (Section 3.2.4).

3.2.1 *MosaStore*

MosaStore is a highly configurable storage system prototype that can be configured to optimize its operations to exploit specific workload characteristics [19]. More relevant to this study, the current version of *MosaStore* can be configured to work as a content addressable storage system.

MosaStore employs an object-based distributed storage system architecture (similar to GoogleFS). Its three main components are (Figure 2): a centralized metadata manager,

the storage nodes, and the client’s system access interface (SAI) which uses the FUSE [20] kernel module to provide a POSIX file system interface.

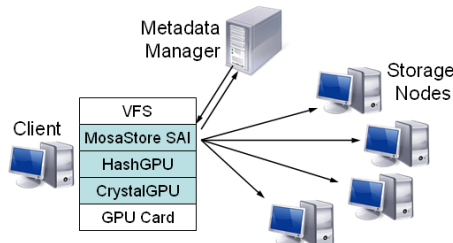


Figure 2. System architecture. At the client, we use the FUSE Linux module to implement a user-level filesystem (MosaStore SAI). Linux virtual file system (VFS) calls FUSE, which in turn calls our user-level primitives that implement MosaStore file system functionality. Note that the GPU is needed only on the client machines.

Each file is divided into blocks that are stored on the storage nodes. The metadata manager maintains a block-map for each file which contains the file’s blocks information including the hash value of every block. The SAI implements the client-side content-based addressability mechanisms. The current implementation (and, by consequence, our experimental evaluation) uses fixed-size blocks. In Section 5 we discuss the impact of detecting blocks boundaries based on content (as described in Section 2.1), which requires support for blocks of variable size.

To write to *MosaStore*, the SAI first retrieves the file’s previous-version block-map from the manager, divides the new version of the file into blocks, computes the hash value of every block, and searches the file’s previous-version block-map for equivalent block hash values. The SAI stores only the blocks with no match at the storage nodes, saving considerable storage space and network effort. Once the write operation is completed (as indicated by the `release` POSIX call) the SAI commits the file’s block-map including the blocks’ hash values to the metadata manager.

3.2.2 HashGPU

This section presets the high-level design of the *HashGPU* direct hashing module. For an in-depth description of the sliding-window hashing module we refer the reader to our previous publication [4].

Most widely used hash functions follow the *sequential* Merkle-Damgård construction approach [21, 22]. In this approach, a large data block is split into fixed-size segments. The first segment is processed to produce a fixed-size output which is used as the input to the hashing function together with the next segment. The process continues sequentially until all segments are processed. This sequential construction does not allow multiple threads to operate concurrently to hash the data. To exploit GPUs’ parallelism, *HashGPU* employs the *parallel* Merkle-Damgård construction: the sequential hash function is used as a building block to process multiple segments of data in parallel on the GPU. We note that the resulting hash is as strong as the original, sequentially built, hash, as demonstrated by Damgård [22].

Computing the hash is carried in five stages (as described in section 2.2). Once the input data is transferred from the CPU, it is divided into segments that are hashed in parallel. Each segment is hashed using a standard sequential hashing

function (MD5 in all our experiments). The result is placed in a single output buffer which is itself hashed to produce the final hash value. Two aspects are worth mentioning. First, as there are no dependencies between the intermediate hashing computations, each computation can execute in a separate GPU thread. Second, this design uses the CPU to compute the final hash of the intermediary hashes. The reason is that synchronizing all running GPU threads is not possible [17].

Optimized Memory Management. Although the design of the hashing module is relatively simple, optimizing for performance is a challenging task. For example, one aspect that induces additional complexity is maximizing the number of threads to extract the highest parallelism (around 100K threads are created for large blocks) while avoiding bank conflicts and maximizing the use of each processors’ registers. To address this issue, we have implemented a shared-memory management sub-system that:

- First, to reduce memory access latency, the memory management sub-system allocates to each thread a fixed-size workspace located in the shared memory. Additionally, to avoid bank conflicts, the workspaces of threads that are co-scheduled are allocated on separate shared memory banks. When a thread starts, it copies its data from the global memory to its shared memory workspace, hence avoiding subsequent accesses to the slower global memory.
- Second, to reduce the complexity of the programming task (as the workspace allocation technique just described makes programming more complex; since a shared memory bank is not mapped to a contiguous memory address space) the memory management sub-system abstracts the shared memory to allow the thread to access its workspace as a contiguous address space.

In effect, the shared memory management sub-system reduces memory access latency by avoiding bank conflicts while reducing the programming effort by providing a contiguous memory address abstraction.

3.2.3 CrystalGPU

Although *HashGPU* optimizes the execution of hashing a single block, there is still room for improvement. Let us consider a workload that is a stream of hashing computations. In this case, three opportunities for performance improvement exist: input/output buffer reuse to amortize buffer allocation overhead, ability to harness multiple GPUs on multi-GPU systems, and exploiting the CUDA2 [17] capabilities to overlap the data transfer of one block with the computation of a previous block.

To exploit these opportunities, we developed *CrystalGPU*, a standalone abstraction layer that exploits these three opportunities. *CrystalGPU* runs entirely on the host machine as a management layer between the application and the GPU runtime system. *CrystalGPU* manages the execution of GPU operations (e.g., transferring data from/to the GPU and starting computations on the GPU) and includes a memory management layer that enables the reuse of non-pageable memory.

At a high level, the metaphor *CrystalGPU* offers is that of a task-management environment for GPU tasks. Our design aims for (i) *generality*, to facilitate the support of a wide range of GPGPU applications, (ii) *flexibility*, to enable deployment on various GPU models while hiding the heterogeneity in task

management across models, and (iii) *efficiency*, to maximize the utilization of GPU resources (processing units and I/O channels). We achieve these goals via an interface that is agnostic to the upper level application, and an internal design that avoids extra data copies and shared data-structure bottlenecks. Due to space limitations we do not present the detailed *CrystalGPU* design and API.

3.2.4 Integration and System Prototype

Our prototype (Figure 2) integrates the three aforementioned components into a storage system able to exploit the GPU computational power to support content addressability. The integration entailed two main changes:

- We modified the *HashGPU* library to manage GPU operations using *CrystalGPU*. To this end, we applied two changes: allocated the data buffers through *CrystalGPU*, and retrofitted the hash computation into a *CrystalGPU* ‘task’ (*CrystalGPU*’s abstraction for a unit of GPU computation and the associated data transfers). The resulting library is able to harness optimizations available for a single hash operation as well as across a stream of hash operations.
- Further, we modified MosaStore’s SAI to use the *HashGPU* library to offload the hash computations to the GPU. This modification required two main changes: First, all hash function calls needed to be changed to the new *HashGPU*-provided hash function. Second, to facilitate optimized data transfers and buffer reuse during GPU calls, memory buffers needed to be allocated using the *HashGPU* library call instead of standard memory allocation functions (e.g. `malloc`). The resulting storage system is able to offload the hash computation to the GPU.

These changes required changing only 22 lines of the original *MosaStore* implementation (over 18K lines of code). We estimate that integrating GPU offloading with other storage systems that make use of hashing is equally straightforward.

4. EXPERIMENTAL EVALUATION

This section starts by highlighting the benefits brought by *CrystalGPU* (Section 4.1). While we include information to present the speedup offered by *HashGPU* when used independently we refer the reader to our previous publication for a complete evaluation of the *HashGPU* library [4]. Next, the evaluation focuses on the integrated system and analyzes the performance improvements GPU support brings to the whole storage system (Section 4.2). Finally the evaluation focuses on the impact of offloading to the GPU on concurrently running applications (Section 4.3).

4.1 Evaluating the Performance Gains Enabled by CrystalGPU

To motivate the performance gains enabled by the *CrystalGPU* layer this section starts by analyzing the overheads of the various processing steps within the *HashGPU* library. We then evaluate the performance gains enabled by each of the optimizations enabled by *CrystalGPU*: buffer reuse, overlap of data transfers and computations, and transparent utilization of multiple GPUs.

Analyzing the Overheads. Figure 3 presents the proportion of the total execution time that corresponds to each stage (outlined in Section 2.2) for the direct hashing module on the

GeForce 8800 GTX card. These results show the major impact of memory allocation and copy-in stages on the total execution time, 70-90% of the total execution time.

Two opportunities exist to reduce the overhead of these two stages: first, the overhead of memory allocation stage can be avoided by reusing memory buffers. Second, copy-in overheads can be hidden, by overlapping data transfers and computation. Given the generic nature of these optimizations, *CrystalGPU* offers them in an application agnostic layer. The rest of this section presents an evaluation of the benefits of integrating *HashGPU* on top of *CrystalGPU*.

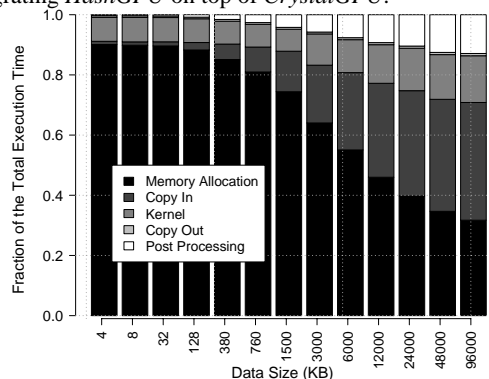


Figure 3: Percentage of total HashGPU execution time spent on each stage.

Evaluating the Performance Gains Enabled by CrystalGPU.

The evaluation uses a NVIDIA GeForce 9800GX2 GPU. The card is a dual GPU that is equivalent to two GeForce 8800 GTS GPUs. The host is an Intel quad-core 2.66 GHz machine with PCI Express 2.0 x16 bus. Note that the two internal GPUs share the same PCI bus.

Figure 4 shows the speedup obtained when using *HashGPU* on top of *CrystalGPU* for the direct hashing module to hash a stream of 8 data blocks (our evaluation shows that a batch of at least 3 blocks is needed to maximize the performance gains). The figure uses the performance on a single CPU core as the baseline.

The experiment shows the effect of block size on the achieved speedup. For smaller block sizes, and irrespective of which optimization is enabled, the memory allocation and data transfer overheads overshadow the performance improvement in the computation (i.e. calculating the hash); conversely, for larger block sizes, the aforementioned overheads are amortized over the longer computation time, hence achieving better speedups compared to the CPU.

This experiment demonstrates that using all *CrystalGPU* optimizations enables up to 18x performance gains (for large data blocks), compared to up to 5x performance gains provided by *HashGPU* alone. Further, while the original *HashGPU* performance lags behind the CPU performance for data blocks smaller than 1MB, *CrystalGPU* starts to achieve speedup for blocks as small as 64KB.

Despite all the optimizations, offloaded hashing performance lags behind the CPU for blocks smaller than 64KBs. This is mainly due to the lack of enough data-level parallelism to exploit all the computational capability of the GPU for small blocks.

The figure demonstrates that the buffer reuse optimization is able to amortize the memory management costs; enabling, as a result, up to 9x speedup for large data blocks. The

overlap feature enables an additional speedup increase that corresponds to the portion of time spent on data transfer operations (to up to 14x). Using both GPUs available on the card increases the speedup further (to up to 18x). Two contention points prevent dual GPU cards to achieve linear speedup in our case: First, the ratio of time spent on data transfers increases as the data size grows, thus placing more contention on the shared PCI buss. Second, direct hashing post-processing stage is performed sequentially on the CPU.

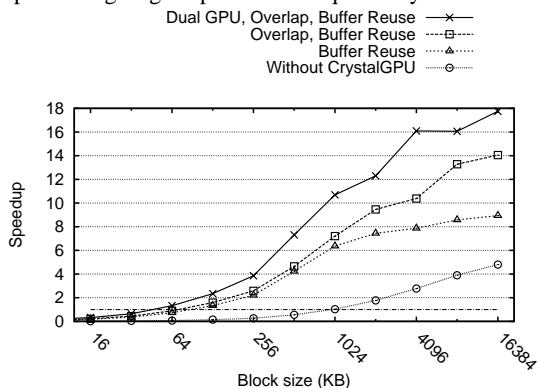


Figure 4. Achieved speedup for direct-hashing for a stream of 8 jobs. Values below 1 indicate slowdown. The baseline is the performance on a single core on Intel Core2 Duo 6600 2.40GHz.

4.2 System Evaluation

This section evaluates the gains enabled by integrating the *HashGPU/CrystalGPU* module with *MosaStore*. We evaluated the integrated system on a 22 nodes cluster of 2.33GHz Intel Xeon Quad-core CPU, 4GB memory nodes connected at 1Gbps. The storage system client machine has an Intel quad-core 2.66 GHz processor, PCI Express 2.0 x16 bus, and NVIDIA GeForce 9800GX2 GPU. We present here averages and 95% confidence interval over 20 runs. At the time of writing this paper the retail price of the CPU is about \$350 while that of the GPU is about \$550.

The storage system prototype is configured to employ fixed-size blocks (1MB, the default block size in *MosaStore*) and hash-based content addressability. The client SAI is configured to stripe the write operations to four storage nodes in parallel.

To explore the gains enabled by GPU offloading we compare the performance of the following three configurations of the storage system:

- *non-CA*, in which the content addressability module is disabled and all data is written directly to the storage nodes (i.e., without any hashing and similarity detection overheads).
- *CA-CPU*, in which all processing related to content addressability is done by the CPU (i.e., hashing of data blocks and hash comparisons). Compared to the previous configuration, this configuration exposes the performance impact of hashing data to support content addressability.
- *CA-GPU*, which uses the integrated *HashGPU/CrystalGPU* stack to offload the computation of hashes to the GPU. Compared to the previous configuration, this configuration exposes the gains brought by offloading computationally intensive operations to a GPU.

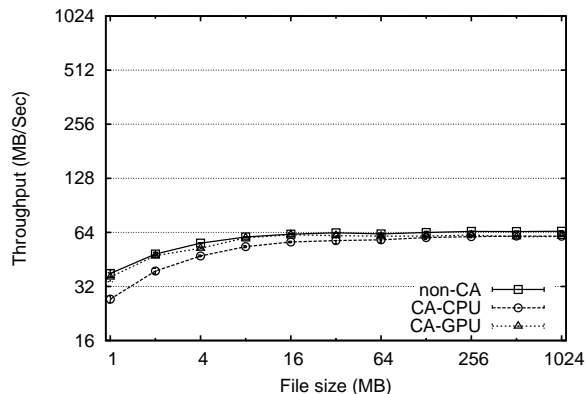


Figure 5: Average throughput while writing 40 different files back-to-back. Note the logarithmic y-axis.

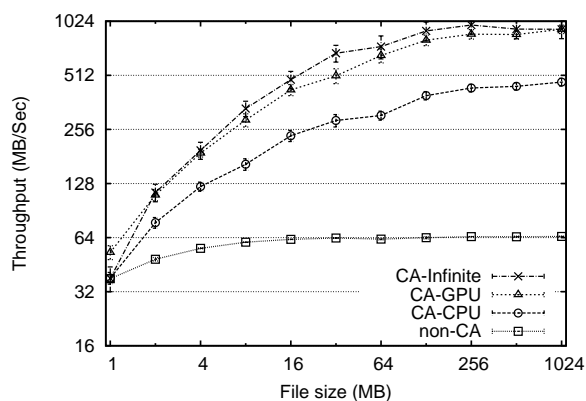


Figure 6: Average throughput while writing the same file 40 times back-to-back. We discuss the CA-Infinite configuration in section 5. Note the logarithmic y-axis.

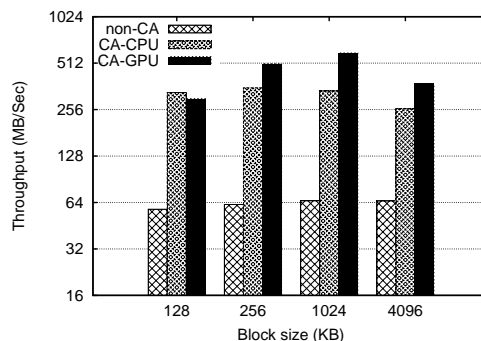


Figure 7: Average throughput while writing 100 BLAST checkpoints back-to-back while varying the block size. Note the logarithmic y-axis.

Note that the performance of the system when using content addressability varies depending on the degree of data similarity present in the workload. To evaluate the entire performance spectrum we use the following three workloads:

- *Different*: The first workload consists of writing a set of completely different files. This workload exposes all overheads as all data need to be hashed and transferred across the network to the storage nodes. Moreover, no similarity can be detected between writes, which implies no opportunity to reduce space or bandwidth usage.

Exploring this workload has a second advantage: the performance comparison holds for systems that use hashing for other goals than content addressability (e.g., for data integrity checks only).

- *Similar*: The second workload represents the other end of the spectrum: it exposes an upper bound for the performance gains that can be obtained using content addressability and maximizes the hash-computation overheads in relation with other storage overheads. When the files are identical, data is transferred only once across the network yet similarity detection overheads still exist.
- *Checkpoint*: Finally the third workload represents a real application data. This third workload involves 100 successive checkpoint images, taken at 5 minute intervals for the BLAST [23] application using BLCR [24] checkpointing library (the average checkpoint size is 280MB). The average similarity detected between successive checkpoint images is 23% for 1MB blocks.

Figure 5 presents the write throughput with the *different* workload. The figure shows that under workloads with completely non-similar data, the write throughput is roughly the same for all configurations. The reason is that, in this case, on our experimental platform, the throughput is limited by the capabilities of the network cards; i.e. the bottleneck in this scenario is the data transfer to the storage node and not the hashing overheads.

Figure 6 presents the write throughput with the *similar* workload. The figure shows that under workloads with completely similar data, *MosaStore* with *HashGPU* significantly outperforms the CPU version and achieves two times higher throughput for files larger than 8MB. This is because the *similar* workload is compute bound: only the first file needs to be transferred over the network to the storage nodes, while the computationally intensive hash computations will detect that the other files are similar. Under this workload *HashGPU* enables doubling the system throughput by accelerating the hash computations.

Varying the block size. Figure 7 presents the write throughput with the *checkpoint* workload for all system configurations while varying the block size. When using the default block size (the same as used in the previous experiments - 1MB), offloading to the GPU enables about 76% write throughput improvement compared to a content addressable system that does not offload.

Three factors influence the system throughput and explain the results of the experiments presented in Figure 7:

- First, the speedup offered by offloading to the GPU increases with the block size (as evaluated in section 4.1 and section 4.2). This is the reason the system that offloads to the GPU performs increasingly better for 128KB, 256KB, and 1MB blocks.

When compared to exploiting all four cores of the client node, offloading to the GPU starts to offer sizeable benefits only for blocks larger than 256KB. This is the reason the system configuration that offloads to the GPU (*CA-GPU*) performs better than the one that does not (*CA-CPU*) for block sizes larger than 256KB. (Note that since *MosaStore* uses independent threads to manage the connection to each storage node, all CPU cores are used for hashing in the *CA-CPU* configuration).

- Second, while for blocks smaller than 1MB, the rate of detected similarity is relatively identical (at about 24%) regardless of the block size, for 4MB blocks it drops to 19%. This is the reason for the decrease in apparent system throughput for large 4MB blocks compared to 1MB blocks. However, even for 4MB blocks, offloading to the GPU enables 45% throughput improvement.
- Finally, and with the smallest impact, overheads decrease with block-size. This is reflected in the throughput increase correlated with the block size increase for the *non-CA* system configuration.

In summary, for workloads that display data similarity that can be detected using blocks larger than 256KB, exploiting GPU's computational power can bring tangible performance benefits to content addressable storage systems. Further, in systems that use hashing only for data integrity checks (as indirectly revealed by our *different* workload), hashing can be efficiently offloaded to the GPU.

4.3 The Impact on Competing Applications

While the previous section has demonstrated that offloading computationally intensive primitives to the GPU can improve the system's throughput, the impact of this approach on the overall client system performance still needs to be evaluated. On the one hand, offloading frees CPU cycles on the client system. On the other hand, offloading might add a significant load on the client's kernel and the I/O subsystems to handle data transfers to and from the device. Consequently, this technique may negatively impact applications that are running concurrently, especially those with high I/O load.

This section evaluates the impact of the proposed approach on concurrently running applications. We use two workloads to reproduce the diversity of applications' possible resource usage pattern: a compute bound application (we use a multi-threaded prime number search application), and an I/O-bound application (we use the compilation of the Apache web server v2.2.11 as a representative I/O-bound application, which particularly stresses the disk I/O channel).

To measure the performance impact, we time the application run while the system client is also serving an intense write workload: writing 1GB files back to back. The performance baseline for all results presented in the figures below is the execution time of the application on an unloaded system (i.e., neither the *MosaStore* SAI client nor other applications are running on the system at the time). We present averages over 20 runs.

The impact on a compute bound application. Figure 8, Figure 9, and Figure 10 present the *MosaStore* achieved throughput (left), and the compute bound application execution time *increase* (i.e., application slowdown – the lower, the better) (right) under the three workloads presented in Section 4.2. These results confirm that:

- First, outsourcing to the GPU frees CPU cycles that can be effectively used by a compute intensive competing application. In all three experiments, with different workloads, the competing application performs faster on a client system that offloads to the GPU than on a client system that computes hash functions on the CPU. The difference can vary from as high as reducing the slowdown by half (for the 'different' workload) to reducing the slowdown by 10-20% (for the other two workloads).

- Second, outsourcing to the GPU enables better storage system throughput (around 2.5x better under the ‘similar’ workload, and 2x better under the checkpoint workload) even when a competing compute intensive application is present on the client node.
- Third, the throughput of the GPU-enabled storage system is only slightly affected by the competing application: less than 18% throughput loss compared to the case when the client system runs on a dedicated node.

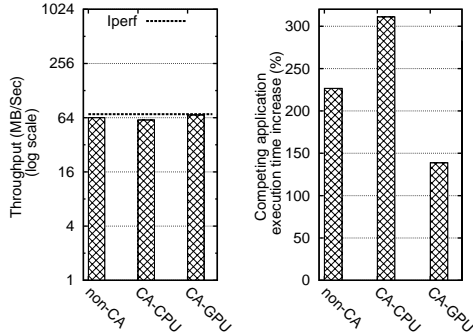


Figure 8: (Left) MosaStore average achieved throughput under the different workload while running a competing compute intensive application. (Right) Competing application slowdown (the lower the better).

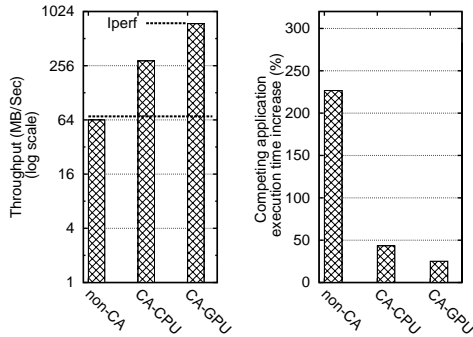


Figure 9: (Left) MosaStore average achieved throughput under the ‘similar’ workload while running a competing compute intensive application. (Right) Competing application slowdown (the lower, the better).

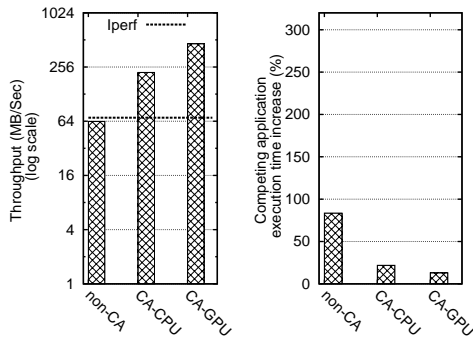


Figure 10: (Left) MosaStore average achieved throughput under the ‘checkpoint’ workload while running a competing compute intensive application. (Right) Competing application slowdown (the lower, the better).

We note that the *non-CA* system imposes a significant burden on the competing application (between 225% and 80% slowdown depending on workload). This is due to high TCP

processing overheads (we have verified this by running the competing application while continuously generating TCP traffic using *iperf*. In this case *iperf* caused 185% application slowdown).

A second, surprising, observation is that under the *different* workload, *CA-GPU* has lower impact on the competing compute-bound application than the *non-CA* system configuration (that does not consume CPU cycles for hashing). While we do not have a precise understanding for this performance disparity, our intuition is that it is related to the blocking of the SAI threads (introduced by GPU calls) that yield more frequently to the application.

The Impact on an I/O bound application. Figure 11, Figure 12, and Figure 13, present MosaStore’s achieved throughput (left) and the disk I/O-bound application slowdown (right), under the three workloads presented in Section 4.2.

There are three main takeaways from these experiments:

- First, offloading to the GPU does not introduce a bottleneck for a competing I/O-intensive application. The competing application slowdown is marginally (5-15%) lower (thus better) than when hashing on CPU.
- Second, outsourcing to the GPU enables better storage system throughput (around 2x better under the ‘similar’ and 1.7x better under the checkpoint workload) even when a competing I/O intensive application is present on the client node.
- Third, the throughput of the GPU-enabled storage system is only slightly affected by the competing application: less than 6% throughput loss compared to the case when the client system runs on a dedicated node.

5. DISCUSSION

This section discusses a number of interrelated issues:

1. *What would be the impact of employing content-based block boundary detection (as opposed to fixed-size blocks as used in our experiments)?*

Our experiments present evidence that GPUs can be exploited to improve storage system throughput by outsourcing hash-based mechanisms (e.g. similarity detection) based on fixed-size blocks. A different approach is to use content-based boundary detection (which leads to variable size blocks) as in Low Bandwidth File System [3] and JumboStore [16].

This approach can bring significant gains: for instance, for the checkpoint images we used as one of our workloads, content-based block boundary detection can enable detecting up to 82% average similarity between successive images (as opposed to an average of only 23% when using fixed-size blocks) [25]. This improvement is due to this mechanism’s resilience to data insertions and deletions. The tradeoff is computational intensity: content-based block boundary detection is significantly more computationally intensive.

Thus, the fact that offloading the hashing operations that enable this technique, namely the sliding-window hashing, offers higher speedups than direct hashing is not surprising. Our evaluation (Figure 14) shows that *HashGPU* enables up to 24x performance gain on the computation of block boundaries. Further speedup gains can be achieved by integrating with *CrystalGPU*.

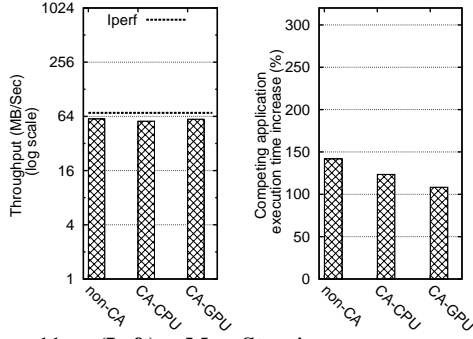


Figure 11: (Left) MosaStore's average achieved throughput while running an I/O intensive application with the different workload. (Right) Competing application slowdown (lower is better).

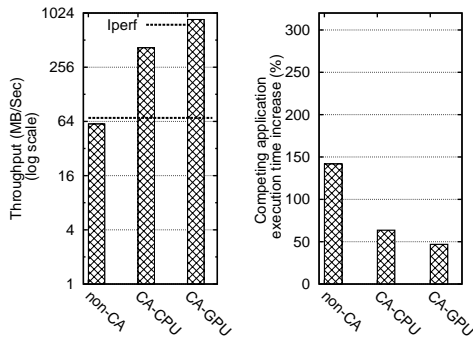


Figure 12: (Left) MosaStore's average achieved throughput while running an I/O intensive application with the similar workload. (Right) Competing application slowdown (lower is better).

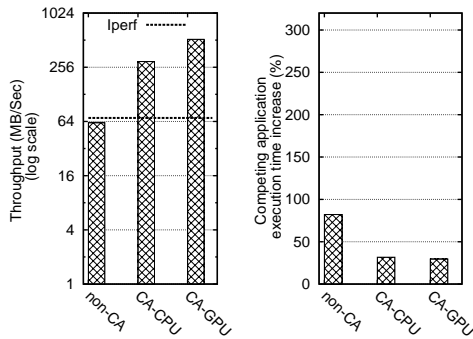


Figure 13: (Left) MosaStore's average achieved throughput while running an I/O intensive application with the checkpoint workload. (Right) Competing application slowdown (lower is better).

We are currently extending *MosaStore* to support variable block sizes required for data similarity detection that uses content-based block boundary detection. We plan to evaluate the end-to-end benefits of this technique on complete prototype under real application workloads for the full version of this paper. Considering the workloads of storage systems that adopt content-based block boundary detection (e.g., the majority of the files handled with JumboStore [16] are larger than 90MBs and, on average 44% similarity is detected), we expect our approach to bring significant performance gains.

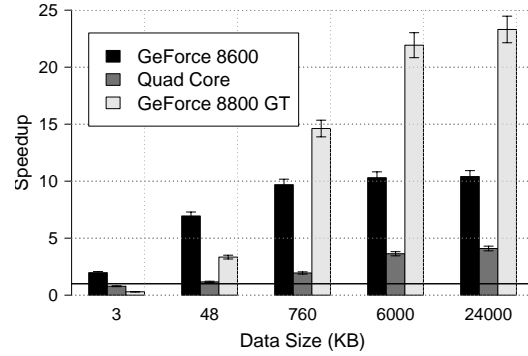


Figure 14. HashGPU sliding-window hashing module speedup for MD5 (the module is configured to hash every 20 bytes with offset of 4 bytes). The figure presents the speedup on GeForce 8600 and 8800, and a multithreaded CPU implementation harnessing all available cores on an Intel Core2 Quad Q6700 2.66GHz. The baseline is the performance on a single core. The value $x=1$ separates the speedup (top) from the slowdown values (bottom). All data points present the average speedup and 95% confidence intervals.

2. What would the system performance be if we had infinite compute power?

While our results (Figure 6) show that significant performance gains can be achieved by offloading the hash computation to the GPU, it is important to estimate the practicality of further investing in hardware or software optimizations to obtain higher throughput. To answer this question, we experimented with a hypothetical storage system configuration (*CA_Infinite* in (Figure 6)) that represents a system with infinite compute power for computing hash functions. This configuration uses an oracle that ‘computes’ the hash function instantly (thus emulating infinite compute power to compute hashes). The results in Figure 6, (which presents the throughput of the system when using the *similar* workload) show that *CA_GPU* performance is close to optimal (as represented by *CA_Infinite* plot line): the throughput loss is lower than 20% for files smaller than 32MB, and lower than 10% for larger files. Thus, we can argue that offloading to the GPU, which in this configuration and workload combination enables up to 2x higher system throughput, will offer most of the possible benefits that could be obtained by accelerating the hash computation.

3. What if chip multiprocessors had many cores? Would a GPU-driven solution be necessary?

Processor manufacturers expect to develop many-core architectures over the next decade [11]. While we have proposed *HashGPU* as a solution that addresses germane problems for storage systems with current technology, the high-level principles embodied in our work will extend to the many-core arena as well. Some of the challenges that we encountered in our development of *HashGPU* relate to the efficient handling of memory transfers and shared memory. Such concerns are likely to exist even with many cores on a single chip; it is expected that many core architectures will utilize heterogeneous cores and non-uniform memory accesses (NUMA) and such architectures will continue to

require careful memory management to achieve significant speedups.

6. RELATED WORK

Hashing. Hashing is commonly used by storage systems to support: content addressability [1, 2], data integrity checking [26, 27, 28], load balancing [29, 30], data similarity detection [3, 16], deduplication backup operations [7], and version control (e.g. rsync [31]). The use of hashing in these systems differs along multiple axes. We enumerate only two: block sizes - from fixed-sized blocks [1, 2] that use hashing similar to our direct-hashing module, to detecting block boundaries based on content [3, 7, 16] and use, what we call sliding-window hashing in Section 2.1; and targeted deployment environment - from personal use [2], peer to peer [27] to data center [7]. Other uses of hashing include: detection of copyright violations [32, 33], compact set representation [34], and various security mechanisms [26].

GPU harnessing. Exploiting GPUs for general purpose computing has recently gained popularity, particularly as a mean to increase the performance of scientific applications. We refer the reader to Owens et al. [35] for a comprehensive survey.

More related to our infrastructural focus, Curry et al. [14] explore the feasibility of using GPUs to enhance RAID system reliability. Their preliminary results show that GPUs can be used to accelerate Reed Solomon codes [36], used in RAID6 systems. Along the same lines Falcao et al. [37] show that GPUs can be used to accelerate Low Density Parity Checks (LDPC) error correcting codes.

Harrison and Waldron [38], study the feasibility of GPUs as a cryptographic accelerator. To this end they implement the AES encryption algorithm using CUDA and report 4x speedup. Finally, Moss et al. [39] study the feasibility of accelerating the mathematical primitives used in RSA encryption. They use OpenGL and render the application into a graphics application and report up to 3x speedup. Moreover, recently exploiting GPUs to accelerate security operations was adopted in few security products, including Kaspersky antivirus [40] and Elcomsoft password recovery software [41].

Our study is different from the above studies in two ways. First, unlike the previous studies that focus on accelerating standalone primitives, we evaluate the viability of exploiting the GPU computational power in the context of a complete system design. Second, the hashing primitives we focus on are data-intensive with a ratio of computation to input data size of at least one order of magnitude lower than in previous studies.

7. CONCLUSIONS

At a high level of abstraction, computing system design is a multi-dimensional constraint optimization problem: designers have at hand various components that offer different cost-to-performance characteristics as well as the techniques to put these components together. These techniques have their own overheads that lead to performance tradeoffs and new limitations. In this context, a system designer optimizes for key success metrics (such as latency, throughput, storage footprint, and data durability) within specific constraints (such as cost and availability).

Historically, the unit costs for raw storage, computation,

or network transfers have evolved largely in sync. Recently, however, massively multi core commodity hardware (such as GPUs) holds the promise of a sharp, one order of magnitude, drop in computation costs. This drop in the cost of computation, as any order-of-magnitude drop in the cost per unit of performance for a class of system components, triggers the opportunity to redesign systems and to explore new ways to engineer them to recalibrate the cost-to-performance relation.

This study demonstrates the feasibility of exploiting GPU computational power to support distributed storage systems, by accelerating popular compute-intensive primitives. We prototyped a storage system capable of harvesting GPU power, and evaluated it under two system configurations: as a content addressable storage system and as a traditional system that uses hashing to preserve data integrity. Further, we evaluated the impact of offloading to the GPU on competing applications' performance, and the impact of competing applications on the storage system performance. Our results show that this technique can bring tangible performance gains enabling close to optimal system performance under the set of workloads and configurations we considered without negatively impacting the performance of concurrently running applications.

Whereas we have effectively demonstrated the feasibility of tapping into GPUs computational power to accelerate compute-intensive storage system primitives, we view this effort as a first step towards the larger goal of understanding how we can use heterogeneous multicore processors for independent system-level enhancements rather than application-level speedups.

We note that future single-chip massively multicore processors will likely be similar to our experimental platform along two directions: they will be heterogeneous (e.g., include different cores with different execution models, SIMD vs. MIMD) and will offer complex application manageable memory hierarchies. In this context performance gains can only be obtained by careful orchestration of data transfers, data placement, and task allocation. Our experience with a combination of parallel hashing on a GPU, optimized memory handling, and task scheduling leads us to believe that these goals can be effectively be mapped to independent abstraction layers.

An ongoing exploration is extending the storage system prototype to support variable-size blocks (and thus support the use of techniques based on content-based block boundaries). Our results suggest that we can unleash the benefits of this mechanism (i.e., much better similarity detection rates) without introducing a system bottleneck.

8. REFERENCES

- [1] S. Quinlan and S. Dorward. *Venti: A New Approach to Archival Data Storage*. in *USENIX Conference on File and Storage Technologies (FAST)*. 2002. Monterey, CA.
- [2] S. C. Rhea, R. Cox, and A. Pesterev. *Fast, Inexpensive Content-Addressed Storage in Foundation*. in *USENIX Annual Technical Conference*. 2008.
- [3] A. Muthitacharoen, B. Chen, and D. Mazieres. *A Low-bandwidth Network File System*. in *Symposium on Operating Systems Principles (SOSP)*. 2001. Banff, Canada.

- [4] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, et al. *StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems*. in *ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC)*. 2008.
- [5] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide v0.8*. 2008.
- [6] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, et al. *Efficient Replica Maintenance for Distributed Storage Systems*. in *3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI)*. 2006.
- [7] B. Zhu, K. Li, and H. Patterson. *Avoiding the disk bottleneck in the data domain deduplication file system*. in *USENIX Conference on File and Storage Technologies (FAST)*. 2008.
- [8] D. Dabiri and I. F. Blake, *Fast parallel algorithms for decoding Reed-Solomon codes based on remainder polynomials*. *IEEE Transactions on Information Theory*, 1995. **41**(4): p. 873-885.
- [9] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. *Parallelizing security checks on commodity hardware*. in *international conference on Architectural support for programming languages and operating systems (ASPLOS)*. 2008. Seattle, WA.
- [10] R. Datta, Dhiraj Joshi, J. Li, and J. Z. Wang, *Image Retrieval: Ideas, Influences, and Trends of the New Age*. *ACM Computing Surveys*, 2008. **40**(2): p. 1-60.
- [11] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, et al. *Larrabee: A Many-Core x86 Architecture for Visual Computing*. in *IEEE Micro*. 2009.
- [12] M. D. Hill and M. R. Marty, *Amdahl's Law in the Multicore Era*. *IEEE Computer*, 2008. **41**(7): p. 33-38.
- [13] M. Polte, J. Simsa, and G. Gibson. *Comparing Performance of Solid State Devices and Mechanical Disks*. in *Petascale Data Storage Workshop* 2008.
- [14] M. L. Curry, A. Skjellum, H. L. Ward, and R. Brightwell. *Accelerating Reed-Solomon Coding in RAID Systems with GPUs*. in *IEEE International Parallel and Distributed Processing Symposium, IPDPS*. 2008.
- [15] S. Ghemawat, H. Gobioff, and S.-T. Leung. *The Google File System*. in *19th ACM Symposium on Operating Systems Principles*. 2003. Lake George, NY.
- [16] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, et al. *JumboStore: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service*. in *USENIX FAST*. 2007.
- [17] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide v2.0*. 2008.
- [18] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, et al. *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA*. in *ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 2008.
- [19] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. *The Case for Versatile Storage System*. in *Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 2009.
- [20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, et al. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. in *SIGCOMM 2001*.
- [21] R. Merkle. *A Certified Digital Signature*. in *Advances in Cryptology - CRYPTO*. 1989: Lecture Notes in Computer Science.
- [22] I. Damgard. *A Design Principle for Hash Functions*. in *Advances in Cryptology - CRYPTO*. 1989: Lecture Notes in Computer Science.
- [23] S. F. Altschul, W. Gish, W. Miller, E. Myers, et al., *Basic Local Alignment Search Tool*. *Molecular Biology*, 1990. **215**: p. 403-410.
- [24] P. H. Hargrove and J. C. Duell. *Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters*. in *Proceedings of SciDAC*. 2006.
- [25] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, and A. Gharaibeh. *stdchk: A Checkpoint Storage System for Desktop Grid Computing*. in *International Conference on Distributed Computing Systems (ICDCS '08)*. 2008. Beijing, China.
- [26] A. R. Yumerefendi and J. S. Chase. *Strong Accountability for Network Storage*. in *FAST'07*. 2007.
- [27] L. P. Cox and B. D. Noble. *Samsara: honor among thieves in peer-to-peer storage*. in *ACM Symposium on Operating Systems Principles*. 2003.
- [28] R. Kotla, L. Alvisi, and M. Dahlin. *SafeStore: A Durable and Practical Storage System*. in *USENIX Annual Technical Conference*. 2007.
- [29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, et al. *Dynamo: Amazon's Highly Available Key-value Store*. in *SOSP07*. 2007.
- [30] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, et al. *Wide-area cooperative storage with CFS*. in *18th ACM Symposium on Operating Systems Principles (SOSP '01)*. 2001. Chateau Lake Louise, Banff, Canada.
- [31] *rsync files synchronization tool*. [cited 2009; <http://www.samba.org/rsync/>].
- [32] S. Brin, J. Davis, and H. Garcia-Molina. *Copy detection mechanisms for digital documents*. in *ACM SIGMOD*. 1995.
- [33] S. Schleimer, D. S. Wilkerson, and A. Aiken. *Winnowing: local algorithms for document fingerprinting*. in *ACM SIGMOD international conference on Management of data*. 2003.
- [34] B. Bloom, *Space/time Trade-offs in Hash Coding with Allowable Errors*. *Communications of the ACM*, 1970. **13**(7): p. 422-426.
- [35] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, et al., *A Survey of General-Purpose Computation on Graphics Hardware*. *Computer Graphics Forum*, 2007. **26**(1): p. 80-113.
- [36] I. S. Reed and G. Solomon, *Polynomial Codes Over Certain Finite Fields*. *Journal of the Society for Industrial and Applied Mathematics*, 1960. **8**(2).
- [37] G. Falcao, L. Sousa, and V. Silva. *Massive Parallel LDPC Decoding on GPU*. in *ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*. 2008. Salt Lake City.
- [38] O. Harrison and J. Waldron. *Practical Symmetric Key Cryptography on Modern Graphics Hardware*. in *USENIX Security Symposium*. 2008. San Jose, CA.
- [39] A. Moss, D. Page, and N. Smart. *Toward Acceleration of RSA Using 3D Graphics Hardware*. in *Cryptography and Coding*. 2007.
- [40] *Kaspersky Antivirus* [cited 2008; <http://www.kaspersky.com/>].
- [41] *Elcomsoft password recovery software* [cited 2008; <http://www.elcomsoft.com>].