Cluster Comput (2009) 12: 123–140
DOI 10.1007/s10586-009-0076-0

On GPU's viability as a middleware accelerator

Samer Al-Kiswany · Abdullah Gharaibeh · Elizeu Santos-Neto · Matei Ripeanu

Received: 1 January 2009 / Accepted: 5 January 2009 / Published online: 17 January 2009
© Springer Science+Business Media, LLC 2009

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 and we design StoreGPU, a library that accelerates a number of hashing-based middleware primitives popular in distributed storage system implementations. Our evaluation shows that StoreGPU enables up twenty five fold performance gains on synthetic benchmarks as well as on a high-level application: the online similarity detection between large data files.

Keywords Middleware · Storage system · Graphics Processing Unit · GPU hashing · StoreGPU

S. Al-Kiswany (✉) · A. Gharaibeh · E. Santos-Neto · M. Ripeanu
Electrical and Computer Engineering Department, The University of British Columbia, Vancouver, BC Canada, V6T 1Z4
e-mail: samera@ece.ubc.ca

A. Gharaibeh
e-mail: abdullah@ece.ubc.ca

E. Santos-Neto
e-mail: elizeus@ece.ubc.ca

M. Ripeanu
e-mail: matei@ece.ubc.ca

1 Introduction

Recent advances in processor technology [1] have resulted in a wide availability of massively parallel Graphics Processing Units (GPUs). Low-end GPUs like NVIDIA's GeForce 8600 priced at about $100 have 32 processors and 256 MB of memory while high-end GPUs, like the NVIDIA 8800 GTX priced at about $300, have up to 128 processors running at 575 MHz and 768 MB of memory, for instance. With these characteristics, GPUs are often underutilized in desktops deployments (as these 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 the 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 [2]. Although these efforts confirmed that significant speedup is achievable, the development cost for this specialized platform was often prohibitive. Recently, however, the introduction of general-purpose programming models (e.g., NVIDIA's CUDA [3]) lowered the development cost making GPUs attractive to a broader spectrum of applications. Additionally, improvements on GPUs architecture created the opportunity to data intensive applications to benefit from GPUs.

This study starts from the observation that a number of techniques that enhance the reliability and/or performance of distributed storage systems (e.g., content addressability in data storage [4, 5], erasure codes [6], on-the-fly data similarity detection [7]) 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 25× 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.*

In particular, this project proposes StoreGPU a library that enables transparent use of GPUs to support specialized uses of hashing for content addressability, on-the-fly similarity detection, data integrity, and load balancing all key techniques to design efficient distributed systems. By making the StoreGPU library 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 argue that this approach can be extended to other routines that support today's distributed systems like erasure coding [8], compact dataset representation using Bloom filters [9], data compression [10], and data filtering. A library that transparently outsources these computationally demanding operations to the GPU will dramatically reduce the CPU load on the hosting machine and enhance overall system performance.

The contribution of this work is fourfold:

- First, this project explores a new territory: the use of GPUs to support acceleration of middleware functionality (as opposed to specialized scientific applications). We show that exploiting GPU in this context brings valuable performance gains. Moreover, we present preliminary 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 study has attempted to use the GPUs to enhance the performance of this category of applications.
- Second, we explore techniques to efficiently use this processing resource. Additionally we provide a memory management subsystem that can be reused across applications and GPU models to efficiently harness the device's shared memory and to reduce the programming effort.
- Third, we present a minimal 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 systems.[1] Depending on its configuration and the target application's data usage patterns, StoreGPU enables significant performance gains. When comparing the performance enabled by a commodity GPU (the NVIDIA 8800 GTX) and one core on a commodity CPU (Intel Core2 Duo 6600), StoreGPU achieves up to 25-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 to focus on optimizing hash-based operations through a survey of hashing use in storage systems (Sect. 2.1) and describes the GPU programming model and the main factors influencing application performance when using GPUs (Sect. 2.2). Section 3 details the design of StoreGPU library. Section 4 presents our experimental results, Sect. 5 surveys the related work, and Sect. 6 presents a discussion of our approach. We conclude in Sect. 7.

2 Background

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

2.1 Use of 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 [4] 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, to support the argument that their computational overheads prevent their use in conventional high-performance systems. Second, to derive the scenarios which inform the design of StoreGPU.

2.1.1 Content addressable storage

In systems that support content addressability [4], data blocks are identified based on their content. In this context, hashing is used as an identification technique: data-block

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

identifiers are simply the hash value of the data [4, 5, 11]. 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 for 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 [7]: 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. Previous studies report that space savings can be as high as 60% in production [4] and research systems [12].

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 [13], Samsara [14], SFS [15], and SafeStore [16], 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, systems based on consistent-hashing [17–19] use a hash value of the data to assign it to nodes [20, 21]. A good hashing function that minimizes collisions leads to an efficient data distribution 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 large 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 blocks' start and end) are defined based on file content. For instance, the Low-Bandwidth File System (LBFS) [7] and JumboStore [22] 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 [7].

2.1.6 Summary of usage scenarios

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

– *Direct Hashing*, where the hash of an entire data block is computed (to support, for example, content addressability, data integrity, or fixed-block similarity detection), and
– *Sliding Window Hashing*, where a large number of hashes of possibly overlapping windows in a large data block are computed (to support detection of content-based block boundaries as outlined in Sect. 2.1.5).

Finally, we note that some of the aforementioned techniques (e.g., data integrity, content addressable storage) 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) [3]. We have selected the NVIDIA cards for two reasons. First, it has the largest market share [23, 24] and second, the CUDA programming model is the most mature GPU programming environment. Recently, other vendors have developed similar programming environments: for example AMD's Stream Computing SDK [25, 26], Apple's OpenCL [27] and RapidMind Development Platform [28], in addition to research based environments such as BrookGPU [29] and Sh [30].

NVIDIA GPUs have a Single Program, Multiple-Data (SPMD) architecture. It offers a number of Single Instruction Multiple Data (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

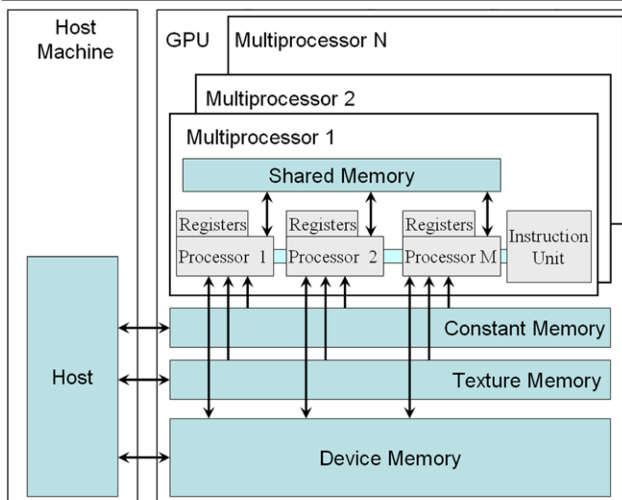

Fig. 1 GPU architecture

that expose the GPU's memory and execution models to the developers, and a runtime library that manages the execution of programs on the GPU. Moreover, CUDA provides an API for GPU-specific functions, such as multi-GPU support, timers, memory management, and per-multiprocessor thread synchronization mechanisms.

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) programming model. Moreover, despite the abstractions provided by CUDA, it is still challenging to make efficient use of GPU resources; for example, as our performance analysis (Sect. 4) shows, suboptimal 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 *shared memory*, a small (16 KB in all NVIDIA's GPUs including the GeForce 8600 GTS and GeForce 8800 GTX) but fast memory, shared by all processors in the multiprocessor. Additionally, all multiprocessors have access to three other GPU-device-level memory modules: the *global* (a.k.a. device memory), *texture*, and *constant* memory modules. 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 1024 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 [3, 31].

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

Programming an application that efficiently exploits the GPU resources 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 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 helps hiding global memory access latency as the threads that have to wait for access to memory are not scheduled on the processors. However, increasing the number of threads 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. Recent studies by Ryoo et al. [32] and by Che et al. [33] has reached this same observation. Obviously, the optimal resource usage configuration is tightly related to the application's characteristics (e.g., the data access patterns) and GPU hardware specifications (the number of registers in the multiprocessor or the size of the 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.)

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

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 memory.
(5) Post-processing	5.1.Post-processing ($T_{PostProc}$)	Application-specific post processing on CPU resource release

data parallel application can be modeled as:

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

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 GPU execution overheads and possibly 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 GeForce 8600 GTS card, and 172.8 GIPS for GeForce 8800 GTX card), the data transfer from/to the two GPUs is limited at 4 GB/s, the theoretical maximum bandwidth of PCIe 16× interface.

To give the reader an intuition of how the various overheads interplay, we present the time breakdown to hash a 96 MB data block: On the GeForce 8600 GTS, transferring the data to the GPU takes 37.4 ms (for an achieved throughput of 2.5 GBps), hashing takes 41.8 ms (using the 32 GT8600 stream processors), and copying the results back takes 1.0 ms. Overall, in this configuration, the memory transfers represent over 48% of the execution time. While on the GeForce 8800 GTX, transferring the data to the GPU takes 37.06 ms (for an achieved throughput of 2.5 GBps), hashing takes 14.7 ms (using the 128 GT8800 stream processors), and copying the results back takes 0.72 ms. Overall, in this configuration, the memory transfers represent over 71% of the execution time.

3 StoreGPU design

The design of StoreGPU is driven by storage systems' use of hashing as presented in Sect. 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 [34, 35]. 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 to hash 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 Sect. 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 (C syntax):

```
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 (Fig. 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 [3].

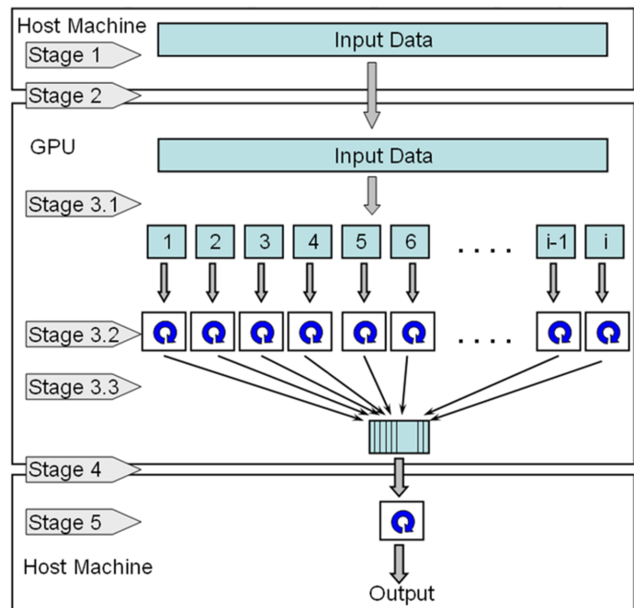

Fig. 2 Direct hashing module architecture. The *blocks* with *circular arrows* represent the standard hashing kernel. Stages numbers correspond to Table 1

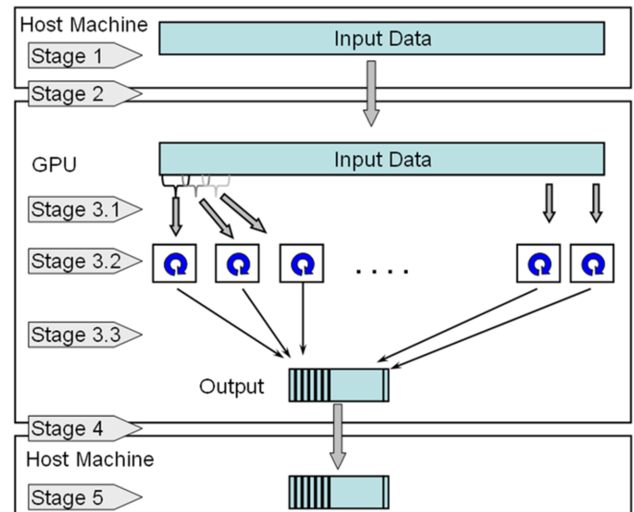

Fig. 3 Sliding window hashing module architecture. The *blocks* with *circular arrows* represent the standard hashing kernel. Stage numbers correspond to Table 1

3.3 Design of 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 Fig. 3.

Each of the hash functions in Fig. 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 is 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 100 K 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 memory management sub-system for shared memory. This sub-system has two main goals. First, to reduce memory access latency, the memory management sub-system allocates a memory workspace for each thread. This workspace has a fixed-size and is located in the shared memory. Additionally, also to lower the access latency, the memory management sub-system allocates workspaces for different threads on a separate shared memory banks, effectively avoiding bank conflict problem. 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, while allocating the thread's workspaces as described offers better access latency, it complicates application programming since a shared memory bank does not represent a contiguous memory address space. To avoid complicating the programming task, 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 increases the shared memory performance by avoiding bank conflicts while reducing the programming effort by providing a contiguous memory address abstraction.

3.5 Other optimizations

In addition to optimizing the shared memory usage, we consider 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 between the host memory and 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 pinned buffer. However, allocating pinned memory adds some overhead since the kernel is involved in finding and preparing a contiguous set of memory pages before locking it. Our performance numbers do not show a pronounced effect for this overhead. Moreover, if the overhead of allocating pinned memory buffers becomes an issue, allocated buffers can be reused by subsequent library calls and thus this overhead can be 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 (Sect. 4.1) and an application driven benchmark: similarity detection between multiple versions of the same file (Sect. 4.2).

4.1 Synthetic benchmarks

This section presents the performance and speedup delivered by StoreGPU under a synthetic workload: it first compares GPU-supported performance with the performance for 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 usage scenarios of hashing described in Sect. 3 (i.e., Direct Hashing and Sliding Window Hashing).

Table 2 summarizes the factors considered in the performance evaluation. 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 overhead between the host and the GPU. Additionally the sliding-window hashing technique introduces two specific parameters: the window and offset sizes.

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

Our experiments follow a factorial experimental design and evaluate the impact of each combination of factors presented in Table 2. For all performance data points, we report

Table 2 The list of 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	4 KB to 96 MB
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 speedup computed from the average execution time collected from 40 experiments. We confirmed that this number of experiments is sufficient to guarantee an average speedup estimate with confidence level of 95%. The following sections present a summary of these experiments.

The devices used in the performance analysis are: an Intel Core2 Duo 6600 2.40 GHz processor (released late 2006), an Intel Core2 Quad Q6700 2.66 GHz processor (released mid 2007), an NVIDIA GeForce 8600 GTS GPU (released mid 2007) and an NVIDIA GeForce 8800 GTX (released late 2006). The GeForce 8600 GTS GPU was installed on a machine with Intel Core2 Duo 6600 2.40 GHz processor, running WindowsXP, and CUDA 1.0 driver and runtime library. While the GeForce 8800 GTX was installed on a machine with Intel Core2 Duo E6850 3 GHz processor, running Linux 2.6.24, and CUDA 2.0 driver and runtime library.

We note that, in all cases, our implementation uses out-of-the-box hash function implementations. These original implementations are single-threaded and use only one core of the Intel processor; when we use them on multi-core architecture, we execute in parallel multiple instances of the original hash function implementation.

For this reason, in the rest of this section, we use the performance the original single-threaded implementation running on a single core on the Intel Core2 Duo 6600 2.40 GHz processor as the baseline to compute and compare the speedups achieved by StoreGPU configurations. To offer a more comprehensive perspective on the achieved performance we also offer a speedup evaluation when using multiple traditional cores.

Due to space limitations, we do not report performance on all the platforms we use for all experiments. Figure 4, however, offers a first indication that StoreGPU provides better performance when compared to an optimized multi-threaded CPU implementation harnessing all cores of a traditional (e.g., Intel) multicore architecture and optimized for maximum parallelism. Although for small data sizes, the

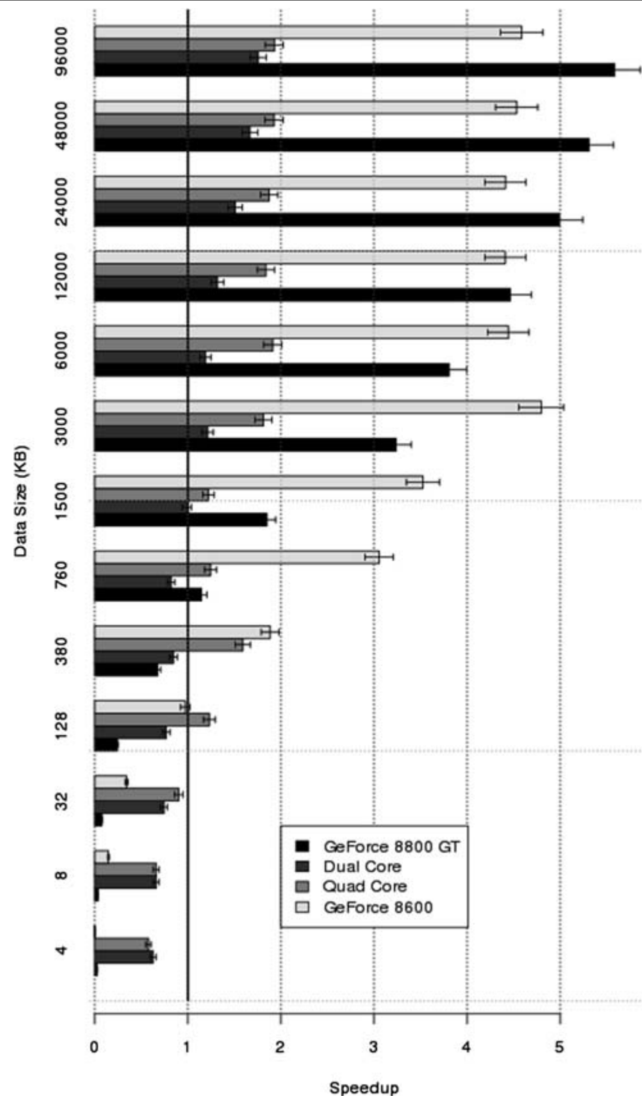

Fig. 4 Speedups for MD5 direct hashing module for fully optimized GPU implementations running on GeForce 8600 and 8800, and multithreaded CPU implementations harnessing all available cores. The value $x = 1$ separates the speedup (*right*) from the slowdown values (*left*)

multithreaded CPU implementation has better performance than StoreGPU, as the data size increase (i.e., data larger than 1 MB), StoreGPU achieves much higher speedups. It is worth noting that, even though not presented here, the sliding window module and SHA1 algorithm have a similar behavior.

This comparison highlights that for a comparable price (or even lower), a GPU offers much higher performance than a high end CPU. A more comprehensive discussion on the impact of the experiment platform choices is presented in Sect. 6.

We note that the lower-end GPU (GeForce 8600) achieved a better performance for smaller data sizes than the high-end GPU model (GeForce 8800). This is due to two main rea-

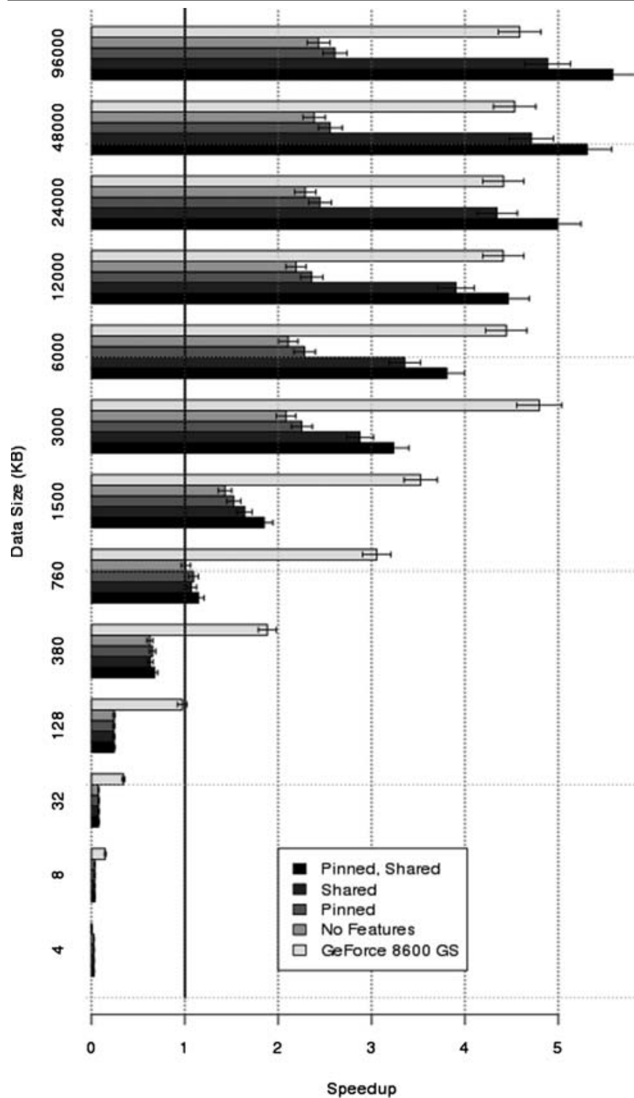

Fig. 5 StoreGPU speedup for the MD5 implementation of the direct hashing module

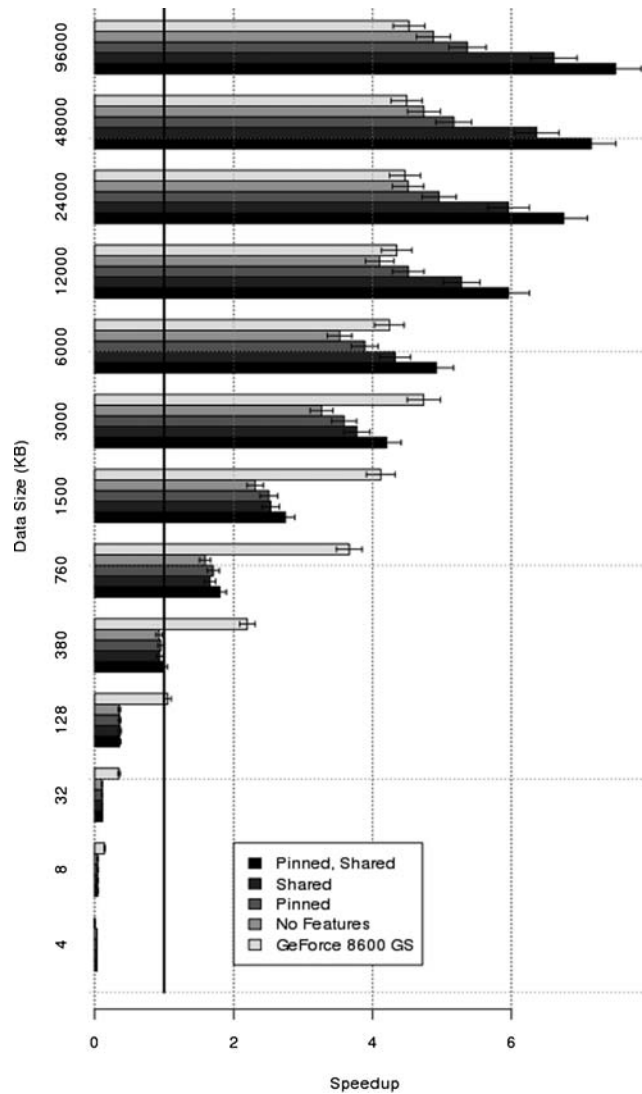

Fig. 6 StoreGPU speedup for the SHA1 implementation of the direct hashing module

sons; first, a careful examination of the results break down reveals that CUDA 2.0 run time library takes considerably longer time to allocate pinned memory buffers (around 14× slower) than CUDA 1.0. Second, the GeForce 8600 GPU cores (a.k.a. shaders) are clocked at 1450 MHz while the newer GeForce 8800 GTX GPU shaders are clocked at 1350 MHz; consequently, for small data sizes that do not utilize the 16 multiprocessors of the GeForce 8800, the GeForce 8600 will complete the computation faster.

4.1.2 Experimental results

The first question addressed by our experiments is: *What is the speedup offered by StoreGPU compared to the original single-threaded 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 5 and Fig. 6 show the speedups 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* (this is indicated by a line at $x = 1$). The results show that the fully optimized (i.e., pinned and shared memory optimizations enabled) StoreGPU starts to offer speedups for blocks larger than 700 KB and offer up to 6× speedup for large data blocks.

Note that as the data size increases, the performance improvement reaches a saturation point. Further, optimizing for shared memory accesses has the biggest impact on the achieved speedup. This highlights the fact that efficient memory management is paramount to gain maximum performance when considering data-intensive applications. It

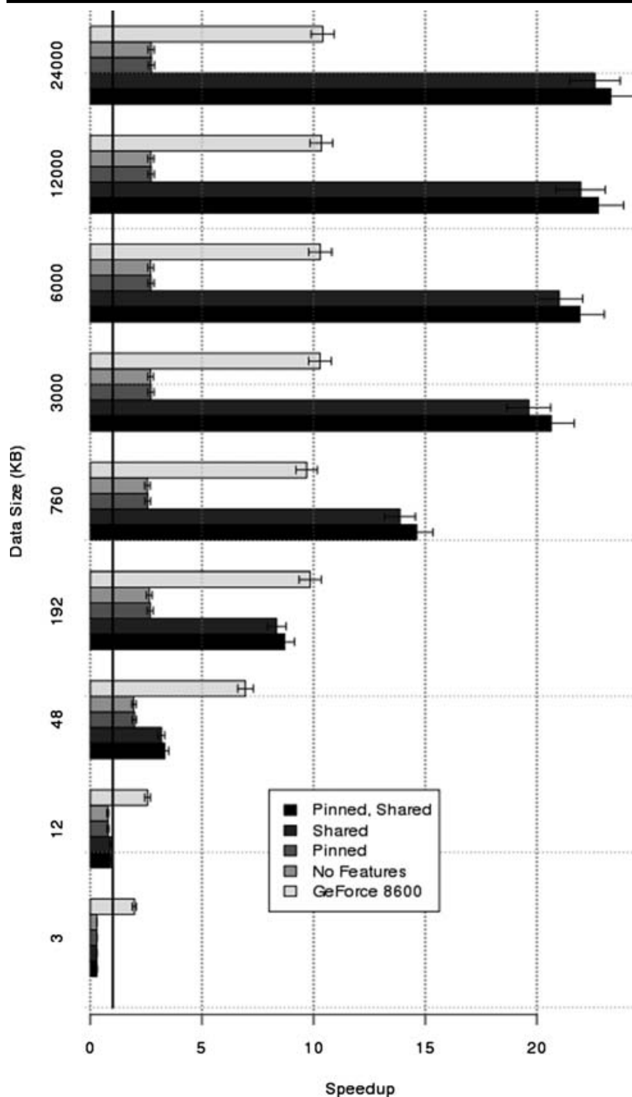

Fig. 7 StoreGPU sliding-window hashing module speedup for MD5. Window = 20 bytes, offset = 4 bytes, reduced hash

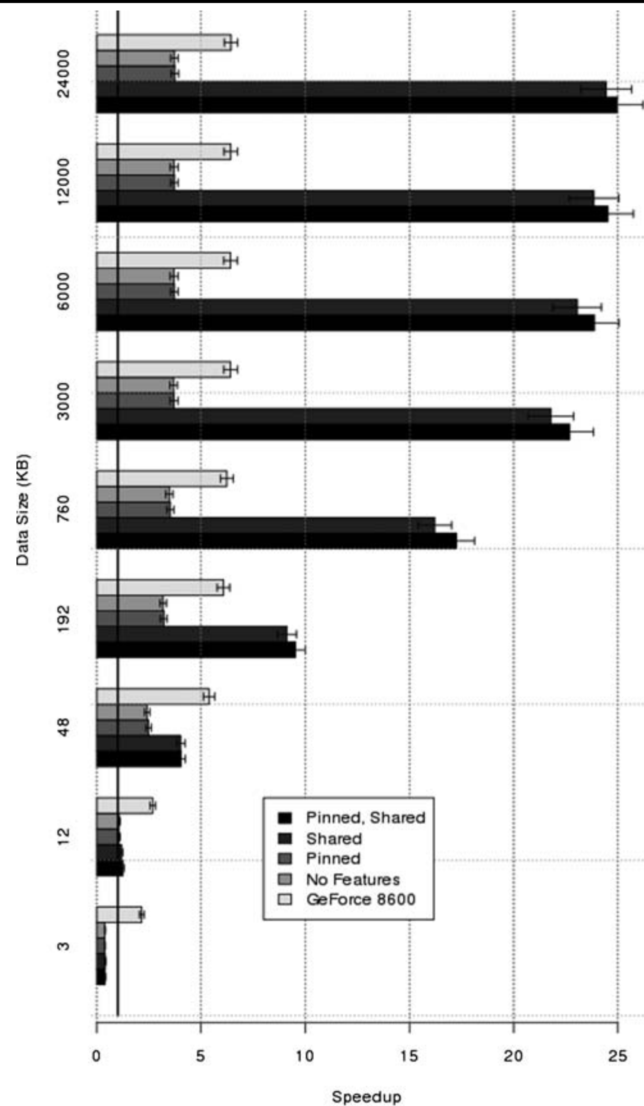

Fig. 8 StoreGPU sliding window hashing module speedup for SHA1. Window = 20 bytes, offset = 4 bytes, reduced hash

is also important to observe that for small blocks the GPU performs much worse than its CPU counterpart (e.g., when memory accesses are not optimized, the performance can decrease up to 37× slow down for 4 KB and MD5). This is mainly due to the overhead of host to device data transfers compared to the processing cost. We discuss the latter point in more detail in the next section.

Figure 7 to Fig. 10 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 significantly higher (up to 25× speedup).

The 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 [7], Jumbostore [22], and stdchk [12]. Due to space constraints, we present the results of window size of 20 bytes and offset of 4 bytes, and window size of 52 bytes and offset of 52 bytes. Although not reported here the other combinations of window sizes and offset present the same characteristics.

Figure 7 and Fig. 8 show 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 stdchk), StoreGPU hashes the input data approximately 20× faster for MD5 and up to 25× faster for SHA1. Figure 9 and Fig. 10 present the results for larger chunks (52 bytes) and offset (52 bytes). It is interesting to note that StoreGPU performs a little slower when

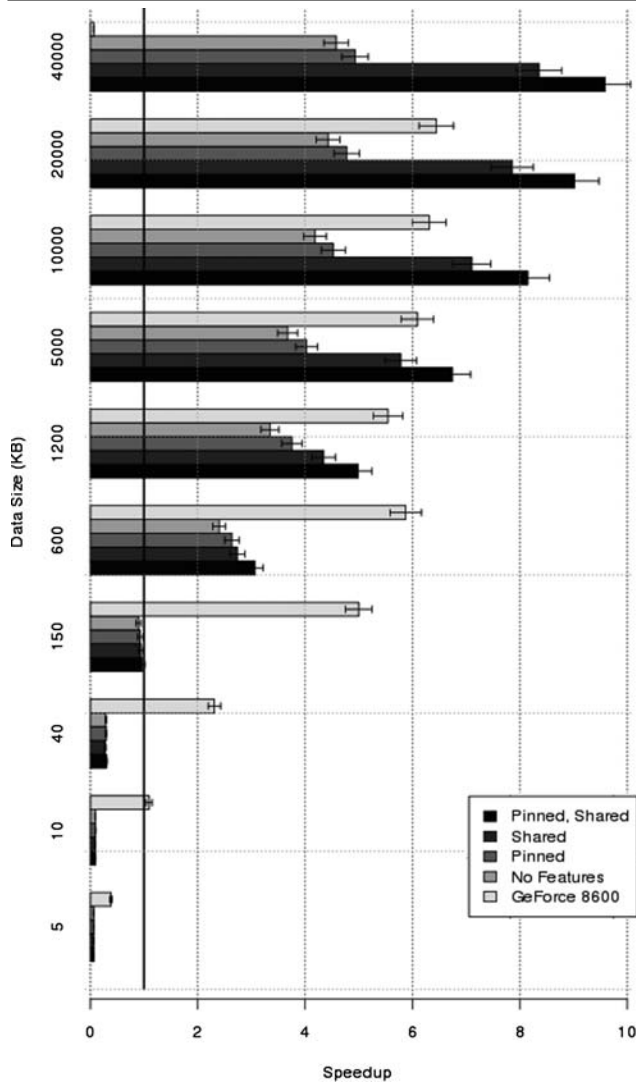

Fig. 9 StoreGPU sliding-window hashing module speedup for MD5. Window size = 52 bytes. Offset = 52 bytes

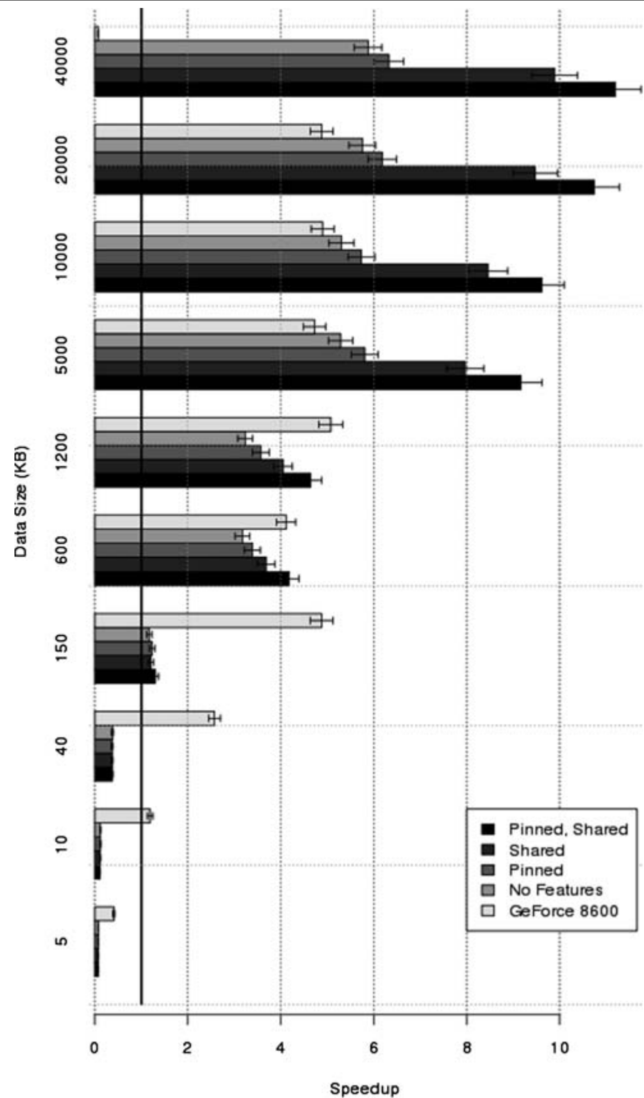

Fig. 10 StoreGPU sliding-window hashing module speedup for SHA1. Window size = 52 bytes. Offset = 52 bytes

compared to the previous, more computing intensive, scenario.

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 minimum 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 due to its intrinsic parallelism. This is also the reason we observe larger speedups with smaller window sizes and offsets.

Finally, we observed that the speedup achieved for SHA1 is better than MD5. We attribute that to SHA1 being more computationally demanding than MD5 algorithm and hence fitting better GPU's application domain.

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 Sect. 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: use of pinned memory, optimized use of

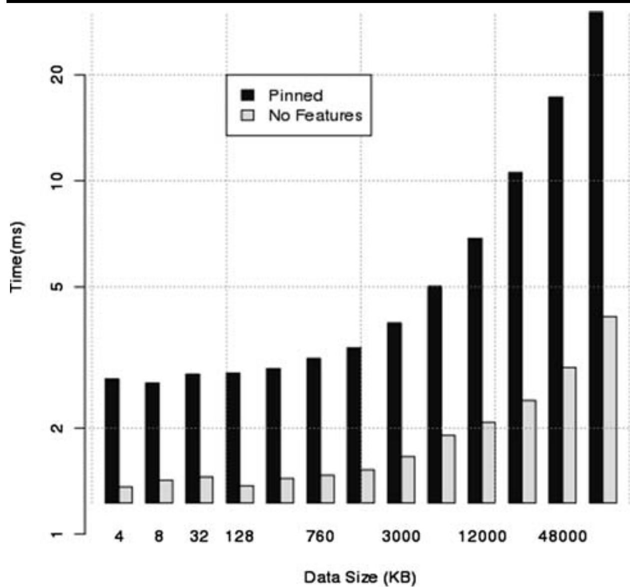

Fig. 11 Stage 1 (preprocessing and initialization) execution time for MD5 direct hashing module of StoreGPU

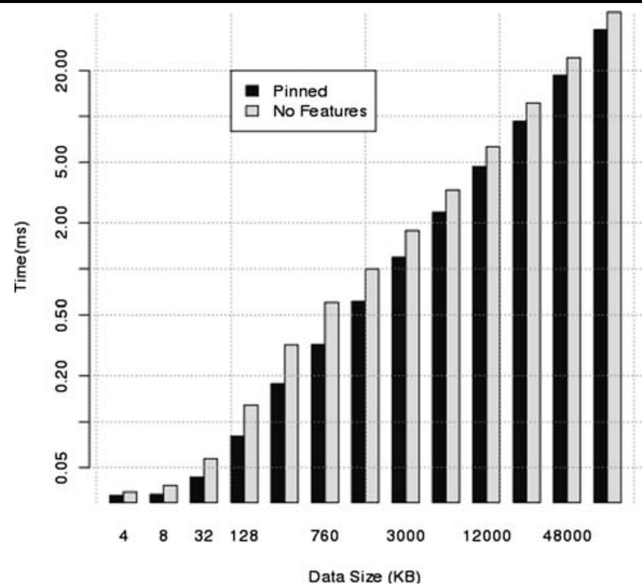

Fig. 12 Stage 2 (input data transfer) execution time of MD5 direct hashing module with and without using pinned memory feature

shared memory, and reduced output 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. Therefore, this stage consists of memory allocation and GPU initialization 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 3 ms and 30 ms depending on the data size and whether the pinned memory optimization is enabled (Fig. 11). The initialization takes longer with pinned memory and larger data sizes as it is costly to find contiguous pages to accommodate larger data sizes.

At a first glance, the proportional overhead due to the initialization time may seem significant for the overall StoreGPU performance (*Stage 1* in Fig. 16 and Fig. 17). However, the pinned memory allocation impact can be reduced by reusing buffers once allocated.

Stage 2: Data Transfer In. The host-to-device transfer time varies depending on the data size and on whether Pinned Memory optimization is used. As expected, although using pinned memory slows down Stage 1, it significantly improves transfer performance (Fig. 12). Compared to the theoretical 4 GBps peak throughput of the PCIe 16× bus, we obtain, for large blocks, 2.5 GB/s with pinning and 1.7 GB/s without.

Stage 3: Data Processing. The performance of kernel execution is highly dependent on the utilization of shared

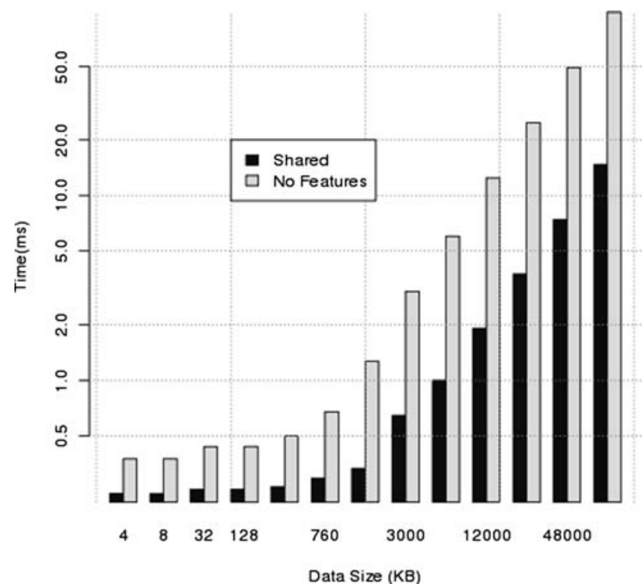

Fig. 13 Kernel execution time for MD5 direct hashing module with/without shared memory optimization enabled

GPU memory and its optimized use (i.e., avoiding bank conflicts—Fig. 13). For large data volumes, without the optimized memory management, the kernel contributes approximately up to 65% to the overall operation time (Fig. 16). When all optimizations are enabled, efficient use of shared memory reduces the kernel execution impact to about 20% of the total execution time (Fig. 16 and Fig. 17).

Stage 4: Data Transfer Out. Transferring the output (Fig. 14) causes proportionally less impact on the overall execution than transferring the input (Fig. 16 to Fig. 17). The

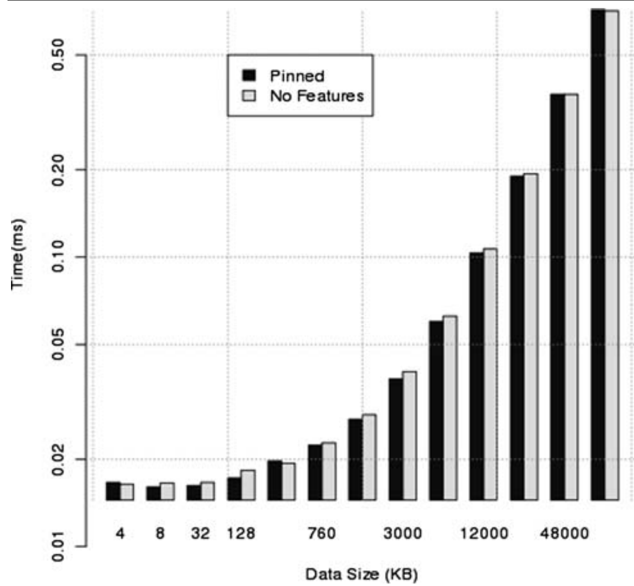

Fig. 14 Execution times for MD5 direct hashing module data transfer operation from the GPU global memory to the host memory with and without using pinned memory feature

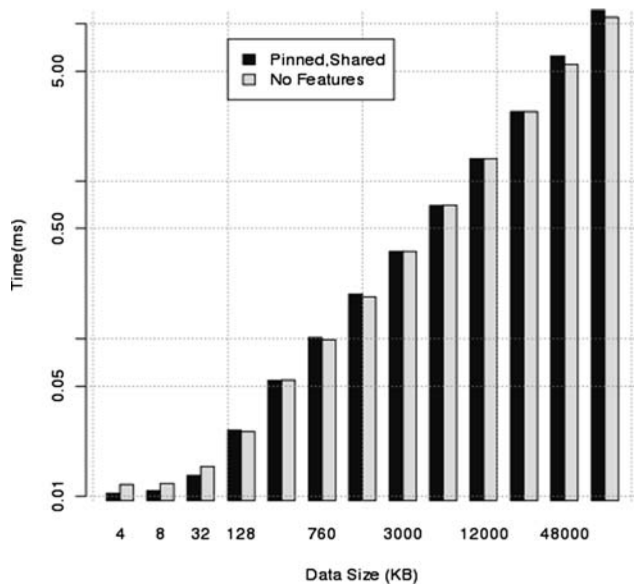

Fig. 15 Execution times for the last stage—the hash aggregation

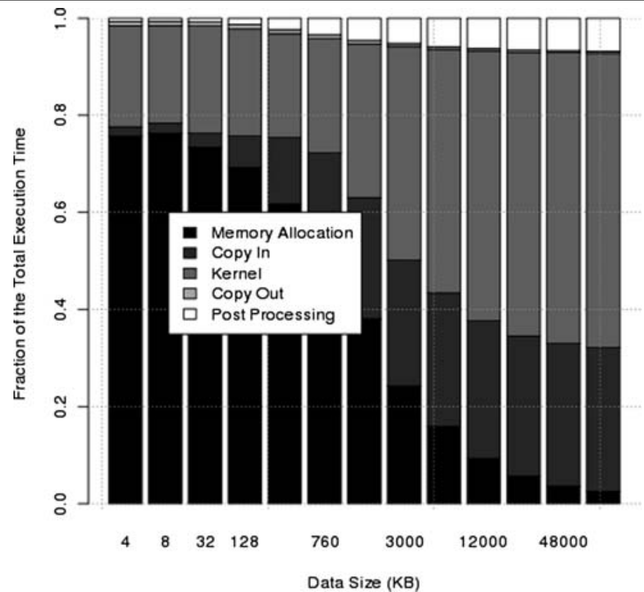

Fig. 16 Percentage of total execution time spent on each stage when none of the optimizations are enabled

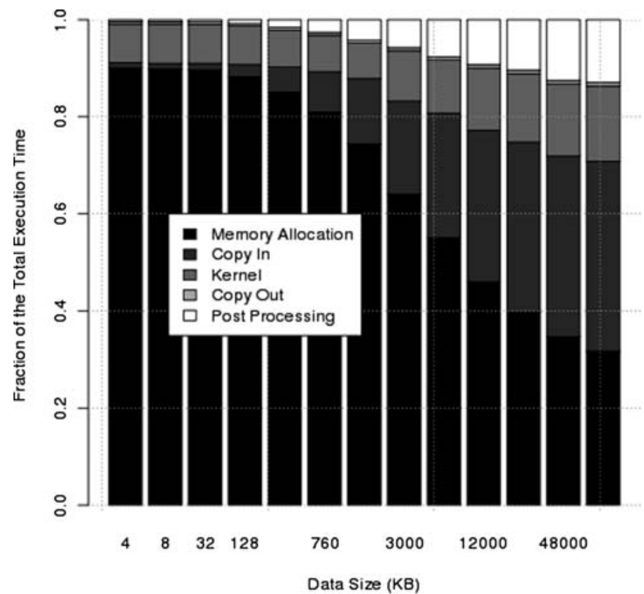

Fig. 17 Percentage of total execution time spent on each stage with pinned and shared memory optimizations enabled

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 (Fig. 15). Enabling GPU optimizations do not influence the performance of the last stage (hash aggregation), since the execution is performed on the CPU.

Figure 16 and Fig. 17 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 Fig. 16 and Fig. 17), while using the shared memory reduces kernel execution impact

(compare Stage 3 in Fig. 16 and Fig. 17). 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 Fig. 17).

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 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. [12] detect up to 82% similarity).

We have collected the checkpoint images using BLCR checkpointing library [36] from 24 hour-long runs of BLAST, a popular bioinformatics application [37]. The interval between the checkpoints is 5 min. The average image size is 279 MB.

Table 3 and Table 4 compare the throughput of online similarity detection between using standard hashing functions running on a single core of Intel Core2 Duo 6600 2.40 GHz processor and using StoreGPU running on NVIDIA 8800 GTX. The fixed block size similarity detection uses a block size of 20 MB, and the variable block size similarity detection uses window size of 20 bytes and an offset of 4 bytes. These results show dramatic improvement in the throughput of online similarity detection with both fixed and variable size blocks. The results indicate that fixed-block similarity detection can be used even on 10 Gbps systems while the variable block size technique can be used for systems connected with 1 Gbps links without introducing a performance bottleneck. Note that this experiment considers the similarity detection mechanism at the application level only, without integrating with a file system.

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

	Throughput (MBps)		Similarity
	StoreGPU	Standard	ratio detected
Fixed block size (using direct hashing)	1015.9	155.76	23%
	Speedup: 6.5×		
Variable block size (LBFS technique using sliding window hashing)	194.24	8.06	82.0%
	Speedup: 24.1×		

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

	Throughput (MBps)		Similarity
	StoreGPU	Standard	ratio detected
Fixed block size (using direct hashing)	1101.6	234.17	23%
	Speedup: 4.7×		
Variable block size (LBFS technique using sliding window hashing)	255.21	11.05	80%
	Speedup: 23.1×		

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. [2] for a comprehensive survey.

A number of science-oriented applications stand out. Liu et al. [38] implement 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 report a 16× 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. [39] compare 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. Kruger et al. [40] implement linear algebra operators for GPUs and demonstrate the feasibility of offloading a number of matrix and vector operations to GPUs.

More related to our infrastructural focus, Govindaraju et al. [41] implement a number of database operations for GPUs, including conjecture selection, aggregation, and semi-linear query operations. Curry et. al [42] 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 [43], used in RAID 6 systems, to facilitate generating more than two parity blocks without affecting system overall performance. Along the same lines Falcao et al. [44] show that GPUs can be used to accelerate Low Density Parity Checks (LDPC) error correcting codes. Their LDPC GPU implementation achieves up to 700× speedup compared to CPU implementation, a performance that enables using LDPC codes on commodity systems without special decoding hardware.

Harrison and Waldron [45] studied the feasibility of using the recent GPUs as a cryptographic accelerator. To this end they implemented the AES encryption algorithm using OpenGL instead of CUDA. Although their implementation did not outperform the CPU implementation, they show that the AES cryptography can successfully be offloaded from the CPU to the GPU. Their moderate performance gains may be attributed to using OpenGL and rendering the AES encryption process to match the graphical pipeline processing which does not leave space for GPU side optimization. In a latter attempt Harrison and Waldron [46] implemented AES using CUDA and reported 4× speedup. Manavski [47] also implement AES encryption using CUDA and report up to 20× speedups. Finally, Moss et. al [48] 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 3× speedup. Moreover, recently exploiting GPUs to accelerate security operations was adopted in few security products, including Kaspersky antivirus [49] and Elcomsoft password recovery software [50].

Our study is different from the above studies in four ways. First, we employ the latest GPU generation and the CUDA programming model which are more suitable for general purpose programming. Unlike most of the previous studies, this relieves us from having to retrofit the problem we solve into a graphics problem. Second, unlike the previous studies, we focus on accelerating primitives that support a broad range of storage system techniques and place them in a library, thus providing infrastructure for a large set of applications. Third, we provide a memory management subsystem that can be reused across application and GPU models to efficiently harness the device's shared memory and reduce the programming effort. Finally, 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.

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 [35]. 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, as it is 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 v2.0)?*

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 16× interface) [51] 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 v2.0 which supports, among other features, overlapping computation and data transfers (through the *streaming API*). A stream (not confused with stream processing) is a sequence of operations that execute in a strictly serial order. Different streams, however, may execute operations out of order, with respect to one another or concurrently. This creates an opportunity for concurrent memory transfers and computations on the GPU, thus reducing the data transfer overhead between the host and the GPU [3]. For example, stream *S1* can be defined to hash block *B1* by the following operations: (i) copy block *B1* to the GPU, (ii) execute the hash kernel on *B1*, (iii) copy the hash result out. Since these operations belong to the same stream *S1*, they are executed sequentially. However, if another stream *S2* is defined with the same operations but for block *B2,* the operations of the two streams can execute concurrently. For instance, the execution of the hash kernel on *B1* can overlap with the copy of block *B2* to the GPU, effectively hiding the data transfer overhead of bock *B2*.We conjecture that by exploiting these features, we will be able to provide additional speedups.

3. *Can other middleware primitives 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 [52] and can be deployed on GPUs; on the other hand, Gilchrist [53] proposes a parallel implementation of the bzip2 loss-less data compression algorithm that may benefit from GPU support, and Nightingale et al. [54] design Speck a framework aimed at parallelizing different types of security checks. 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 host-to-device memory transfer overheads.

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 specification [55] is as a high 32 GB/s for GeForce 8600 GTS GPU and 86.4 GB/s for GeForce 8800 GTX GPU, the real memory access bottleneck is the PCIe bus, listed at 4 GB/s in each direction. This is congruent to our experiments, which show that pinned memory transfers achieve up to 2.48 GB/s. Furthermore, we estimate NVIDIA's GeForce 8600 theoretical non floating point instruction exaction 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.

As a mater of fact, recently, Bakhoda et al. [56] built a detailed micro-architecture performance simulator for NVIDIA cards and evaluated the performance of a dozen of non graphical application on the GPU including our StoreGPU code. The study concludes with insightful guidelines for optimizing general purpose programming on GPUs. Based on micro-architecture level information, Bakhoda et al. [56] estimate that StoreGPU uses the shared memory efficiently: over 90% of the memory accesses are to shared memory and less than 10% to global memory. Moreover the study estimates that StoreGPU is able to maximally use all the running threads and avoid stalls.

5. *Is the comparison fair?*

We have used four devices: Intel Core2 Duo 6600 and Intel Core2 Quad Q6700 2.66 GHz processors, and NVIDIA GeForce 8600 GTS and GeForce 8800 GTX GPUs, for our comparison. In all cases, we used the unmodified hashing functions (with best compiler options). On the CPU-side, an additional optimization may also be considered: using Intel's Streaming SIMD Extensions (SSE).

We believe that not exploring this optimization does not impact the validity of our argument that GPUs can effectively be used to accelerate distributed system middleware. Moreover, 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 speedups as high as 25× 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 a mainstream feature, 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 CUDA 2.0 will certainly occupy our minds with exciting investigations in the near future.

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

References

1. Moya, V., Gonzalez, C., Roca, J., Fernandez, A., et al.: Shader performance analysis on a modern GPU architecture. In: IEEE/ACM International Symposium on Microarchitecture, MICRO-38, 2005

2. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., et al.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007). doi:10.1111/j.1467-8659.2007.01012.x
3. NVIDIA CUDA Compute Unified Device Architecture: Programming Guide v2.0 (2008)
4. Quinlan, S., Dorward, S.: Venti: a new approach to archival data storage. In: FAST, Monterey, CA, 2002
5. Twisted Storage. http://twistedstorage.sourceforge.net/ (2008)
6. Weatherspoon, H., Kubiatowicz, J.: Erasure coding vs. replication: a quantitative comparison. In: IPTPS, 2002
7. Muthitacharoen, A., Chen, B., Mazieres, D.: A low-bandwidth network file system, In: SOSP, 2001
8. Chun, B.-G., Dabek, F., Haeberlen, A., Sit, E., et al.: Efficient replica maintenance for distributed storage systems. In: NSDI, San Jose, CA, (2006)
9. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970). doi:10.1145/362686.362692
10. Huffman, D.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952). doi:10.1109/JRPROC.1952.273898
11. Vilayannur, M., Nath, P., Sivasubramaniam, A.: Providing tunable consistency for a parallel file store. In: USENIX Conference on File and Storage Technologies, 2005
12. Al-Kiswany, S., Ripeanu, M., Vazhkudai, S., Gharaibeh, A.: STDCHK: a checkpoint storage system for desktop grid computing. In: ICDCS, Beijing, China, 2008
13. Yumerefendi, A.R., Chase, J.S.: Strong accountability for network storage. In: FAST'07, 2007
14. Cox, L.P., Noble, B.D.: Samsara: honor among thieves in peer-to-peer storage. In: ACM Symposium on Operating Systems Principles, 2003
15. Fu, K., Kaashoek, M.F., Mazières, D.: Fast and secure distributed read-only file system. In: OSDI, 2000
16. Kotla, R., Alvisi, L., Dahlin, M.: SafeStore: a durable and practical storage system. In: USENIX Annual Technical Conference, 2007
17. Karger, D.R., Lehman, E., Leighton, F.T., Panigrahy, R., et al.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: Symposium on Theory of Computing, 1997. ACM, New York (1997)
18. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., et al.: Chord: a scalable peer-to-peer lookup service for Internet applications. In: SIGCOMM 2001, San Diego, USA, 2001
19. Rowstron, A., Druschel, P.: Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, 2001
20. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., et al.: Dynamo: Amazon's highly available key-value store. In: SOSP07, 2007
21. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., et al.: Wide-area cooperative storage with CFS. In: SOSP, 2001
22. Eshghi, K., Lillibridge, M., Wilcock, L., Belrose, G., et al.: JumboStore: providing efficient incremental upload and versioning for a utility rendering service. In: FAST, 2007
23. Jon Peddie Research Report: NVIDIA on a roll, grabs more desktop graphics market share in 4Q. http://www.jonpeddie.com/about/press/MarketWatch_Q405.shtml (2006)
24. 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)
25. AMD Stream Computing SDK. Available from: http://ati.amd.com/technology/streamcomputing/ (2008)
26. ATI Close To Metal (CTM) Technical Reference Version 1.01 Manual (2008)
27. Open, C.L.: Available from: http://www.khronos.org/opencl/ (2008)
28. RapidMind Development Platform. Available from: http://www.rapidmind.net/ (2008)
29. Buck, I., Foley, T., Horn, D., Sugerman, J., et al.: Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph. **23**(3), 777–786 (2004). doi:10.1145/1015706.1015800
30. McCool, M., Toit, S.D.: Metaprogramming GPUs with Sh. AK Peters, Wellesley (2004)
31. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA tesla: a unified graphics and computing architecture. In: IEEE Micro, pp. 39–55, 2008
32. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., 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
33. Che, S., Boyer, M., Meng, J., Tarjan, D. et al.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. **68**(10), (2008). doi:10.1016/j.jpdc.2008.05.014
34. Merkle, R.: A certified digital signature. In: Advances in Cryptology—CRYPTO. Lecture Notes in Computer Science. Springer, Berlin (1989)
35. Damgard, I.: A design principle for hash functions. In: Advances in Cryptology—CRYPTO. Lecture Notes in Computer Science. Springer, Berlin (1989)
36. Hargrove, P.H., Duell, J.C.: Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. In: Scientific Discovery through Advanced Computing Program (SciDAC), 2006
37. Altschul, S.F., Gish, W., Miller, W., Myers, E., et al.: Basic local alignment tool. Mol. Biol. **215**, 403–410 (1990)
38. Liu, W., Schmidt, B., Voss, G., Schroder, A., et al.: Bio-sequence database scanning on a GPU. In: IPDPS, 2006
39. Thompson, C.J., Hahn, S., Oskin, M.: Using modern graphics architectures for general-purpose computing: a framework and analysis. In: ACM/IEEE International Symposium on Microarchitecture, 2002
40. Kruger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. In: ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, 2003
41. Govindaraju, N.K., Lloyd, B., Wang, W., Manocha, M.L.: Fast computation of database operations using graphics processors. In: ACM SIGMOD International Conference on Management of Data, 2004
42. Curry, M.L., Skjellum, A., Ward, H.L., Brightwell, R.: Accelerating Reed–Solomon coding in RAID systems with GPUs. In: IPDPS, 2008
43. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. J. Soc. Ind. Appl. Math. **8**(2), 300–304 (1960). doi:10.1137/0108018
44. Falcao, G., Sousa, L., Silva, V.: Massive parallel LDPC decoding on GPU. In: ACM SIGPLAN Symposium on Principles and practice of Parallel Programming (PPoPP), Salt Lake City, 2008
45. Harrison, O., Waldron, J.: AES encryption implementation and analysis on commodity graphics processing units, In: Workshop on Cryptographic Hardware and Embedded Systems (CHES), Vienna, Austria, 2007
46. Harrison, O., Waldron, J.: Practical symmetric key cryptography on modern graphics hardware. In: USENIX Security Symposium, San Jose, CA, 2008
47. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: IEEE International Conference on Signal Processing and Communications (ICSPC), Dubai, United Arab Emirates, 2007
48. Moss, A., Page, D., Smart, N.: Toward acceleration of RSA using 3D graphics hardware. In: Cryptography and Coding, 2007

49. Kaspersky Antivirus. Available from: http://www.kaspersky.com/ (2008)
50. Elcomsoft password recovery software. Available from: http://www.elcomsoft.com (2008)
51. Geforce 9 Series. http://www.nvidia.com/object/geforce9.html (2008)
52. Dabiri, D., Blake, I.F.: Fast parallel algorithms for decoding Reed–Solomon codes based on remainder polynomials. IEEE Trans. Inf. Theory **41**(4), 873–885 (1995). doi:10.1109/18.391235
53. Gilchrist, J.: Parallel compression with BZIP2. In: IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), 2004
54. Nightingale, E.B., Peek, D., Chen, P.M., Flinn, J.: Parallelizing security checks on commodity hardware. In: ASPLOS, Seattle, WA, 2008
55. Geforce 8 Series. http://www.nvidia.com/page/geforce8.html (2008)
56. Bakhoda, A., Yuan, G., Fung, W.W.L., Wong, H., et al.: Performance analysis of GPU compute workloads via detailed simulation. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, 2009

Samer Al-Kiswany is a Ph.D. student in the Electrical and Computer Engineering Department at the University of British Columbia (UBC). He received his B.Sc. degree from Jordan University of Science and Technology, Jordan in 2003, and his M.Sc. degree from UBC in 2007. Samer research interests are in distributed systems with special focus on high performance computing systems.

Abdullah Gharaibeh is a M.A.Sc. student in the Electrical and Computer Engineering Department at the University of British Columbia (UBC). Abdullah's research interests are focused on the design and evaluation of high performance distributed computing systems.

Elizeu Santos-Neto is a Ph.D. student in the Electrical and Computer Engineering Department at the University of British Columbia (UBC). Before joining UBC, he received a B.Sc. and a M.Sc. from Universidade Federal de Alagoas and Universidade Federal de Campina Grande, respectively. Currently, Elizeu's interests are in the characterization and design of large scale distributed systems. In particular, his research focuses on peer-to-peer content sharing systems and online social networks.

Matei Ripeanu received his Ph.D. degree in Computer Science from The University of Chicago in 2005. After a brief visiting period with Argonne National Laboratory, Matei joined the Electrical and Computer Engineering Department of the University of British Columbia as an Assistant Professor. Matei is broadly interested in distributed systems with a focus on self-organization and decentralized control in large-scale Grid and peer-to-peer systems.