# BAG: Managing GPU as Buffer Cache in Operating Systems

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

**Abstract**—This paper presents the design, implementation and evaluation of BAG, a system that manages GPU as the buffer cache in operating systems. Unlike previous uses of GPUs, which have focused on the computational capabilities of GPUs, BAG is designed to explore a new dimension in managing GPUs in heterogeneous systems where the GPU memory is an exploitable but always ignored resource. With the carefully designed data structures and algorithms, such as concurrent hashtable, log-structured data store for the management of GPU memory, and highly-parallel GPU kernels for garbage collection, BAG achieves good performance under various workloads. In addition, leveraging the existing abstraction of the operating system not only makes the implementation of BAG non-intrusive, but also facilitates the system deployment.

**Index Terms**—Graphics Processing Unit (GPU), CUDA, Buffer Cache, Operating Systems.

## 1 Introduction

Despite the considerable endeavors, the operating systems support for GPU resource management in commodity software is still highly limited. Little work has been conducted to promote research in this area. In fact, recent operating system (OS) designs like Helios [14] and Barrelfish [5] have investigated the importance of heterogeneity in contemporary computer systems. However, both targeted heterogeneous CPU-cores in traditional OS contexts featuring preemptive scheduling, interrupts, and direct IO access, which are all absent in current GPU computing platforms. Different from CPUs, GPUs are managed by the OS as peripheral devices rather than as shared computing resources, which severely limits the applicability of GPU to certain application domains. Lack of support for OS interfaces, such as GPU scheduler and GPU resource management that are all encapsulated in vendor-specific kernel drivers and runtime systems, rules out many potential usage scenarios of GPUs.

In response to the practical issues of GPUs, a recent work [15] proposed new OS abstractions called PTask API to support a dataflow programming model in the OS. By promoting GPUs to first-class citizens, PTask provides the OS kernel sufficient visibility and control over GPUs to achieve fairness and performance isolation. Another work [7] proposed a system Pegasus to address similar problems as with PTask, but in virtualized environments. We strongly agree with the proposals attempting to better utilize heterogeneous GPU computing resource in the OS. However, in this work, we explore a new dimension in managing GPU resources of heterogeneous systems where the GPU memory is a type of exploitable but always ignored resource. In contrast to current research activities focusing on the computational power of GPUs, we argue that the GPU memory is also a valuable resource in heterogeneous systems and can be leveraged to extend the functionality of the OS. Specifically, we have designed and implemented a system called BAG (Buffer cAche on GPU) to demonstrate the feasibility and performance gains of a GPU-enhanced buffer cache in the OS.

The main challenge in designing BAG is to avoid introducing extra complexities into the existing OS kernel with a good abstraction for GPUs and to retain high performance at the same time. To this end, we built BAG as a virtual block device, which is the interface provided by most operating systems to abstract away the complexity of interaction across different storage components. The proposed abstraction for GPUs guarantees maximal reuse of the storage stack in the OS and eliminate redundant hard-coded logic. We have implemented *indirector*, one component of BAG to transparently intercept file-system IO requests, which are then redirected to the GPU-based block device via block IO interfaces in the Linux kernel. Considering the performance goal, we organize the data storage on GPU in a log structure. Coupled with multi-threaded design, internal read/write queues, and unique parallel garbage collector, the log structure not only optimizes random read/write operations, but also eases memory management.

The asymmetric programming model of GPU computing frameworks severely limits the programmability and efficiency of accelerator-based systems. Emerging efforts have tried to address the issue of scheduability of GPUs as compute nodes in heterogeneous systems. However,

- *Hao Chen, Jianhua Sun, Kenli Li, and Huailiang Tan are with School of Information Science and Engineering, Hunan University, Chang Sha, 410082, China.*
  *E-mail: haochen@aimlab.org, jhsun@aimlab.org, lkl510@263.net, tanhuailiang@hnu.edu.cn*
- *Ligang He is with Department of Computer Science, University of Warwick, Coventry, CV47AL, United Kingdom.*
  *E-mail: liganghe@dcs.warwick.ac.uk*

another inherent problem in the GPU computing, a unified data access model, deserves at least equal attention. Although focusing on the buffer cache as an illustrating example to address how to exploit the GPU memory as a heterogeneous resource in the OS, we believe that the abstraction we proposed has important implications for the GPU programming model, which is beneficial to both user-level GPU applications and OS designs.

We make the following contributions.

- We identified the feasibility of exploiting GPU memory as a heterogeneous resource to augment the OS kernel, and used virtual block device as the OS abstraction to support the design of the GPU-based buffer cache.
- We identified a key design choice of using log-structured data store in the management of GPU memory, which provides high performance read and write accesses. Together with the log structure, we also implemented an efficient parallel CUDA kernel to garbage collect orphaned data blocks and security mechanisms to protect data confidentiality.
- We built a prototype system BAG to demonstrate the effectiveness of a GPU-based buffer cache, and conducted extensive performance evaluation to validate our choices of the OS abstraction and algorithmic designs.

## 2 Motivation

This section presents the motivation in the design and implementation of BAG. More background about the GPU architecture is given in the supplementary file.

Recent trends show that memory capacity is increasingly becoming a limiting factor in commodity systems, due to the continued growth of big-data workloads, the technology shift to Cloud computing based on virtual machine consolidation, and the increasing demand for desktop applications such as video games. To address this emerging memory wall problem, hardware and software based approaches have been demonstrated to be effective in scaling the memory capacity and improving the utilization of physical memory.

Disaggregated memory [11], [12] is a new architectural extension, called memory blade, which expands local memory with a remote memory that can be dynamically partitioned and assigned to blade servers. The authors [12] developed a software-based prototype by augmenting the Xen hypervisor to evaluate the design of disaggregated memory, which leverages hardware support for virtualization to manage remote memory as a transparent demand-paging store. Transcendent memory (Tmem) [4] is a recently proposed approach to improving the utilization of physical memory. Tmem is a collection of idle physical memory in a system, managed as a fast pseudo-RAM of dynamically variable size. Tmem can be configured either as persistent or as ephemeral, and is only addressable indirectly by the kernel through a well-defined API that can be used by clients to implement

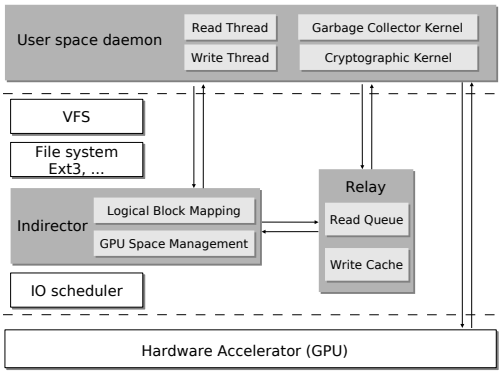



Fig. 1. System architecture of BAG

a variety of functionalities, such as clean page cache and fast swap device. Recent studies [1], [6], [16] show that low-latency and high-speed solid-state drives can be leveraged as an effective buffer cache at the block layer.

High performance demand of applications stimulates the rapid development of the GPU technology, resulting in not only more computing power but also larger memory capacity in GPUs. For example, some GPUs released to the market are equipped with up to 6 GB memory that is even larger than the main memory. In systems that is not intended for graphics or GPGPU computing most of the time, a large proportion of GPU memory would be left unused. In such systems, we can view GPU memory as a special kind of disaggregated memory, and manage it with interfaces similar to Tmem to augment the OS services, such as buffer cache and swap device, due to its abundant capacity and low access latency. All these above observations directly motivated the design and implementation of BAG as we elaborate next.

## 3 Design and Implementation

We start with an overview of the system architecture of BAG, and then discuss various design choices and implementation issues in constructing BAG.

### 3.1 Architectural Overview

BAG has three main components (shaded areas) as shown in Figure 1. The *indirector* is responsible for intercepting every block I/O request from the upper layer to the disk drive, and redirecting the I/O request to the virtual block device (*relay*) as necessary. BAG redirects read-hit requests and all write requests to the *relay*, and passes read-miss requests to the disk. Upon receiving I/O requests from the *indirector*, the *relay* copies write requests to an internal buffer and queues read requests temporarily, and then maps the I/O data to the user space daemon (referred to as the *daemon* in the following) in batches. The *daemon* fulfills the real task of storing/loading I/O blocks to/from the GPU memory organized in log structure, by encrypting/decrypting

data blocks with the corresponding cryptographic kernels. The garbage collector kernel periodically checks if the state of the GPU cache necessitates a defragmentation operation. If such an operation is required, the garbage collector starts to reorganize the GPU memory by reclaiming free slots and sorting the occupied slots in the order specified by the cache eviction policy (LRU). In the rest of this section, we elaborate each BAG component.

### 3.2 Logical Block Mapping

Positioning the *indirector* at the block I/O layer has three benefits. First, the *indirector* can communicate with the *relay* using a common interface without introducing OS kernel modifications, because the *relay* is also located at the same layer. Second, the manipulation of I/O requests, such as interception and redirection, is straightforward at this layer. Third, caching operations and configurations are transparent to the upper-level components, so that different types of file systems can be seamlessly supported. Next, we describe how to map logical disk blocks to the GPU memory that acts as a write-through cache.

#### 3.2.1 Aligning I/O Requests

In BAG, we maintain the metadata of I/O requests at the granularity of a page (4KB), which reduces memory consumption of metadata significantly, as compared to the granularity used in the sector-based addressing mode (512Byte). However, it complicates the handling of I/O requests that are not page-aligned or span multiple pages. For each request issued to the disk by the OS, the *indirector* first splits it into a series of blocks, each representing a potential page in the cache, and then uses the first logical block address (LBA) in each block as the index to determine if there is a cache-hit or cache-miss for that block.

We adopt an approach similar to the one described in [13] to address the issue of block alignment. Normally, the starting sector of a file system is rarely the first sector of the disk partition on which the file system is installed, or strictly aligned on a specific boundary, so I/O requests issued by the OS may cross page boundaries. For example, in our case, a page contains eight sectors, but the LBAs for the indexing purpose are not always guaranteed to be aligned on the eight-sector boundary. As a result, we need to identify the file system offset to make indexing LBAs aligned when splitting I/O requests. The side effect of splitting I/O requests and processing each separately is that it would inevitably lead to additional cache (GPU memory) and disk accesses as we show next.

#### 3.2.2 Splitting I/O Requests

Any given I/O request can be composed of many cache blocks, each of which may or may not be present in the cache. Worse, in certain cases, a cache block may be subject to the partially-addressed access, which means the location of the starting LBA of the request is in the middle of the cache block. These observations indicate that we can not handle the I/O request by simply sending it as a whole to the cache or the disk.

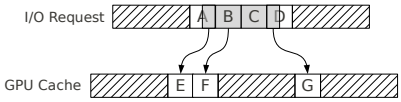



Fig. 2. I/O request example

Figure 2 shows an illustrating example, in which a read request consists of a cache-hit block B mapped to cache block F, a cache-miss block C, and two cache-hit blocks A and D that are not page-aligned and mapped to cache blocks E and G respectively. Processing such a complicated request results in a series of cache and disk accesses, and memory copies that finally merge the individually requested data. In this example, the *indirector* groups block A and B into one request and issues them to the cache together, since they are consecutively situated. The *indirector* passes Block C to the disk as a single request, and redirects Block D to the cache even though only part of the block is accessed as in the case of Block A. Only until the three requests return, can the *indirector* acknowledge the completion of the original I/O request. The indirector forwards the non-aligned write requests to the disk directly, because caching them would invalidate a portion of the target cache block, which makes it difficult to bookkeep the cached data.

### 3.3 GPU Storage Management

The management of GPU storage space plays a vital role in the whole system. Optimized algorithms and well-designed data structures have a significant impact on both I/O throughput and latency.

#### 3.3.1 Overview of the GPU Storage

Given that the bulk data transfer is preferable for GPUs, we maintain the I/O blocks on GPU in a log-like structure to allow write operations to be sequentially organized, thereby not only speeding up writes but improving overall performance (see Section 3.4.1). The GPU cache is regarded implicitly as a circular buffer, which represents a contiguous block of logical addresses with a *tail* pointing to the least-recently-written page and a *head* pointing to the most-recently-written page. As a consequence of the GPU cache being an append-only structure, the blocks previously written may become invalidated or orphaned upon the arrival of new writes, making the cache space fragmented. Thus, when the usage of the GPU cache exceeds a certain threshold or the degree of cache fragmentation surpasses a predefined value, the garbage collector wakes up to perform a cleaning operation that reclaims orphaned blocks and rearranges the active blocks in sorted order.

### 3.3.2 Data Structures and Associated Operations

We use a hash table to maintain the mapping between logical block addresses and GPU cache pages, and choose hopscotch hashing [8] to resolve hash collisions. The hopscotch hashing is an open-addressing based hash algorithm, and is well-suited for our scenario because of its three key advantages. The first is that it exhibits good performance at very high load factors of the hash table. The second is that the multi-threaded implementation of hopscotch hashing proves to be highly scalable. The last advantage is that hopscotch hashing is insensitive to the choice of hash function, and particularly suggests simple ones that are close-to-universal.

The hash table maintains the key-value association in an array, in which each slot contains a pointer to a node (located in a doubly-linked list) that stores the key-value pair with the disk LBA as the key and the logical page number of the GPU cache as the value. As shown in Figure 3, the hash table provides interfaces for processing the I/O requests, while the doubly-linked list defines the access order in which the nodes are read, updated, and inserted through the hash table. In a convenient manner, this data structure not only offers predictable performance such as fast access to the hash table and constant time manipulation of the linked list, but also naturally expresses the LRU access order for I/O requests in the course of system evolution.

Three basic operations (*lookup*, *insert*, *delete*) exposed by the hash table can be used for manipulating the I/O requests. We use the *lookup* operation to query the existance of the I/O data, and the *insert* operation to add new items to the cache. For non-aligned write-hit requests, we need to *delete* the corresponding cache slots to ensure that there do not exist partially cached blocks in the cache. Furthermore, as the system progresses and the working set changes, we also need to *delete* the aged blocks in the cache to make room for new I/O requests. To aid the selection of cache blocks for eviction, we use a linked list to maintain the LRU order of I/O requests present in the hash table.

The *indirector* intercepts all I/O requests regardless of if they are forwarded to the disk or the cache, so the *lookup* operation is always involved. For read requests, a hash table hit incurs a *READ* operation on a node of the LRU list and returns the node's content immediately, and the *READ* operation only changes the node position. For write requests, a hash table hit will cause the invalidation of an existing slot on GPU and an *UPDATE* operation on the LRU list; when a miss occurs, we insert a new entry into the table and perform an *ADD* operation. In both cases of write-hit and write-miss, we append I/O pages of the write-requests to the cache and increment the head indicating the most-recently-written page accordingly. In addition, if the cache is full when inserting a new entry, we evict the oldest page in the cache and *DELETE* the corresponding node in the LRU list.

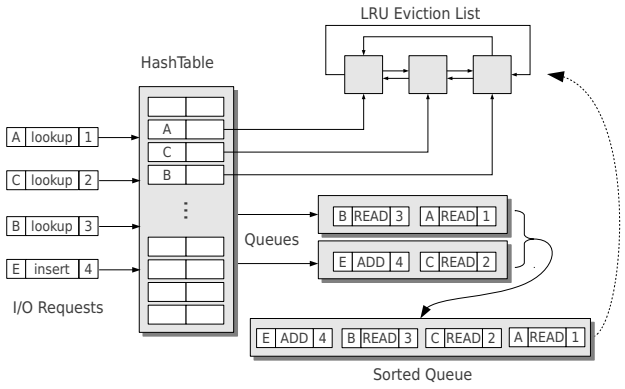



Fig. 3. Hash table and LRU list data structures.

### 3.3.3 Algorithmic Design

Since the *indirector* is in charge of interpreting all I/O requests, the processing speed in the respective operations such as looking up the hash table and maintaining the eviction list in the LRU order, is critical to system performance. These concerns motivate the algorithmic design that aims to maximize the exploitation of parallelism in the management of the GPU storage space. To provide concurrent accesses to the hash table, we adopt a multi-threaded design, and each thread can perform *lookup*, *insert*, and *delete* operations independently. BAG schedules requests in a round-robin manner to balance load across service threads. The concurrency control for the hash table is as follows. At any time instant, there may be multiple reads with the same LBA and multiple reads/writes with different LBAs. This strategy can not only exploit as much parallelism as possible but avoid the ordering issues arising from the asynchronous processing of I/O requests in other modules of BAG. We partition the hash table into multiple segments with each having a lock to mediate concurrent accesses. Two threads need to compete for the segment lock when they concurrently access different entries in the same segment. Accesses to distinct segments and concurrent hash table lookups are lock-free. This design strikes a balance between the concurrency level and the overhead of maintaining one lock per hash table entry.

Only parallelizing the accesses to the hash table is not sufficient to improve overall performance, because strict synchronization between the hash table and the LRU list would lead to sequential updates to the list and lock contention among service threads, neutralizing the proposed optimizations. To address this issue, we make a key observation that we can queue the operations to be applied to the list and keep the LRU list eventually consistent with the hash table, because the system only occasionally accesses the LRU list when evicting and reclaiming the cache slots. To this end, each service thread independently records the operations in an internal queue, and the queues are merged and then drained in batches when one or more predefined conditions are

satisfied, for example, when the queues are full or when the garbage collector performs synchronization between the LRU list and the GPU cache. Although multiple queues improves concurrency as compared to a single queue that may become another point of contention, we can no longer maintain the order of the I/O requests with multiple queues. The solution is to attach a unique *id* to each request before dispatching the request to a service thread, which allows the queues to be merged in sorted order prior to draining them. We use counting sort to perform sorting due to its linear time complexity for small integer values. After the queues are processed, the counter used to generate *id* values is reset to avoid numeric overflow. The proposed strategy for optimizing access to the LRU list avoids lock contention and only incurs slight overhead by spreading penalty across multiple threads.

#### 3.3.4 Garbage Collection

As a result of the log-structured design and the *insert* (*update*) and *delete* operations, the orphaned pages accumulate in the GPU cache. These garbage pages scatter among active pages and can not be effectively used. Therefore, when a certain amount of orphaned pages is present in the GPU cache, we need to invoke the garbage collector to clean and compact the GPU cache. The cleaning operation first makes the state of the LRU list up-to-date by draining queued operations, and then calls a series of GPU kernels to rearrange active pages in the order specified by the LRU list. The functionalities of the GPU kernels range from analyzing the LRU list to prepare a plan for page moves, to copying memory to compact the cache space. Preserving the LRU order in defragmenting the GPU cache is beneficial in two aspects. On one hand, with an ordered cache, it is straightforward to displace the aged pages indicated by the *tail* of the cache with new content when the cache eviction is needed, guaranteeing not to break the large I/O buffer written to GPU (discussed in Section 3.4.1) into small chunks. On the other hand, unordered organization of the cache makes it complicated to maintain the mapping between LBA and logical page number, because the system needs to record new location for each cache page to update the hash table accordingly after the cleaning operation. The maintenance procedure may involve scanning the LRU list or a set of recorded locations multiple times ($O(n^2)$ time complexity and $O(n)$ space complexity). However, this procedure is much easier to achieve if we keep the order of the cache the same as that of the LRU list.

Besides space efficiency considerations, the garbage collector also performs cleaning operations under circumstances where the aged pages may be frequently accessed due to large working sets or irregular I/O patterns. The situation is exacerbated if eviction decision is dependent on pending operations performed on pages that are exactly the candidates for eviction. For example, when the oldest page is to be evicted, but a *READ* operation on this page is still in the queue. In such a case, we can not guarantee the consistency when the queued operation is finally drained, or extra programming logic may be needed to avoid potentially inconsistent state. To address these issues, we maintain the access history for an amount of aged pages to assist in predicting the consequences of evicting the tracked pages reasonably. Negative predictions would cause the rearrangement of cache space. Furthermore, if the number of free pages is less than the expected threshold value when cleaning the cache space, we intentionally deallocate a percentage of aged pages (say, up to 10%) to reduce explicit eviction in subsequent execution. In doing so, the problems, such as inconsistent state and extra logic, can be alleviated at the cost of incurring cache misses due to the deallocated pages. While taking into account the limited performance impact of cache misses, the alleviation of design issues, and most importantly, the time consumption of the garbage collector (typically tens of milliseconds), we believe this strategy is cost-effective. The algorithm of garbage collection is present in the supplementary file.

#### 3.3.5 Memory Evacuation

It may not make sense to exclusively use all or part of the GPU memory as buffer cache, because normal GPU applications may fail to run due to inadequate memory. Managing memory effectively has been an active research topic, but implementing a custom memory management framework is tedious and error-prone. In the following, we present a simple strategy that achieves automatic evacuation of the memory reserved by BAG when needed.

We implemented a dynamic library to intercept the *cudaMalloc* call (using the LD_PRELOAD trick) to mediate memory allocation requests from both BAG and other applications. The intercepted *cudaMalloc* call only tracks the memory usage of each request, and forwards the real memory allocation to the CUDA runtime. BAG acts as a server that communicates with client applications using the message queue IPC mechanism because the dynamic library is linked to different address spaces. The server initially allocates a portion of GPU memory that is split into blocks of equal size to facilitate fine-grained memory (de)allocation, and this reserved space may shrink or grow dynamically. Clients obtain the amount of memory reserved by BAG when *cudaMalloc* is called at the first time, and only need to communicate with the server when the system can not fulfill the allocation request, but the requested memory size is less than the sum of the available memory and the memory used by BAG. Upon receiving client requests, the server releases one or more memory blocks according to the specified allocation size, which may incur a garbage collection operation. We found that the time taken by each *cudaMalloc* or *cudaFree* call is relatively independent of the memory size, which indicates that we can arbitrarily choose the size for memory blocks. However, large blocks may make memory management inefficient (comparing memory requests of size 1M to blocks of size

32 MB); small blocks would lead to excessive memory allocation time (with block size of 1MB, allocating 32MB memory needs to invoke the *cudaFree* call 32 times). Thus, we make the block size a configurable parameter.

### 3.4 Moving Data to GPU

The kernel-level *relay* and the user-level *daemon* together form a virtual block device that cooperatively serves I/O requests issued by the *indirector*. In this section, we presents the rationale behind the design choices of the two components.

#### 3.4.1 Relay

The *relay* is responsible for transferring and buffering I/O data between the *daemon* and the *indirector*. As a pseudo block device, the implementation of the *relay* leverages existing block I/O interfaces in the Linux kernel. The *relay* processes I/O requests asynchronously with one thread receiving requests from the *indirector* and another two threads dispatching read and write requests to the *daemon* respectively. Because of the lack of a kernel-facing interface to communicate with the GPU, this indirection layer is indispensable for delegating OS data to user space. A naive method for transferring data between OS kernel and user space is to deliver each I/O request individually. Obviously, this would result in low throughput due to frequent context switches and underutilized parallelism of the GPU due to insufficient I/O data. As a result, BAG buffers the I/O requests in the *relay* and delivers the pending requests to the *daemon* in batches. We deal with read and write requests differently as described next.

Read requests are aggregated in a fixed-length queue and processed when the queue is full or a predefined timeout interval expires. This design can help reduce the amortized cost of communication between kernel space and user space, and avoid queuing the requests for an excessively long time. However, the physical memory pages (see *bv_page* member in *bio_vec* structure in the Linux kernel) allocated for the individual read requests are typically not contiguous, so we need to invoke *mmap* system call multiple times to map all the I/O requests to user space. For large queues, the overall overhead will be dominated by too many *mmap* calls. Thus, we allocate a data buffer with the desired number of contiguous physical pages, which acts as an intermediate staging post for reads. Although involving considerable memory copies, this strategy improves read performance remarkably.

In contrast, writes are copied to an internal buffer, and mapped to user space once the buffer size has reached maximum capacity. Despite the penalty incurred by memory copies, the write buffer has several benefits. First, if there is room in the buffer, we can complete write requests (e.g. by calling *bio_end* in the Linux kernel) as soon as possible, reducing response latency. Second, reads can be firstly served by querying the write buffer, and a read hit in the buffer will save the cost of visiting user space. Third, as in the case of reads, by combining scattered memory pages into a write buffer with contiguous address space, we can map the buffer to user space using one *mmap* call. Fourth, buffering write requests reduces bandwidth contention with read requests, since read requests are generally more important than write requests, and often granted higher priority when mapping I/O data between kernel space and user space. Finally, frequent writes targeting the same block can be absorbed by the buffer and acknowledged before they go to user space. This strategy has a positive impact on performance, but it would break the intended log structure of the cache. Therefore, we reject this idea and maintain structural consistency between the write buffer and the GPU cache. We present the optimization against the structure of the write buffer in Section 3.4.2. The supplementary file provides more details about the kernel-user communication mechanisms.

#### 3.4.2 User Space Daemon

The *relay* processes I/O requests in separate threads, so we use two threads to handle data transfer between host and device in the *daemon* accordingly. We assign threads with the same type (read or write) in the *relay* and *daemon* to a single core to avoid context switching and cache bouncing costs. The daemon calls cryptographic GPU kernels in processing I/O data due to the reasons discussed in Section 3.5. In the read thread, we configure the decryption kernel to make the output (a page array residing in the device memory) properly arranged in the order in which I/O requests are mapped into the user-space page array, so that we need only one device-to-host memory copy to transfer the output back to the host. In the write thread, since the write buffer in the *relay* may contain invalidated pages as described in Section 3.4.1, the encryption kernel excludes these pages to save computational resources, even though the whole write buffer is present in the GPU cache. The garbage collector kernel obtains its input from the *indirector* using the *procfs* interface as well, and the communication protocol between them is similar to that used in the *relay*.

CUDA devices with Compute Capability 2.0 support concurrent GPU kernel execution and data copy for better utilization of the GPU. We do not currently realize concurrent data copy that requires memory to be allocated in page-locked mode, which is not supported in our implementation of memory mapping. GPUstore [18] has a flexible framework for integrating GPU computing into storage systems. In future work, we plan to reinforce BAG with GPUstore's mechanisms. Our current implementation works well due to the goal of using GPU as buffer cache that involves less data transfer, as compared to GPUstore that performs cryptographic operations on all data stored in the system.

### 3.5 Data Confidentiality

GPU device provides shared access to user-level applications, so without appropriate protection, backing up

the OS kernel data on GPUs may lead to the risk of information disclosure. In this section, we first show the inadequacy of memory protection implemented in the current generation of GPUs, and then present the approaches to securing data storage on GPUs.

#### 3.5.1 Threat Model

We assume that the operating system is trusted, and only the GPU's global memory is potentially vulnerable to compromise. In other words, our assumption is that the adversary has necessary privileges to compute on GPU device, but its activity is under the control of the standard Linux security model. Privilege escalation to the OS kernel is beyond the scope of this paper because attackers with root access to all system resources would have no intentions to subvert data protection on GPUs.

#### 3.5.2 CUDA Memory Protection Model

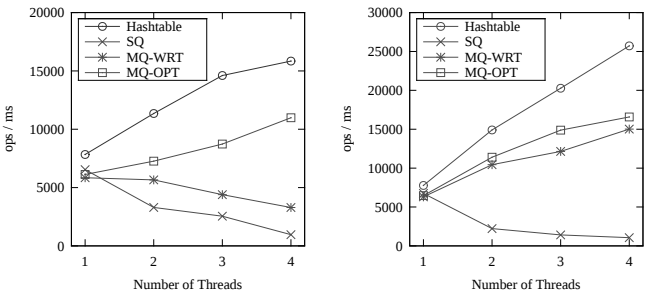
It is worth noting that, in CUDA, the virtual memory system coupled with context-based memory isolation guarantees that distinct and simultaneously running contexts would not interfere with each other. In the following, we demonstrate two properties (not officially documented) of CUDA that motivated us to develop solutions to address data confidentiality concerns.

***Property 1: The GPU does not scrub the global memory between kernel invocations.*** We can verify this property as follows. We first launch a CUDA kernel generating a random string in global memory. Then, after its completion, we can fully recover the randomly generated string by searching the GPU memory dump in another CUDA context with distinct user privilege.

***Property 2: Kernel's code memory is not accessible via the global memory space.*** Official documents of CUDA do not explicitly specify the storage location or the accessibility of kernel code memory. However, experimental measurements we conducted show that it is not feasible to read or write the code memory via global memory space, because by calculating the MD5 hashes of global memory space in two subsequent kernels (both of which do not involve modification to global memory), we can always obtain the same value.

#### 3.5.3 Data Protection with GPU-accelerated AES

On one hand, to address the security issue impilied by *Property 1*, we implemented GPU-accelerated AES kernels that are detailed in the supplementary file. On the other hand, data confidentiality is based on the assumption that cryptographic keys are kept absolutely secret. In our setting, ~~managing keys in memory spaces that are vulnerable to memory disclosure attacks, would compromise data security~~. Therefore, the cryptographic kernels generate keys ~~dynamically~~ on the GPU to eliminate persistent storage for the keys. Inspired by *Property 2*, we developed a custom key generator that produces keys directly in shared memory. It is also possible to store keys in registers, but due to the per-thread register limit, this option may cause register spilling into local memory. Additionally, according to the choice of storage location for keys, we also derive the round key in shared memory. While incurring a constant computational overhead, this design eliminates expensive global memory access and consequently reduces latency.




(a) Write-heavy workload (b) Write-light workload

Fig. 4. Evaluation results of the hash table.

## 4 Evaluation

In this section, we evaluate the performance of BAG using micro-benchmarks and macro-benchmarks. The experiments were conducted on a machine equipped with a AMD phenomII X6 1055T CPU, 4 GB memory, and a Nvidia GTX 480 GPU. The GPU card has 15 streaming multiprocessors (SMs), each containing 32 cores (480 cores in total), and 1.5 GB device memory. The maximum amount of shared memory for each SM is 48 KB. The operating system was Ubuntu Linux 11.04 with NVIDIA driver version 280.13 installed.

### 4.1 Micro-benchmarks

We first examine the performance of each main component in BAG individually (The results for ASE evaluation are shown in the supplementary file). The experimental results were all averaged over 10 runs.

#### 4.1.1 Hashtable

We evaluate the performance of hashtable using four configurations: 1) The original hashtable implementation without maintaining the LRU list (Hashtable). 2) A single queue connecting the hashtable and LRU list (SQ). 3) Multiple queues that are drained upon a *put*/*delete* operation (MQ-WRT). 4) Optimized design of multiple queues as described in this paper (MQ-OPT). For this test, we populated a hash table that has approximately $2^{23}$ items to a high density 90%. To examine the performance variations under different loads, we performed this experiment with a write-heavy workload that consists of 60% *get*, 30% *put* and 10% *delete*, and a write-light workload that contains 90% *get*, 5% *put* and 5% *delete*.

As shown in Figure 4, the hopscotch hash algorithm scales equally well under different loads, but exhibits better performance measured in terms of operations per

TABLE 1
Overhead of garbage collection.

| ms | 128M | 256M | 512M | 768M | 1024MB |
|---|---|---|---|---|---|
| GPU (load factor 0.9) | 3.05 | 6.02 | 12.07 | 18.14 | 24.21 |
| CPU (load factor 0.9) | 181.48 | 361.99 | 725.05 | 1093.52 | 1461.31 |
| GPU (load factor 0.8) | 2.86 | 5.67 | 11.41 | 17.20 | 23.01 |
| CPU (load factor 0.8) | 139.77 | 280.11 | 560.73 | 842.25 | 1129.34 |

TABLE 2
Breakdown of garbage collection overhead.

| ms | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Sum |
|---|---|---|---|---|---|---|
| 128M | 0.13366 | 0.01078 | 0.01517 | 2.87248 | 0.01955 | 3.05164 |
| 256M | 0.13398 | 0.01366 | 0.02278 | 5.83702 | 0.00941 | 6.01685 |
| 512M | 0.17360 | 0.02861 | 0.07117 | 11.78397 | 0.01648 | 12.07383 |
| 768M | 0.32685 | 0.02608 | 0.08589 | 17.68806 | 0.01699 | 18.14387 |
| 1024M | 0.39882 | 0.03056 | 0.12870 | 23.63338 | 0.01834 | 24.2098 |

millisecond for write-light workload (compare the scale of y-axis in two sub-figures). In contrast, the performance of SQ drops quickly as the number of threads increases (from 6537 ops/ms at 1 thread to 960 ops/ms at 4 threads in Figure 4(a)), due to the significant overhead incurred by lock contention among threads. Although MQ-WRT shows good scalability for workload containing fewer writes as shown in Figure 4(b), it degrades performance under write-heavy workload as shown in Figure 4(a) because intensive writes cause frequent synchronization between the queues and the LRU list. The optimized design of multiple queues, MQ-OPT, not only outperforms SQ and MQ-WRT, but scales with the number of threads under both workloads.

#### 4.1.2 Garbage Collection

Table 1 compares the time consumption of garbage collection between two implementations. One is the GPU-accelerated implementation as proposed in this paper, and the other is a single-threaded implementation on CPU using the CUDA API *cudaMemcpy* with the parameter *cudaMemcpyDeviceToDevice* specified. We conducted the measurements using the same memory layout and load factors. The results for the GPU version only contain the kernel execution times~~, and~~ the data transfer for kernel input, such as the auxiliary array, consumes less than 1ms. As shown in Table 1, for all ~~the~~ cache sizes, the GPU version outperforms the CPU version by a factor of 60 when the cache is 90% full. In comparison, the speedup factor is reduced to 49 with a less loaded cache. Table 2 shows the breakdown of garbage collection overhead into five categories corresponding to the five steps (see the supplementary file) described in Section 3.3.4. We can observe that the vast majority of overhead comes from memory copy for chains (Step 4), implying the presence of a small number of loops and the significance of exploiting data-parallelism on GPUs.

#### 4.1.3 Memory Evacuation

Table 3 shows the overhead of our memory evacuation mechanism discussed in Section 3.3.5. The row marked as 'NAT' is the baseline measurement of the native *cudaMalloc* call, and the row marked as 'INT' presents the time consumption of the intercepted *cudaMalloc*. The later does not consider the overhead in scenarios where an allocation request may result in multiple memory deallocation operations in BAG. For example, if the memory block size is 16MB, and we have a request of allocating 64MB memory. Then, we need to free 4 memory blocks, which would incur an additional cost of invoking *cudaFree* 4 times.

TABLE 3
Overhead of memory evacuation.

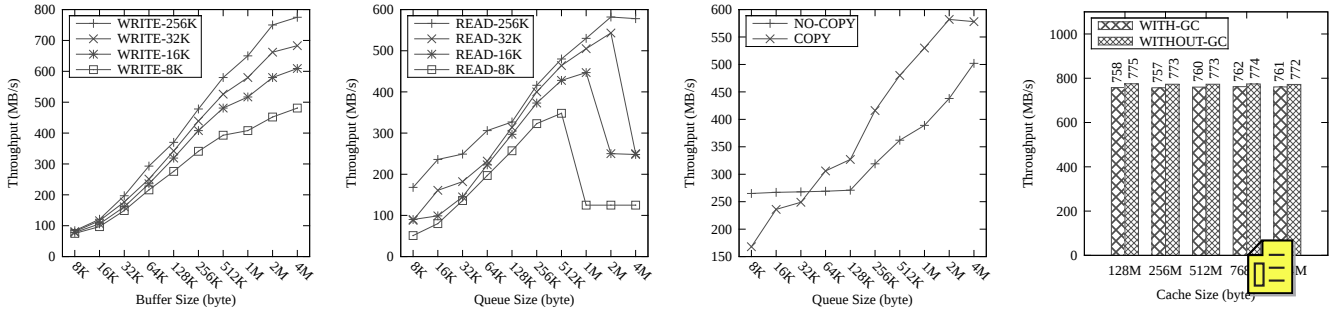
| ms | 1M | 2M | 4M | 8M | 16MB | 32M | 64M | 128M |
|---|---|---|---|---|---|---|---|---|
| NAT | 0.098 | 0.069 | 0.067 | 0.067 | 0.068 | 0.071 | 0.082 | 0.124 |
| INT | 0.134 | 0.118 | 0.113 | 0.114 | 0.117 | 0.117 | 0.126 | 0.177 |

#### 4.1.4 Virtual Block Device

We used the Intel Open Storage Toolkit [3] to generate complex I/O patterns to evaluate the performance of reads and writes of the virtual block device. For each test scenario, we set the number of outstanding requests to 64 with various request sizes (8KB, 16KB, 32KB, 256KB). Each workload was configured to access raw block devices directly to bypass the buffer cache and file system. All read and write requests were synchronous I/O with no think time.

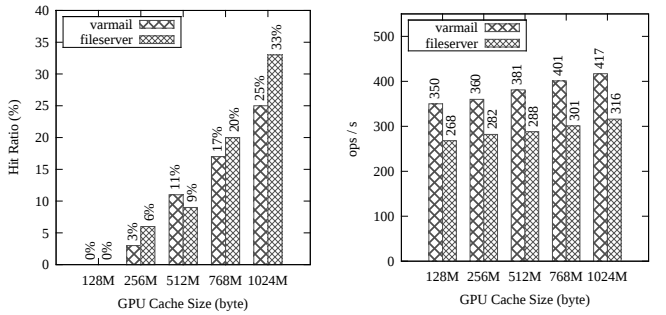
Figure 5(a) depicts the write performance. As expected, the throughput increases linearly with the buffer size. For example, with a request size of 8KB, writes using a buffer size of 4MB achieve 6.4 times higher bandwidth as compared to a buffer size of 8KB. As request size increases to 256KB, the relative throughput gain of writes increases to 9.2 times. Note that random reads and writes can achieve almost identical performance as sequential reads and writes, so we only present the results of sequential workloads in this experiment. As for reads, increasing the queue size does not always help improve performance, because the queuing delay may outweigh the benefits of batching process of read requests, especially for large queues with small request sizes. For example, as shown in Figure 5(b), with a request size of 8KB, the read performance drops dramatically from 348MB/s to 125MB/s when the queue size exceeds 512KB. For a larger request size of 256KB, the throughput reaches a maximum of 582MB/s around the queue size of 2MB. These observations indicate that we need an adaptive queuing policy for reads to handle the real-life workloads that can be a mix of various access patterns. For example, we can use differentiated queuing service for read requests with varying access patterns. We leave this optimization to future work.

Figure 5(c) quantifies the copy-based optimization for reads as described in Section 3.4.1. The results were obtained using the request size of 256KB. When the queue size is less than 32KB, mapping each read request individually is more preferable than memory copy.




(a) Write Performance (b) Read Performance (c) Copy Based Read (d) Overhead of GC

Fig. 5. Evaluation results of the virtual block device.




(a) Cache Hit Ratio (b) Impact on Performance

Fig. 6. Performance and cache hit ratios of workloads.

However, as the queue size increases, the superiority of copy-based method becomes more and more evident. For example, with a queue size of 2MB, "COPY" beats "NO-COPY" by a factor of 1.3. To estimate the overhead of garbage collection, we performed two experiments (using the same configuration, writes with request size of 256KB), in which the garbage collection is enabled and disabled respectively. Figure 5(d) shows that the overhead caused by garbage collection is at most 2.1%.

### 4.2 Macro-benchmarks

To demonstrate the performance benefits of BAG for real system workloads, we used FileBench [2] to generate file system workloads. We chose two common server workloads: mail server and file server (other workloads show similar trends, but the chosen workloads are more representative because of the desired write frequency). Two basic performance metrics, throughput measured in terms of operations per second and cache hit ratio, are reported. In mail server workload, FileBench spawns 16 threads to perform a sequence of operations to imitate reading mails (*open*, *read* the whole file, and *close*), composing (*open*/*create*, *append*, *close*, and *fsync*) and deleting mails. The average file size is 32KB, and the read-write ratio is 1:1. This workload generates 1872MB data including 60,000 files. The file server workload emulates servers hosting home directories of multiple users. It spawns 50 threads each performing a series of *create*, *delete*, *append*, *read*, *write*, and *stat* operations. The average file size is 256KB, and the read-write ratio is 1:2. This workload generates 2482MB data including 10,000 files. The results shown below were obtained by taking into account all the overhead incurred by the components of BAG detailed in Section 4.1.

Figure 6 shows that BAG effectively improves hit ratio as well as throughput for both workloads. As shown in Figure 6(a), with the increase of cache space, the hit ratios of the two workloads are increased up to 33% and 25% respectively when the cache size reaches 1GB. Here, we measured the results by considering the system cache and GPU cache as a whole. Due to the large synthetic workload and limited system memory, we can observe from Figure 6 that adding 128MB GPU memory to the cache has no impact on cache hit ratio. The resulting cache hit ratio directly translates into throughput improvement. Figure 6(b) shows that 1GB cache can improve performance by 19% and 18% for *varmail* and *fileserver* respectively, if we use the throughput with cache size of 128MB as the baseline that is slightly lower than actual throughput without cache. Considering that the workloads features large working sets and a significant proportion of write operations (see the read-write ratio), we believe BAG achieves reasonable performance gains for the chosen workloads.

## 5 Related Work

**OS Support for Heterogeneous Processors:** The research community has spent considerable effort on the problem of processor heterogeneity in OS designs. The Helios [14] OS introduces satellite kernels that export a set of OS abstractions for CPUs with distinct ISAs to simply the task of programming heterogeneous systems. The Barrelfish OS [5] treats the underlying hardware as a distributed network with independent OS kernels on each core, communicating via RPC. However, the abstractions proposed by Helios and Barrelfish do not support GPUs, which lack features such as preemptive scheduling, interrupts, and direct IO access to create the full context to run the OS code. PTask [15] addresses the issue of insufficient support of existing OS abstractions for GPU-based

interactive applications. With the dataflow programming model provided by PTask, the programmer can manage computation in a graph structure that consists of OS objects to provide the OS kernel with sufficient visibility and control over the course of the computation, thereby guaranteeing fairness and performance isolation.

Efficient scheduler for heterogeneous systems such as CPU-GPU hybrids has received considerable attention. TimeGraph [9] is a GPU scheduler to support real-time multi-tasking environments, providing isolation and prioritization capabilities in GPU resource management. Currently, TimeGraph only supports graphics workloads. The Pegasus system [7] offers a uniform resource usage model and schedules virtual machines to share accelerators fairly and efficiently, targeting virtualized systems. PTask also provides support for GPU-aware scheduling and makes applications respect scheduling priorities in the OS kernel. In contrast to these research endeavors, we mainly focus on exploiting the potential of GPUs for data storage in the OS kernel, more than just the computational capabilities of GPUs.

Gdev [10] introduces an open source kernel driver and user-level library to manage the GPU as first-class computing resource, facilitating the sharing of GPU resources. GPUfs [17] is a system that exposes POSIX-like file system API to GPU programs. In order to optimize GPU file access, GPUfs also maintain a buffer cache in GPU memory. Although both GPUfs and BAG manages GPU memory as buffer cache, there are many differences between the designs of these two systems because of the distinct goals. GPUstore [18] is a general-purpose framework that is intended to accelerate computational tasks in storage systems. GPUstore provides efficient mechanism for mapping memory pages between kernel and user space, and we hope to integrate it into BAG to further improve performance. The RAID 6 implementation on GPU in GPUstore can also be used in our system for fault tolerance, since only high-end GPUs support ECC for GPU memory. Both GPUstore and BAG operate at the storage layer, but BAG focuses on how to expand memory capacity with GPU's disaggregated RAM, instead of just the computational capability.

**Memory expansion:** Disaggregated memory [11], [12] has been proposed as an effective approach to scaling the local memory capacity of blade servers. The work in [12] also demonstrated the feasibility of enhancing disaggregated memory with content-based page sharing, which would be a good fit for GPUs. Transcendent memory (Tmem) [4] is a new approach to improving the utilization of physical memory, and a well-designed front-end API of Tmem can be used to implement various memory capacity optimizations, such as remote paging and page compression. We believe that developing a back-end for Tmem using GPUs is an interesting future work. Flash has been identified to be a promising way to expand the buffer cache [1], [6], [16]. All these previous studies in part motivated our work. More discussion about the related work can be found in the supplementary file.

## 6 Conclusions

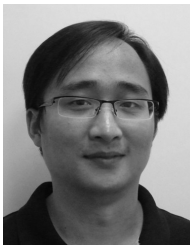
This paper presents the design, implementation and evaluation of BAG, a system managing GPU as the buffer cache to augment the operating system kernel. With efficient data structures and algorithmic designs for GPU storage management, BAG exploits ~~only~~ existing OS interfaces to achieve good throughput and low latency, while providing an abstract view of the GPU memory to retain modularity and composability.

## 7 Acknowledgments

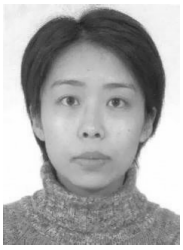
The authors are grateful to the anonymous reviewers for their helpful feedback. This research was supported in part by the National Natural Science Foundation of China under grants 61272190 and 61173166, the Program for New Century Excellent Talents in University, the Leverhulme Trust under grant RPG-101, and the Key Program of National Natural Science Foundation of China under grant 61133005.

## References

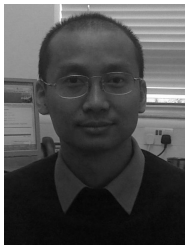
[1] Facebook Flashcache. https://github.com/facebook/flashcache/.
[2] FileBench. http://www.fsl.cs.sunysb.edu/~vass/filebench/.
[3] Intel open storage toolkit. http://www.sourceforge.org/projects/intel-iscsi.
[4] Transcendent Memory. https://oss.oracle.com/projects/tmem/.
[5] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singuhania. The Multikernel: a New OS Architecture for Scalable Multicore Systems. In Proc. SOSP 2009. ACM.
[6] F. Chen, D. Koufaty and X. D. Zhang. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In Proc. ICS 2011, pp.22-32.
[7] V. Gupta, K. Schwan and N. Tolia. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In Proc. USENIX 2011.
[8] M. Herlihy , N. Shavit, and M. Tzafrir. Hopscotch Hashing. In Proc. DISC 2008.
[9] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In Proc. USENIX Annual Technical Conference, Berkeley, CA, USA, 2011. USENIX Association.
[10] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In Proc. USENIX Annual Technical Conference (ATC), June 2012.
[11] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In Proc. ISCA, 2009.
[12] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan and T. Wenisch. System-level Implications of Disaggregated Memory. In Proc. HPCA, 2012.
[13] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. In ACM Transactions on Storage, volume 4, May 2008.
[14] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In Proc. SOSP 2009. ACM.
[15] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In Proc. SOSP 2011. ACM.
[16] M. Saxena, M. M. Swift and Y. Y. Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In Proc. EUROSYS 2012.
[17] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. In Proc. ASPLOS 2013
[18] W. B. Sun, R. Ricci, and M. L. Curry. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In Proc. SYSTOR 2012.



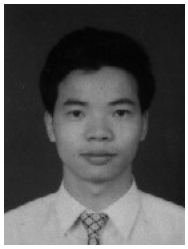
**Hao Chen** received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the PhD degree in computer science from Huazhong University of Science and Technology, China in 2005. He is now an Associate Professor at the School of Information Science and Engineering, Hunan University,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



**Jianhua Sun** is an Associate Professor at the School of Information Science and Engineering, Hunan University, China. She received the Ph.D. degree in Computer Science from Huazhong University of Science and Technology, China in 2005. Her research interests are in security and operating systems. She has published more than 50 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers.



**Ligang He** studied for the Ph.D degree in Computer Science at the University of Warwick, UK, from 2002 to 2005, and then worked as a postdoctor in the University of Cambridge, UK. In 2006, he joined the Department of Computer Science at the University of Warwick as an Assistant Professor. He is now an Associate Professor in the Department of Computer Science at the University of Warwick. His research interests focus on parallel and distributed processing, Cluster, Grid and Cloud computing. He has published more than 40 papers in international conferences and journals, such as IEEE Transactions on Parallel and Distributed Systems, IPDPS, CCGrid, MASCOTS. He has been a member of the program committee for many international conferences, and been the reviewer for a number of international journals, including IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, etc. He is a member of the IEEE.



**Kenli Li** received the Ph.D degree in computer science from Huazhong University of Science and Technology, China, in 2003, and the B.S. degree in mathematics from Central South University, China, in 2000. He has been a visiting scholar at University of Illinois at Champaign and Urbana from 2004 to 2005. He is now a Professor of the School of Information Science and Engineering at Hunan University. He is a senior member of CCF. His major research contains parallel computing, Grid and Cloud computing, and DNA computer.

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