

StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems

Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, Matei Ripeanu
Electrical and Computer Engineering Department
The University of British Columbia
Vancouver, Canada
{samera, abduallah, elizeus, gyuan, matei}@ece.ubc.ca

ABSTRACT

Today Graphics Processing Units (GPUs) are a largely underexploited resource on existing desktops and a possible cost-effective enhancement to high-performance systems. To date, most applications that exploit GPUs are specialized scientific applications. Little attention has been paid to harnessing these highly-parallel devices to support more generic functionality at the operating system or middleware level. This study starts from the hypothesis that generic middleware-level techniques that improve distributed system reliability or performance (such as content addressing, erasure coding, or data similarity detection) can be significantly accelerated using GPU support.

We take a first step towards validating this hypothesis, focusing on distributed storage systems. As a proof of concept, we design StoreGPU, a library that accelerates a number of hashing based primitives popular in distributed storage system implementations. Our evaluation shows that StoreGPU enables up to eight-fold performance gains on synthetic benchmarks as well as on a high-level application: the online similarity detection between large data files.

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

Middleware, Storage System, Graphics Processing Unit, GPU hashing, StoreGPU.

1. INTRODUCTION

Recent advances in processor technology [31] have resulted in a wide availability of massively parallel Graphics Processing Units (GPUs) in commodity desktop systems. For example, commodity GPUs like NVIDIA's GeForce 8600 priced at about \$100 have 32 processors and 256 MB of memory while high-end GPUs, like the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HPDC'08, June 23–27, 2008, Boston, Massachusetts, USA.
Copyright 2008 ACM 978-1-59593-997-5/08/06...\$5.00.

NVIDIA 8800 GTX priced at about \$300, have up to 128 processors running at 575MHz and 768MB of memory. With these characteristics, GPUs are often underutilized in corporate/educational desktops (as they are generally provisioned for graphics-intensive workloads such as high-definition video) and may be cost-effective enhancements to high-end server systems

However, the constraints introduced by a GPU programming model which, until recently, specialized in supporting only graphical processing, have led past efforts aimed at harnessing this resource to focus exclusively on computationally intensive scientific applications [33]. While these efforts have confirmed that significant speedup is achievable, the development cost for this specialized platform is often prohibitive. Recently, however, the introduction of general-purpose programming models (e.g., NVIDIA's CUDA [7]) has lowered the development cost and opened this resource to a new set of applications.

This study starts from the observation that a number of techniques that enhance distributed system reliability and/or performance (e.g., content addressability in data storage [8, 34], erasure codes [39], on-the-fly data similarity detection [32]) incur computational overheads that often preclude their effective usage with today's commodity hardware. We study the viability of offloading these data-processing intensive operations to the GPU. *We demonstrate that GPUs offer up to 8x speedup compared to traditional CPU-based processing. This brings in a new overhead tradeoff balance point where the above techniques can be effectively used to support high-performance computing system middleware.*

While the approach we explore can be employed to support a wide range of distributed systems, we focus on distributed storage. The reason is the increasing performance gap between the storage system and the processor, memory, and network subsystems. This performance gap has made the cost-effective engineering of a distributed storage system capable of keeping up with the other components of a distributed system (e.g., data-processing, visualization) an increasingly challenging task.

This project proposes StoreGPU, a library that enables transparent use of a GPU's computational power to support data- and compute-intensive primitives used in distributed storage systems. In particular, we focus on specialized use of hashing to support content addressability, on-the-fly similarity detection, data integrity, and load balancing. By building the StoreGPU library and making it available to the community, we open the possibility of efficiently incorporating these mechanisms into distributed storage systems, thereby unleashing a valuable set of optimization techniques. Furthermore, we present evidence that this approach

can be extended to other routines that support today’s distributed systems like erasure coding [14], compact dataset representation using Bloom filters [11], data compression [25], and data filtering. For instance, in the digital fountain approach for data dissemination [13], the data source continuously generates erasure codes of the original data blocks. A library that transparently outsources the computationally demanding encoding operation to the GPU will dramatically reduce the load of the source CPU and enhance overall system performance.

The contribution of this work is threefold:

- First, this project explores a new territory: the use of GPUs to support general purpose computing middleware (as opposed to specialized scientific applications). We show that exploiting GPU in this context brings valuable performance gains. Moreover, we present evidence that GPUs can enhance the performance of storage systems, a usage scenario where the challenge lays in the data-intensive nature of system operations. In this scenario, large volumes of data need to be sequentially processed; an operational case outside the scope of the original GPU design. To the best of our knowledge, no previous studies have attempted to use the GPUs to enhance the performance of this category of applications.
- Second, we present a performance model that allows the estimation of a data-processing application’s performance on a given GPU model. The performance model can be used to evaluate whether modifying an application to exploit GPUs is worth the effort. We also present a detailed analysis of the factors that influence performance for a subset of applications and quantitatively evaluate their effect.
- Finally, we make the StoreGPU library available to the community. This library can be used to harness the computational power of GPUs with minimal modifications to current systems¹. Depending on its configuration and the target application’s data usage patterns, StoreGPU enables significant performance gains. When comparing the performance enabled by a low-end GPU (the NVIDIA 8600GTS) and a commodity CPU (Intel Core2 Duo 6600), StoreGPU achieves up to eight-fold performance gains on not only synthetic benchmarks but also when supporting a high-level application.

The rest of the paper is organized as follows. The next section justifies our choice of optimizing hash-based operations through a survey of hashing use in storage systems (Section 2.1) and describes the GPU programming model and the main factors influencing application performance when using GPUs (Section 2.2). Section 3 details the design of StoreGPU library. Section 4 presents our experimental results, section 5 surveys the related work, and section 6 presents a discussion of our approach. We conclude in section 7.

2. BACKGROUND

This section surveys the use of hashing techniques to support efficiency and reliability in data storage systems (whether distributed or not) and presents the architecture and programming model of the used GPU.

2.1 Hashing in Storage Systems

Hash-based primitives are commonly used by data-oriented system middleware. Content addressability, data integrity, load

balancing, data similarity detection, and compact set representation are all middleware primitives with best implementations based on various uses of hashing. Yet the computational overheads of these implementations sets them apart as potential bottlenecks [34] in today’s high-performance distributed systems that commonly employ multi-Gbps optical links.

This section briefly details some of these hashing-based primitives with two goals in mind: First, we support the argument that their computational overheads prevent their use in conventional high-performance systems. Second, we present the usage scenarios that inform the design of StoreGPU.

2.1.1 Content Addressable Storage

In systems that support content addressability [34], data blocks are named based on their content. In this context, hashing is used as a naming technique: data-block names are simply the hash value of the data [8, 34, 38]. There are multiple advantages to this approach: it provides a flat namespace for data-block identifiers and a naming system which, in turn, simplifies the separation of file-metadata from data-block metadata. However, the overhead required to compute block hashes may limit performance in workloads that have frequent updates.

2.1.2 Data-similarity detection

Content addressability enables tradeoffs between computation and storage space overheads. Consider a versioning file system [32]: When a client saves a new version of a file, the file system divides the file into blocks, computes their identifiers (often hashes of the block), and sends these identifiers to the storage system. The storage system, in turn, compares the identifiers received with the identifiers of blocks of previous versions of the file to detect which blocks have changed and need to be persistently stored. The client, informed of the presence of similar blocks, will not store them again, saving considerable storage space and network bandwidth. Experience shows space savings can be as high as 60% in production [34] and research systems [9].

2.1.3 Data Integrity

In an untrustworthy environment, hashing is used to support data integrity and non-repudiability guarantees. For example, in accountable storage systems [40], Samsara [15], SFS [21], and SafeStore [27], integrity of the stored data is protected using digital signatures. To keep the overhead of signing and verifying integrity manageable, only the hash of the data is signed and stored together with the data and the public credentials of the signing entity.

2.1.4 Load Balancing

Hashing is used to load-balance a distributed storage system. For instance, hashing is used in systems based on consistent-hashing techniques [26, 35, 36] to assign loads to nodes [16, 19]. A good hashing function that minimizes collisions leads to an efficient data distribution technique since blocks are distributed evenly between the storage nodes.

2.1.5 Computing Block Boundaries

To implement the aforementioned techniques, storage systems need to divide files into multiple blocks. To this end, two approaches are possible: fixed- or variable-size blocks. In the first approach, the file is divided into a set of equally-sized blocks. In the second approach, block boundaries (i.e., markers for block start and end) are defined based on file content. For instance, the

¹ StoreGPU is an open source project, the code can be found at: <http://netsyslab.ece.ubc.ca>

Low-Bandwidth File System (LBFS) [32] and JumboStore [20] both detect block boundaries by passing all successive 48 byte ‘windows’ of the file through a hash function and declaring a block boundary if the last 20 bits of the hash value are all zero. The advantage of this approach is that, unlike fixed-blocks, the ability to detect block similarities is preserved even in the presence of data insertion and deletion. However, this technique is computationally intensive since a large number of hashes need to be computed to determine the block boundaries, therefore imposing a high overhead and making usage in the context of general-purpose data storage systems difficult. In fact, the low throughput provided by this technique is the main reason its proponents recommend its use in storage systems supported by low-bandwidth networks [32].

2.1.6 Summary of Usage Scenarios

We can reduce the use cases presented to two uses of hashing:

- Direct Hashing, where the hash for an entire data block is computed (to support content addressability, data integrity, or fixed-block similarity detection), and
- Sliding Window Hashing, where a large number of possibly overlapping windows in a large data block are computed (to support similarity detection based on variable-block sizes).

Finally, we note that the aforementioned techniques require a collision-resistant hash function, such as MD5 or SHA, which are more computationally intensive than simple hashing functions (e.g., CRC codes).

2.2 GPU Programming

This section presents an overview of the latest GPU models’ architecture, main performance factors, and the programming model. We focus on NVIDIA’s architecture and programming environment: the Compute Unified Device Architecture (CUDA) [7]. Other vendors (e.g., AMD) have also developed similar architectures and programming environments [1]. We have selected the NVIDIA cards for two reasons. First, it has the largest market share [5, 6] and second, the CUDA programming model is more suitable for general purpose computing. Unlike AMD’s Close To Metal (CTM) programming model [1], which provides a low-level assembly language for GPU programming, CUDA supports the use of C programming language for application programming.

The GPU has a Single-Instruction Multiple-Data (SIMD) architecture. It offers a number of SIMD multiprocessors and four different memories each with their own performance characteristics (detailed in the next subsection). The CUDA programming model extends the C language with directives and libraries to abstract the GPU architecture. CUDA allows the programmer to control application variables’ allocation (e.g., the memory type in which the variable resides), and provides an API for GPU specific functions, such as device properties querying, timing, memory management, and per multiprocessor thread synchronization.

Although using C as the programming language lowers the barriers to developing general purpose code on GPUs, the programming model requires that the application fits the Single-Instruction Multiple-Data (SIMD) model. Moreover, despite the abstractions provided by the CUDA API, it is still challenging to make efficient use of the GPU memory. As our performance analysis (Section 4) shows, poor memory management may critically impair performance.

2.2.1 GPU Architecture

Figure 1 presents a high-level view of NVIDIA’s GPU architecture. The device is composed of a number of SIMD multiprocessors. Each multiprocessor incorporates a small (16KB in the GeForce 8600GTS) but fast memory, shared by all processors in the multiprocessor. All multiprocessors have access to three other device-level memory modules: global (a.k.a. device memory), texture, and constant. These memories are also accessible from the host machine. The global memory supports read and write operations and it is the largest memory in the GPU (with size ranging from 256 to 768 MB). In comparison, the texture and constant memories are much smaller and have restricted access policies. Apart from size, the critical characteristic of the various GPU memory modules is their access latencies. While accessing an entry in the shared memory takes up to four cycles, it takes 400 to 600 cycles to access the global memory [7].

Typically, a general purpose application will first transfer the application data from host (CPU) memory to the GPU global memory and then try to maximize the usage of the shared memory throughout the computation.

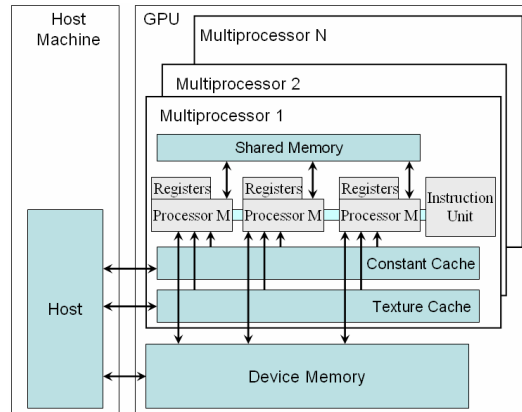


Figure 1. GPU Architecture.

Programming efficient applications to exploit GPUs implies extracting the target application’s parallelism and employing efficient memory and thread management techniques. Improper task decomposition, memory allocation, or memory transfers can lead to dramatic performance degradation. Particularly, efficient use of the shared memory is a challenging task for three reasons. First, the shared memory is often small compared to the volume of data being processed. Second, the shared memory is divided into 16 banks and all read and write operations that access the same bank are serialized, hence, reducing concurrency. Consequently, to maximize the performance, an application should schedule concurrent threads to 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, increasing the number of threads per multiprocessor does not directly lead to a linear performance gain although it may help hide global memory access latency. The reason is that increasing the number of threads decreases the amount of shared memory available per thread. Obviously, the optimal resource usage configuration is tightly coupled with the application characteristics (e.g., the data access patterns) and GPU hardware specifications (the number of registers in the multiprocessor or the size of shared memory available).

2.2.2 GPU Performance Factors

When using the GPU, an application passes through five main stages: preprocessing, host-to-GPU data transfer, processing, GPU-to-host results transfer, and post-processing. Table 1 describes these stages, identifies the main performance factors for each stage, and introduces the notation used throughout the rest of this paper to model performance. (We note that not all applications will have the preprocessing or post-processing stages.)

Table 1. Application processing stages and performance factors.

Stage	Sub-stages	Operations performed
(1) Preprocessing	1.1. GPU initialization ($T_{GPUInit}$)	GPU initialization
	1.2. Memory allocation ($T_{MemAlloc}$)	Memory allocation at the host and the GPU
	1.3. Pre-processing ($T_{PreProc}$)	Application-specific data preprocessing on the CPU
(2) Data Transfer In	Data transfer to GPU ($T_{DataHtoG}$)	Data transfer from host's memory to GPU global memory
(3) Processing	3.1. Data transfer to shared memory ($T_{DataGtoS}$)	Data transfer from global GPU memory to shared memories.
	3.2. Processing ($T_{GPUProc}$)	Application 'kernel' processing
	3.3. Data transfer to device global memory ($T_{DataStoG}$)	Result transfer from shared memory to global memory
(4) Data Transfer Out	4.1. Output data transfer ($T_{DataGtoH}$)	Transfer the results to the host system RAM.
(5) Post-processing	5.1. Post-processing ($T_{PostProc}$)	Application-specific post processing on CPU resource deallocation

For a data-parallel application, the processing step is usually repeated multiple times until all input data is processed. During each iteration, parts of the data are copied from global memory to each multiprocessor's shared memory and processed by the application 'kernel' before the results are then transferred back to the global memory. Thus, the runtime of a data parallel application can be modeled as:

$$\begin{aligned}
 T_{Total} &= T_{Preprocessing} + T_{DataHtoG} + T_{Processing} + T_{DataGtoH} + T_{PostProcH} \\
 &= T_{GPUInit} + T_{MemAlloc} + T_{PreProc} + T_{DataHtoG} + \\
 &\quad \frac{DataSize}{N \times SMSize} \times (T_{DataGtoS} + T_{Proc} + T_{DataStoG}) + \\
 &\quad T_{DataGtoH} + T_{PostProc}
 \end{aligned} \tag{1}$$

where $DataSize$ is the size of an application data set, N is the number of multiprocessors, and $SMSize$ is the size of the multiprocessor's shared memory. The parameters that influence the formula above (e.g., host-to-memory transfer throughput, device global-to-shared memory throughput, initialization overheads) can be either benchmarked or found in the GPU data sheets.

Equation 1 allows system designers to estimate the benefits of offloading processing to the GPU and to identify parts of the application that need optimization. GPUs are known for their ability to accelerate number-crunching applications, but are less efficient when hashing large volumes of data. This is due not only to the overheads incurred when transferring large amounts of data to and from the device, but also to the fact that the various floating point units are not used. In fact, trivial data processing, such as a simple XOR between two data blocks, even on a large amount of data, is faster on the CPU than on the GPU. While the GPU can perform computations at a huge theoretical sustained instruction-per-second peak rate (46.4 GIPS -Giga Instruction Per Second for the NVIDIA 8600 GTS card), the data transfer from the machine

to the GPU is limited at 4GB/s, the theoretical maximum bandwidth of PCIe 16x interface.

To give the reader an intuition of how the various overheads interplay, we present the time breakdown to hash a 96MB data block: transferring the data to the GPU takes 37.4ms (for an achieved throughput of 2.5GBps), hashing takes 41.8ms (using the four GT8600 multiprocessors), and copying the results back takes 1.0ms. Overall, in this configuration, the memory transfers represent over 48% of the execution time.

3. StoreGPU Design

The design of StoreGPU is driven by storage systems' use of hashing as presented in section 2.1. This section presents StoreGPU's application programming interface (API) and a high-level design overview. We present a number of performance-oriented design improvements in the evaluation section.

SHA1 (RFC 3174) and MD5 (RFC 1321), as well as most widely used hash functions, follow the sequential Merkle-Damgård construction approach [18, 30]. In this sequential approach, at each stage, one chunk of data is processed to produce a fixed size output. The output of each stage is used as an input to the following stage together with a new data chunk. This sequential construction does not allow multiple threads to operate concurrently in hashing the data. To exploit the highly parallel GPU architecture, our design uses the original hash functions as a building block to process multiple chunks of data in parallel. The discussion section presents evidence that the hash function we build is as strong as the original, sequentially built, hash function.

3.1 StoreGPU API

We designed StoreGPU API to correspond to the two main use cases presented in Section 2.1.

Direct Hashing, i.e., hashing large blocks of data, with size ranging from kilobytes to megabytes or more. To address this scenario, the library provides the following interface:

```
char* SHA(char* DataBuffer, int DataBufferSize)
char* MD5(char* DataBuffer, int DataBufferSize)
```

Sliding Window Hashing. As opposed to the first case, content-based detection of block boundaries requires hashing numerous small data blocks (sized from tens to hundreds of bytes). To address this usage pattern, the library provides the following interface:

```
char* SHA(char* DataBuffer,
           int DataBufferSize, int WinSize, int Offset)
char* MD5(char* DataBuffer,
           int DataBufferSize, int WinSize, int Offset)
```

This API returns an array of hashes, where each entry of this array is the result of hashing a window of data of size $WinSize$ at shifting offset $Offset$.

The rest of this section presents the two main modules of StoreGPU with a focus on parallelizing hash computations.

3.2 Design of the Direct Hashing Module

Figure 2 presents StoreGPU's direct hashing module design. Once input data is transferred from the CPU, it is divided into smaller blocks and, every small block is hashed. The result is placed in a single output buffer and, finally, the output buffer is hashed to produce the final hash value.

Two aspects are worth mentioning. First, there are no dependencies between the intermediate hashing computations in Step 2 (Figure 2). Consequently, each computation can be

executed in a separate thread. Second, this design uses the CPU to aggregate the intermediary hashes (Step 3). The reason is that synchronization of GPU threads across the blocks inside the GPU is not possible.

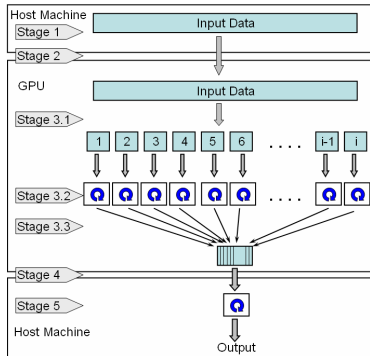


Figure 2: Direct hashing module architecture. The blocks with circular arrows represent the standard hashing kernel. Stages numbers correspond to Table 1.

3.3 The Sliding Window Hashing Module

To parallelize the computation of a large number of small hashes drawn from a large data block, we hash in parallel all the small blocks and aggregate the result in a buffer. This module’s architecture is presented in Figure 3.

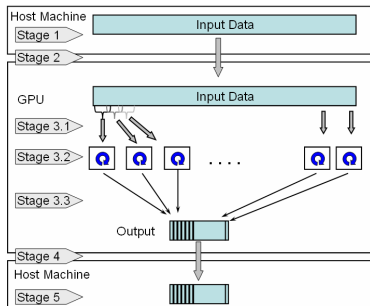


Figure 3: Sliding window hashing module architecture. The blocks with circular arrows represent the standard hashing kernel. Stage numbers correspond to Table 1.

Each of the hash functions in Figure 3 can be executed in a separate thread since there are no dependencies between computations. The challenge in implementing this module lies in the memory management to extract maximum performance. Note that the input data is not divided into smaller blocks as the previous case. The reason is that the input data for each thread may overlap with the neighboring threads.

3.4 Optimized Memory Management

Although the design of the two modules presented here are relatively simple, optimizing their performance for GPU deployment is a challenging task. For example, one aspect that induces additional complexity is maximizing the number of threads to extract maximal parallelism (around 100K threads are created for large blocks) while avoiding multiple threads accessing the same shared memory bank and maximizing the use of each processors’ registers. To this end, we have implemented our own shared memory management mechanism with two main functions. First, it allocates a fixed space for every thread in a single shared memory bank and avoids assigning workspaces allocated on the same memory bank to concurrent threads in the same multiprocessor. 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, it abstracts the shared memory to allow the thread to access its workspace as a contiguous address space. Effectively the shared memory management mechanism serves to increase the shared memory performance through avoiding bank conflicts while providing a contiguous memory address abstraction.

3.5 Other Optimizations

In addition to optimizing the shared memory usage, we considered two other optimizations: the use of pinned memory, and reducing the size of the output hash.

Allocating and initializing the input data in host’s *pinned memory* (i.e., non-pageable memory) saves the GPU driver from an extra memory copy to an internal pinned memory buffer. In fact, the GPU driver always uses DMA (Direct Memory Access) from its internal pinned memory buffer when copying data from the host memory to the GPU global memory. Therefore, if the application allocates the input data in pinned memory from the beginning, it saves the driver from performing the extra copy to its internal buffer. However, allocating pinned memory adds some overhead since the kernel is involved in finding and preparing a contiguous set of pages before locking it. Our performance numbers do not show a pronounced effect for this overhead, since pinned memory buffers can be reused over subsequent library calls and thus this overhead is amortized.

Additionally, we allow users to specify the size of the desired output hash. The rationale behind this feature is that, some applications such as block boundary for similarity detection only need the first few bytes of the hash value.

4. EXPERIMENTAL EVALUATION

We evaluate StoreGPU both with synthetic benchmarks (Section 4.1) and an application driven benchmark: we estimate the performance gain of an application using StoreGPU to compare similarities between multiple versions of the same checkpoint image (Section 4.2).

4.1 Synthetic Benchmarks

This section presents the performance speedup delivered by StoreGPU under a synthetic workload. We first compare GPU-supported performance with the performance of the same workload running on a commodity CPU. Next, this section investigates the factors that determine the observed performance.

4.1.1 Experiment Design

The experiments are divided into two parts, each corresponding to the evaluation of one of the two uses of hashing described in Section 3 (i.e. Direct Hashing and Sliding Window Hashing). The performance metric of interest is execution speedup.

Table 2 summarizes the factors that influence performance. Currently, StoreGPU provides the implementation of two hashing algorithms: MD5 and SHA1. The data size variation is intended to expose the impact of memory copy between the host and the GPU. Additionally the sliding-window hashing module has two specific parameters: window and offset size.

Additionally, we explore the impact of the three performance optimizations presented in section 3 : i) the optimized use of shared memory; ii) memory pinning; and iii) reduced output size.

Table 2: Factors considered in the experiments and their respective levels. Note that the sliding-window hashing module has extra parameters.

Direct and Sliding Window Hashing	
Factors	Levels
Algorithm	MD5 & SHA1
Data Size	8KB to 96MB
Shared Memory	Enabled or Disabled
Pinned Memory	Enabled or Disabled
Sliding-Window Hashing only	
Window Size	20 or 52 bytes
Offset	4, 20 or 52 bytes
Reduced Hash Size	Enabled or Disabled

The devices used in the performance analysis are: an Intel Core2 Duo 6600 processor (released late 2006) and an NVIDIA GeForce 8600 GTS GPU (released early 2007). We note that, in both cases, our implementation uses out-of-the-box hash function implementations. These implementations are single-threaded and use only one core of the Intel processor. We defer the discussion on the impact of the experiment platform choices to Section 6.

For all performance data points, we report averages over multiple experiments. The number of experiments is adjusted to guarantee 90% confidence intervals. We applied a full factorial experimental design to evaluate the impact of each combination of factors presented in Table 2. The following sections present a summary of these experiments.

4.1.2 Experimental Results

The first question addressed by our experiments is: *What is the execution time speedup offered by StoreGPU compared to a CPU implementation?* To answer this question, we determine the ratio between the execution time on the GPU and the CPU for both MD5 and SHA1 hashing algorithms.

Figure 4 shows the speedup achieved by StoreGPU for MD5 and SHA1 respectively for the Direct Hashing module. Values larger than one indicate performance improvements, while values lower than one indicate a slow down. The results show that the optimized (pinned and shared memory optimizations enabled) StoreGPU starts to offer speedups for blocks larger than 300KB and offer up to 4x speedup for large data blocks (>5MB).

Note that as the data size increases, the performance improvement reaches a saturation point. It is also important to observe that non-optimized GPU implementations may perform much worse than its CPU counterpart. When memory accesses are not optimized, the performance can decrease up to 30x for small blocks (8KB and MD5). This fact highlights two aspects: first, efficient memory management is paramount to achieving maximum performance in data-intensive applications running on GPUs; second, as the data size grows, the impact of the overhead in moving the data from the host to the device lowers compared to the processing cost. We discuss the latter point in more detail in the next section.

Figure 5 and Figure 6 present the results of experiments for the sliding-window hashing module. Qualitatively, the observed behavior is similar to the direct hashing module. Quantitatively, however, the speedup delivered by StoreGPU is much higher. The figures show the results for MD5 sliding window hashing module. Other parameter choices and choosing SHA1 lead to similar patterns. Hence we do not include these results here. We direct the reader to our technical report.

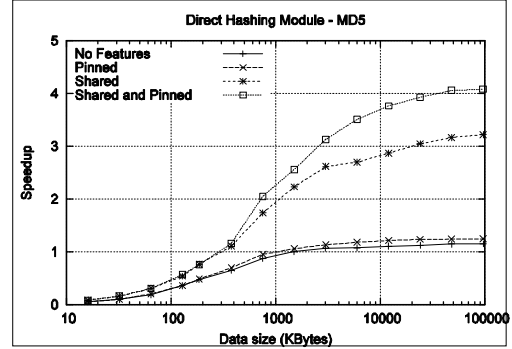


Figure 4: StoreGPU speedup for MD5 implementations for direct hashing. SHA1 performs similarly.

Sliding window hashing introduces two extra parameters that influence performance: the window size and the offset. The window size determines how much data is hashed while the offset determines by how many bytes the window is advanced after each hash operation. The experiments explore four combinations for these two factors with values chosen to match those used by storage systems like LBFS [32], Jumbostore [20], and stdchk [9].

Figure 5 shows the results for a configuration that leads to intense computational overheads: a window size of 20 bytes and an offset of 4 bytes. In this configuration (in fact suggested by LBFS), StoreGPU hashes the input data up to 9x faster for MD5 and up to 5x faster for SHA1. For slightly larger chunks (56 bytes), StoreGPU performs a little slower when compared to the previous scenario. The speedup offered is over 7x for the MD5 algorithm and about 4.8x for SHA1. The same trend is also observed for experiments where the offset is increased to 20 bytes, as shown in Figure 6 .

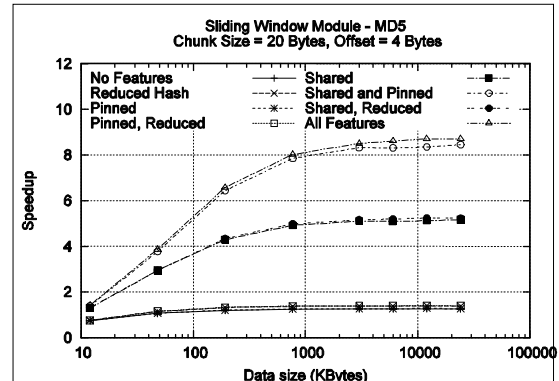


Figure 5: StoreGPU sliding-window hashing module speedup for MD5. Window size=20 bytes, offset=4 bytes.

The sliding window hashing achieves higher speedup compared to direct hashing module for two reasons: First, the CPU implementation of the sliding window hashing will pay an additional overhead of a function call to hash each window, while StoreGPU spawns one thread per window that can execute in parallel. Second, since the window size is usually less than 64 bytes (the input size for SHA or MD5), every window is padded to complete the 64 bytes. This translates to hashing considerably larger amounts of data for the same given input data, making this module more computationally intensive and thus a better fit for GPU processing. This is also the reason we observe larger speedups with smaller window sizes and offsets.

Finally, we observed that the speedup achieved for MD5 is better than SHA1. Although we do not have a precise understanding of the reasons for this performance disparity, our intuition is that this is due to the intrinsic characteristics of the algorithms.

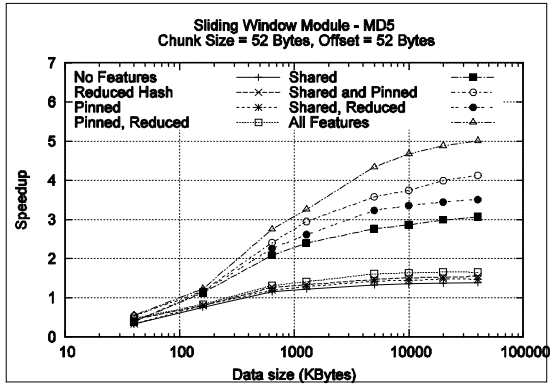


Figure 6: StoreGPU sliding-window hashing module speedup for MD5 Window size=56 bytes. Offset=56 bytes.

4.1.3 Dissecting the Overheads

The execution time of a particular computation on the GPU can be divided into the five stages outlined in section 2.2.2.: preprocessing, host-to-GPU data transfer, code execution, GPU-to-host result transfer, and finally post-processing operations on CPU (e.g., result aggregation, release of resources).

This section analyzes how the execution time of each of these stages is affected by the three optimization features available: pinned memory, shared memory and reduced hash size. Due to space constraints, we limit our analysis to the direct hashing module and MD5 algorithm implementation. Although not reported here, the sliding-window module and the SHA1 implementations present the same characteristics.

Stage 1: Preprocessing. Our application does not have a special data preprocessing operation, consequently this stage is effectively reduces to memory allocation only. The allocation of memory buffers (host and GPU) and the allocation of the buffer for returned results on the host main memory take between 0.3ms and 14ms depending on the data size and whether the pinned memory optimization is enabled (Figure 7). The initialization takes longer with pinned memory and larger data sizes as it is costly to find contiguous pages to accommodate larger data sizes. However, the proportional overhead implied by the initialization time follows is negligible (Step 1 in Figure 10 to Figure 11).

Stage 2: Data transfer In. The host-to-device transfer time varies depending on the data size and whether Pinned Memory optimization is used. As expected, although using pinned memory slows down Step I, it significantly improves transfer performance (Figure 8). Compared to the theoretical 4GBps peak throughput of the PCIe 16x bus, we obtain, for large blocks, 2.5GB/s with pinning and 1.7GB/s without.

Stage 3: Data processing. The performance of kernel execution is highly dependent on the utilization of shared GPU memory and its optimized use (i.e. avoiding bank conflicts - Figure 9). For large data volumes, without the optimized memory management, the kernel contributes up to 80% to the overall operation time (Figure 10). When all optimizations are enabled, efficient use of shared memory reduces the kernel execution impact to about 40% of the total execution time (and Figure 11).

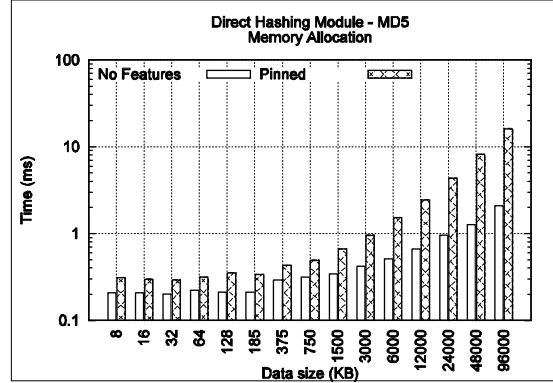


Figure 7: Stage 1 duration with and w/o pinned memory.

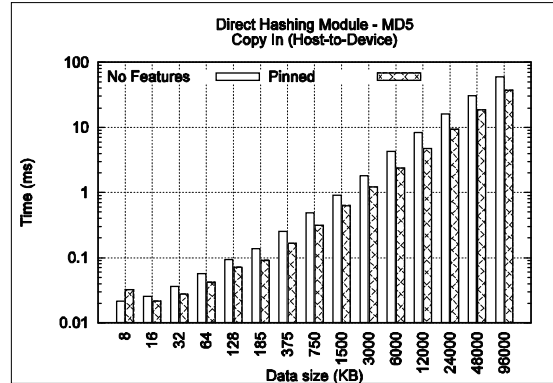


Figure 8: Stage 2: Input transfer time.

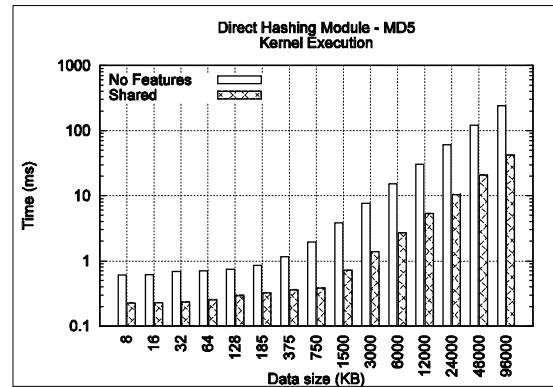


Figure 9: Stage 3: Time spent on kernel execution with/without shared memory optimization enabled.

Stage 4: Data Transfer Out. Transferring the output causes proportionally less impact on the overall execution than transferring the input (Figure 10 to Figure 11). The reason is that, for direct hashing, the output size is several orders of magnitude smaller than the input. Moreover, the output buffers are always pinned; therefore, this step always benefits from the high throughput achieved by using pinned memory pages. As a result, we do not observe any major difference in terms of the impact caused by the output transfer across tested configurations.

Stage 5: Post-processing. Finally, the aggregation of the kernel output into one hash value takes only up to a few milliseconds and has a minor impact on the overall execution time. Enabling GPU optimizations do not influence the performance of the last stage (hash aggregation), since the execution is performed on the CPU.

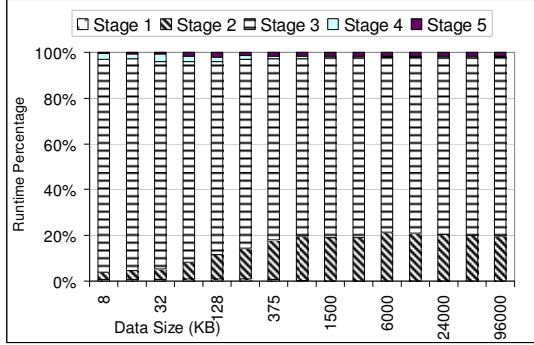


Figure 10: Percentage of total execution time spent on each stage when none of the optimizations are enabled.

Figure 10 and Figure 11 illustrate the proportion of total execution time that corresponds to each execution stage. These results show the major impact of pinned and shared memory optimizations on the contribution of each stage to the total runtime. Using pinned memory reduces the impact of data transfer (compare Stage 2 in Figure 10 to Figure 11), while using the shared memory reduces kernel execution impact (compare Stage 2 in Figure 10 to Figure 11). Finally, enabling both optimizations increases the impact of the copy operation, since pinning memory demands a higher overhead during the allocation stage (Stage 1 in Figure 11).

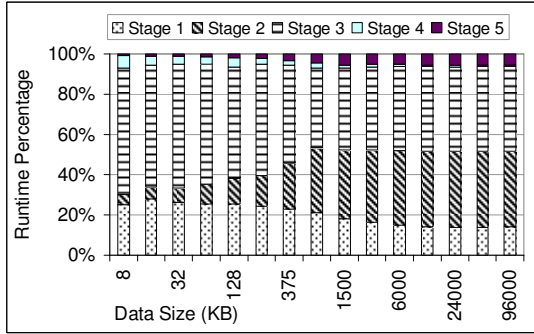


Figure 11: Percentage of total execution time spent on each stage with pinned and shared memory optimizations enabled.

4.2 Application Level Performance

This section complements the synthetic benchmarks presented so far. We evaluate the application-level gains achieved by using StoreGPU. Concretely, we evaluate the speedup offered by using StoreGPU to detect similarities between successive checkpoint images of the same application.

Checkpointing is an indispensable fault tolerance technique adopted by long-running applications. These applications periodically write large volumes of snapshot data to persistent storage in an attempt to capture their current state. In the event of a failure, applications recover by rolling-back their execution state to a previously saved checkpoint. Consecutive checkpoint images have often a high degree of similarity (for instance, Al-Kiswany et al. [9] detect up to 82% similarity).

We have collected the checkpoint images using BLCR checkpointing library [24] from 24 hour-long runs of BLAST, a popular bioinformatics application [10]. The interval between the checkpoints is 5min. The average image size is 279MB.

Table 3. Online similarity detection throughput (in MBps) and speedup using SHA1.

	Throughput (MBps)		Similarity ratio detected
	StoreGPU	Standard	
Fixed block size (using direct hashing)	460.2	126.3	23.4%
	Speedup: 3.6x		
Variable block size (LBFS technique using sliding window hashing)	51.9	9.7	82.0%
	Speedup: 5.4x		

Table 4. Online similarity detection throughput (in MBps) and speedup using MD5.

	Throughput (MBps)		Similarity ratio detected
	StoreGPU	Standard	
Fixed block size (using direct hashing)	840	193	23%
	Speedup: 4.3x		
Variable block size (LBFS technique using sliding window hashing)	114	13.5	80%
	Speedup: 8.4x		

Table 3 and Table 4 compare the throughput of online similarity detection between using standard hashing functions running on CPU and using StoreGPU. These results show dramatic improvement in the throughput of online similarity detection with both fixed and variable size blocks. Despite the fact that we are using a lower-end GPU, the results indicate that fixed-block similarity detection can be used even on 10Gbps systems while the variable block size technique can be used for systems connected with 1Gbps networks without introducing a performance bottleneck. We anticipate that using high-end graphical cards, e.g., NVIDIA GeForce 8800 series, will significantly enhance performance even further.

5. RELATED WORK

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. [33] for a comprehensive survey.

A number of science-oriented applications stand out. Liu et al. [29] implemented the Smith-Waterman algorithm, which compares two biological sequences by computing the number of steps required to transfer one sequence to the other, for a GPU platform and reported an 16x speedup in some cases. The algorithm is often used by bioinformatics applications to compare an unknown sequence with a database of known sequences.

Thompson et al. [37] compared GPU with CPU implementations for a variety of programs, such as matrix multiplications and a solver for the 3-SAT problem. They also suggest minor extensions to current GPU architectures to improve their effectiveness in solving general purpose problems. Following this trend, Buck et al. [12] proposed a programming environment for general-purpose computations on GPUs that provides developers with a view of the GPU as a streaming coprocessor. Kruger et al. [28] implemented linear algebra operators for GPUs and demonstrated the feasibility of offloading a number of matrix and vector operations to GPUs.

More related to our infrastructural focus, Govindaraju et al. [23] have implemented a number of database operations for GPUs, including conjecture selection, aggregation, and semi-linear queries operations.

Our study is different from the above studies in three ways. First,

we employ the latest GPU generation and the CUDA programming modes which are more suitable for general purpose programming. Unlike most of the previous studies [37], this relieves us from having to retrofit the problem that we solve into a graphics problem. Second, unlike the previous studies, we focus on primitive storage system operations and place them in a library, thus providing infrastructure for a broad set of applications. Finally, we focus on data-intensive processing applications with a ratio of computation to input data of at least one order of magnitude lower than previous studies.

6. DISCUSSION

This section focuses on a number of interrelated questions:

1.) *Are StoreGPU hash function implementations strong? Is the system backward compatible?*

Most of today's hash functions are designed using the Merkle-Damgard construction [18]. Merkle and Damgard show that collision-free hash functions can divide data into fixed-sized blocks to be processed either sequentially, using the output of each stage as the input to the next, or in parallel and then concatenating and hashing the intermediate hash results to produce a single final hash value. Most hash functions such as MD5 and SHA adopt the iterative model because it does not require extra memory to store the intermediate hash results.

Our approach for the direct-hashing module is based on the parallel construction. This choice has two implications. First, as a direct implication of the Merkle and Damgard argument, the resulting hash function will still have the same strength as the original sequential construction. Second, while our sliding-window module is still backward-compatible, hashing only small data windows, our direct-hashing technique produces different hash values compared to the sequential MD5 or SHA versions. This does not have an impact on the StoreGPU usability as long as all entities in the storage system use the same library. While we are still investigating alternatives to maintain backward-compatibility, one way to reduce the migration burden is to provide CPU implementations of StoreGPU that implement the same algorithm.

2.) *What are the implications of newer GPU cards (e.g., NVIDIA GeForce 9800 priced at \$300) and of the new programming model (CUDA 1.1)?*

Two enhancements in the newer GPU cards have the potential to increase GPU-supported application performance: First, high-end cards are much more powerful. For example the GF9800 GTX has four times as many processors (128 cores), two times higher memory bandwidth (70.4 GB/s), and two times higher host-to-GPU bandwidth (due to using PCIe 2 16x interface) [4] than the card we use. This additional capacity should speed up the GPU-supported execution. Second, our work is based on CUDA v1.0. NVIDIA has recently released CUDA v1.1 which supports, among other features, an API which allows asynchronous memory copies for pinned memory and kernel launches [2]. This introduces the possibility to overlap GPU kernel execution and memory transfers. We estimate that by exploiting these features, we will be able to provide additional speedups.

3.) *Can other middleware benefit from this idea?*

We believe that a host of other popular primitives used in distributed systems can benefit from GPU support, such as erasure coding, compressed set representation using Bloom filters, and data compression among others. For example, different parallel algorithms for Reed-Solomon coding exist [17] and can be

deployed on GPUs; on the other hand, Gilchrist [22] proposes a parallel implementation of the bzip2 loss-less data compression algorithm that may benefit from GPU support. Currently, we are experimenting with a GPU-optimized Bloom filter implementation. In general, we believe that GPUs can be used by any data-parallel application to provide significant performance improvements, provided that the number of operations performed per byte being processed is sufficiently high to amortize the additional overheads due to host-device memory transfers.

4.) *How does StoreGPU perform against the theoretical peak?*

Since StoreGPU is a data-intensive application as opposed to a compute-intensive one, we first consider memory access throughput. While the memory bandwidth listed in NVIDIA's GeForce 8600 specification is as a high 32GB/s [3], the real memory access bottleneck is the PCI-Express bus, listed at 4GB/s in each direction. This is congruent to our experiments, which show that pinned memory transfers achieve up to 2.48GB/s. Furthermore, we estimate NVIDIA's GeForce 8600 theoretical non floating point instruction execution peak rate at to 46.4 GIPS (Giga Instruction Per Second). Our StoreGPU kernel performs at up to 19.54 GIPS, a slowdown compared to the peak rate mainly due to internal memory copy operations inside the GPU.

5.) *Is the comparison fair?*

We have used two low-end devices, an Intel Core2 Duo 6600 processor and an NVIDIA GeForce 8600 GTS GPU, for our comparison. In both cases, we used the unmodified hashing functions (with best compiler options). On the CPU-side, two additional optimizations may also be considered: first, using a multi-threaded implementation to exploit all CPU cores, and second, using Intel's Streaming SIMD Extensions (SSE).

We believe that not exploring these optimizations does not impact the validity of our argument that GPUs can effectively be used to accelerate distributed system middleware for three reasons. First, in most deployments, CPUs are shared by multiple applications. We demonstrate that CPU-intensive middleware primitives can be effectively offloaded to a GPU to reduce the load of the main processor. Second, to use the SSE computational units the application needs to be transformed into vector processing operations, an operation that complicates the development if done manually and for which compiler support has just begun to emerge.

7. CONCLUSIONS

This study demonstrates the feasibility of harvesting GPU computational power to support distributed systems middleware. We focus on accelerating compute- and data-intensive primitives of distributed storage systems. We implemented StoreGPU, a library which enables distributed storage system designers to offload hashing-based operations to GPUs, demonstrating 8x speedup when comparing StoreGPU performance to a standard CPU implementation. Additionally, we show that applications that depend on hashing computations to effectively identify similarity among large volumes of data, such as comparing two checkpoint images, benefit from the throughput boost enabled by StoreGPU.

Despite the positive outcome of our study, it is important to highlight the challenges involved in the process. As pointed out by our experiments and discussion section, careful optimization of memory access patterns is paramount to achieving such levels of performance. Nevertheless, we expect that, as offloading computations to GPU becomes mainstream, hardware vendors

will offer better support for memory access optimizations in the form of compilers, profilers, etc.

An immediate future exploration is a deeper performance analysis with a broader set of techniques used in distributed storage systems, such as erasure coding and Bloom filters. Furthermore, we plan to fully integrate the library with a distributed storage system prototype to evaluate its impact from an application perspective. The efforts briefly described above and the further exploration of the new features in CUDA1.1 will certainly occupy our minds with exciting investigations in the near future.

8. ACKNOWLEDGMENTS

We thank Sathish Gopalakrishnan and the anonymous reviewers for their insightful comments on earlier versions of this paper.

9. REFERENCES

1. ATI Close To Metal (CTM) Technical Reference, 2008.
2. CUDA 1.1 Beta. <http://developer.nvidia.com>, 2007.
3. Geforce 8 Series, <http://www.nvidia.com/>. 2008.
4. Geforce 9 Series, <http://www.nvidia.com/>. 2008..
5. Jon Peddie Research Report: Nvidia on a roll, grabs more desktop graphics market share in 4Q, http://www.jonpeddie.com/about/press/MarketWatch_Q405.s.html. 2006.
6. Jon Peddie Research Report: Overall GPU market was up an astounding 20% – desktop displaced mobile http://www.jonpeddie.com/about/press/2007/GPU_market_Q307.shtml. 2007.
7. NVIDIA CUDA Compute Unified Device Architecture: Programming Guide v0.8. 2008.
8. Twisted Storage, <http://twistedstorage.sourceforge.net/>. 2008.
9. Al-Kiswany, S., et al. stdchk: A Checkpoint Storage System for Desktop Grid Computing. in ICDCS '08. 2008. Beijing, China.
10. Altschul, S.F., et al., Basic Local Alignment Tool. *Molecular Biology*, 1990. 215: p. 403–410.
11. Bloom, B., Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 1970. 13(7): p. 422-426.
12. Buck, I., et al., Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 2004. 23(3): p. 777-786.
13. Byers, J.W., et al. A Digital Fountain Approach to Reliable Distribution of Bulk Data. in SIGCOM. 1998.
14. Chun, B.-G., et al. Efficient Replica Maintenance for Distributed Storage Systems. in 3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI). 2006. San Jose, CA.
15. Cox, L.P. and B.D. Noble. Samsara: honor among thieves in peer-to-peer storage. in ACM Symposium on Operating Systems Principles. 2003.
16. Dabek, F., et al. Wide-area cooperative storage with CFS. in 18th ACM Symposium on Operating Systems Principles (SOSP '01). 2001. Chateau Lake Louise, Banff, Canada.
17. Dabiri, D. 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.
18. Damgard, I. A Design Principle for Hash Functions. in *Advances in Cryptology - CRYPTO. 1989: Lecture Notes in Computer Science*.
19. DeCandia, G., et al. Dynamo: Amazon's Highly Available Key-value Store. in SOSPO7. 2007.
20. Eshghi, K., et al. JumboStore: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service. in USENIX FAST 2007.
21. Fu, K., M.F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. in USENIX OSDI. 2000.
22. Gilchrist, J. Parallel Compression with BZIP2. in IASTED PDCS, 2004.
23. Govindaraju, N.K., et al. Fast Computation of Database Operations using Graphics Processors. in ACM SIGMOD International Conference on Management of Data. 2004.
24. Hargrove, P.H. and J.C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. in *Scientific Discovery through Advanced Computing Program*. 2006.
25. Huffman, D., A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 1952. 40(9).
26. Karger, D.R., et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. in STOC, 1997.
27. Kotla, R., L. Alvisi, and M. Dahlin. SafeStore: A Durable and Practical Storage System. in USENIX Conference, 2007.
28. Kruger, J. and R. Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. in ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques. 2003.
29. Liu, W., et al. Bio-sequence database scanning on a GPU. in *Parallel and Distributed Processing Symposium, IPDPS. 2006*.
30. Merkle, R. A Certified Digital Signature. in *Advances in Cryptology - CRYPTO. 1989: Lecture Notes in Computer Science*.
31. Moya, V., et al. Shader performance analysis on a modern GPU architecture. in *IEEE/ACM International Symposium on Microarchitecture, MICRO-38. 2005*.
32. Muthitacharoen, A., B. Chen, and D. Mazieres. A Low-bandwidth Network File System. in *Symposium on Operating Systems Principles (SOSP). 2001. Banff, Canada*.
33. Owens, J.D., et al., A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 2007. 26(1): p. 80-113.
34. Quinlan, S. and S. Dorward. Venti: A New Approach to Archival Data Storage. in USENIX FAST 2002.
35. Rowstron, A. and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. in *Middleware'01*.
36. Stoica, I., et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. in *SIGCOMM 2001. 2001. San Diego, USA*.
37. Thompson, C.J., S. Hahn, and M. Oskin. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. in *ACM/IEEE international symposium on Microarchitecture. 2002*.
38. Vilayannur, M., P. Nath, and A. Sivasubramaniam. Providing Tunable Consistency for a Parallel File Store. in *USENIX Conference on File and Storage Technologies. 2005*.
39. Weatherspoon, H. and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. in *International Workshop on Peer-to-Peer Systems IPTPS. 2002*.
40. Yumerefendi, A.R. and J.S. Chase. Strong Accountability for Network Storage. in *FAST'07. 2007*.