J. Parallel Distrib. Comput. 83 (2015) 44–57

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A holistic approach to build real-time stream processing system with GPU

Kai Zhang, Jiayu Hu, Bei Hua [*]

University of Science and Technology of China, Suzhou Institute for Advanced Study, USTC, China

HIGHLIGHTS

- We propose a GPU-centric model for real-time GPU scheduling.
- A latency-driven GPU-based framework is designed for networking systems.
- A high-performance SRTP reverse proxy is built which utilizes GPU for real-time stream processing.
- The sufficient condition for real-time GPU scheduling is studied, and a mechanism is proposed for admission control.
- CPU limits system performance even the computation heavy jobs are offloaded to GPU.

ARTICLE INFO

Article history:
Received 8 May 2014
Received in revised form
24 March 2015
Accepted 18 May 2015
Available online 27 May 2015

Keywords:
GPU
High-speed networking
Real-time
Stream processing

ABSTRACT

Stream processing needs to process huge volume of data with strict deadline requirements. These applications generally consume large amount of network bandwidth and involve compute-intensive operations. Accelerating such operations with general purpose GPU has drawn a lot of attention from both academia and industry. However, GPU has not been applied to real-time stream processing due to its programming paradigm and unpredictable latency.

In this paper, we study the problem of applying GPU to real-time processing and propose a holistic approach for building real-time stream processing system with GPU. Based on the proposed techniques, we build a GPU-accelerated SRTP reverse proxy that achieves more than 10Gbps overall throughput and guarantees low latency. Our work demonstrates that using GPU in high-speed real-time stream processing is both feasible and attractive.

1. Introduction

An important class of web applications have emerged across many domains such as Online Transaction Processing (OLTP), mobile interactive applications, cloud gaming, and Live interactive IPTV. These applications are required to process huge amount of data with strict deadlines. Specifically, low-latency processing is a critical requirement as it enables swift reaction to human or real-world events. It is believed that "the future belongs to services that respond in real time to information provided either by their users or by nonhuman sensors" [2,24]. Study [8] has shown that unpredictable latency for web applications may impact both market share and company revenue.

Furthermore, data volume is growing exponentially and market data feeds can generate hundreds of thousands of messages per second. Network bandwidth is growing from 10 to 40 Gbps, and the upcoming 80 and 100 Gbps NIC have already been conducted with active research. These web applications, which serve millions of users around the world, generally consume large network bandwidth and involve compute-intensive and memory-intensive operations, such as encryption and compression. These operations under high input data rate may exert a significant impact to server design and implementation, and lead to unpredictable latency. Therefore, stream processing systems are required to offer both highly optimized for performance and real-time guaranteed latency [28,20,27,11].

Attributed to low cost, easy programming and debugging, developing high-end network devices with general purpose off-the-shelf commodity hardware such as multi-core CPU has attracted a lot of attention from both academia and industry [36,35,5]. However, as link speed continues to grow and network services become more and more complex (e.g., require deep packet

* Corresponding author.
E-mail addresses: kay21s@mail.ustc.edu.cn (K. Zhang), hjysama@mail.ustc.edu.cn (J. Hu), bhua@ustc.edu.cn (B. Hua).

http://dx.doi.org/10.1016/j.jpdc.2015.05.002
0743-7315/© 2015 Elsevier Inc. All rights reserved.

inspection), CPU becomes incompetent at handling memory-intensive and compute-intensive tasks at very high speed. For example, RouteBricks [5] figures out that CPU is the performance bottleneck of a 10 Gbps router built on a general purpose multi-core server. In recent years, Graphics Processing Unit (GPU) has found its application in general purpose computing to accelerate memory-intensive and compute-intensive tasks with inherent parallelism [4]. As network applications usually have inherent packet-level parallelism, much work has been done to accelerate stream processing in network application using GPU [16,26,30,19].

Previous works always use GPU to improve system throughput, e.g., PacketShader [9] and SSLShader [13] demonstrate that GPU is competent to accelerate router and security gateway, respectively. However, GPU is considered unsuitable to real-time applications [6,15]. GPU relies on batch processing to take full advantage of hundreds of GPU cores; moreover, GPU processing time depends on the slowest thread. These two factors normally introduce unpredictable and long latency, making well-studied real-time CPU scheduling algorithms unsuitable to GPU. Therefore, unpredictable latency is the biggest obstacle to use GPU in real-time processing.

In this paper, we study the problem of applying GPU to real-time processing and propose a holistic approach to build GPU-accelerated stream processing system. To verify the proposed technique, we take the SRTP (Secure Real-Time Transport Protocol) reverse proxy as an example, and implement a GPU-accelerated SRTP reverse proxy that performs encryption and message authentication on RTP packet streams. The main contributions of this paper are as follows.

- The problem of using GPU to stream processing is studied, and sufficient conditions for a feasible GPU scheduling scheme that meets all the deadline requirement are given.
- A holistic approach to build a GPU accelerated stream processing system is proposed, including a GPU-centric model and a high efficient framework which takes high-speed network I/O into account.
- As a concrete and useful application, a high-performance SRTP reverse proxy is built, which is capable of encrypting and authenticating RTP packet streams at 10 Gbps+ speed.

Roadmap of the paper is as follows. Section 2 introduces background and motivation. Section 3 gives design goals of a GPU-accelerated stream processing system. Section 4 studies the problem of real-time GPU scheduling, and Section 5 discusses the GPU-centric system framework. Section 6 describes a GPU-accelerated SRTP reverse proxy based on the above techniques. Section 7 introduces experimental setup, and Section 8 reports and analyzes the experimental results. Section 9 discusses the implementation and Section 10 concludes the paper.

2. Motivation and background

2.1. Limitation of CPU

Stream processing generally involves tasks that require intensive memory access and computation, such as packet classification, encryption/decryption, data compression, and pattern matching. Over the past decades, due to imbalanced development of processor and memory technology, gap between CPU speed and memory access speed continuously enlarges. For example, an instruction usually takes 0.5 ns to complete in a 2 GHz CPU; however, one memory access may take 50–100 ns. Therefore, memory access has become the main bottleneck in modern computer architectures.

In the last decade, network speed grows from 1 to 10 Gbps. It is believed that 40 and 100 Gbps network will come in the near future. High-speed network combined with relatively slow memory access poses great challenge on CPUs. Taking 10 Gbps network as an example, a minimal Ethernet packet (64 bytes) must be processed in 67 ns, however one memory access would take 50–100 ns to complete.

Previous work claims to have achieved 20 Gbps+ RX speed and 10 Gbps forwarding speed with a single core [9,22]. However, the above performance is achieved with no extra processing overhead and the forwarded packets are just received and are still in cache. Situation is different for stream processing, where packets must pass through TCP/IP stack and usually need to be copied one or more times among different memory spaces. mTCP [14] develops a lightweight TCP/IP stack on I/O engine [12]. With an 8-core Intel Xeon E5-2690 CPU, mTCP achieves 10 Gbps throughput with 1-KB packets, and only 6 Gbps with 64-byte packets.

We measured the CPU usage in a lightweight network system that is built on a hexa-core Intel Xeon E5-2620 CPU and equipped with a 10 Gbps Ethernet port and I/O engine. Incoming packets are copied and batched in a buffer, and output packets are constructed from memory. Without any further processing, CPU usage is as high as 45%. This test shows that packet I/O, data copy, and operating system may consume a large portion of CPU cycles; therefore offloading compute-intensive tasks from CPU is necessary for high-speed network system.

2.2. Power of GPU

With CPU showing its limitations in such compute and memory intensive processing, GPU is considered to be the ideal candidate. First, different with CPU, GPU devotes most of its die area to a large array of Arithmetic Logic Units (ALUs), and executes code in SIMD (Single Instruction, Multiple Data) fashion, i.e., multiple threads share the same code path and work on multiple data simultaneously. In this way, GPU offers much higher throughput (4.5 Teraflops of single precision, Nvidia GTX TITAN) than CPU (156 Gflops, Intel Core i7-4960X).

In the aspect of hiding memory access latency, GPU performs better than CPU. Limited by resources like instruction window size and number of Miss Status Holding Registers (MSHRs), CPU can handle only a small quantity of memory misses. For example, Intel X5550 CPU is reported to handle only 4–6 memory misses [9]. However, GPU can effectively hide memory access latency with hundreds of threads; with this ability, memory stalls can be minimized or even eliminated. Moreover, GPU has much larger memory bandwidth than CPU. For example, Nvidia GTX TITAN has 288.4 GB/s memory bandwidth, while Intel Core i7-4960X only has 59.7 GB/s memory bandwidth.

Benefited from high computation power and memory bandwidth, GPU has great advantages over CPU. However, not all applications are suitable to run on GPU. Applications that can take advantage of GPU should have massive data parallelism, as application data must be divided among hundreds of cores with little or no data dependency. Fortunately, network applications usually have packet level parallelism that can be utilized. All in all, GPU is an ideal candidate to offload compute-intensive tasks from CPU in high-speed network systems.

2.3. Preliminary understanding of GPU latency

General purpose computing with GPU is divided into three stages. Firstly, input data is transferred to GPU device memory. Secondly, GPU code segment called *kernel* is launched to execute the computation. Thirdly, computation results are copied back to host memory for further use. Some important architectural features of GPU are as follows. (1) GPU's massive computing power comes from executing hundreds or thousands of lightweight (a

Table 1
GPU latency performance evaluation.

P/S	1 stream		2 streams		4 streams		8 streams	
	Thr.	Lat.	Thr.	Lat.	Thr.	Lat.	Thr.	Lat.
1 K	5.0	2.1	5.6	3.7	6.0	7.0	5.9	14.3
2 K	8.2	2.6	10.0	4.2	11.3	7.5	12.0	14.2
3 K	10.5	3.0	13.4	4.7	15.6	8.1	17.1	14.9
4 K	10.9	3.9	13.5	6.3	15.4	11.0	16.5	20.5

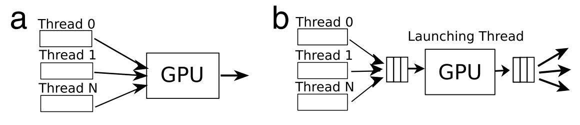


Fig. 1. Two ways to use GPU.

synonym for low performance) threads in SIMD fashion. (2) Due to the SIMD fashion, execution time of a task depends on the slowest thread. (3) Input and output data should be transferred between CPU and GPU via PCIe interface. (4) GPU kernel is non-interruptible and non-preemptible. The above four characteristics are the root cause of unpredictable and long execution latency of GPU processing.

Modern GPUs show great power in performing symmetric-key algorithms, such as AES and DES [10,33,17]. To have a preliminary understanding of GPU's processing latency, we did AES encryption and HMAC-SHA1 message authentication on RTP packets, and measured the throughput and latency with varying workloads and CUDA stream numbers. Experimental results are shown in Table 1. Column P/S represents workload that is the number of RTP packets processed by each CUDA stream. Column *Thr.* and *Lat.* represent throughput (in Gbps) and latency (in millisecond), respectively.

As shown in the table, both throughput and latency rise with the increasing number of CUDA number due to enlarged batching size. On the other hand, when CUDA stream number is fixed, both throughput and latency rise with the increasing workload of each CUDA stream, also due to the enlarged batching size. However, due to complex interactions among data batching, memory transfer, and kernel scheduling, it is difficult to predict throughput and latency when different workloads and configurations are used. Therefore, to have predictable latency and optimal throughput, GPU must be used in a controllable way.

2.4. How GPU is used

In a multi-threading networking system, GPU is generally used in two ways, as shown in Fig. 1. In Fig. 1(A), CPU threads compete to access GPU, causing unpredictable throughput and latency. Fig. 1(B) is a typical N-1-N pipeline, where two buffers (input, output) are shared among all CPU threads, and a specific CPU thread (Launching Thread) takes charge of launching GPU kernels to process packets in input buffer. However, this model has poor scalability and the two shared buffers may cause unpredictable queuing latency. Furthermore, to amortize synchronization overhead on shared buffer, receiving threads usually batch packets before inserting them into the buffer, which further introduces the unpredictability.

2.5. Related works

SSLShader [13] is the first work to use GPU as a SSL accelerator. The authors claim to achieve 9 K transactions per second for small files and 13 Gbps throughput for large files on a commodity server machine. They also measure the distribution of processing latency of SSLShader: for 1 K concurrent connections, the 50th and 99th percentiles are 39 ms and 64 ms, respectively; and for 4 K concurrent connections, the 50th and 99th percentiles are 74 ms and 518 ms, respectively. There is no mention in the paper that packets can be processed with guaranteed latency.

[32] proposes an algorithm to accelerate real-time data processing with GPU and CPU, which assigns incoming streams to CPU or GPU according to their rates and deadlines. This work is further improved with multiple GPUs in [31]. There are two major differences between our systems. On one hand, network processing overhead is huge in high-speed networking systems. For example, 70% of processing time is taken by network processing in [18], a high-speed key-value store system with optimized network processing, and MemC3 [7] suffers 7X performance degradation with network processing overhead. However, their work did not take networking overhead into consideration in system design and implementation. Consequently, the performance numbers in their paper are obtained without network processing. On the other hand, according to total stream rate, data may be processed not on time in their work. There is a mechanism in our system to reject new streams when the total rate exceeds our system capability, guaranteeing that established streams are served in real-time. By applying the admission control, our system guarantees that each packet is processed with predictable latency.

3. Design goals

Generally speaking, functions of a network device include packet receiving, application specific processing, and packet/result sending. To provide high-throughput, real-time services, every part of the device including packet I/O, data transfer, CPU and GPU processing should be considered systematically. Our goal is to study the problem of using GPU in high-throughput real-time streaming applications and propose a CPU–GPU collaboration framework that can make optimal use of CPU and GPU to achieve both high throughput and low latency. Specifically, our design goals are as follows:

1. **Hard real-time guarantee**: Packets are processed before their deadlines.
2. **High performance**: System is capable of handling multi-Gbps network traffic.
3. **Generality**: The CPU–GPU collaboration framework can be generalized to a class of stream processing applications.
4. **Scalability**: System throughput scales with the number of CPUs and GPUs.

To achieve these goals, we develop a holistic approach for building real-time stream processing systems with GPU, which includes a real-time GPU scheduling algorithm and a high-performance system framework. To overcome the limitations of GPU, we (1) adopt a GPU-centric model and a periodical GPU launching scheme to guarantee that each packet is processed before its deadline; (2) propose a zero-copy pipelining approach for efficient data movement; and (3) utilize CUDA streams to overlap computation and data transfer between CPU and GPU. The overall design is described in the following two sections.

4. Real-time GPU scheduling

4.1. Application model

We define the application model as follows. A periodic data stream S is described by a three-tuple $\langle w, d, v\rangle$, where w is the packet size, d is the maximum processing time of a packet, and v is the maximum speed of the stream. The deadline for a packet that arrives at t is $t+d$, and it should be processed to completion before $t + d$.

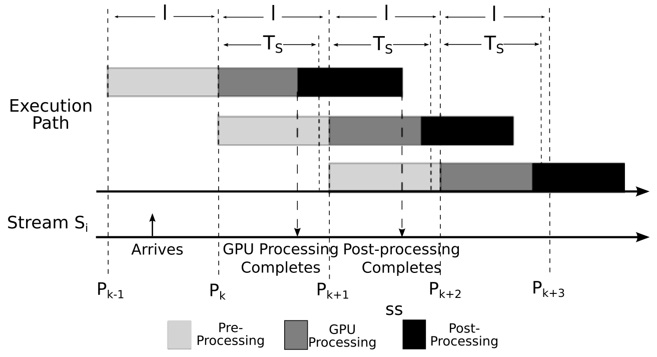


Fig. 2. Life cycle of a packet in a periodically scheduled system.

We denote a set of concurrent data streams in an application as $\mathbb{S} = \{S_1, S_2, \dots, S_n\}$, whose characteristics are described by a three-tuple $\langle \omega_S, \theta_S, \beta_S \rangle$, where $\omega_S = \max\{w_i | S_i \in \mathbb{S}\}$, $\theta_S = \min\{d_i | S_i \in \mathbb{S}\}$, $\beta_S = \sum_{i=1}^{n} v_i, S_i \in \mathbb{S}$. ω_S is the maximum packet size of the data stream set. θ_S is the maximum packet processing time of all the streams. Making each packet be processed within θ_S can guarantee that all packets of the streams meet their deadlines. β_S is the total throughput of all streams in the stream set.

4.2. Periodic kernel launch

There are basically two factors that lead to the unpredictable GPU processing time, one is the GPU kernel launch time, the other is the GPU kernel execution time. To guarantee hard real-time deadlines, we launch GPU kernels periodically to get predictable launch time. The fixed periodic launch time is called scheduling cycle in our scheduling policy.

We denote the scheduling cycle as I, and a sequence of scheduling time points as $\mathbb{P} = \{P_1, P_2, \dots | \forall k \geq 1, P_{k+1} - P_k = I\}$. At each time point P_k, packets that are accumulated in the previous cycle $[P_{k-1}, P_k)$ are handled to GPU for processing.

All packets arrived during the previous scheduling cycle must be processed to completion within current cycle to get predictable execution time. The key problem here is to guarantee the second requirement, i.e., the workloads collected in the previous cycle must be processed to completion within current cycle. Apparently, a scheduling cycle should be longer than the worst case GPU processing time in a cycle; however, GPU processing time in a cycle depends on the workloads collected in previous cycle, which in turn depends on the scheduling cycle length and traffic speed. Obviously, the scheduling cycle is the most important parameter that must be chosen properly to meet the hard real-time deadlines.

Given a stream set $\mathbb{S}$ and a scheduling cycle I, GPU processing time in a cycle is defined as the sum of data transfer time and kernel execution time. Data transfer time includes transferring input data from main memory to GPU memory, and transferring output data back from GPU memory to main memory. The worst case GPU processing time (denoted as T_S) is defined as the maximum time for data transfer and kernel execution. T_S can be measured through experiments by launching $N = \sum \lceil I \times v_i / w_i \rceil, S_i \in \mathbb{S}$ threads, with each thread processing a sequence of packets with the maximum packet size ω_S. For most operations, such as AES in CBC mode and CRC, the processing of each byte in a packet depends on the results of previous data. Therefore, each packet is suitable to be processed only by one GPU thread, and the thread with the maximum packet size may be the slowest one. Considering that the GPU kernel execution time depends on the slowest thread, the maximum packet size ω_S is reasonable to be used here.

Fig. 2 shows the lifecycle of a packet in the system, which is roughly divided into three stages: pre-processing, GPU processing, and post-processing. Pre-processing is performed when packet is received, which parses packets, extracts data to be processed, batches data in the buffer, and records stream information for forwarding. Post-processing constructs packets from GPU results and forwards the packets. Pre-processing and post-processing are performed by CPU. As we mainly concern GPU scheduling, we assume that any packet arrives during $[P_{k-1}, P_k)$ is processed to completion before $P_k + T_S$ ($T_S \leq I$), and post-processing finishes within I.

Many factors influence the GPU processing time, such as hardware architecture, driver implementation, and thread scheduling policy. Currently we find it difficult to model the relationship between I and S with a closed formulation; therefore, we choose proper value of I via experiments (Section 7.3).

4.3. Schedulability conditions

We define a stream set $\mathbb{S} = \{S_i | i = 1, \dots, n\}$ is schedulable, if and only if all packets of S_i ($S_i \in \mathbb{S}$) can be processed before their deadlines.

Theorem 1. *Given a stream set $\mathbb{S}$ characterized by $\langle \omega_S, \theta_S, \beta_S \rangle$ and a GPU scheduling cycle I, all packets belonging to $\mathbb{S}$ can be processed before their deadlines if $3 \times I + \mu < \theta_S$ and $T_S < I$, where T_S is the worst GPU processing time for the jobs accumulated in I, and μ is the pre-processing time for one packet.*

Proof. Let j' be a packet that belongs to stream $S_i \langle w_i, d_i, v_i \rangle$ and arrives at $l \in [P_{k-1}, P_k)$. Depending on the time that its pre-processing completes, j' may be handed to GPU processing in different scheduling cycles, i.e., P_k or P_{k+1}. Both of the cases are proved in the following, and Fig. 2 is used as a reference that illustrates the most common case (case 1).

(1) If j' completes pre-processing before P_k, it will be handed to GPU at P_k. After GPU processing, it is then handed to CPU for post-processing before $P_k + T_S$, and finally processed to completion before $P_k + T_S + I$. Therefore, the overall processing time t' of packet j' is $t' \leq P_k - l + T_S + I$. Given $3 \times I + \mu < \theta_S$ and $T_S < I$, we have $t' \leq P_k - l + T_S + I < I + T_S + I < 3 \times I < \theta_S \leq d_i$, which means that j' will be processed before its deadline.
(2) If j' arrives just before P_k and cannot finish pre-processing before P_k, then it will be batched with the jobs that arrive between $[P_k, P_{k+1})$ and handled to GPU at P_{k+1}. Its total processing time is $t' \leq \mu + I + T_S + I < 3 \times I + \mu < \theta_S \leq d_i$. Therefore, j' can also be processed before its deadline. □

The pre-processing time of a single packet μ is only tens or hundreds of nanoseconds, while the scheduling cycle in our system is 10–50 µs, which is 10^5 higher. For instance, if the scheduling cycle is 30 ms, the overall processing time for a packet will be less than $(3 \times 30 + \mu)$. Since we have measured in our system that the pre-processing time μ for a packet is only about 10^{-4} ms, it can be negligible. As both the GPU processing time T_S and the post-processing time are less than I, even a packet is postponed to the next cycle, its overall processing time can have a great possibility to be less than $3 \times I$.

With this scheduling policy, if we can find a value I that satisfies the conditions in Theorem 1, all packets are guaranteed to be processed to completion before their deadlines.

4.4. Maintain schedulability in runtime

In real world networks, the stream set $\mathbb{S}$ changes with new streams established or old streams terminated. Once the accumulated workload is too large to be processed within current cycle (i.e., $T_S > I$), the next batch of workload will miss its launch time, then more batches will miss their launch time, and quickly system

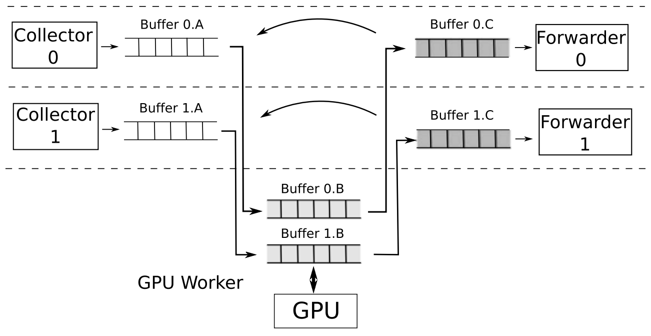


Fig. 3. The GPU-centric model.

crashes. Therefore, system workload should be controlled not to overload the system.

In order to guarantee all packets' deadlines, according to Theorem 1 the following conditions must be satisfied: $T_S < I$, and $3 \times I + \mu < \theta_S$. As T_S varies with system workload, for a given I, we define the maximum traffic not to overload the system as B_I, which is estimated through experiments.

To monitor system workload, we maintain two variables: β_S and θ_S. When a stream $S_i\langle w_i, d_i, v_i\rangle$ terminates, β_S is updated as $\beta_S = \beta_S - v_i$, and θ_S is updated only when $\theta_S = d_i$, and the stream set $\mathbb{S}$ is still schedulable. When a new stream $S_i\langle w_i, d_i, v_i\rangle$ arrives, this stream is accepted only when the following conditions are satisfied:

$$\begin{cases} \beta_S + v_i \le B_I \\ \min(\theta_S, d_i) > 3 \times I + \mu. \end{cases}$$

If stream S_i is accepted, β_S is updated as $\beta_S = \beta_S + v_i$, and θ_S is updated as $\theta_S = \min(\theta_S, d_i)$.

5. System framework for GPU real-time scheduling

Fig. 1 shows two usage modes of GPU in a networking system in previous works. In usage mode A, each thread accumulates packets in its buffer, and competes to use GPU after a batch of packets has been collected. In usage mode B, each thread inserts packets in a shared buffer (actually inserts a batch of packets each time to amortize the synchronization overhead), and a separate thread launches the GPU to process the packets in shared buffer. In both modes, the overall processing time of a packet in the system is unpredictable. The uncertainty comes from two aspects: packet buffering, and competition for the GPU (in usage mode A) or the shared buffer (in usage mode B).

We collectively call the two usage modes in Fig. 1 as CPU-centric model or throughput-centric model, as GPU is only used as a co-processor to maximize the system throughput. However, this model cannot be applied to latency-sensitive applications due to the uncertainty of packet processing time.

5.1. GPU-centric model

According to the analysis in Section 4, we propose a GPU-centric model or a latency-centric model, where GPU is used as a real-time processing engine rather than a co-processor, to provide deadline guarantees to latency-sensitive applications.

According to the life cycle definition of a packet in Fig. 2, packet processing tasks are partitioned into three stages: pre-processing, GPU processing, and post-processing. Each stage is assigned to one or more CPU threads. The GPU-centric model is shown in Fig. 3.

Collector threads do packet I/O and packet pre-processing, then batch the pre-processed packets in their buffers. GPU worker thread is the central of the model that acts as a GPU scheduler. At every launch time point, it gets packets from all the Collector threads, launches the GPU kernels to do computation, and then outputs the results to corresponding forwarders. Each Forwarder thread gets data from its buffer, and does the post-processing work, such as encapsulating packets and sending them out.

In GPU-centric model, there is no resource competition among Collectors or Forwarders, and GPU is launched periodically regardless of the workload accumulated in the previous cycle.

5.2. Zero-copy pipelining

In GPU-centric model, multiple pipelines exist, and each pipeline consists of a Collector, the shared GPU worker, and a Forwarder. To allow the pipelines to work in parallel, we assign each pipeline three buffers, each buffer to a stage, as shown in Fig. 3. In this way, memory competition is completely avoided. As the maximum workload processed in each launch is $B \times I$, the maximum buffer size allocated in the system is $3 \times B \times I$, where B is the link bandwidth.

Data copy is considered to be expensive in high-speed networking systems, therefore we do not copy packets between neighboring stages in the pipeline. Instead, buffer swapping is used to transfer data between two stages. Suppose there are N pipelines in the system, the working flow of GPU-centric model is as follows. At any time, each Collector works on its buffer to batch incoming packets. When the launch time arrives, GPU worker swaps buffer with all the N Collectors. After finishing the computation, GPU worker swaps buffers with all the N forwarders. Finally, after finishing the post-processing work, each Forwarder immediately marks its buffer as available. Since both the worst case GPU processing time T_S and the worst case post-processing time are less than I in the model, there are always buffers available to Collectors.

The buffer swapping process of a pipeline is shown in Table 2. (1) In the beginning, Collector, GPU Worker, and Forwarder are working independently on buffers a, b, and c. (2) When Forwarder finishes forwarding all packets of buffer c, it immediately marks it as available. (3) GPU worker will handle the buffer b to Forwarder after it finishes processing. If GPU finish processing before Forwarder, it will wait until Forwarder becomes available. Then the GPU worker waits until the scheduling time point arrives, and it gets buffer a from Collector and assigns the available buffer c to Collector. Now the three workers are all working on their new jobs in the buffers where no data copying is needed.

Collector does not know when time arrives and GPU Worker swaps its buffer, therefore, it has to guarantee that each query is successfully batched in the buffer. Lock is a general solution, however, this leads to locking on the critical path of each query, which violates our design principle. Instead, we adopt an opportunistic way for batching query. Before query is added into the buffer, we record the current buffer ID, and check if buffer has been swapped after completing query processing. If buffer is swapped during the procedure, Collector is not sure whether the query has been inserted successfully. Therefore, the query is added into the new buffer again. If we find the buffer is swapped, the total number of jobs in the buffer is not "+1" so that the new job, whether or not it has been processed by GPU, will not be sent by Forwarder in that batch.

5.3. Asynchronous concurrent execution

Newer CUDA devices with Computer Capability 2.0 or above provide concurrent operations for better utilization of GPU, including concurrent GPU kernel execution (execute multiple kernels concurrently), concurrent data transfer (concurrently copy from host to GPU and GPU to host), and overlapping data transfer

Table 2
Buffer swapping process.

	Buf. used by Collector	Buf. used by GPU worker	Buf. used by forwarder	Available buffer pool
1	a	b	c	
2	a	b		***c***
3	a		***b***	c
4	***c***	***a***	b	

with kernel execution. The Fermi-based Tesla M2090 supports up to 16 concurrent kernels within a single GPU.

We make good use of these features in our design. For *N*-1-*N* pipelines, GPU worker launches *N* kernels, with each kernel corresponding to one pipeline. Accompanied with dedicated buffers assigned to each pipeline, all the *N* pipelines can be executed in parallel. This design not only improves performance, but also eases programming as resource competition is eliminated.

6. SRTP reverse proxy

Data transferred between client and server generally contains personal privacy or business secrets, and it is the responsibility of service providers to prevent it from being eavesdropped. Protocols such as SRTP [29], RTMPS [1] and RTMPE [1] have been proposed and standardized to provide encryption and message authentication for transmitting streaming data on network. Unfortunately, encryption protection has not been widely adopted due to the heavy overhead of cryptographic computation. Nowadays with the popularization of multi-core processors, even a smart phone is powered by an 8-core processor, not to speak of PC. Therefore, today's end-user devices have had enough computing power to carry out tasks like on-line decryption and decompression. Challenges are on the server side, as servers have to handle thousands or millions of clients simultaneously. In this situation, compute-intensive operations pose unaffordable burden on servers, and normally high-cost specialized hardware has to be used.

To verify the effectiveness of our GPU-centric system model on latency-sensitive applications, we build a SRTP reverse proxy that provides secure streaming delivery services based on the framework described in Section 5.

6.1. Real-time network protocols

Real-time network applications usually use RTP (Real-time Transport Protocol) [23] over UDP (User Datagram Protocol) to stream data (e.g. video and audio) over IP networks. RTCP (RTP Control Protocol) is a sister protocol of RTP that does not transport any media streams itself, but provides out-of-band statistics and control information for an RTP flow. RTSP (Real Time Streaming Protocol) [21] is used to control streaming media servers, such as establishing and controlling media sessions between end points.

RTP does not provide flow encryption or authentication servers, such servers can be provided by SRTP (Secure Real-time Transport Protocol) [29,25] if needed. SRTP also has a sister protocol called SRTCP (Secure RTCP), which provides the same security-related features to RTCP. For encryption and decryption of the data flow, SRTP (together with SRTCP) utilizes AES (Advanced Encryption Standard) as the default cipher. To authenticate message and protect their integrity, HMAC-SHA1 algorithm is used.

For VoIP applications, data are encapsulated in MPEG packets. Each MPEG packets is 188 bytes, and one IP packet may contain one or several MPEG packets. Since the ethernet MTU is 1500 bytes, there can be up to 7 MPEG packets in one IP packet, which is 1370 bytes in total. With more MPEG packets, less transfer overhead (the headers). Therefore, 7 MPEG packets are generally adopted in systems with high encapsulation efficiency [3]. Table 3 shows the general structure of a MPEG-embedded RTP packet.

Table 3
MPEG-embedded RTP packet.

Ethernet header	14 bytes
IP header	20 bytes
UDP header	8 bytes
RTP header	12 bytes
Payload	$N \times 188$ bytes
Total	54 bytes overhead $+N\times$ 188 bytes

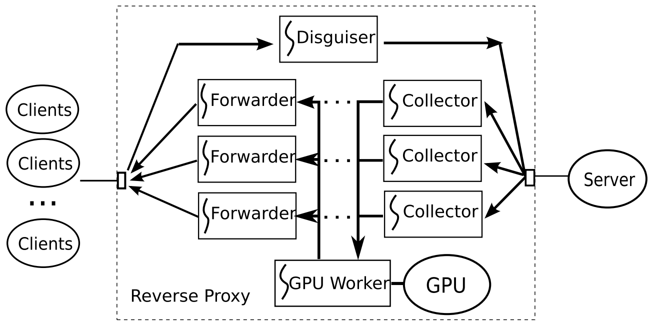


Fig. 4. Execution framework.

6.2. Role of SRTP reverse proxy

A reverse proxy generally deceives the clients as if it were the server, so that it is transparent to the clients. To this end, reverse proxy must maintain separate connections with clients and server, and record mappings between client-side connections and server-side connections in a stream table. When there are numerous active streams, maintaining large number of connections and their mappings are extremely expensive; therefore, this mechanism is poorly scalable. In order to avoid the high overhead, we make the SRTP reverse proxy work as a transparent relay. It relays client requests to server and server responses to clients without changing the network addresses of the packets. The only change is that it replaces the plaintext payload of data packets with encrypted payload, or vice versa. In this way, SRTP reverse proxy is transparent to both clients and server.

We identify the tasks of a SRTP reverse proxy as follows.

1. Performing AES decryption and HMAC-SHA1 authentication on SRTP requests from clients, and forwarding them to server.
2. Performing AES encryption and HMAC-SHA1 authentication on RTP messages from server, and forwarding them to clients.
3. Managing encryption keys for SRTP.

In streaming delivery applications, size of client requests is relatively small, while size of server responses is very large. Therefore, in our implementation we only use GPU to accelerate AES encryption and HMAC-SHA1 authentication on RTP messages, which are sent from server, and make CPU do all the other work.

6.3. Working threads in the system

The execution framework of our SRTP reverse proxy is shown in Fig. 4, which has four types of working threads. In addition to Collector, GPU worker and Forwarder that have been described in Section 5.1, Disguiser handles application-specific tasks.

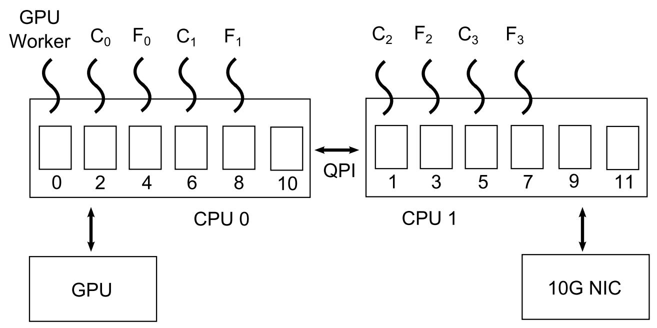


Fig. 5. System architecture and thread bonding.

Collector. Collector receives a packet from server, identifies its type, and proceeds as follows.

1. If the packet is a RTSP SETUP message, records the port number the server uses in its Global Port List, sets the transport profile to "RTP/SAVP", and forwards the message to client.
2. If the packet is a RTP message (its source port is in the Global Port List), copies the packet payload and relevant parameters like AES key and packet length into its batch buffer, and copies the packet header (IP/UDP header) into a header buffer.
3. If the packet is a RTSP DESCRIBE message, changes the "RTP/AVP" to "RTP/SAVP" in SDP media line ("m="), and adds "crypto" line in SDP protocol.
4. For other packets, directly forwards them to client.

Forwarder. Forwarder gets encrypted payloads from GPU worker or gets packets from Collector, assembles the packets if necessary, and forwards them to client.

GPU worker. GPU worker periodically launches GPU kernels to do computation on collected payloads, and delivers encrypted payloads to Forwarders.

Disguiser. Except for the above GPU-related work, Disguiser performs all the other work as follows.

1. Takes charge of key management for Collectors.
2. When receiving a RTSP SETUP message from a client, sets the transport profile to "RTP'AVP", and sends it to server.
3. When receiving a SRTP/SRTCP message from a client, decrypts the message and sends it to server.
4. For all other packets from clients, forwards them to server.

6.4. Selection of I

Many factors influence the GPU processing time, including GPU architecture, kernel scheduling policy, configurations, computation tasks, and traffic speed, etc. It is difficult to build a theoretical model to describe the relationship between scheduling cycle I and all the influencing factors (Section 2.3). Therefore, we get value I through experiments that will be laborated in Section 7.3.

7. Experimental setup

7.1. Hardware platform

We build the runtime system on a Dell R720 server, which is equipped with two Intel Xeon E5-2620 hexa-core processors running at 2.0 GHz. Each processor has a 15 MB L3 cache and an integrated memory controller installed with 8 GB memory. The two processors are connected via QuickPath Interconnect (QPI) at 7.2 GT/s. There is an NVIDIA Tesla M2090 GPU in the server, which has 512 processing cores running at 1.3 and 6 GB GDDR5 device memory. The operating system is 64-bit Ubuntu Server 11.10 with Linux kernel version 3.0.0–12. We also use another two servers in the experiments to simulate clients and server.

Table 4
GPU processing time under different configurations.

I	Stream					
	1	2	4	6	8	10
10 ms	10.5	**5.4**	**7.5**	11.3	14.1	17.2
15 ms	17.9	15.4	**8.7**	**11.0**	**14.0**	17.9
25 ms	30.3	25.4	**20.2**	**14.2**	**14.7**	**17.7**
35 ms	*42.7*	37.0	**34.8**	**27.5**	**19.8**	**18.7**
45 ms	55.3	45.2	**44.5**	**39.2**	**34.2**	**25.4**

Each machine installs an Intel 82 599 dual port 10GbE card, and we use the open source packet I/O Engine [12] as the driver for high-speed I/O. To maximize the degree of parallelism, each Collector and Forwarder is assigned a dedicated hardware queue for packet receiving and packet transmitting.

7.2. NUMA-aware thread bonding

Intel Sandy Bridge architecture has incorporated I/O hub (IOH) on the die, so each CPU has its own IOH that can connect to one or more PCIe buses. In this architecture, each I/O device has a local CPU and one or more remote CPUs. Data transferring between I/O device and remote CPUs takes longer time. Fig. 5 shows the system architecture of our Dell R720 server, where CPU0 is GPU's local CPU, and CPU1 is NIC's local CPU.

By experiments we find that random thread bonding may lead to severe system performance degradation. The optimal thread bonding scheme is shown in Fig. 5, where GPU worker is bonded to CPU0, and each Collector–Forwarder pair must be bonded to the same CPU to avoid NUMA access to packet header buffer. When hyper-threading is turned on, each physical core works as two logic cores. Therefore, there can be at most 10 Collector–Forwarder pairs in our server, with one core reserved to GPU worker thread.

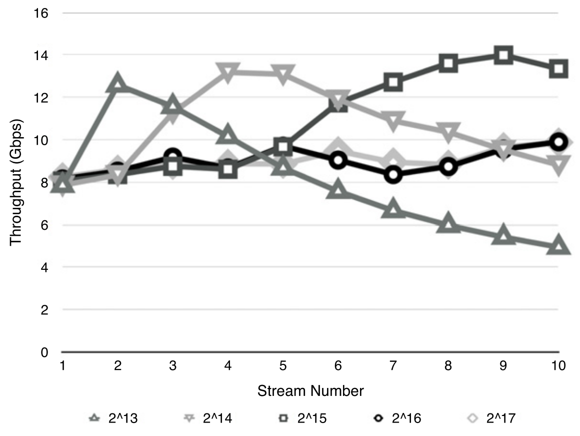
7.3. Get I through experiments

In this section, we illustrate how to get I through preliminary experiments. Stream set $\mathbb{S}\langle \omega_S, \theta_S, \beta_S \rangle$ is used as an example, where $\omega_S = 1370$ bytes, $\theta_S = 80$ ms, and $\beta_S = 10$ Gbps. To simulate multi-client situation, 1000 RTP flows are used, and the rate of each flow is 10 Mbps. Maximum packet size 1370 bytes is used as the packet size, and the workload of GPU is performing AES–CBC and HMAC-SHA1 operations.

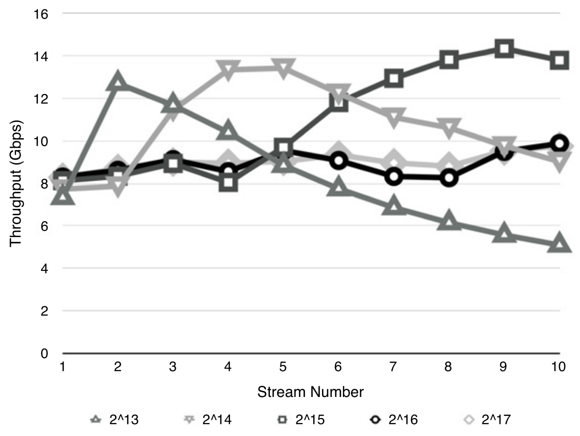
For a given I, run the proxy with different number of CUDA streams, and measure the maximum GPU processing time T_S in each scheduling cycle. If $T_S \leq I$ in all cases, a feasible configuration (i.e., the scheduling cycle I and the number of CUDA streams) is found. As for the setting of I, values less than 10 ms are not suggested. This is because that PCI-e data transfer throughput may become very low with small batch of data, which makes data transfer become bottleneck and make a significant impact to system performance. The maximum value and granularity of changing of value in the measurement experiment are determined by specific application requirement.

Part of the experimental data is shown in Table 4, where in the first column are I values, and in other columns are T_S values measured when different numbers of CUDA streams are used. Given a fixed I, the total number of packets received during a scheduling cycle is known to be $N_I = 10\,\text{Gbps} \times I/(8 \times 1370\,\text{bytes})$. If M CUDA streams are used, each CUDA stream processes N_I/M packets.

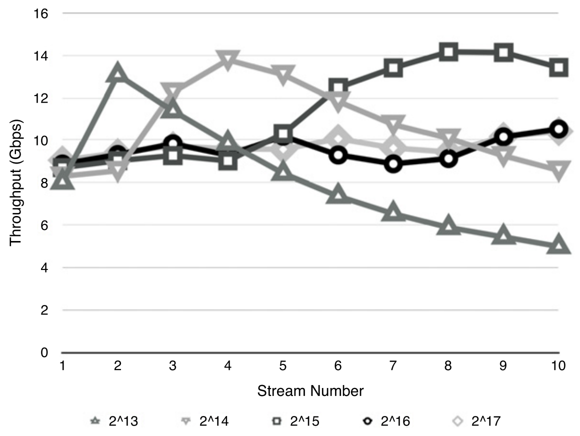
In Table 4, boldface numbers denote schedulable configurations. With this table, when a processing deadline θ_S is given, I is chosen from $[10\text{ ms}, \theta_S/3]$, and Table 4 is looked up to find a



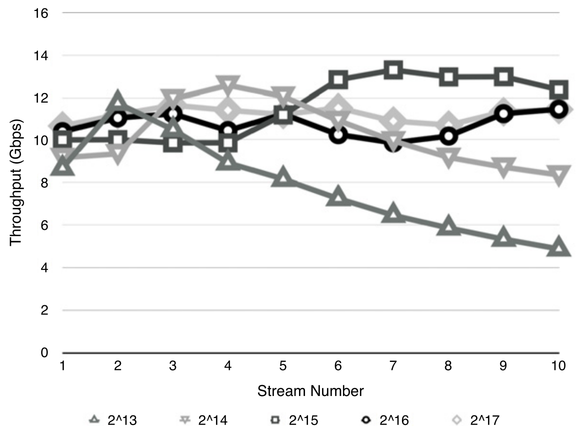
(a) GPU throughput for 1370 (54 + 7 ∗ 188) bytes packets with various batch size.



(b) GPU throughput for 994 (54 + 5 ∗ 188) bytes packets with various batch size.



(c) GPU throughput for 618 (54 + 3 ∗ 188) bytes packets with various batch size.



(d) GPU throughput for 242 (54 + 1 ∗ 188) bytes packets with various batch size.

Fig. 6. GPU performance with various batch size.

schedulable configuration. For example, if processing deadline is 46 ms, I can be chosen from [10 ms, 15.3 ms]. If $I = 15$ ms is chosen, then three schedulable configurations are found in Table 4, in which 4, 6, or 8 CUDA streams are needed, respectively.

8. Experimental evaluations

In this section, we evaluate GPU performance and the overall system performance. We identify that system performance may be influenced by the number of pipelines and scheduling cycle. The major bottleneck in current system is studied with three experiment modes.

8.1. GPU performance

8.1.1. GPU throughput evaluation

Fig. 6 illustrates GPU performance for different size of RTP packets with different batch size. We use 2^{13}, 2^{14}, 2^{15}, 2^{16}, and 2^{17} batch size to measure GPU performance. As is shown in the figure, the maximum throughput that can be achieved by GPU is 14.2 Gbps. With 2^{13}, 2^{14}, and 2^{15} batch size, GPU performance first increases to a peak and then drops. We conclude that single stream reaches maximum performance with 2^{12} (4096) jobs, and drops with smaller batching size. Therefore, more streams will not lead to higher performance when batch size is small, and this is because that small batch size will result in low PCI-e data transfer rate and low GPU utilization, which leads to low performance. If batch size is large enough, as 2^{16} and 2^{17}, GPU performance becomes stable at 8–10 Gbps. Therefore, it is critical in controlling batch size.

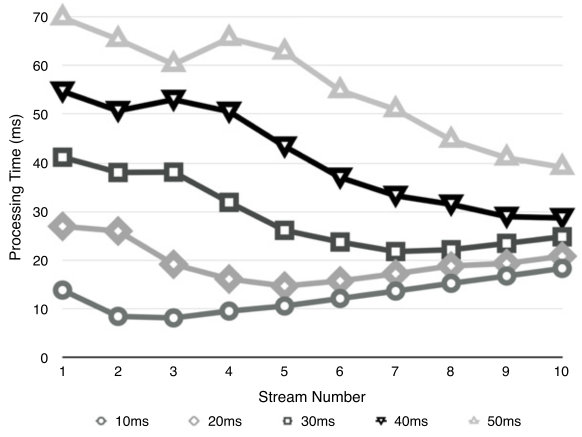
We also evaluate GPU performance with different packet size. Typical MPEG embedded RTP size 1370 bytes, 994 bytes, 618 bytes, and 242 bytes are used in our evaluation. All of them exhibit the same performance pattern. Therefore, the above conclusions can be applied to different size of input. Furthermore, we found that packet size has limited impact to GPU throughput.

8.1.2. GPU execution time evaluation

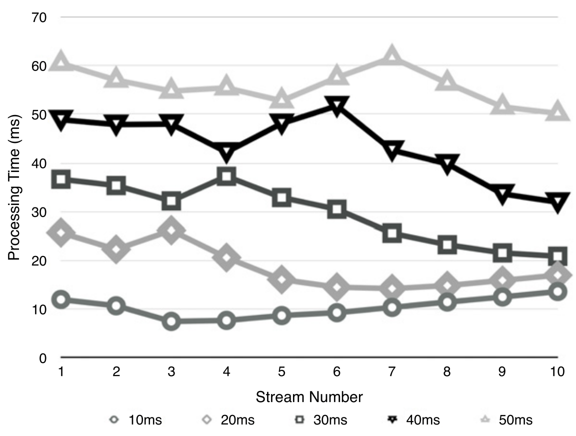
Fig. 7 evaluates GPU processing time with different batching time under 10 Gbps network input speed. As can be clearly seen from the figures, GPU processing increases correspondingly with the batching time.

Small batching time for large packets leads to much higher processing time. This is because that the number of jobs is small, and GPU computational resources cannot be fully utilized, while smaller size packets have more jobs and can use more GPU threads in processing. That is why GPU performance for small size packets stays stable with different number of streams.

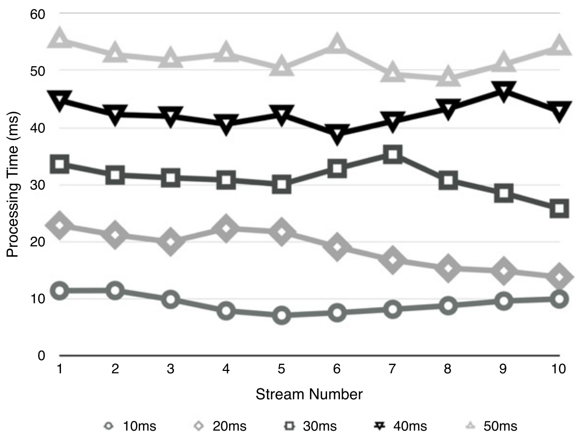
To conclude, both pipeline number (stream number) and batch size (scheduling cycle) exert a significant impact to system performance, which should given serious consideration in system setup configuration. Since the performance is determined by the number of batched jobs and number of pipeline, given fixed input speed and number of pipeline, scheduling cycle I can be adjusted to achieve the maximum performance.



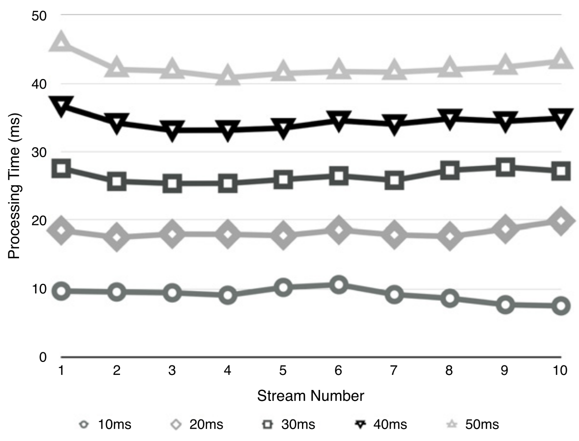
(a) GPU execution time for packet size 1370 (54 + 7 ∗ 188) bytes with different batching time under 10 Gbps.



(b) GPU execution time for packet size 994 (54 + 5 ∗ 188) bytes with different batching time under 10 Gbps.



(c) GPU execution time for packet size 618 (54 + 3 ∗ 188) bytes with different batching time under 10 Gbps.



(d) GPU execution time for packet size 242 (54 + 1 ∗ 188) bytes with different batching time under 10 Gbps.

Fig. 7. GPU execution time with various batch size.

8.2. Experimental method

To evaluate system performance and have a deep understanding of the GPU-centric pipelining, we run the system in three modes in each experiment: *Collector-only* mode, *Zero-load* mode, and *Full-load* mode.

- *Collector-only* mode evaluates the processing speed of Collectors, whose tasks include receiving packets, doing IP/UDP processing, and copying packet payloads to a specific memory buffer. Note that the destination of memory copy has been kept in CPU cache. This mode is used for evaluating the maximum I/O throughput, and checking whether system performance is blocked by I/O.
- *Zero-load* mode evaluates the pipeline throughput excluding GPU processing. In this mode, GPU worker swaps buffers with Collectors and Forwards every scheduling cycle *I*, but does not hand the workload to GPU. Different from *Collector-only* mode, the destination of memory copy is not in cache. Without GPU workload, this mode aims at evaluating our system framework and the efficiency of buffer swapping mechanism.
- *Full-load* mode evaluates the system performance as a whole. With a comparison with *Zero-load* mode, impact from GPU can be evaluated and analyzed.

We get two stream sets

- Stream set $\mathbb{S}_{\mathbb{A}}\langle \omega_S, \theta_S, \beta_S \rangle$, where $\omega_S = 1370$ bytes, $\theta_S = 80$ ms, and $\beta_S = 10$ Gbps. For the streams in the stream set $\mathbb{S}_{\mathbb{A}}$, all attributes are the same. Packet size w is fixed to 1370 bytes, $d \geq 80$ ms, and $v = 6$ Mbps.
- Stream set $\mathbb{S}_{\mathbb{B}}\langle \omega_S, \theta_S, \beta_S \rangle$, where $\omega_S = 1370$ bytes, $\theta_S = 80$ ms, and $\beta_S = 10$ Gbps. For the streams in the stream set $\mathbb{S}_{\mathbb{A}}$, the attributes are not the same. Packet size w is evenly distributed among the 7 MPEG embedded RTP packet sizes, i.e., 242, 430, 618, 806, 994, 1182, and 1370 bytes. $d \geq 80$ ms, and v follows normal distribution $v \sim \mathcal{N}(\mu, \sigma^2)$, where $\mu = 6$ Mbps, $\sigma = 1$.

Stream set $\mathbb{S}_{\mathbb{B}}$ is generated in random, and the average packet length varies each time, since streams may be set with various packet sizes and speed. For the experiments that running with stream set $\mathbb{S}_{\mathbb{B}}$, the numbers are average results with 10 runs. In the following experiments, we use the two stream sets to evaluate system throughput, and use stream set $\mathbb{S}_{\mathbb{A}}$ in measuring system latency.

8.3. Throughput evaluation

In this section, we evaluate and analysis system throughput with 10 Gbps and 20 Gbps input, respectively. With the three modes mentioned above, we are trying to identify the bottleneck in our system. To measure the impact from input stream characteristics and I configurations, different I (15 and 25 ms), and different stream set ($\mathbb{S}_{\mathbb{A}}$ and $\mathbb{S}_{\mathbb{B}}$) are used in evaluation. Less than 30 ms scheduling cycle are adopted in all experiments, because we found the overall processing latency would be too high with $I > 30$ ms,

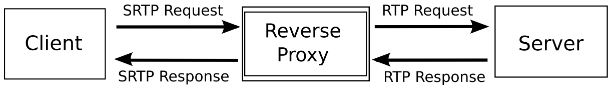


Fig. 8. Test network with 10 Gbps input.

which will be over 90 ms. Furthermore, as shown in previous section, larger batch size will not lead to higher performance.

8.3.1. Experiment in 10 Gbps environment

Fig. 8 shows the experiment environment that consists of three machines. One acts as SRTP reverse proxy, another two act as server and client, respectively. Server and client are connected to the SRTP reverse proxy through separate 10 Gbps links. Server sends original RTP packets to SRTP reverse proxy, which performs encryption and message authentication on the payloads, and then forwards the encrypted packets to client.

Experiments are conducted by running the system in three modes, each with different number of pipelines. Fig. 9 shows system's throughput with stream set $\mathbb{S}_{\mathbb{A}}$ and $\mathbb{S}_{\mathbb{B}}$, respectively. The horizontal axis denotes the number of pipelines used, and the vertical axis shows the processing speed in Gbps.

As shown in the figure, system throughput (in *Full-load* mode) rises with the increasing number of pipelines, and almost reaches 10 Gbps when six or more pipelines are used. When fewer pipelines are used, the processing speed of *Zero-load* mode is significantly lower than that of *Collector-only* mode, indicating that CPU are not capable of processing that high speed when data copying buffer is not in cache. However, with more cores/pipelines, system performance becomes the same with *Zero-load* mode, which means the overhead from CPU memory copy has been alleviated. In *Full-load* mode, performance is almost the same with *Zero-load* mode, indicating that system performance is mainly blocked by CPU, while GPU is competent in handling 10 Gbps throughput.

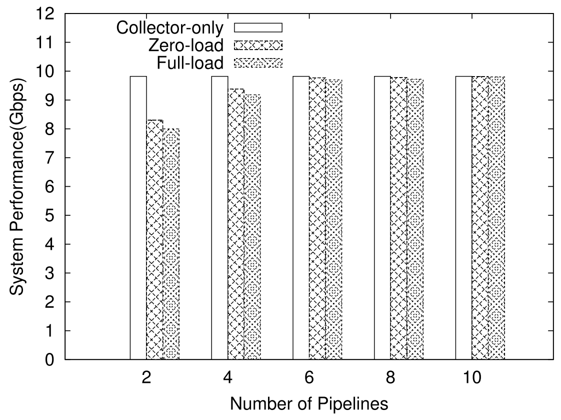
The performance with stream set $\mathbb{S}_{\mathbb{A}}$ and $\mathbb{S}_{\mathbb{B}}$ are almost the same with all the three modes, showing that our system is free of impact from the workload variations. Since CPU, not GPU, blocks our system performance, different scheduling cycles 15 and 25 ms also show same performance. If processing speed of CPU not exceeds that of GPU, these configurations will have limited influence on our system's performance.

According to this experiment and Fig. 6, we conclude that GPU is competent in handling 10 Gbps traffic, especially when there are enough CPU cores and CUDA streams.

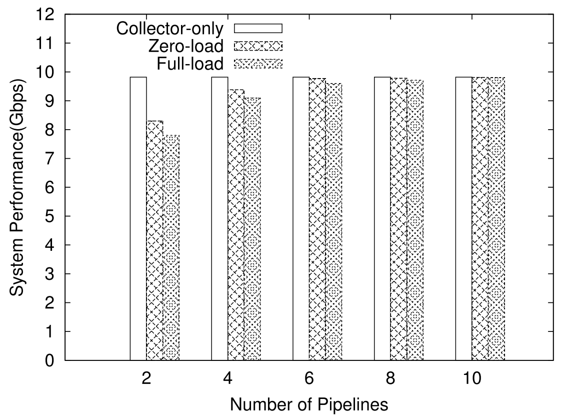
8.3.2. Maximum system throughput

To explore the maximum system throughput, we build the testing network as shown in Fig. 10. Two machines are connected to the SRTP reverse proxy through separate 10 Gbps links, but each machine acts as both server and its client. For example, server-A sends RTP packets to SRTP reverse proxy, which after doing the encryption and message authentication, sends the encrypted packets to client-A through the same NIC port. Therefore, the SRTP reverse proxy receives 2 × 10 Gbps traffic.

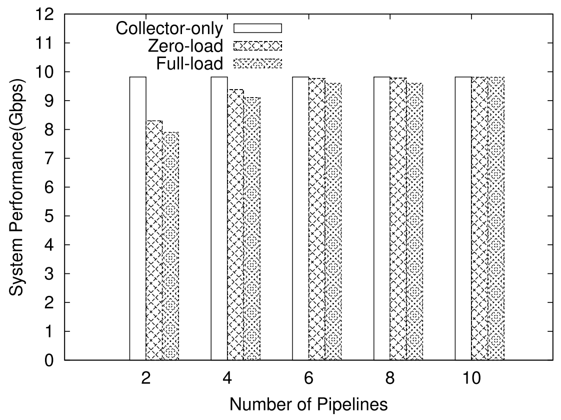
Like the experiments in Section 8.3.1, system runs in three modes, each with different number of pipelines. Fig. 11 shows the processing speed in each scenario. The horizontal axis denotes the number of pipelines, and the vertical axis shows the processing speed in Gbps.



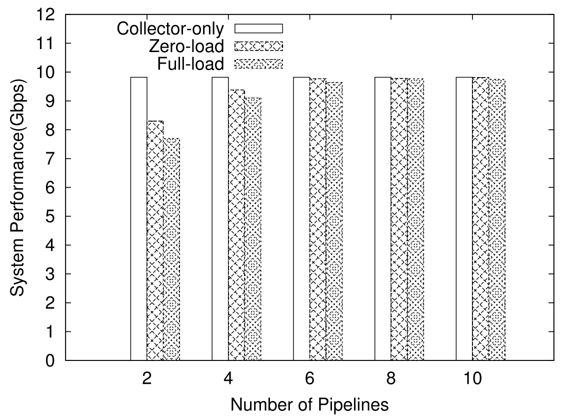
(a) Input stream set $\mathbb{S}_{\mathbb{A}}$, $I = 25$ ms.



(b) Input stream set $\mathbb{S}_{\mathbb{B}}$, $I = 25$ ms.



(c) Input stream set $\mathbb{S}_{\mathbb{A}}$, $I = 15$ ms.



(d) Input stream set $\mathbb{S}_{\mathbb{B}}$, $I = 15$ ms.

Fig. 9. System performance with 10 Gbps input.

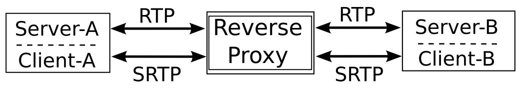


Fig. 10. Test network with 2 × 10 Gbps input.

As expected, system throughput (in *Full-load* mode) rises with the increasing number of pipelines, and same with the 10 Gbps case, performance for both stream sets are almost the same. In *Zero-load* mode, input speed may reach 16.8 Gbps with 10 *Collectors*. With 8 or 10 pipelines, system throughput reaches 13.3 Gbps, which is the maximum speed in our experiments. In all the circumstances, network I/O (*Collector-only* mode) is much faster than the processing speed of *Zero-load* mode and *Full-load* mode, and the later two modes have similar processing speeds. This result indicates that, same with the 10 Gbps experiments, the memory copy operations in CPU is the major overhead that limits our system's performance.

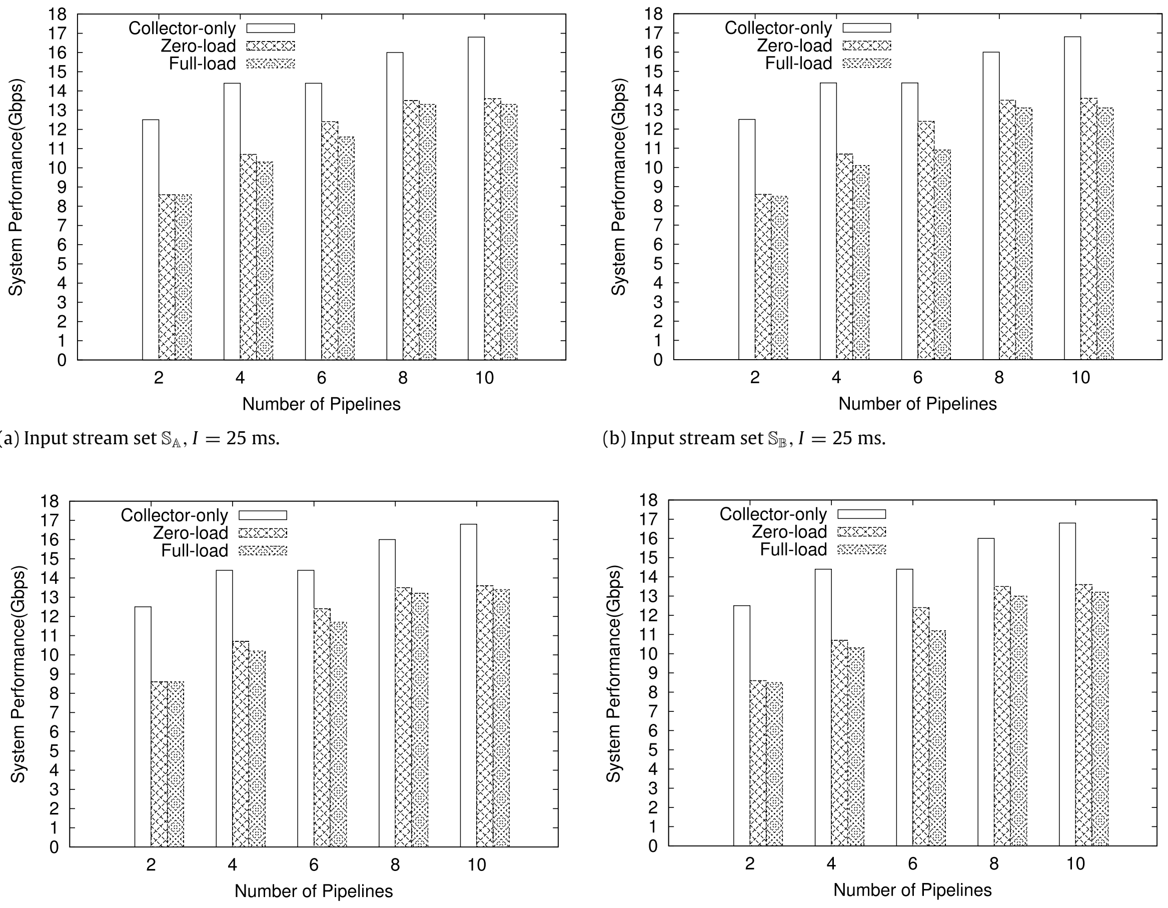
It is worth noting that with 6 pipelines, GPU becomes the bottleneck and lots of deadline misses have been reported in our system if our admission control technique is not used. After applying the admission control technique as described in Section 4.4, system becomes capable of handling such situations.

In summary, we find that neither NIC I/O nor GPU is system bottleneck, but memory copy operation in CPU slows down the framework. Again, as in most cases CPU is the bottleneck that limits system performance (*Zero-load*), system configurations such as I and input stream characteristics have limited influence on system performance.

8.4. *Evaluation of system latency*

This section evaluates the system latency of SRTP reverse proxy, which is measured as the elapsed time between the server sending a message and the client receiving the message. In fact, this time is larger than the actual system latency of SRTP reverse proxy. However, to ease the programming and avoid measurement overhead on SRTP reverse proxy, we use the testing network shown in Fig. 10, and measure the latency at the server/client machine. The server program keeps sending a sequence of RTP packets to the SRTP reverse proxy; each RTP packet carries a unique RTP sequence number and a timestamp. Once the client program receives a packet, it records the packet arriving time, as well as the RTP sequence number and timestamp in the packet header. After the test program stops, a separate program calculates the latency of each packet. Since server and client reside on the same machine, no system clock synchronization is needed.

Intuitively, scheduling cycle and system workload have significant influence on system latency. We first evaluate the impact of scheduling cycle on system latency. We fix the traffic speed to 2 Gbps, 6 Gbps and 10 Gbps, respectively; With each traffic speed, we change the scheduling cycle I, and measure the system latency. Fig. 12 shows the cumulative distribution function (CDF) on system latency. Roughly speaking, system latency rises with the increase of scheduling cycle, just as we expected. However, there is an exception in Fig. 12(c), where system latency with $I = 20$ ms is greater than that with $I = 25$ and $I = 30$ ms. We guess this is caused by some unpredictable interactions with, for example, architectural factors.



(a) Input stream set $\mathbb{S}_{\mathbb{A}}$, $I = 25$ ms.
(b) Input stream set $\mathbb{S}_{\mathbb{B}}$, $I = 25$ ms.
(c) Input stream set $\mathbb{S}_{\mathbb{A}}$, $I = 15$ ms.
(d) Input stream set $\mathbb{S}_{\mathbb{B}}$, $I = 15$ ms.

Fig. 11. System performance with 2 × 10 Gbps input.

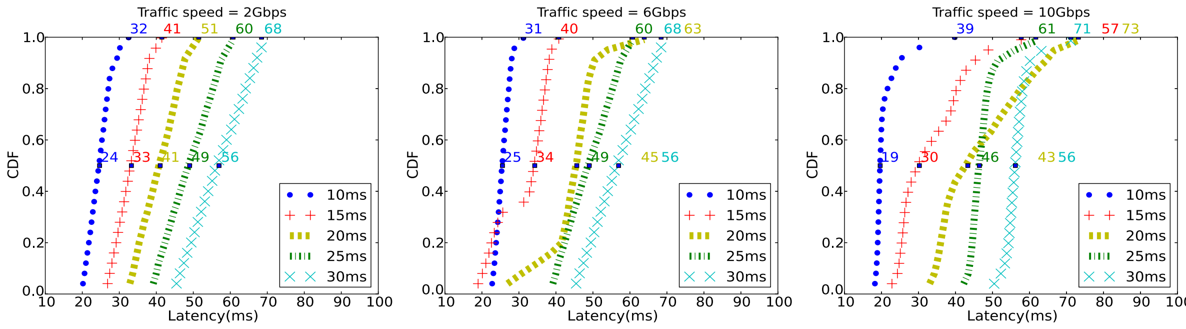


Fig. 12. Impact of scheduling cycle on system latency.

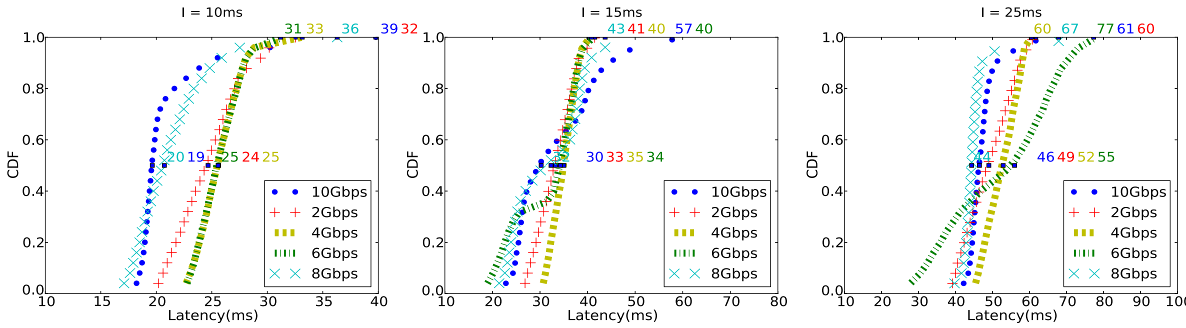


Fig. 13. Impact of traffic speed on system latency.

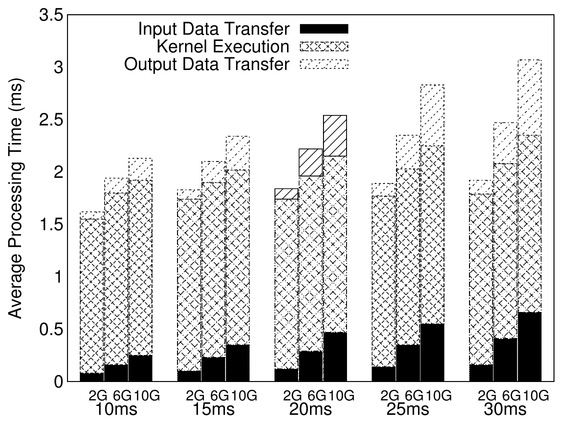


Fig. 14. Decomposition of GPU processing time.

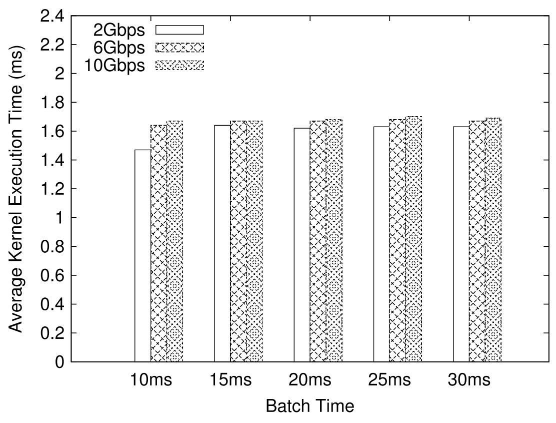


Fig. 15. GPU kernel execution time.

We then evaluate the impact of system workload on latency. We fix the scheduling cycle I to 10 ms, 15 ms, and 25 ms, respectively; then with each scheduling cycle, we change the traffic speed, and measure the system latency. Fig. 13 shows the experimental results. Out of our expectation, system latency does not rise monotonically with the increase of traffic speed. This validates our conjecture that it is difficult to build a theoretical model to deduce a proper I from given conditions. It is also shown that the maximum system latency is relatively stable even with dramatic traffic fluctuation, indicating that our system can work stably in real world networks.

8.5. Decomposition of GPU processing time

GPU processing time includes data input, kernel execution, and data output. We measure the average time of each stage in a single stream with different scheduling cycles and traffic speeds. In the experiment, stream set $\mathbb{S}_{\mathbb{A}}$ is used as the input, and system is configured with 10 pipelines. The numbers measured in the figure are single stream processing latency, which are the average of 10 CUDA streams.

The decomposition of GPU processing time is shown in Fig. 14. As expected, data transfer time (both input and output) grows with the increasing data size. However, the kernel execution time, shown in Fig. 15, stays relatively stable at around 1.6–1.7 ms. L_{kernel} takes the largest portion of the GPU processing time and remains relatively stable This is because the worst case single thread throughput is limited by the memory access latency. Although GPU schedules dynamically to hide single thread memory access latency, each thread's worst case processing time remains stable before reaching GPU's maximum computation throughput.

With 10 pipelines, as data transfer is overlapped with kernel execution, the overall GPU processing time is calculated as $L_{input} + 10 \times L_{kernel} + L_{output}$. It also explains the stability of system latency when traffic speed fluctuates as shown in Fig. 13.

9. Experiences and discussion

In this section, we summarize and discuss the pros and cons of applying GPU in real-time stream processing.

The GPU-centric model. In GPU-centric model, system is organized around a GPU that is used as a real-time processing engine rather than just a co-processor. In this model, CPU only receives and forwards packets, and does preparation jobs for GPU. GPU is scheduled periodically to process batched workload with the scheduling cycle chosen carefully to guarantee all packets' processing deadlines.

Selecting a proper scheduling cycle seems to be the most ambiguous part in our system implementation. With given conditions, one must first do a trial run to find a feasible configuration that includes the scheduling cycle and number of pipelines (Section 7). However, as it is difficult to build a theoretical model to get such a schedulable scheme, currently we find this step inevitable.

Applicable to other GPU-accelerated stream processing systems. There are many compute-intensive or memory-intensive operations in stream processing system that are extremely suitable for GPU, such as pattern matching [16,26,30] and packet classification [19,9]. GPU is even used to accelerate data base queries [34]. From the point of view of GPU, stream processing tasks can be divided into three stages: pre-GPU stage, GPU stage, and post-GPU stage. To design a GPU-accelerated stream processing system, one first figures out the system work, then partitions it into three stages, and finally applies the GPU-centric model to the system. For complicated functions, extra worker threads can be added to the system, such as the Disguiser thread in SRTP reverse proxy.

Easily scales up with more CPU cores or more GPUs. GPU-centric model organizes the system into pipelines, which makes the system easily scale up with more CPU cores (or CPUs) or more GPUs. For example, if pre-GPU stage or post-GPU stage becomes bottleneck, more CPU cores (CPUs) can be used to enlarge the number of pipelines. If GPU becomes bottleneck, more GPUs can be added into the system. By assigning the CPU workers to different GPUs, their jobs can be naturally partitioned and dispatched among GPUs. This can be simply implemented by launching different CUDA streams on different GPUs.

10. Conclusions

In this paper, applying GPU to real-time stream processing is studied. A holistic approach to build a GPU-accelerated stream processing system is proposed, which includes a real-time GPU scheduling scheme, a GPU-centric model, and a pipelined system framework. To verify our techniques, a GPU-accelerated SRTP reverse proxy is built on the proposed approach with commodity general-purpose CPU and GPU hardwares. It achieves more than 10 Gbps encryption and message authentication throughput, and meanwhile still offers predictable processing latency. Our work demonstrates that using GPU in high-speed real-time stream processing is feasible and attractive.

In the future, we will port our platform to APU architectures. The newly released APU platform allows zero-copy between CPU and the integrated GPU, which significantly mitigates data transfer overhead in current GPU systems. Furthermore, a programming API is planned to be provided on our real-time processing framework which is designed for facilitating building more real-time applications. Aiming at building an open platform for GPU-accelerated real-time stream processing, multi-GPU support will also be studied in the future.

References

[1] Adobe flash access overview on protected streaming, http://www.adobe.com/products/flashmediaserver/pdfs/flashaccess_wp_protectstreaming.pdf.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, M. Zaharia, Above the clouds: A berkeley view of cloud computing, Tech. Rep., UC Berkeley, 2009.

[3] M. Burza, J. Kang, P.V.D. Stok, Adaptive streaming of MPEG-based audio/video content over wireless networks, J. Multimed. 2 (2) (2007).

[4] S. Chey, J. Liz, J.W. Sheaffery, K. Skadrony, J. Lach, Accelerating compute-intensive applications with GPUs and fpgas, in: Symposium on Application Specific Processors, 2008.

[5] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, S. Ratnasamy, Routebricks: Exploiting parallelism to scale software routers, in: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2009.

[6] G.A. Elliott, J.H. Anderson, Real-world constraints of GPUs in real-time systems, in: IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2011.

[7] B. Fan, D.G. Andersen, M. Kaminsky, MemC3: Compact and concurrent memcache with dumber caching and smarter hashing, in: Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation, (NSDI 13), USENIX, Lombard, IL, 2013, pp. 371–384. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

[8] J. Hamilton, The cost of latency, http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx.

[9] S. Han, K. Jang, K. Park, S. Moon, PacketShader: A GPU-accelerated software router, in: Proceedings of the SIGCOMM'10 Conference, 2010.

[10] O. Harrison, J. Waldron, Practical symmetric key cryptography on modern graphics hardware, in: USENIX Security Symposium, 2008.

[11] J.-H. Hwang, U. Çetintemel, S. Zdonik, Fast and highly-available stream processing over wide area networks, in: Proceedings of the 24th International Conference on Data Engineering, ICDE, Washington, DC, 2008, pp. 404–813.

[12] IOEngine, http://shader.kaist.edu/packetshader/io_engine/.

[13] K. Jang, S. Han, S. Han, S. Moon, K. Park, SSLShader: cheap SSL acceleration with commodity processors, in: Proceedings of the 8th USENIX Conference on Networked Systems and Implementation, NSDI'11, USENIX Association, 2011.

[14] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, K. Park, mTCP: a highly scalable user-level TCP stack for multicore systems, in: 11th USENIX Symposium on Networked Systems Design and Implementation, (NSDI 14), USENIX Association, Seattle, WA, 2014, pp. 489–502. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[15] S. Kato, K. Lakshmanan, R. Rajkumar, Y. Ishikawa, TimeGraph: GPU scheduling for real-time multi-tasking environments, in: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'11, USENIX Association, Berkeley, CA, USA, 2011, URL: http://dl.acm.org/citation.cfm?id=2002181.2002183.

[16] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, J.-M. Shyu, Accelerating string matching using multi-threaded algorithm on GPU, in: IEEE Globecom, 2010, pp. 1–5.

[17] S. Manavski, CUDA compatible GPU as an efficient hardware accelerator for aes cryptography, in: IEEE International Conference on Signal Processing and Communications, 2007, pp. 65–68.

[18] Z. Metreveli, N. Zeldovich, M.F. Kaashoek, CPHASH: A cache-partitioned hash table, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'12, ACM, New York, NY, USA, 2012, http://dx.doi.org/10.1145/2145816.2145874. URL: http://doi.acm.org/10.1145/2145816.2145874.

[19] S. Mu, X. Zhang, N. Zhang, J. Lu, Y.S. Deng, S. Zhang, IP routing processing with graphic processors, in: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 93–98.

[20] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, Network-aware operator placement for stream-processing systems, in: Proceedings of the 22th International Conference on Data Engineering, ICDE, 2006, pp. 49–60.

[21] Real time streaming protocol, http://www.ietf.org/rfc/rfc2326.txt.

[22] L. Rizzo, Netmap: A novel framework for fast packet I/O, in: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12, USENIX Association, Berkeley, CA, USA, 2012, p. 9. URL: http://dl.acm.org/citation.cfm?id=2342821.2342830.

[23] RTP: A transport protocol for real-time applications, RFC 3550, http://tools.ietf.org/html/rfc3550.

[24] L. Siegele, Let It Rise: A Special Report on Corporate It, 2008.

[25] Skype, Tls and srtp for skype connect technical datasheet, https://support.skype.com/resources/sites/SKYPE/content/live/DOCUMENTS/0/DO14/en_US/skype-connect-technical-datasheet.pdf.

[26] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, C. Estan, Evaluating GPUs for network packet signature matching, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2009.
[27] U. Srivastava, K. Munagala, J. Widom, Operator placement for in-network stream query processing, in: Proc. the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principle of Database Systems, PODS, 2005, pp. 250–258.
[28] M. Stonebraker, U. Çetintemel, S. Zdonik, The 8 requirements of real-time stream processing, SIGMOD Rec. 34 (4) (2005) 42–47.
[29] The secure real-time transport protocol (SRTP), RFC 3711, http://tools.ietf.org/html/rfc3711.
[30] G. Vasiliadis, S. Antonatos, M. Polychronakis, E.P. Markatos, S. Ioannidis, Gnort: High performance network intrusion detection using graphics processors, in: 11th International Symposium on Recent Advances in Intrusion Detection, pp. 116–134.
[31] U. Verner, A. Schuster, A. Mendelson, M. Silberstein, Scheduling processing of real-time data streams on heterogeneous multi-GPU systems, in: Proceedings of the 5th Annual International Systems and Storage Conference, SYSTOR'11, ACM, 2012.
[32] U. Verner, A. Schuster, M. Silberstein, Processing data streams with hard real-time constraints on heterogeneous systems, in: Proceedings of the 25th International Conference on Supercomputing, ICS'11, ACM, 2011.
[33] J. Yang, J. Goodman, Symmetric key cryptography on modern graphics hardware, in: ASIACRYPTO, 2007.
[34] Y. Yuan, R. Lee, X. Zhang, The Yin and Yang of processing data warehousing queries on GPU devices, Proc. VLDB Endow. 6 (10) (2013) 817–828. http://dx.doi.org/10.14778/2536206.2536210.
[35] K. Zhang, J. Wang, B. Hua, L. Lu, Dhash: A cache-friendly tcp lookup algorithm for fast network processing, in: The 38th IEEE Conference on Local Computer Networks, LCN, 2013.
[36] K. Zhang, J. Wang, B. Hua, X. Tang, Building high-performance application protocol parsers on multi-core architectures, in: Proceedings of 17th IEEE International Conference on Parallel and Distributed System, ICPADS, 2011.

Kai Zhang is currently a Ph.D. candidate in Computer Science and Technology in University of Science and Technology of China. He received his B.E. degree in Computer Science and Technology from Qingdao University. His research interests include computer networks, parallel computing, and big data.

Jiayu Hu received her bachelor's degree in Computer Science and Technology from Sichuan University. She is currently a postgraduate student in Computer Science and Technology from the University of Science and Technology of China. Her research interests include computer networks and parallel computing.

Bei Hua is a professor at School of Computer Science and Technology, University of Science and Technology of China. She got her Bachelor's degree in Electronics Engineering at University of Science and Technology of China in 1990, got her Master's degree in Electronics Engineer at Peking University in 1993, and got her doctorate in Computer Science at University of Science and Technology of China in 2005. Her research interests are computer networks and parallel computing.