An Architecture Design of GPU-Accelerated VoD Streaming Servers with Network Coding

Jin Zhao*†, Xinya Zhang*† and Xin Wang*†
*School of Computer Science, Fudan University
Shanghai 200433, China
†Shanghai Key Lab of Intelligent Information Processing
Shanghai 200433, China
Email: {jzhao, 06300720198, xinw}@fudan.edu.cn

***Abstract*—Graphics processing unit (GPU) has evolved into a general-purpose computing platform. Inspired by the GPU technology advantage, this paper concerns the design and performance evaluation of practical GPU-accelerated server architecture for Video-on-Demand (VoD) services with network coding. Following the proposal of an optimized network coding algorithm based on parallel threads on GPU, a GPU-Accelerated Server (GAS) for VoD streaming is designed to leverage the workload between GPU and CPU and thus improve the performance of the VoD server. Extensive real-world experimental results have proved that compared with the approaches with network coding performed only on CPU or GPU, the proposed GAS architecture is more advantageous in serving capacity, response time, and CPU usage. Our study has investigated a way of designing high performance VoD streaming servers with network coding and GPU-acceleration incorporated.**

***Index Terms*—Video-on-demand, network coding, GPU, streaming server**

I. Introduction

Video-on-Demand (VoD) services are becoming increasingly popular over the Internet [1]. It has been known that centralized server approach is expensive in terms of both bandwidth cost and serving capacity. As an alternative, peer-to-peer (P2P) network is a cost-effective solution to providing VoD streaming to a large number of users [2]–[4]. In a typical P2P-VoD architecture, peers are organized in a tree [5], [6] or mesh [7], [8] overlay structure to collaboratively relay data blocks for each other. The server's bandwidth cost and workload can thus partially be alleviated by leveraging each peer's bandwidth and capacity [2].

Network coding provides an information-theoretical approach to network throughput improvement [9]. The idea behind network coding is to allow coding operations on data flows to be performed at intermediate nodes. Since the landmark work on randomized network coding [10], [11], many research efforts have been directed towards the practical application of network coding in the P2P networks, such as Avalanche [12] and R^2 [13]. In general, network coding eliminates the need for block reconciliation and thus is highly resilient to peer dynamics. In the context of VoD, Guha et al. [16] investigated the feasibility of providing VoD with low start-up delays using network coding. Chi et al. [15] proposed a scheduling algorithm based on deadline-aware network coding to meet the playback deadline constraints. It has been demonstrated that P2P VoD can benefit a lot from network coding in terms of video delivery efficiency. An commencial P2P system, UUSee [28], has also reported the effectiveness of network coding in VoD service.

Unfortunately, for the P2P-VoD systems with network coding, most existing results have focused on the following issues [15]–[17]: 1) How can network coding simplify data scheduling? 2) What strategy can be designed to improve peer resilience? 3) Which overlay topology should be constructed for network coding? Little has been found on a sufficient analysis of the network coding overhead and on the performance improvement of the VoD server in the P2P-VoD systems with network coding.

In this paper, we focus on the problem of optimizing the network coding overhead at VoD servers. The problem is motivated by the following observations.

First, despite of the benefits of network coding, it is, however, important to recognize that the performance gain of network coding is obtained at the cost of higher coding complexity at each peer including the server. Such computational overhead adds to VoD servers workload, especially as the number of blocks to be coded scales up.

Second, VoD client's intrinsic behaviors in asynchronous requests and interactive operations further exaggerate the problem. In contrast to live streaming, a VoD client may start to request a video at any time, and may randomly access arbitrary part of a video. Such asynchrony may lead to peer's failure in finding collaborating partners with similar block interests. In this case, the peer have to resort to the server for the desired video blocks.

These characteristics make VoD server's computing capability a bottleneck. A VoD server has to be able to perform network coding operations and at the same time serve hundreds (or even thousands) of directly connected peers. Consequently, achieving high throughput and fast response times at the VoD server presents a challenging task for the P2P-VoD systems with network coding, especially in the case to sustain hundreds of concurrent video channels. As an evidence, a runtime trace of a real-world commercial P2P streaming system has also revealed that the available server computing capacities are unable to keep up with the increasing user demand in

COLLABORATECOM 2010, October 9-12, Chicago, USA

DOI 10.4108/icst.collaboratecom.2010.37

video channels and have become a bottleneck to the streaming service [14]. Even when efficient algorithms are employed for block scheduling and overlay construction, the network coding overhead may still yield a degradation of the streaming quality.

Derived from conventional wisdom, this paper attempts to investigate a new approach for bulding VoD servers using graphics processing units (GPUs). Our approach is inspired by the rapid increase in GPU's hardware performance and recent improvements in its programming flexibility. Modern GPUs offer large numbers of computing cores that operate in parallel. Compared to CPUs, commodity off-the-shelf GPUs have a much more favorable price-performance ratio. For example, NVIDIA GeForce 8800GT GPU is over three times faster than Intel Core2 Quad 3.0GHz CPU in terms of floating point operations per second (FLOPS). In addition, flexible programming environments [24], [25] let any application tap into the vast GPU computational power previously available only to graphics applications. For example, CUDA (Compute Unified Device Architecture) provides a C-language development environment for NVIDIA GPUs [24]. The growing computing ability has transformed the GPU into a massively parallel general-purpose accelerator processor for non-graphics problems, ranging from image processing to scientific computing [27].

We believe that VoD servers with network coding provides a potential application domain for GPUs. However, building an efficient VoD server with GPU is a non-trivial task. To provide high throughput and fast response times, a VoD server needs to be carefully structured to conform to data-parallel programming model provided by the GPU, and also to balance the workload between CPU and GPU.

We propose a GPU-Accelerated Server (GAS) architecture which embeds the optimized GPU-based network coding components. Together with the GAS architecture, a flexible scheduling assignment is also investigated. Using real-world experiments, we evaluate the performances of both the GPU-accelerated network coding and the GAS architecture with a set of measurement metrics. The real-world experiments are designed with NVIDIA GeForce 8800GT GPU [26] and its development platform CUDA [24]. The experiment results show that the proposed GPU-accelerated network coding algorithm on NVIDIA 8800GT achieves a doubled coding throughput compared with that on Intel Quad-Core CPU with SSE2 acceleration. The experiment results also show that the GAS architecture achieves performance improvements of approximately 2 times in both average response time and serving capacity compared with the pure CPU implementation.

To our best knowledge, this paper provides, for the first time, the detailed measurements of streaming server capacity utilization in a GPU-accelerated P2P-VoD streaming system. It is also worth while mentioning that the proposed GAS architecture can be deployed in a wide range of software and hardware environments, including Microsoft Windows, Linux, and Mac OS with the CUDA driver support by NVIDIA [24].

The rest of this paper is organized as follows. Section II describes the background and related work. Section III presents the design of VoD server architecture and scheduling algorithm. Section IV analyzes the experimental results. Finally, Section V concludes this paper.

II. Background and related work

In this section, we position our work in the context of related research.

A. Random Linear Network Coding

Network coding [9] has been originally proposed for achieving the theoretical multicast capacity of a network. Using random network coding, the server or a peer is allowed to combine a number of data blocks and encode them into one or several outgoing blocks, specified by independently and randomly chosen coding coefficients. Let $\boldsymbol{b}$ denote a group of m data blocks to be encoded at the server, which is represented by a vector with block elements as follows

$$\boldsymbol{b} = [\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_m]^T \quad (1)$$

where each data block $\boldsymbol{b}_j$, $j = 1, 2, \ldots, m$, has a fixed size of k bytes, and the superscript T denotes the transpose operation.

For any specific vector $\boldsymbol{b}$, in order to produce one outgoing block x_i, a server first generates a coding coefficient vector $\boldsymbol{a}_i = [a_{i1}, a_{i2}, \ldots, a_{im}]^T$, where each coding coefficient a_{ij} is randomly chosen from Galois field GF(2^8). With linear network coding, the new block x_i is created by

$$x_i = \sum_{j=1}^{m} a_{ij} \cdot b_j \quad (2)$$

The corresponding coefficient vector $\boldsymbol{a}_i$ is embedded in each block's header during transmission.

Each coded block $\boldsymbol{x}_i$ is sent together with the corresponding coefficient vector $\boldsymbol{a}_i$ from the server to its downstream peer. Clearly, an overhead of m bytes per coded block is yielded by the accompanying coefficient vector since it contains m elements from $GF(2^8)$, or equivalently m bytes. Such an overhead is subtle for a large block size k and a moderate group size m.

After a peer has received m linearly independent coded blocks $\boldsymbol{x} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m]^T$,, it begin to decode using the following matrix operations. The m coding coefficient vectors $\boldsymbol{a}_i$, $i = 1, 2, \ldots, m$ in fact constitute an $m \times m$ matrix A. Each row in A corresponds to the coefficients of one coded block.

$$\boldsymbol{b} = \boldsymbol{A}^{-1} \cdot \boldsymbol{x} \quad (3)$$

Note that the inverse of A is possible if and only if its rows are linearly independent, i.e., A has full rank. The inverse of $\boldsymbol{A}$ can be implemented efficiently using Gauss elimination or its extension Gauss-Jordan elimination.

With network coding, all blocks are mixed and treated equally, which in turn eliminates the need to schedule the "rarest block" first or find a specific block. As long as a peer can collect enough number of linearly independent blocks, it can decode the original content.

B. GPU-based Acceleration

The primary practical problem network coding confronts is the coding complexity. As has been observed in a real-world experiment, network coding is only computationally feasible for the number of blocks to be coded less than a thousand [18]. To accelerate network coding throughput, a parallelized progressive approach has been proposed with SSE2 and AltiVec single instruction - multiple data (SIMD) vector instructions performed on Intel x86 and PowerPC processors, respectively [19].

More recently, preliminary results have been reported on employing graphics processing units (GPUs) to accelerate network coding by mapping network coding operations to GPUs [20]–[23]. The approaches to GPU-based network coding behind these results are in common.

In order to help better understand the GPU-accelerated processing of network coding, we first use NVIDIA Geforce 8800GT (henceforth used in this paper) as an example to briefly introduce the GPU architecture. The GPU contains a total of 112 streaming processors (SP) that are grouped to 14 streaming multiprocessors (SM), with each SM containing 8 SPs. The GPU employs a single-instruction multiple-thread (SIMT) architecture which is akin to SIMD. In the SIMT architecture, when an SM is allocated for a set of threads to execute, it groups the threads into 32 as one SIMT scheduling unit, or warp. Each unit are exectued in 4 cycles on the 8 SPs, with 1 thread per SP each cycle. If one unit stalls, the SM can execute another unit with zero-overhead hardware-based scheduling. Meanwhile, the 14 SMs can run different sets of threads independently and simultaneously. Therefore, the GPU is particularly suitable for data-parallel computations with data processing tasks mapped to parallel threads.

In order to perform the parallel network coding on the GPU, the following two issues must be considered:

1) Divide the coding computations into tasks that can be executed concurrently.
2) Map the tasks to physical streaming processors.

Suppose each data block $\boldsymbol{b}_j$, $j = 1, 2, \ldots, m$ has a fixed size of k bytes. Clearly, $\boldsymbol{b}_j$ can be partitioned into up to k columns in terms of bytes. Whereas the conventional encoding processing works over these columns in a sequential order. In this parallel processing mode, k parallel threads are used in correspondences with the columns of $\boldsymbol{b}$, with each thread independently calculating a dot product and providing an output byte x_{ij}, $j = 1, 2, \ldots, k$. To generate a full set of n coded blocks for a data group b, a total of $n \cdot k$ threads are thus required.

With CUDA, the mapping was fairly straightforward. Fig. 1 describes the principle of mapping the encoding processing to the GPU's processors. On each SM, threads are split into warps. In each warp, the 32 parallel threads are scheduled for a group of 8 threads in 4 cycles.

Previous results [20]–[22] have shown that commodity GPUs are able to achieve the better encoding throughput as compared to mainstream multi-core CPUs.

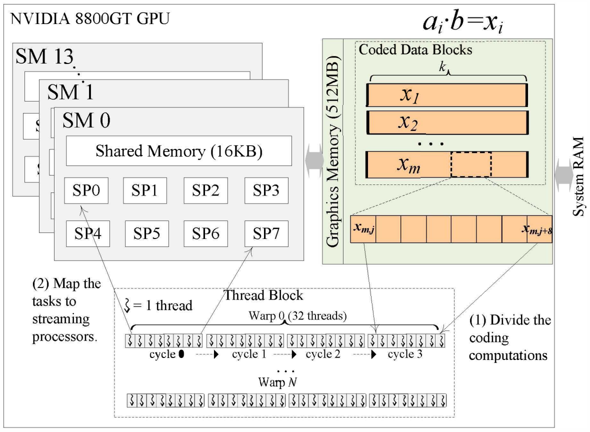


Fig. 1. Mapping of coding processing tasks to threads on GPU.

C. Our Contribution

Our work in this paper is different from the already available related results on GPU-based network coding, such as [20], [22]. While these results are aimed to simply accelerate the raw network coding throughput using GPUs, our work here provides a new point of view for the design of practical VoD systems in order to accelerate the performance of the VoD servers. The raw coding throughput only reflect the feasibility of network coding on GPUs. However, the design of VoD server involves further considerations. The server with GPU need to be structured to provide high aggregate streaming rates and fast response time. Our paper is quite orthogonal to the above work. We investigate the challenges and benefits of GPU-based network coding on VoD servers. By taking the advantages of GPU, we are ready to propose optimized VoD server architecture. The complexity and challenges of GPU-accelerated network coding have not been previously examined in the context of VoD server. Therefore, we also implement an real-world experimental prototype to form an unbiased evaluation of the VoD server's performance.

III. GPU-Accelerated Server for VoD Streaming

A. Assumptions and Design Space

A VoD streaming server with network coding needs to serve the continuous requests from its directly connected peers. Meanwhile, it also needs to perform random network coding and to packetize the coded blocks. There are many aspects that affect the performance of the VoD streaming servers with network coding, such as network bandwidth, CPU overhead, and disk I/O. Therefore, to provide high coding throughput and fast response times, there are many possible design spaces in the VoD server architecture.

In this section, we focus on the performance optimization of the VoD streaming servers by alleviating the CPU load with GPU acceleration in a cost-effective way. We believe that this part is most suitable for implementation on GPUs. For this purpose, we propose an architecture of GPU-Accelerated Servers (GAS), together with a scheduling assignment.

Of course, if disk I/O and network bandwidth are the main bottlenecks, then any method based on optimizing CPU or GPU performance is futile. In our design, we assume that disk I/O and network bandwidth are not the bottleneck. We will see in the experiments that such assumptions are realistic provided that the server is equipped with Giga-byte Ethernet interface and RAID disk.

B. The GAS Architecture

While network coding has proved effective for video streaming in the P2P networks, it is also accompanied with a large portion of the total CPU workload on the server. The network coding overhead may dramatically degrade the performance of the VoD streaming server, especially in the cases of high stream bit rate and massive scale-up of the number of blocks to be coded.

In order to reduce the server's workload, one possible approach is to employ an offline pre-coding procedure. With offline pre-coding approach, the video blocks are pre-encoded by a dedicated server, and the coded blocks are then stored on the VoD server's disk. However, this approach is usually impractical since it requires a large extra storage space with a prohibitive cost of creation and maintenance. In addition, the coding parameters are fixed once the coded blocks have been generated and stored on the disk.

In our design, we consider on-the-fly coding. In on-the-fly coding approach, the video blocks are encoded by the server upon the arrival of a request from a peer, and are then sent to the the peer. In comparison with offline pre-coding approach, on-the-fly coding approach may prove more advantageous. First, with on-the-fly coding, it is easy to configure the coding parameters, including block size and finite field, in order to meet various on-demand needs from different peers. Second, on-the-fly coding may generate as many coded blocks as needed for the original blocks, whereas the offline pre-coding approach usually stores only a limited number of coded blocks for avoiding large costs of extra storage.

With the rapid improvements in performance and programmability as well as the increasing performance-to-price ratio of GPUs, it is reasonable to equip the VoD streaming servers with off-the-shelf graphics cards to take over a portion of the CPU workload.

Fig. 2 presents a cost-effective and high-performance architecture of GPU-Accelerated Servers (GAS) for VoD Streaming. The computing capacity of GPU is exploited to alleviate the CPU workload and hence achieve a performance improvement of the VoD streaming server. The GAS architecture in Fig. 2 consists of the following basic modules:

1) *GPU/CPU Coder:* to maintain the encoding threads on the GPU and the CPU, respectively.
2) *Task Manager:* be responsible for scheduling user requests and reading data blocks from video files.
3) *Request/Data Dispatcher:* to accomodate the client requests from peers and send the coded blocks to peers, respectively.

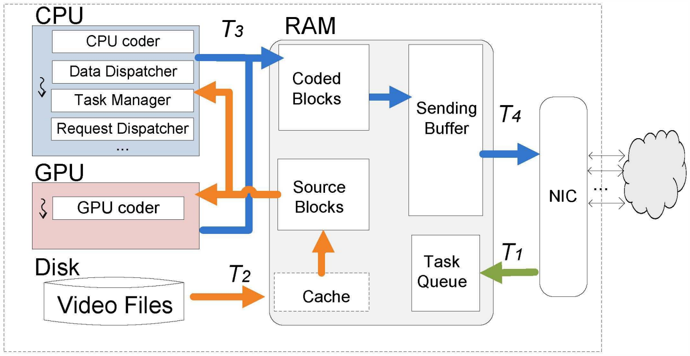


Fig. 2. The architecture of GPU-accelerated server (GAS) for VoD streaming.

```
1. Request Dispatcher
   receive user request from network;
   pass the request to Task Manager;
2. Task Manager
   parse (video file, block group id, block length);
   if (the requested block group are not in memory)
      read the blocks from disk;
   generate an encoding task;
   push the task to Task Queue;
3. GPU Coder
   wait until GPU is idle;
   read task from Task Queue;
   run the encoding task;
   pass the coded block to Data Dispatcher;
4. CPU Coder
   wait until CPU is idle;
   read task from Task Queue;
   run the encoding task;
   pass the coded block to Data Dispatcher;
5. Data Dispatcher
   pack the coded block;
   send the coded block to user;
```

Fig. 3. Pseudo-code for the working threads at the VoD streaming server.

In the GAS architecture, the video stream is divided into blocks of a fixed size k, and each block is assigned a sequence number to represent its playback order in the stream. Network coding operates over a group of m blocks as described in the previous section. The GAS architecture supports both push-based and pull-based streaming protocols. Either when a peer requests a coded block, or when the streaming server needs to send out a coded block, the on-the-fly network coding scheme in GAS is presented in the following. The request dispatcher will pass the block information to the task manager. The task manager then reads all the blocks within a coding group into memory and pushes the encoding tasks into the task queue, and the GPU and CPU coders schedule the encoding tasks from the task queue to run. Finally, the generated coded blocks are sent by the data dispatcher to the peers.

The pseudo-code for the working threads is given in Fig. 3.

C. Scheduling

In the streaming server design, one of the major objectives is to maintain a high serving capacity and provide a small response time. As one step in this direction, the GAS architecture allows the simultaneous use of CPU and GPU at the VoD Streaming servers. For the GAS architecture to be efficient, a properly designed task scheduler is required to balance the load between CPU and GPU on the server.

The load balance between CPU and GPU is indeed a multiprocessor scheduling problem, which needs to consider the following two issues:

1) Which task runs next?
2) Which processor runs the next task?

Recall that the Request Dispatcher queues all the incoming data requests in Task Queue. We assume that there are N tasks in the Task Queue J, $j_i \in J$,$i = 1, 2, \ldots, N$. We denote the CPU and GPU as processor set P, where p_0 and p_1 are CPU and GPU respectively. Suppose the amount of time delay needed for encoding is d_{ik} if task j_i is scheduled on p_k. Our goal is to schedule these tasks such that

1) no task takes longer than a given maximum time U_i (say, the video playback delay constraint),
2) the average delay of a task is as small as possible.

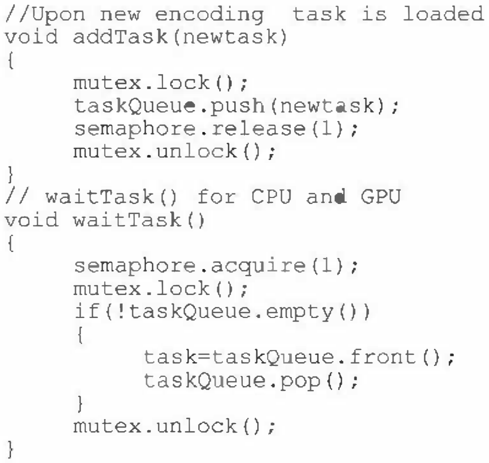
The Task Manager's objective is to a find a scheduling function $f : T \mapsto [0, \infty) \times [p_0, p_1]$ that specifies the task execution sequences on each processor.

The probem is indeed a job scheduling problem on two processors which is NP-Complete [29]. Therefore we provide heuristic approaches to address the problem. An intuitive assignment of tasks is the even assignment which distributes the coding load evenly across the processors. To this end, there are several possible processor assignment policies such as random and round robin. However, the even assignment of tasks to processors may entail an unwanted situation in which one processor is idle whereas in the meantime, another processor has a backlog since both CPU and GPU vary in computing capacity. With multiprocessors, it is paramount to keep each processor busy as much as possible in order to exploit their computing capacities and thus obtain the best performance.

In our GAS design, we provide a simple yet efficient scheduling method. The arrived coding tasks are scheduled in a first-come first-serve (FCFS) fashion. We believe that the differentiated services can easily be incorporated into the GAS design using a priority queue that is maintained for the coding tasks from different peers.

When assigning the scheduled coding tasks to processors, the GAS treats both CPU and GPU as a pooled resource, where both CPU and GPU coder threads employ self-scheduling when they are able to accept the encoding tasks. Consequently, both CPU and GPU in the GAS architecture will be kept busy once the task queue is not empty. Clearly, the self-scheduling assignment must satisfy the following two requirements:

1) Make sure that the two threads do not choose the same encoding task;



```
//Upon new encoding  task is loaded
void addTask(newtask)
{
      mutex.lock();
      taskQueue.push(newtask);
      semaphore.release(1);
      mutex.unlock();
}
// waitTask() for CPU and GPU
void waitTask()
{
      semaphore.acquire(1);
      mutex.lock();
      if(!taskQueue.empty())
      {
            task=taskQueue.front();
            taskQueue.pop();
      }
      mutex.unlock();
}
```

Fig. 4. Scheduling of the encoding task queue for both CPU and GPU.

2) Make sure that no encoding task is neglected.

To address the challenge, GAS maintains a semaphore in the encoding task queue which is given in Fig. 4. When the file manager thread has completed the loading of the encoding data into the main memory, it creates a new encoding task and pushes it into the task queue. Both CPU and GPU controller threads run in a blocking manner during the encoding process, that is, these threads can invoke the waitTask() function and schedule a new task only when they have completed the current tasks. With this scheduling algorithm, the encoding tasks dequeue in a FCFS fashion and get assigned to a current idle processor. As a result, the network coding throughput on the streaming server could be maximally accelerated by balancing the load between CPU and GPU.

In particular, priority could be trivially combined with our approach as it does not impact the proposed server architecture. If any request approaches its playback time constraint, Task Manager may prioritize this request and set it at the top of the task queue. Hence it will be scheduled immediately.

D. Analysis on GAS Design

We now investigate the important factors that account for the server performance under GAS architecture. As we saw in the previous subsection, there are four key steps for generating coded blocks in GAS:

- T_1: Request Dispatcher accommodates the client requests;
- T_2: Task Manager loads the data blocks from hard disk to the main memory;
- T_3: GPU and CPU Coder schedules coding tasks to run respectively;
- T_4: Data Dispatcher packetizes the coded blocks and sends them to the peers.

All the steps may contribute to the server's performance bottlenecks. The first step T_1 involves reading the client requests from network interface and passing the requests to Task Manager. Ignoring the latency of copying requests from the Ethernet interface to the main memory, which is almost constant (say, several μs), T_1 is only limited by the efficiency of Request Dispatcher threads. Since the server needs to sustain the demands from hundreds, even thousands, of directly connected peers, a pool of Request Dispatcher service threads is necessary to meet the simultaneous requests.

In the second step, the Task Manager needs to parse the requests and read the required data blocks into the main memory. Parsing a client request only contributes a minor part to T_2 since the data structure of the request is simple. Indeed, reading data blocks from hard disk to the main memory is costly compared with pure memory operations. The overhead includes opening the video file if it is not ready, seeking the desired block position, and loading a sequence of m data blocks into the main memory. One possible improvement is to merge the requests from different peers into one continuous disk access and to transfer the continuous blocks to the main memory as a whole. The mergence is possible only when the requests are within the same video file and different block sequences are continuous or overlapping. Another key insight that enables us to optimize T_2 is to maintain a disk I/O cache. With a proper cache replacement policy, we can allocate a space in the main memory to cache some popular blocks. Once the data blocks are hit in cache, no extra disk I/O is needed. In our experiment, we do not consider additional cache since the RAID5 SATA disk can already provide a reading throughput which is beyond the GPU and CPU's aggregate coding throughput requirement. However, such optimizations can be included to reduce disk access frequency in case of practical VoD server deployment.

The third step essentially consists of two parallel parts: GPU coding and CPU coding. The overall coding throughput is roughly the sum of the individual throughputs on the GPU and the CPU. The CPU coding efficiency could be improved by taking the advantage of modern multi-core CPUs and SIMD vector instruction sets, eg., Intel SSE2. Nevertheless, CPU coding is not our major concern here. How to improve the coding throughput on GPU has been introduced in section II-B. Specifically, we identify further improvement of coding performance on GPU through a number of optimization schemes. As shown in Fig. 1, the GPU is integrated with 512MB of global graphics memory, and meanwhile, each SM contains 16KB of on-chip shared memory. However, compared with the on-chip shared memory which is as fast as register, the global memory has a relatively longer access time which is usually ranged from 400 to 600 clock cycles. Notice that following the 8-bit byte-based encoding processing represented in (2), a thread needs first to load required bytes from global memory to shared memory prior to its running Galois operations. As discussed above, the byte-by-byte memory access may suffer from a long latency. In order to reduce the memory access latency, we use the following two optimizations:

1) Use 32-bit integer to access global memory in a single instruction.
2) Coalesce the simultaneous memory accesses of data, which lie in the same segment by different threads within a warp, into a single memory transaction.

The use of 32-bit integer, instead of 8-bit byte, avoids practically casting between integer and byte. In this case, moreover, a thread encodes 4 bytes, and thus only $k/4$ parallel threads are needed to obtain a coded block, yielding a reduction of $3/4$ in the total number of required parallel threads. For these purposes, in our implementation, both the block size k and the group size m are chosen as a multiple of 4, respectively.

We now proceed to study the forth step T_4. In the forth step, the Data Dispatcher read the coded blocks from the main memory and sends them to network interfaces. The optimization for T_4 is similar with T_1. A pool of Data Dispatcher service threads is activated to send the coded data blocks as fast as possible. Since step 1 and 4 are not our major focus, we omitted further discussion on the optimizations for network socket I/O. We simply employ a relatively large pool of threads that could saturate the Ethernet interface capacity to ensure that network I/O is not the bottleneck.

Warping up all the modules, GAS is now ready to bring high-performance network coding throughput to streaming servers by leveraging GPU's many-core computing power.

IV. Performance Evaluation

In this section, we present the performance evaluation results on both the GPU-accelerated network coding and the VoD server using the GAS architecture. All the results reported in this section have been obtained by averaging over ten runs of the real-world experiments.

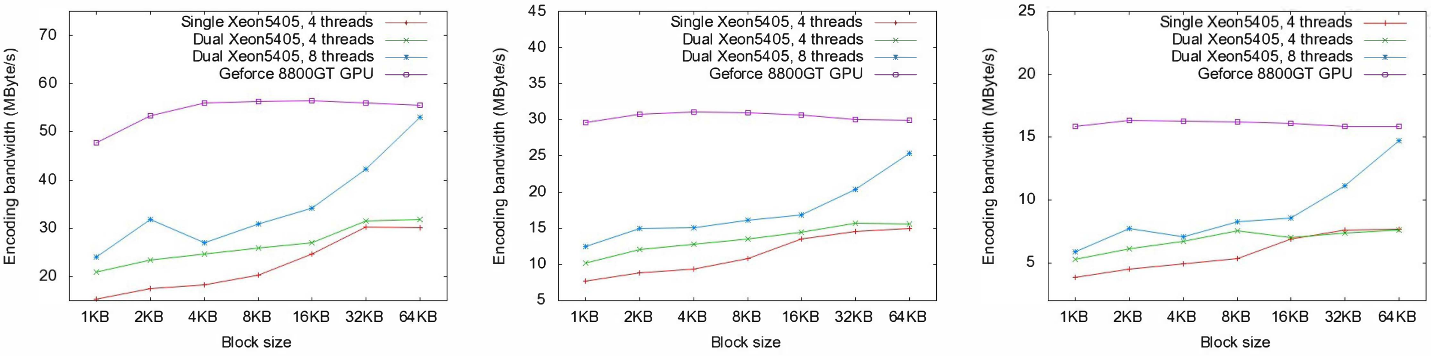
A. Experiment Environments

To evaluate the performances of GPU-accelerated network coding and GAS-based VoD streaming server, the random network coding algorithms are designed over $GF(2^8)$, and the GAS architecture is deployed on a commodity VoD streaming server. The server is equipped with two Intel Pentium Xeon 5405 2.0GHz Quad-Core processors, $2 \times$ 2GB ECC RAM, one NVIDIA Geforce 8800GT GPU, 160GB SATA RAID5 hard disk, and one Gigabit Ethernet adapter. Both the network coding and the GPU code are implemented with CUDA 2.0β.

B. Performance of GPU-Accelerated Network Coding

As an important metric to gain insights on the server performance, we first validate the raw coding bandwidth under our optimized GPU code implementation. The impact of GPU-acceleration on the network coding throughput is evaluated in terms of network coding bandwidth by using a set of experiments. For a fair comparison, a baseline C++ implementation for the CPU schemes is incorporated with the SSE2 optimization. The experiments for the CPU implementation are run with three different CPU setups: 1) Single Xeon 5405 with 4 threads; 2) Dual Xeon 5405 with 4 threads; 3) Dual Xeon 5405 with 8 threads.

In the first set of experiments, the number of data blocks to be coded is fixed at 64, 128, and 256, respectively. Fig. 5 presents a comparison of the encoding bandwidth as a function of the block size between the GPU scheme and the three CPU schemes above. Obviously, the encoding bandwidth of

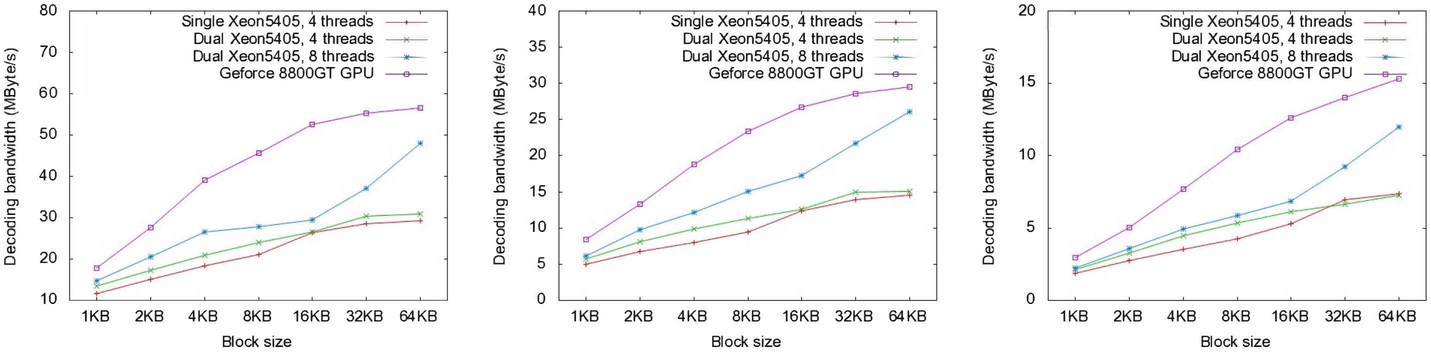


(a) block number 64

(b) block number 128

(c) block number 256

Fig. 5. Encoding bandwidth with fixed block number 64, 128 and 256



(a) block number 64

(b) block number 128

(c) block number 256

Fig. 6. Decoding bandwidth with fixed block number 64, 128 and 256

our GPU-accelerated implementation is larger than that of all the cutting-edge CPU implementations. The improvement in the encoding bandwidth comes from the use of GPU parallel computing architecture. As the block size increases, the encoding bandwidth of each CPU scheme increases significantly, especially in the case of 8 threads. However, the encoding bandwidth of the GPU-accelerated scheme varies marginally. From the three sub-figures in Fig. 5, it is also seen that the encoding bandwidth decreases with the number of data blocks. This is explained by the fact that the increase in the number of data blocks leads to an increased memory accesses and thus results in a reduction of the encoding bandwidth.

The experimental results on the decoding are shown in Fig. 6 for the comparison of the decoding bandwidth as a function of block size between the above schemes. Again, the GPU-accelerated scheme achieves the best performance compared with the CPU schemes. Clearly, the decoding bandwidth of each scheme is smaller than the corresponding encoding bandwidth due to the fact that the coefficient matrix inversion involves serial operations. This may also explain the increasing of the decoding bandwidth for the GPU-accelerated scheme with the block size. With the increase in block size, the time spent in matrix inversion is no longer dominant. As a result, the decoding bandwidth would approach that of encoding.

In conclusion, with respect to both encoding and decoding bandwidths, the GPU-accelerated scheme outperforms the CPU schemes. Therefore, it is not only feasible but also practical to accelerate network coding using GPU.

C. Performance of GAS

Having employed the GPU-based network coding in GAS, we are now ready to evaluate its performance against CPU-based implementation. In the evaluation of the VoD streaming server performance, we consider the following three metrics:

1) *Serving Capacity:* Which represents the maximum number of simultaneous connections that can be successfully supported by the server;
2) *Response Time:* Which is measured as the time difference between when the server receives a request and when the server sends out the coded blocks;
3) *CPU Usage:* Which reflects the CPU load in percentage at a given time instant.

To fully evaluate the GAS scheme with network coding performed on both CPU and GPU, we also consider other two special schemes to network coding, namely pure CPU approach and pure GPU approach. The pure CPU approach performs network coding solely on CPU, whereas the pure GPU approach performs network coding solely on GPU.

In the set of experiments designed to examine the serving capacity of the three schemes, the server provides only one video channel coded at a specified bit rate of 750 Kbps. The arrival of client requests is modeled by a discrete Poisson process with $\lambda = 2$ and $\delta = 5$s. The clients are supposed to keep on requesting data blocks once their requests have arrived.

Fig. 7a presents the experimental results on the CPU usage. Obviously, the CPU usage of GAS is between that of the two pure schemes. This can be explained by the fact that GAS leverages the CPU load with part of the network coding tasks executed on GPU.

Fig. 7b shows that the response time for all the three schemes increases with the number of simultaneous users. Moreover, GAS can accommodate nearly 500 users, whereas the other two pure schemes can only support about 200 and 350 users, respectively. These results have shown that GAS outperforms both pure schemes in terms of serving capacity in a wide range of response time beyond 100ms.

Fig. 7c presents the results of the CPU load on the streaming server in a more dynamic environment, where the video of size 160MB is served at a streaming bit rate of 750 Kbps, and where client departure behaviors have been taken into account using the churn rate pattern of real-world trace results [17]. In this set of experiments, the arrival of client requests is assumed to follow the same Poisson distribution as mentioned above. The results of Fig. 7c has been obtained by tracing the CPU usage on the server within one hour, and the CPU usage is presented as a function of time. Again, the CPU usage of the GAS scheme is reasonably acceptable.

Fig. 8 gives the results of the response time distribution captured for the three schemes. It is easily seen that GAS minimizes the startup delay which is experienced by end users. These results show that GAS are more stable and scalable in terms of response time.

One interesting question raised now is that how the number of video channels affects the server performance. Unlike previous experiments with only one video channel, the experiments below consider 30 and 200 different video channels, respectively, in order to examine the impacts of multiple video channels. In these experiments, the video popularity model follows Zipf-like distribution, with a Zipf popularity parameter of 0.5. All the videos have the same streaming bit rate of 750 Kbps, and the client requests still follow a Poisson arrival pattern with $\lambda = 2$ and $\delta = 5$s. All the clients will stay in session and keep on requesting blocks from the server.

Fig. 9 depicts the CPU usage actually measured for the three schemes. Again, GAS has a moderate CPU usage compared with the other two schemes. Fig. 10 illustrates how the response time of the VoD server evolves in the case where the server has multiple video channels. Clearly, GAS outperforms the other two schemes as the number of concurrent client requests increases with time. It indicates that a substantial improvement in the serving capacity of the streaming server can be obtained by balancing the load between CPU and GPU. For the number of clients beyond 300, GAS outperforms the pure schemes in both scenarios with 30 and 200 channels, respectively. For response time benchmark, there is a larger performance gain for GAS due to the load balancing design by GAS.

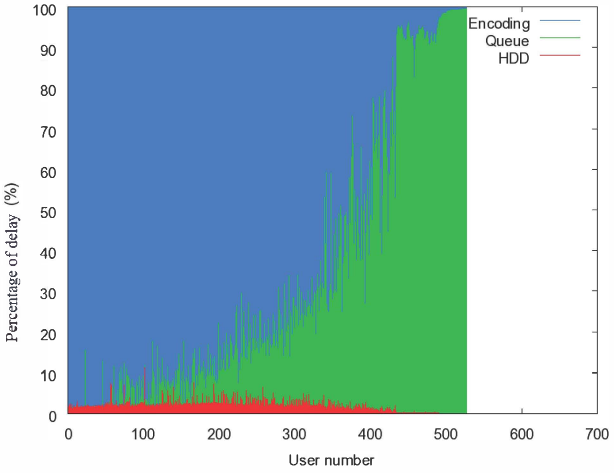
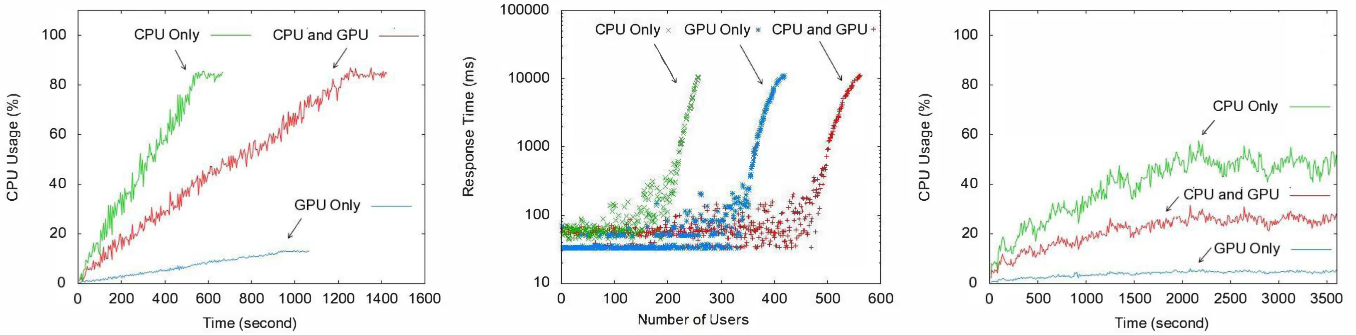


Fig. 11. The percentage of delay components in response time for GAS

As the number of concurrent users increases, the average response time for GAS will degrade. Neglecting the network I/O, the response time includes three delay components: (1)Queueing delay: waiting time in the task queue; (2)I/O delay: the time needed for loading the source blocks from hard disk drive (HDD) to main memory; (3)Encoding delay: encoding the blocks. We now investigate the main contributing factor in the increase of response time. In this set of experiments, the video's bit rate is 750 Kbps,the client requests still follow a Poisson arrival pattern with $\lambda = 2$ and $\delta = 5$s. All the clients will stay in session and keep on requesting blocks from the server. Fig. 11 reflects the average percentage of the three delay components for GAS at a given time. Initially, there's nearly no queueing delay since the number of clients is small. The encoding delay contributes to most of the response time. With the increase in user requests, the queueing delay increases significantly. From Fig. 11, we also observe that disk I/O is actually not a dominant factor in response time.

V. Conclusion

A GPU-Accelerated Server (GAS) architecture for VoD streaming has been proposed based on a GPU-accelerated network coding processing on the NVIDIA GeForce 8800GT GPU. Using real-world experiments, the performances of both the GPU-based network coding and the GAS architecture have been examined. With these results, the main idea of speeding up the VoD streaming server by exploiting the computing power of GPU has been demonstrated. As can be observed, our GAS approach achieves a significant performance gain in terms of the CPU load, the coding throughput, and the response time. This paper provides, in fact, a proof-of-concept in that by offering an attractive performance-price ratio, GPUs can be used to offload the streaming server overhead and thus circumvent the hurdle to the practical deployment of network

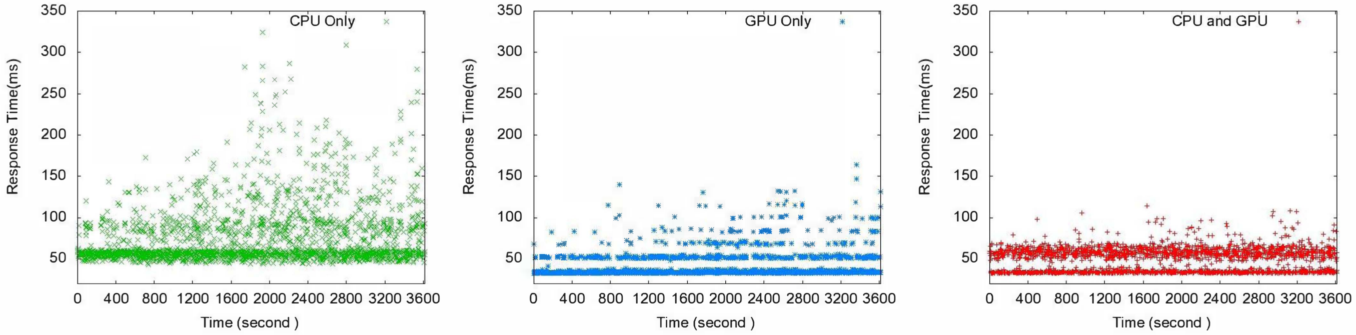


(a) CPU usage

(b) Response time

(c) CPU usage with peer departure

Fig. 7. Serving capacity of the streaming server with single video channel.

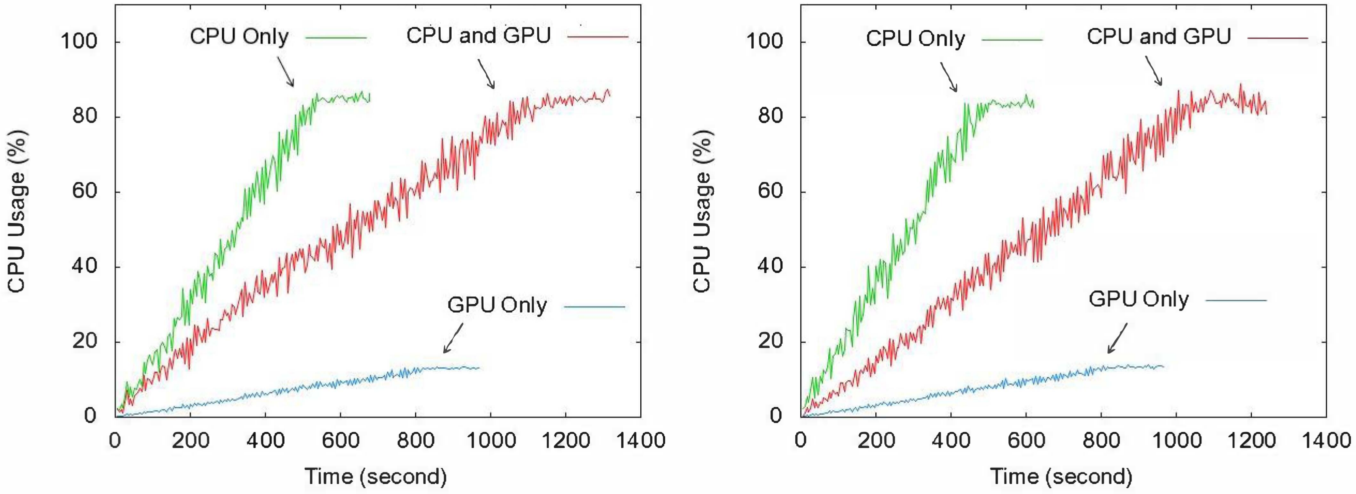


(a) CPU only

(b) GPU only

(c) CPU and GPU

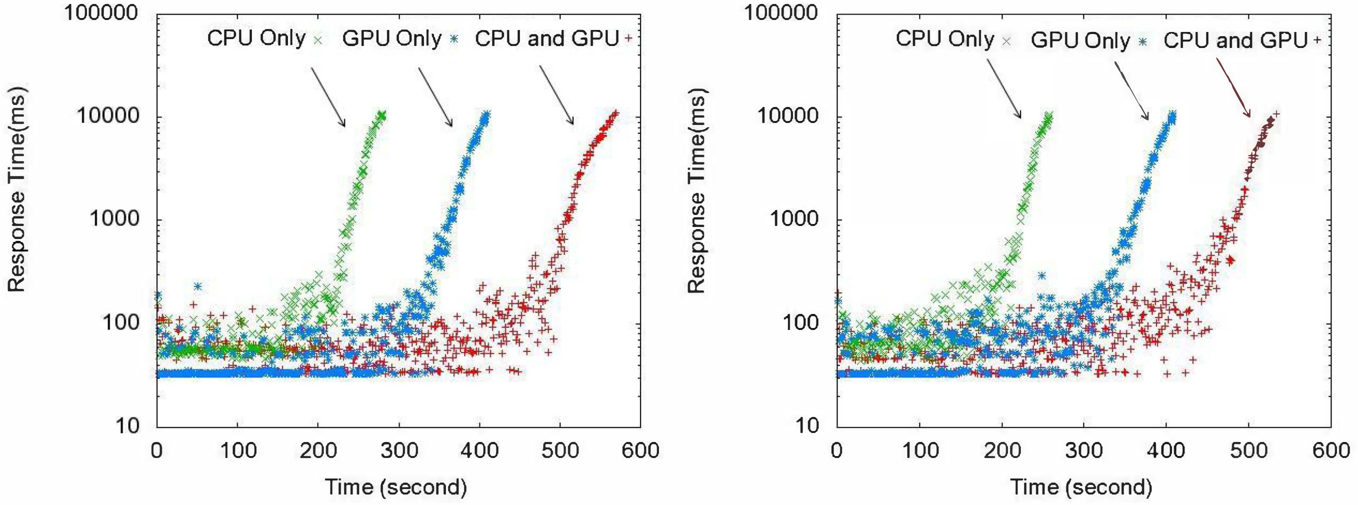
Fig. 8. Response time of GAS, pure CPU scheme, and pure GPU scheme with single video channel.



(a) 30 video channels

(b) 200 video channels

Fig. 9. CPU usage for GAS, pure CPU scheme, and pure GPU scheme with 30 and 200 video channels, respectively.



(a) 30 video channels

(b) 200 video channels

Fig. 10. Response time for GAS, pure CPU scheme, and pure GPU scheme with 30 and 200 video channels, respectively.

coding-based P2P-VoD systems. As part of our ongoing work, several issues still remain open, which include the design and performance evaluation for the application of multiple GPUs in the VoD streaming server.

ACKNOWLEDGMENT

The work was supported in part by 863 program of China under grant No. 2009AA01A348, by NSFC under grant 60803119, and by Science and Technology Commission of Shanghai Municipality under grant 08dz150010E.

REFERENCES

[1] YouTube website. [Online]. Available: http://www.youtube.com/

[2] C. Huang, J. Li, and K. W. Ross, "Can Internet video-on-demand be profitable," in *Proc. ACM SIGCOMM'07*, Kyoto, Japan, 2007, pp. 133–144.

[3] B. Cheng, X. Liu, Z. Zhang, and H. Jin,"A measurement study of a peer-to-peer video-on-demand system" in *Proc. IPTPS*, Bellevue, WA, Feb. 2007.

[4] B. Cheng, L. Stein, H. Jin, and Z. Zhang, "Towards cinematic Internet Video-on-Demand," in *Proc. ACM EuroSys 2008*, Glasgow, Scotland, Apr. 2008, pp. 109–122

[5] Y. Guo, K. Suh, J. Kurose, D. Towsley, "P2Cast: P2P patching scheme for VoD service," in *Proc. WWW'03*, Budapest, Hungary, May 2003, pp. 301–309.

[6] T. Do, K. Hua, and M. Tantaoui, "P2VoD: providing fault tolerant Video-on-Demand streaming in Peer-to-Peer environment," in *Proc. ICC'04*, Paris, France, Jun. 2004, pp. 1467–1472.

[7] K. Suh, C. Diot, J. Kurose, L. Massoulie, C. Neumann, D. Towsley, and M. Valleo, "Push-to-peer video-on-Demand system: design and evaluation," *IEEE J. Select. Areas in Commun.*, vol. 25, no. 9, pp. 1706–1716, Dec. 2007.

[8] C. Huang, J. Li, and K. Ross, "Peer-assisted VoD: making Internet video distribution cheap," in *Proc. IPTPS'07*, Redmond, WA, Feb. 2007.

[9] R. Ahlswede, N. Cai, S. Li, and R. Yeung, "Network information flow," *IEEE Trans. Inform. Theory*, vol. 46, no. 5, pp. 1204–1216, Jul. 2000.

[10] P. Chou, Y. Wu, and K. Jain, "Practical network coding," in *Proc. 41st Allerton Conf. on Communication Control and Computing*, Monticello, IL, Oct. 2003.

[11] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros, "The benefits of coding over routing in a randomized setting," in *Proc. of Int. Symp. on Information Theory ISIT'03*, Jun. 2003, p. 442.

[12] C. Gkantsidis and P. Rodriguez, "Network coding for large scale content distribution," in *Proc. IEEE INFOCOM'05*, Miami, FL, USA, Mar. 2005, pp. 2235–2245.

[13] M. Wang and B. Li, "R^2: random push with random network coding in live peer-to-peer streaming," *IEEE J. Select. Areas Commun.*, vol. 25, no. 9, pp. 1655–1666, Dec. 2007.

[14] C. Wu, B. Li, and S. Zhao, "Multi-channel live P2P streaming: refocusing on servers," in *Proc. IEEE INFOCOM'08*, Phoenix, AZ, May 2008, pp. 1355–1363.

[15] H. Chi, Q. Zhang, J. Jia, and X. Shen, "Efficient search and scheduling in P2P-based media-on-demand streaming service," *IEEE J. Select. Areas Commun.*, vol. 25, no. 1, pp. 119–130, Jan. 2007.

[16] S. Guha, S. Annapureddy, C. Gkantsidis, P. Rodriguez, and D. Gunawardena, "Exploring VoD in P2P swarming systems," in *Proc. IEEE INFOCOM'07*, Anchorage, Alaska, May 2007, pp. 2571–2575

[17] Y. Huang, T Z. J. Fu, D. M. Chiu, J. C. S. Lui, and C. Huang, "Challenges, design and analysis of a large-scale P2P VoD system," in *ACM SIGCOMM'08*, Seattle, WA, Aug. 2008, pp. 375–388.

[18] M. Wang, and B. Li, "How practical is network coding?" in *Proc. IEEE IWQoS 2006*, New Haven, CT, Jun. 2006, pp. 274–278.

[19] H. Shojania and B. Li, "Parallelized progressive network coding with hardware acceleration," in *Proc. IEEE IWQoS'07*, Evanston, IL, 2007, pp. 47–55.

[20] X.-W. Chu, K.-Y Zhao, and M. Wang, "Massively parallel network coding on GPUs," in *Proc. IEEE IPCCC'08*, Dec. 2008.

[21] X.-W. Chu, K. Zhao, and M. Wang, "Practical random linear network coding on GPUs," in *IFIP Networking 09*, Archen, Germany, May 2009.

[22] H. Shojania, B. Li, X. Wang, "Nuclei: GPU-accelerated many-core network coding," in *Proc. IEEE INFOCOM 2009*, Rio de Janeiro, Brazil, Apr. 2009

[23] H. Shojania, B. Li, "Pushing the envelope: extreme network coding on the GPU," in *Proc. IEEE ICDCS 2009*, Montreal, Canada, Jun. 2009, pp. 490–499.

[24] NVIDIA CUDA [Online] Available: http://www.nvidia.com/cuda/.

[25] ATI CTM Guide [Online] Available: http://ati.amd.com/.

[26] NVIDIA GeForce 8800 GPU Architecture Overview [Online] Available: http://www.nvidia.com

[27] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[28] Z. Liu, C. Wu, B. Li, and S. Zhao, "UUSee: large-scale operational on-demand streaming with random network coding," in *Proc. IEEE INFOCOM'10*, San Diego, USA, Mar. 2010.

[29] E. Lawler, J. Lenstra, A. Kan, and D. Shmoys, "Sequencing and scheduling: algorithms and complexity," *Elsevier*, 1993.