XeFlow: Streamlining Inter-Processor Pipeline Execution for the Discrete CPU-GPU Platform

Zhifang Li, Beicheng Peng, and Chuliang Weng

Abstract—Nowadays, GPUs have achieved high throughput computing by running plenty of threads. However, owing to disjoint memory spaces of discrete CPU-GPU systems, exploiting CPU and GPU within a data processing pipeline is a non-trivial issue, which can only be resolved by the coarse-grained workflow of "copy-kernel-copy" or its variants in essence. There is an underlying bottleneck caused by frequent inter-processor invocations for fine-grained batch sizes. This article presents *XeFlow* that enables streamlined execution by leveraging hardware mechanisms inside new generation GPUs. XeFlow significantly reduces costly explicit copy and kernel launching within existing fashions. As an alternative, XeFlow introduces *persistent operators* that continuously process data through *shared topics*, which establish efficient inter-processor data channels via hardware page faults. Compared with the default "copy-kernel-copy" method, XeFlow shows up to $2.4\times \sim 3.1\times$ performance advantages in both coarse-grained and fine-grained pipeline execution. To demonstrate its potentials, this article also evaluates two GPU-accelerated applications, including data encoding and OLAP query.

Index Terms—CPU-GPU programming, heterogeneous memory system, GPU scheduling

1 Introduction

Modern GPUs like the NVIDIA Tesla series, equipped with thousands of cores, have demonstrated the computational power in graphics rendering and deep neural networks. In data processing applications, the CPU-GPU architectures are exploited [1], [2], [3], [4] to accelerate heavy-loaded computations, where the CPU side handles the logic control and offloads computations to the GPU side. However, in these scenarios, input data initially lies on the CPU side. Therefore, inevitable data transfer and synchronization between two sides may weaken the power of using GPU.

Note that two types of mainstream CPU-GPU platforms have different influences. In consumer electronics and embedded systems, "integrated GPUs" like AMD's APU and ARM's Mali are fashionable in the market. This design puts CPU and GPU into a single chip and lets them share a single memory space. Without the concerns of memory coherency, it is easy to adopt existing software techniques like pipeline execution [5] in OpenCL. However, integrated GPUs are unsuitable for computing-intensive workloads due to fewer cores and lower memory bandwidth. Consequently, "discrete GPUs" are the mainstream in data centers and clouds, where the two kinds of processors, CPU and GPU, own discrete memory spaces called *host memory* and *device memory* respectively.

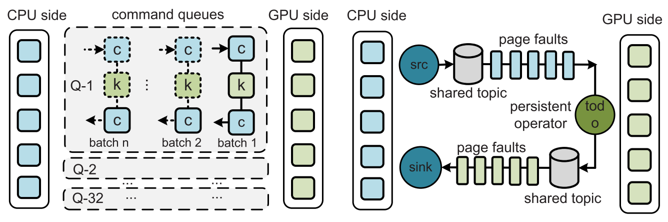
Discrete GPUs are more powerful but lead to challenges in software design. With discrete memory spaces, applications must carefully maintain coherency between two sides by following a workflow of "copy-kernel-copy" (CKC). We take CUDA, the de facto toolkit to program NVIDIA's GPU, as an example to present a CKC workflow. First, the CPU sends a command (e.g., `cudaMemcpy`) that migrates data from host memory to device memory via the PCIe bus and the DMA engine. Then a *kernel* is invoked to run on the GPU, which computes data in device memory with plenty of threads. After waiting for the kernel to finish, the second copy is involved to get results back to the host memory.

In data processing applications like data encoding and analytics, "pipeline" is one of the most common patterns that applies a series of operations on the input data. However, adopting CPU-GPU pipelines to discrete CPU-GPU platforms is not straightforward. Normally, application data is held by the CPU side, and each CKC workflow can only process a limited amount of data. For these reasons, current systems could only support CPU-GPU pipelines at the API level. But in essence, their underlying implementations are still based on the CKC workflows [1], [2], [3], [4], [6]. As shown in Fig. 1a, this approach divides input data into continuous batches with the given size and deals with each batch by a CKC workflow. Due to costly inter-processor invocations—two copies and one kernel launching per batch, this overhead cannot be ignored [7]. CUDA's HyperQ [8] provides multiple command queues to overlap data copy and kernel execution from concurrent CKC workflows.

However, this method does not work well all the time. Here is a trade-off between throughput and latency. With fewer invocations, larger batches provide better throughput but incur a longer latency per batch, where HyperQ could fully overlap concurrent CKC workflows. In contrast,

• *The authors are with East China Normal University, Shanghai 200062, China. E-mail: {zhifangli, beichengpeng}@stu.ecnu.edu.cn, clweng@dase.ecnu.edu.cn.*

Manuscript received 9 June 2019; revised 11 Jan. 2020; accepted 15 Jan. 2020. Date of publication 21 Jan. 2020; date of current version 8 May 2020.
(Corresponding author: Chuliang Weng.)
Recommended for acceptance by J. C. Hoe.
Digital Object Identifier no. 10.1109/TC.2020.2968302



(a) copy-kernel-copy (b) XeFlow

Fig. 1. Two models of inter-processor pipeline execution.

smaller batches shorten this latency but cause underutilization of GPU. Owing to limited tens of command queues (e.g., 32 in NVIDIA GPUs), HyperQ is hard to execute many fine-grained CKC workflows simultaneously. To handle such fine-grained workloads, [9] employs lightweight tasks instead of heavy kernels. Unfortunately, when data lies on the CPU side, this approach still needs explicit copy, and it is incompatible with HyperQ to hide transfer latency.

To the best of our knowledge, the CKC workflow (or its variants) is still the mainstream in CPU-GPU pipelines, even though it leads to performance issues under small batch sizes. Since those software methods based on interprocessor invocations are imperfect, we attempt to explore hardware-based methods. With the advent of new discrete GPUs like NVIDIA's Pascal and Volta, *unified memory (UM)* provides an opportunity to communicate CPU and GPU sides through hardware page fault. Besides, the technique of "persistent thread" [10] enables partial control inside kernel execution.

On the top of UM and persistent thread, we present *XeFlow*, a framework to build data processing pipelines on the discrete CPU-GPU platforms. To deal with issues in fine-grained workloads, we employ a streamlined approach, as shown in Fig. 1b. Our design principle includes two basic ideas. First, we let hardware page faults rather than software invocations handle data transfer and overlapping. Second, we only launch the kernel once and then compute input data throughout the whole running time. To address the challenges encountered in the CPU-GPU pipeline, we make contributions as follows.

Persistent Operator. To execute CPU-GPU pipelines by CKC, each batch of data will launch a kernel for computing. We realize a subtle fact that the same kernel is invoked for different batches repeatedly. Therefore, it makes sense to use a single kernel for the whole input data. From this idea, we introduce the notion of *persistent operator*, which extends persistent thread to reuse the kernel context. The persistent operator wraps user-defined code and lives on the GPU to perform continuous execution, without repetitive kernel launching. Unlike [9] that uses a CPU-managed table to schedule GPU tasks, the persistent operator is controlled by itself, without the assistance of remote CPU.

Shared Topic. Apart from kernel launching, the invocation of explicit copy is another bottleneck. By using page fault of UM to transfer data between processors, we employ *shared topic* to establish an efficient data channel shared by CPU and GPU sides. Our design is inspired by the "subscribe-publish" scheme in the database community, which is used to decouple producers and consumers in a pipeline. Specifically, the persistent operator is notified to read data from the subscribed topic and write computed data to the published topic only when data is ready. It provides an interface based on UM pointers to read or write data directly without explicit copy. If the UM pointer refers to the remote side, the shared topic leverages page faults to deliver desired data and skip unwanted data as well.

Layered Memory Coherency. As shared topics deal with data transfer via UM, the thrashing of page faults between processors is the next troublesome challenge. Besides, the default setting of shared topics does not prevent concurrency hazards. To cope with these issues, we manage UM data according to its access patterns and assign each layer of shared topics with an individual strategy of coherency. By making each side access local data as far as possible, this policy reduces the overhead caused by page faults.

Optimizations of Heterogeneous Workloads. Considering various persistent operators, they can either be computing-intensive or data-intensive, which are bounded by different GPU resources, including computing and transfer bandwidth. To cope with heterogeneous persistent operators, we reorganize the kernel as multiple partitions, where each persistent operator is handled by a partition of threads. By controlling the partitioning of kernel, the kernel could co-utilize two types of resources simultaneously. However, this optimization cannot be achieved easily due to the undocumented implementation of GPU. Therefore, we provide a model to learn the characteristics of GPU, which helps to find an optimal partitioning among persistent operators.

By combining these contributions, XeFlow achieves streamlined pipeline execution on the discrete CPU-GPU systems. In experiments, we design a specialized microbenchmark to evaluate the impact of both CPU-GPU transfer and GPU execution. Compared with the default CKC workflow, experimental results show up to $2.4\times\sim3.1\times$ performance in coarse-grained and fine-grained batch sizes. To show the potentials of XeFlow, we further evaluate two case studies, including data encoding and OLAP query.

We organize the rest of this paper as follows. Section 2 introduces related background. We outline XeFlow's programming model in Section 3. The implementation of shared topics is shown in Section 4. Section 5 explains how to run persistent operators on the GPU. Then Section 6 evaluates the performance of XeFlow through one microbenchmark and two case studies. The related work is discussed in Section 7. Finally, we conclude this paper in Section 8.

2 Background

This section takes CUDA for example to introduce the prior knowledge and challenges in discrete CPU-GPU platforms, which are also similar for other scenarios like OpenCL.

2.1 CPU-GPU Memory Model

CUDA 4.0 with NVIDIA's Fermi GPU (2011) supports 48-bit virtual addressing and memory pinning, which allows GPU to access host memory but not vice versa. CUDA 6.0 with Kepler (2012) first employs imperfect UM that does not allow concurrent access from two sides. In Pascal (2016) or newer GPUs of NVIDIA, CUDA 9.0 supports practicable UM with concurrent access.

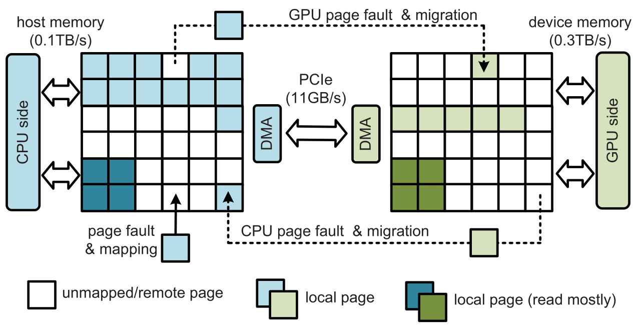


Fig. 2. The paging mechanism of unified memory.

As shown in Fig. 2, UM maps host and device memory into a unified virtual space and maintains the coherency via page faults. We could invoke `cudaMallocManaged` to create UM buffers or define UM variables with `__managed__` prefix. By default, a new virtual page is not mapped to a physical page until a page fault is triggered to map it to a local physical page by modifying the page table. Intuitively, the local page can be directly accessed. As for a remote page, a page fault is triggered to migrate it to the local side via the DMA engine and the PCIe bus. Although UM manages data migration automatically, CUDA 9.0 provides optimization "hints" that can affect preferred page location and mapping.

With UM involved, CUDA suggests a simplified version of CKC, namely *kernel-over-unified-memory (KoUM)*. As its name implied, the kernel is directly launched to scan remote data via UM migration. Even though KoUM removes explicit copy before and after kernel execution, recent research efforts [11], [12] have revealed its drawbacks due to the thrashing of page faults (a page fault can spend more than ten microseconds). Moreover, KoUM still invokes a kernel per batch like CKC. Therefore, simply adopting UM cannot solve the performance issue in fine-grained batch sizes.

2.2 Kernel and Persistent Thread

In CUDA programs, CPU plays as a "driver" that controls GPU behavior through commands such as data copy and kernel launching. The kernel computes data in device memory with plenty of threads, which are further divided into *thread blocks*. GPU hardware dispatches these thread blocks to a dozen of *streaming multiprocessors* (SMs) that execute each dispatched thread block in the unit of *thread warp*—a group of threads that run the same instructions. Each thread handles the given data item according to its "thread ID". SM could hide access latency by switching among thread warps when they encounter cache misses or page faults.

As specific hardware detail of GPU hasn't been disclosed, it is hard to control the scheduling of thread warps/blocks. In prior works [9], [13], [14], [15], *persistent thread* [10] inspires a software method to enable partial kernel control. Its basic idea is to run the fixed number of thread blocks that hardware can schedule concurrently. Each thread computes data items according to an indicated range rather than the default "thread ID". This method often works with other techniques like a CPU-managed queue to launch GPU tasks [9]. Since persistent thread does not consider inter-processor communication, it still needs explicit data copy. To run the pipeline without invocations involved, our idea is to combine persistent thread with UM that could migrate data on the fly.

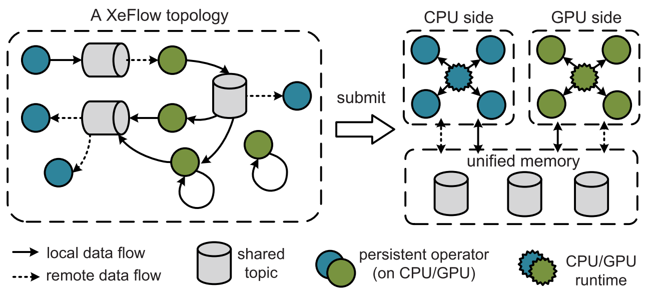


Fig. 3. The overview of XeFlow programming.

3 XeFlow Programming Model

This section presents the programming model of XeFlow that addresses CPU-GPU pipelines for data processing. XeFlow abstracts each application as a "topology" in Fig. 3. We follow three steps to construct such a topology.

Abstracting Data Dependencies as Shared Topics. Unlike previous works that migrate data with explicit copy [3], [8], [9], XeFlow abstracts the data dependency between CPU and GPU as a shared topic, where producers and consumers of pipeline work together with the subscribe-publish interface. In the shared topic, data is organized as *frames*, a bulk of contiguous pages. The producer side could write data into frames continuously and then publish them to the tail of this shared topic to make data accessible. Subsequently, committed frames will be consumed with an increasing offset. Since two operations are performed by two sides simultaneously, Section 4.2 presents our proposed coherency policy to ensure concurrent safety across processors. As this interface only deliveries a handler rather than the whole frame, the shared topic eliminates buffer copy in the CKC workflow.

Filling Computations into Persistent Operators. When writing a CUDA kernel, the programmer should organize computations with plenty of threads and synchronize kernel execution manually. In XeFlow, we can easily put computations into the persistent operator as a payload, who lives on the CPU or GPU side and computes data from the shared topic continuously. The persistent operator provides a basic `CPU/GPU_Operator` class, where its `OnCompute()` is rewritten with computational code by the programmer. Since XeFlow deals with GPU execution through a managed kernel, user-defined code uses virtualized thread IDs to locate data items, rather than CUDA's thread IDs.

Constructing the Topology with CPU-GPU Pipelines. After shared topics and persistent operators have been defined, we construct them as the executable topology. According to their data dependency, we use a C++11 lambda closure to link each triple of <`previous_topic`, `operator`, `next_topic`> as a CPU-GPU pipeline. Taking the most common "ping-pong" topology in Fig. 4 for example, this topology consists of three persistent operators (`Src`, `ToDo`, and `Sink`) and two shared topics (`C2G` and `G2C`).

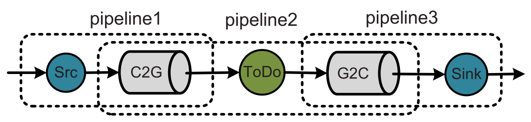


Fig. 4. The ping-pong topology.

These components are connected to three pipelines. The first pipeline < `null`,`Src`,`C2G` > continuously pushes input data from the CPU side. Meanwhile, the second pipeline < `C2G`,`ToDo`,`G2C` > pulls input data from the topic `C2G` and pushes GPU-computed value to the next topic `G2C`. Finally, the third pipeline < `G2C`,`Sink`,`null` > pulls computed data back to the CPU side. The following code template explains how to construct this topology.

```
 #include "xeflow.h"
__managed__ Topic C2G, G2C;
class Src: public CPU_Operator{
 __host__ void OnCompute(Context* ctx){
   int* output = (int*)ctx->NextTopic();
   int* end = output+ctx->MaxPosOfNextTopic
   ();
   for(; output < end; output++)
    *output = RandomInt();
}};
class ToDo: public GPU_Operator{
  __device__ int Multi2(int v){ return v*2;}
  __device__ void OnCompute(Context* ctx){
   // get virtualized thread ID
   int tid = ctx->vThreadID();
   int* input = (int*)ctx->PrevTopic();
   int* output = (int*)ctx->NextTopic();
   output[tid] = Multi2(input[tid]);
}};
class Sink: public CPU_Operator{
  __host__ void OnCompute(Context* ctx){
   int* input = (int*)ctx->PrevTopic();
   int* end = ctx->MaxPosOfPrevTopic();
   for(; input < end; input++)
     printf("%d\n", *input);
}};
int main(){
  InitTopic({&C2G, &G2C});
  auto topo=[]__host__ __device__(XeFlow* x){
   x->Link(nullptr, new Src(), &C2G);
   x->Link(&C2G, new ToDo(), &G2C);
   x->Link(&G2C, new Sink(), nullptr);};
  SubmitAndExec(topo);
}
```

After submitting this topology, XeFlow transparently manages its execution without the concerns on data transfer and scheduling. We leave the details of how the topology works in the next two sections.

4 Shared Topic

To cope with data transfer between persistent operators efficiently, the shared topic is organized as three layers. The catalog layer stores metadata defined by the topology. The offset index layer indicates the physical position of data in the bottom frame layer. This section will present the detailed implementation of shared topics.

4.1 Inter-Processor Synchronization

The first step to build shared topics is establishing efficient primitives for inter-processor synchronization. As the GPU side does not run an operating system, existing mechanisms like mutex and semaphore cannot be used for this purpose. Furthermore, CUDA only allows the CPU to synchronize kernel execution by costly invocations, but not vice versa—the GPU side cannot synchronize with CPU by itself. Therefore, duplex and invocation-free mechanisms are urgently required. Here are three methods we use in the shared topic.

Direct Read. In the shared topic, *direct read* is the uppermost approach we use for data synchronization between two sides. As the name implies, there is no explicit copy before kernel execution. This approach lets persistent operators on the CPU or GPU scan frames on the remote side via UM pointers. This method makes use of page faults to transfer accessed pages of frames between processors. However, direct read works correctly only when the remote side does not modify data simultaneously. The following two mechanisms are employed for concurrent occasions.

Inter-Processor Atomic Instruction. Based on the mechanism of page fault, UM provides "inter-processor" atomic instructions like `atomicAdd_system()`, which could atomically update a concurrent UM variable used by CPU and GPU sides. Hardware page faults are triggered to maintain the memory coherency across two sides. In the shared topic, we leverage atomic instructions to establish the basic atomicity of variables, such as pointers and cursors. However, atomic instructions are quite expensive owing to the overhead of costly PCIe round trips between two sides.

Inter-Processor Spinlock. In common designs, mutex is employed to protect data structures shared by multiple users. Owing to the lack of an operating system, GPU does not support typical mutex. As an alternative, we could implement *inter-processor spinlock* based on the above atomic instructions. Before using spinlock, a lock variable is initialized as 0. To acquire this lock, the caller repeats "compare-and-swap" until it swaps `1` to this lock variable successfully. To release the lock, we reset it back to 0. To avoid potential deadlocks, only the leader thread within the thread warp can lock or unlock this variable.

Considering the high concurrency of CPU-GPU systems, frequent inter-processor synchronizations lead to frequent page faults as well. The design of shared topics should consider the impact of involved page faults. This problem is solved in the next subsection.

4.2 Layered Memory Coherency

To minimize the impact of page faults, we optimize the policy of memory coherency inside shared topics. We observed that different parts of shared topics have individual patterns of data access. For this consideration, we organize the shared topic as multiple layers shown in Fig. 5, and apply a specialized strategy on each layer with CUDA's locality hints. We introduce each layer in a top-down order.

Catalog (Read-Mostly). The top layer stores the metadata of shared topics, including frame sizes and handlers to the below layer. During topology execution, both CPU and GPU sides frequently read metadata from the catalog. Default UM will migrate pages back and forth among two sides. Since all shared topics are pre-defined, the catalog is rarely modified during execution. As recognizing this property, we apply the *read-mostly* policy to the catalog, which maintains a local replica of catalog for each side. Consequently, each side could directly read the local replica without page faults involved. To

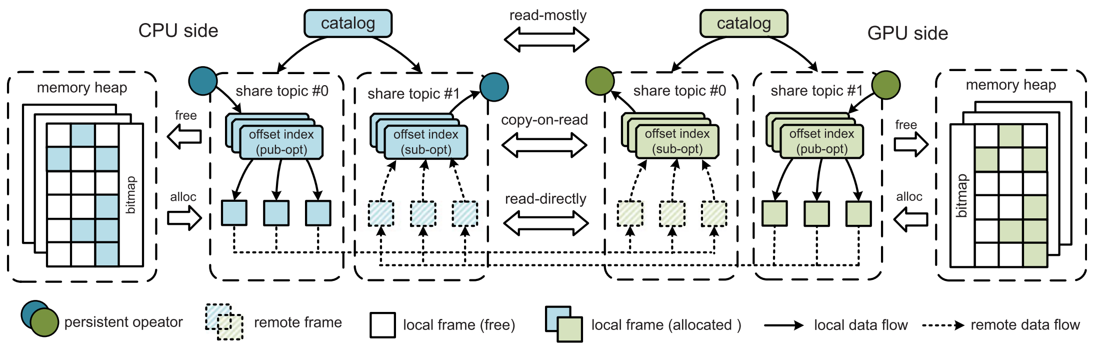


Fig. 5. The architecture of shared topics.

deal with rare updates, two page faults are triggered to modify two replicas, respectively.

Offset Index (Copy-On-Read). The next layer is the offset index that helps persistent operators to read the frame with a subscribed offset. This layer stores the index handler of each published offset along with the pointer of its related frame. Unlike the catalog, this layer is frequently modified to register published frames to the latest offset. Therefore, we adopt the *copy-on-read* policy to decrease page faults involved. It uses two replicas optimized for publishing and subscribing. The publish-optimized replica locates on the CPU or GPU, based on the running location of the previous operator in the pipeline, which will publish frames to this replica. This policy restricts the modified ranges within only one side. Similarly, the subscribe-optimized replica locates on the other side for subscribing to decrease remote access.

To illustrate how the offset index works, we take Fig. 6 as an example. When publishing a frame, the persistent operator inserts an index handler into the index page of publish-optimized replica and modifies the range of valid offset. Considering concurrent modifications, the index page is locked. To read a frame with the subscribed offset, the persistent operator first looks at the subscribe-optimized replica on the local side. If the latest index handlers are not found locally, page faults are triggered to copy them from the other replica. As each published frame registers a 64-byte index handler, each page fault will deliver 64 handlers (i.e., 4 KB/64 = 64) of published frames. After that, the position of index handlers in the page can be easily computed from the given offset. If this frame has not been published, subscribing is failed. This operation can be repeated until

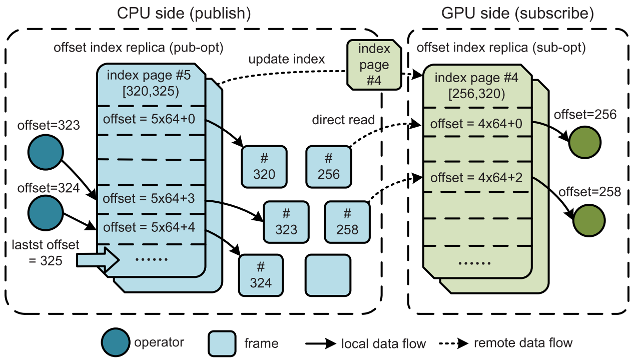


Fig. 6. Two replicas of offset index.

this frame is published. Since modifications only perform on the publish-optimized replica, subscribing does not need to lock the subscribe-optimized replica. After subscribing is finished, the persistent operator could consume subscribed frames with the index handler.

Frame (Read-Directly). The underlying layer stores published frames upon UM data. The CKC model provides a buffer-copy interface, which needs to migrate the buffer between two sides. The overhead cannot be neglected, especially when the buffer size is small. The frame layer provides a read-directly interface. Unlike the above two policies, this layer only maintains a single replica on the published side. When the frame is accessed through UM pointers, GPU's hardware automatically handles transfer and coalesces the requests of contiguous pages. This replica is fixed on the published side to create the opportunity of reducing data to transfer, as presented in the next subsection.

4.3 Skipping Transfer via Fine-Grained Data Structures

Due to limited PCIe bandwidth, CPU-GPU transfer can be the bottleneck in many scenarios. [16] avoids redundant transfer when the areas of accessed data are overlapped and regular. Based on the above read-directly interface, this paper shows another idea of "skipping", where undesired data can be skipped by looking up fine-grained data structures before copying the whole frame. To illustrate its potentials, we take *online analytical processing* (OLAP) [17] as an example, which normally suffers from data transfer. Limited by CKC, GPU-accelerated OLAP query often employs coarse-grained tables. With fine-grained data structures, we bring an opportunity to skip undesired items in the table.

Columnar Paging. CPU-based OLAP systems often organize the table as contiguous rows. To execute an OLAP query by GPU, the entire table is moved to the GPU side reluctantly even if not all columns are desired. Moreover, the row-based format also causes uncoalesced access in device memory. A feasible method is *columnar paging* [18] that arranges values of the same column to the same page in the frame. As direct read allows GPU to access pages via frame pointers, we could easily adapt columnar paging to the shared topic. This optimization enables persistent operators to skip undesired pages and coalesce memory access for each page. Unfortunately, the CKC model is inefficient to manipulate small pages with explicit copy.

Lightweight Indexing. In OLAP, "select" is a standard operator that scans a table to collect rows meeting the given predications. Lightweight indexing brings potential optimizations by dividing each frame as data pages and a few index pages. By looking up index pages in advance, this optimization dramatically reduces the amount of data pages to transfer. For the range predication, index pages could store the minimum and maximum values of each data page. Before scanning a data page, the GPU side could look up the index pages to detect whether it falls into the min-max range. As for the point predication, index pages could store *bloom filter* [19], a hash sketch of data pages to estimate whether a data page contains matched rows.

4.4 User-Space Memory Management

This subsection introduces the memory management strategy in the frame layer. When creating a UM buffer, CUDA's default allocator needs to modify page tables of both CPU and GPU sides. For this reason, allocating each frame on the fly is very costly. Previous research efforts [20], [21] achieve efficient device memory allocation in the user space. We are inspired by this design but extend it to serve both CPU and GPU sides with UM. For each side, XeFlow employs a *memory heap* to allocate memory locally. When the memory heap initializes, UM buffers are created by `cudaMallocManaged`. All pages are pre-mapped to the page table according to the desired memory coherency. With the virtual paging of UM, the capacity of memory heap can oversubscribe total device memory (e.g., 24 GB for NVIDIA P40). Since current UM does not support disk swapping, this capacity cannot exceed host memory.

The memory heap divides each UM buffer into contiguous frames, and a bitmap, where the *ith* bit indicates whether the *ith* frame is free. Each frame contains contiguous 4 KB data pages (i.e., by default, 4 MB frame size). To allocate a new frame, the memory heap tries to flip a bit from 0 to 1 with a linear hash path. In contrast, the memory heap flips the bit back to free a useless frame. Considering concurrent allocation, the modifications of bitmap should use atomic instructions. When one memory heap exhausts all created buffers, it could extend the capacity by creating new buffers or stealing unused buffers from another heap.

5 Runtime System

This section introduces the runtime system to execute persistent operators on the discrete CPU-GPU platform.

5.1 Persistent Operator Execution

For either the CPU or GPU side, the persistent operator has a similar execution flow, as presented in Algorithm 1. The core of persistent operators is a "loop" that executes computations round by round. In each round, it subscribes the input frame and then publishes the computed frame until it encounters the exit flag. Due to the inherent differences between CPU and GPU, lines 6-14 of this algorithm have different implementations for each side.

On the CPU side, it is straightforward to assign CPU threads to compute input frames. In implementation, we can easily use a parallel programming toolkit like Linux Pthread or OpenMP. When the topology contains multiple

Algorithm 1. Template for Topology Execution

```
   Input: a topology $\Gamma$ with a series of piplines like (topic $T_{pv}$,
   operator $\Theta$, topic $T_{nt}$)
 1 for each pipeline $(T_{pv}, \Theta, T_{nt}) \in \Gamma$ do
 2     Initialize a frame handler $f_{pv}$ for $T_{pv}$ ;
 3     Initialize a frame handler $f_{nt}$ for $T_{nt}$ ;
 4     Forward $(f_{pv}, \Theta, f_{nt})$ to a CPU thread or a GPU kernel ;
 5     for round $r = 0$ ; $\Theta$ is not exited; $r++$ do
 6         $f_{pv} \leftarrow$ subscribe a committed frame from $T_{pv}$ ;
 7         for each item $t_i \in f_{pv}$ do
 8             $t_i^* \leftarrow$ apply computations of $\Theta$ on the $t_i$ ;
 9             write $t_i^*$ into $f_{nt}$ by the posistion $i$ ;
10         end
11         if $f_{nt}$ is fulfilled then
12             publish $f_{nt}$ to the $T_{nt}$ ;
13             $f_{nt} \leftarrow$ create an empty frame ;
14         end
15     end
16 end
17 return wait for all $\Theta$ in the $\Gamma$ ;
```

persistent operators, many works have already studied mature strategies to achieve load balancing. Therefore, this paper mainly focuses on the GPU side.

As for GPU execution, the situation gets complicated. Each persistent operator has various demands of GPU resources like computing and bandwidth. As CUDA kernels are scheduled by undocumented hardware, our optimization spaces to co-utilize two resources are restricted by GPU implementation. To overcome this barrier, we divide a single CUDA kernel to execute persistent operators in a managed manner, as shown in Fig. 7.

To exploit parallelism, each thread block of physical kernel only subscribes a part of frames in the shared topic. Each thread block independently runs the above scheduling loop to avoid expensive global synchronizations. In each round, it creates a *work-vector* that indicates the assigned operator ID of each thread warp. This work-vector is stored in GPU's fast on-chip memory to speed up access. Then each thread warp can look up the work-vector and choose the assigned persistent operator to execute in this round.

Considering bothersome logic divergences during GPU execution, all thread warps are assigned to persistent operators in an aligned way. The work-vector also stores frame handlers for each persistent operator, thus multiple operators could subscribe a shared topic via independent offsets. As XeFlow takes over intra-GPU execution, "virtualized"

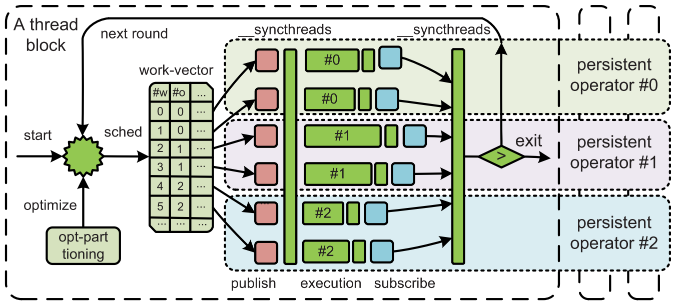


Fig. 7. The virtualized kernel for GPU execution.

thread IDs should be used to replace CUDA's thread IDs for locating data items in the frame. After finishing the execution, the thread block is synchronized to move on to the next round. We conclude the implementation of GPU execution in Algorithm 2.

Algorithm 2. GPU Side Execution

```
Input: a topology $\Gamma$ with a series of piplines like (topic $T_{pv}$, operator $\Theta$, topic $T_{nt}$)
1  $\Gamma^* \leftarrow$ extract all GPU operators from $\Gamma$ ;
2  for round $r = 0$ ; $\Gamma^* \neq \phi$ ; $r++$ do
3      work-vector $\leftarrow$ assign 32 warps to $|\Gamma^*|$ operators;
4      do subscribe ;
5      __syncthreads ;
6      $\Theta \leftarrow$ work-vector[warp-id].operator-id ;
7      if $32 * warp - id \leq thread - id < 32 * (warp - id + 1)$ then
8          vir-thread-id $\leftarrow$ thread-id + r* thread-num ;
9          execute $\Theta$ by using vir-thread-id as the position ;
10     end
11     do publish ;
12     if $\Theta$ is finished then
13         delete $\Theta$ from $\Gamma^*$ ;
14     end
15     __syncthreads ;
16 end
```

5.2 Optimizations for Heterogeneous Workloads

During kernel execution, persistent operators in the different thread warps share GPU hardware via time slicing. When a thread warp begins to transfer data, GPU yields the idle arithmetic units to another thread warp. In essence, different persistent operators prefer various GPU hardware resources. For example, data-intensive ones can be improved by using more thread warps to hide transfer latency, while computing-intensive ones have slight promotion due to limited arithmetic units shared by multiple thread warps.

When creating a work-vector for heterogeneous persistent operators, an optimal partitioning of thread warps will improve the overall performance by co-utilizing different units inside GPU. However, it is almost infeasible to figure out their real resource demands. Current profiling tools like CUDA's `nvprof` are designed for the whole kernel. To estimate persistent operators inside the kernel, we should model the performance of this workload.

We could evaluate the execution time of a workload with n heterogeneous persistent operators as follows. In a scheduling round, we denote R_i as the round time for the *ith* persistent operator that is assigned with N_i thread warps. Intuitively, the overall round time depends on the slowest operator (i.e., $max\{R_i\}$). When a thread block consists of $\mathbf{N}$ thread warps, to get the minimized round time, the optimal resource partitioning $\{N_i\}$ can be given by

$$\underset{\{N_i\}}{\arg\min}\ max\{R_i\} = max\{\ f_i(N_i)\ \}\ |\ \sum N_i = \mathbf{N}, \tag{1}$$

where $f_i(N_i)$ indicates how *ith* operator is affected by the number of assigned thread warp N_i. By default, we could use a following polynomial function as an approximation.

$$f_i(N_i) = \theta_2 N_i^2 + \theta_1 N_i + \theta_0. \tag{2}$$

Data-intensive operators have lower $[\theta_2, \theta_1, \theta_0]$ because more thread warps will alleviate memory latency. In contrast, $[\theta_2, \theta_1, \theta_0]$ tends to be higher in computing-intensive operators due to limited arithmetic units. For a given operator, we cannot know the value of $[\theta_2, \theta_1, \theta_0]$ in advance. Besides, to find an optimal partitioning for n operators, exhaustive search leads to unscalable $O(\binom{N-1}{n-1})$ complexity.

5.3 Auto-Tuning Tool

To estimate $[\theta_2, \theta_1, \theta_0]$ in a scalable way, we provide a auto-tuning tool based on *polynomial regression*, a machine learning algorithm to learn the black-box GPU. When tuning *ith* operator, we could get the sampled R_i from GPU's timer. By using k samples of (N_i, R_i) under varying N_i, we can learn the estimated $[\hat{\theta_2}, \hat{\theta_1}, \hat{\theta_0}]$ through *batch gradient descent* (BGD). Since BGD calculates these samples in limited passes, it reduces the tuning complexity to linear $O(nk)$, which can be finished within milliseconds (see experiments in Table 2).

After that, we let all $\{R_i\}$ be an equal bias f_0 (e.g., the average of sampled f_i) to minimize $max\{R_i\}$. Then the optimal partitioning $\{\hat{N_i}\}$ can be estimated as

$$\{\hat{N_i}\} = \left\{ \frac{\left(\sqrt{\hat{\theta_1}^2 - 4\hat{\theta_2}(\hat{\theta_0} - f_0)} - \hat{\theta_1}\right)/2\hat{\theta_2}}{\sum\left(\sqrt{\hat{\theta_1}^2 - 4\hat{\theta_2}(\hat{\theta_0} - f_0)} - \hat{\theta_1}\right)/2\hat{\theta_2}} \times \mathbf{N} \right\}, \tag{3}$$

where $(\sqrt{\hat{\theta_1}^2 - 4\hat{\theta_2}(\hat{\theta_0} - f_0)} - \hat{\theta_1})/2\hat{\theta_2}$ is the solution of polynomial function $\hat{\theta_2}N_i^2 + \hat{\theta_1}N_i + \hat{\theta_0} = f_0$.

Note that this strategy is mainly to maximize the overall GPU utilization. The future works could extend this method for other dimensions like latency and energy consumption. To reduce incurred overhead, we could perform tuning only when the uniform partitioning leads to an imperfect deviation of $\{R_i\}$ that exceeds an empirical threshold.

6 Evaluation

This section first evaluates the performance of XeFlow via a microbenchmark. Then we evaluate its application potentials by two case studies.

6.1 Experiment Setup

Our experiments employ a machine equipped with Intel Xeon CPU and NVIDIA P40 GPU based on Pascal architecture. Table 1 lists our software and hardware specifications.

We observed that pipeline-like CPU-GPU programming has been studied by various research works [2], [3], [4]. These implementations can be concluded as the following types.

- *CKC.* This method repeatedly invokes a copy-kernel-copy workflow for each batch of input data.
- *CKC+HQx.* To overlap copy and kernel execution of CKC, this strategy leverages CUDA HyperQ to run x concurrent CKC workflows.
- *CTC.* This strategy uses lightweight GPU tasks to substitute CUDA kernels (Pagoda [9]) but still incurs two invocations of copy per batch.

TABLE 1
Software and Hardware Specifications

OS	CentOS 7
Toolchain	CUDA/NVCC 9.1
CPU side	Intel Xeon Silver 4110 @ 2.10 GHz x 2 32 hyper-threading cores, 11 MB L3 cache
	150 GB host memory
GPU side	NVIDIA P40 @ 1.53 GHz x 1 3840 CUDA cores, 1,024 threads per block 24 GB device memory, 346 GB/s bandwidth 11 GB/s PCIe 3.0 x 16, 2 DMA copy engines

- *KoUM.* This strategy removes two copies from CKC by using UM pointers as the kernel input. However, it still launches repetitive kernels for each batch.

To evaluate more general cases, we do not directly compare XeFlow with various systems. As an alternative, we compare their underlying implementations based on CUDA.

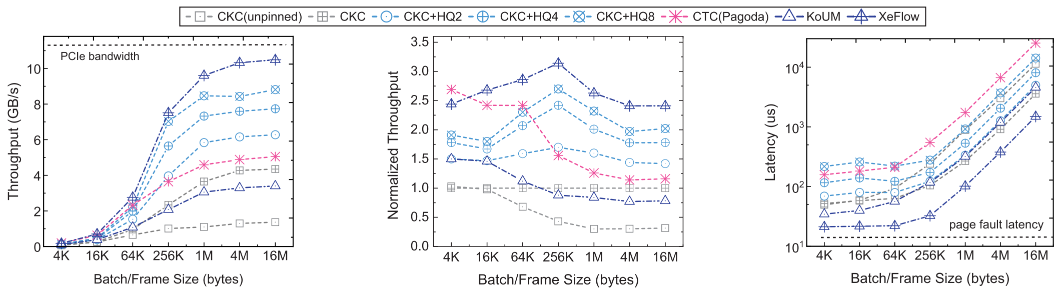
6.2 Performance Evaluation via Microbenchmark

Since this paper addresses CPU-GPU pipelines for data processing, both CPU-GPU transfer and GPU computing will impact the performance. Therefore, we design a specialized microbenchmark based on the ping-pong topology (see Section 3), which could evaluate two factors together. There are two types of variants. One is "$C(GC)_n$" where $n+1$ CPU operators (i.e., C) and n GPU operators (i.e., G) are interleaved in the topology. The other is "$C(G)_nC$" that contains two CPU operators and n adjacent GPU operators.

Overall Performance. For the most common *CGC* (i.e., $C(GC)_1$), Fig. 8 compares the overall performance of various implementations. Before running the microbenchmark, we divide 10 GB data based on the demanded batch sizes (or frame sizes for XeFlow). The experimental results are analyzed below. (1) As data transfer is handled by CPU rather than the DMA engine, CKC with unpinned memory causes a suboptimal performance. (2) With pinned memory involved, CKC is improved but suffers from non-overlapped data copy and kernel execution. (3) When x concurrent CKC workflows (x = 2/4/8) are running to overlap copy and execution, CKC+HQx optimizes the throughput under coarse-grained batch sizes. (4) By replacing kernels with lightweight tasks, CTC (i.e., Pagoda) shows impressive throughput under fine-grained batch sizes. However, CTC still invokes explicit copy and cannot work with HyperQ to hide copy latency. These reasons lead to performance deterioration under coarse-grained batch sizes. (5) Even though KoUM discards explicit copy, it still suffers from low throughput due to frequent page faults and kernel launching. (6) XeFlow not only minimizes runtime invocations but also reduces page faults through layered memory coherency. Compared with CKC, XeFlow achieves up to $2.4\times\sim3.1\times$ throughput and nearly a third of latency.

Computational Workloads. In real applications, GPU is employed to deal with heavy-load computations. To further evaluate these scenarios, we fill the GPU operator in the ping-pong topology with a group of workloads (three computing-intensive and three data-intensive) based on Parboil [22], a widely-used benchmark for GPU computing. Experimental results in Fig. 9 also present noticeable improvements like the previous experiments. We notice the differences between the two types of workloads. For computing-intensive workloads in Fig. 9a, 9b, and 9c, the improvements decrease significantly when the batch size grows. This phenomenon implies that computing instead of CPU-GPU transfer becomes the bottleneck under coarse-grained batch sizes. In contrast, as for data-intensive workloads in Fig. 9d, 9e, and 9f, data transfer is still the primary bottleneck even for the cases with larger batch sizes.

Detailed Profiling. To prove our analysis of performance, we use CUDA's `nvprof` to get detailed profiling parameters. Fig. 10a shows the number of involved inter-processor invocations to process per GB data. CKC introduces three invocations per batch (i.e., copy+kernel+copy). CTC reduces it to two invocations of copy, and KoUM reduces it to one invocation of kernel launching. For this reason, the above methods still lead to performance issues under fine-grained batch sizes. In contrast, XeFlow only launches a single kernel to server persistent operators in the topology. Fig. 10b shows the number of involved page faults during execution. KoUM and unoptimized XeFlow cause frequent page faults. To evaluate the promotion of layered memory coherency, we gradually add the optimized policy of the catalog, index, and frame layer to unoptimized XeFlow. Experimental results in Fig. 10b and 10c explain three interesting facts as below. (1) For small batch sizes, the optimization of layered coherency is imperative. Otherwise, XeFlow is even worse than KoUM. Compared with unoptimized XeFlow, three optimized layers provide about 21, 52, and 334 percent improvements. (2) For larger batch sizes, about 79 percent promotion is contributed by the optimized frame layer because this layer stores large amounts of data consumed by the GPU side. (3) When the batch size is very large, the



(a) Throughput. (b) Speedup over CKC. (c) Latency per Frame.

Fig. 8. The overall performance with varying batch/frame sizes.

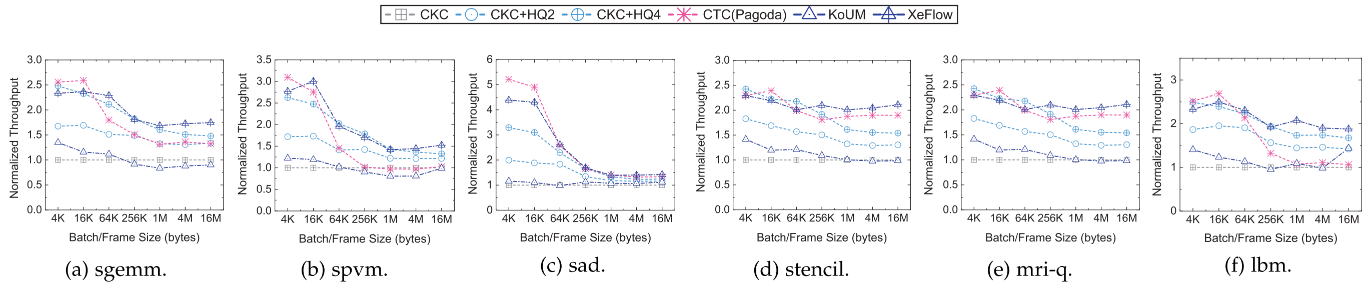


Fig. 9. The performance of computational workloads.

relative speedups decrease slightly because hardware can merge the requests of contiguous page faults.

Scheduling Overhead Analysis. To deeply understand the cost of using persistent operators, we collect the execution time in the scheduling round by CUDA clock. Fig. 11 shows the distribution of each step. When persistent operators are scheduled in each round, a tiny work-vector will be calculated, which only incurs a slight $0.02\%\sim2.5\%$ overhead of runtime. As for very small frames, publishing and subscribing take most of the execution time. Due to more expensive locking, publishing incurs larger overhead, while subscribing only needs to fetch an index handler. However, under larger frames, operator execution can utilize more GPU resources but also enlarges the response latency of each frame. To make a trade-off between utilization and latency, the programmer needs to set appropriate frame size.

Scalability. This experiment evaluates whether XeFlow could scale with more operators. With a varying GPU operator number n (1 to 4), we evaluate $C(GC)_n$ and $C(G)_nC$ topologies under fine-grained (16 KB) and coarse-grained (4 MB) batch/frame sizes. We adopt a uniform partitioning of thread warps for n GPU operators. We do not evaluate CKC and KoUM for their relatively poor performance. Fig. 12 demonstrates the throughput with varying n. Under the fine-grained batch, CKC+HQ4 and Pagoda introduce a noticeable degradation for $C(GC)_n$ because this topology imposes $n{+}1$ CPU-GPU copies per batch. Differently, $C(G)_nC$ leads to slight degradation for constant two copies per batch. By mitigating the overhead of invocations, XeFlow only shows a slight $1.7\%\sim3.6\%$ degradation for $C(GC)_n$ and $C(G)_nC$. For the larger frames, the main bottleneck turns into the limited PCIe bandwidth shared by n operators. Even though all three methods degrade with the increasing n, XeFlow is still better than CKC+HQ4 and Pagoda due to its fewer invocations. We also observed that XeFlow and CKC+HQ4 have higher throughput than Pagoda that does not optimize these coarse-grained cases.

Heterogeneous Workloads. When a workload contains heterogeneous persistent operators (e.g., data-intensive and computing-intensive), the partitioning of thread warps is crucial to performance. To study its impact, we construct such a heterogeneous workload, where each GPU operator performs x times *fused multiply-add (FMA)* for each input item. By changing the value of x, we can simulate either data-intensive or computing-intensive cases. First, we run XeFlow's auto-tuning tool to learn from FMA operators with varying x (short for FMAx). To simplify the analysis, we only compare two FMAx operators in each experiment. Technically, it can be applied to n operators (see Section 5.2). As shown in Table 2, this microsecond tuning time can be neglected by long-running applications. According to learned models, we estimate heterogeneous workloads under varying partitioning cases of 32 thread warps (from

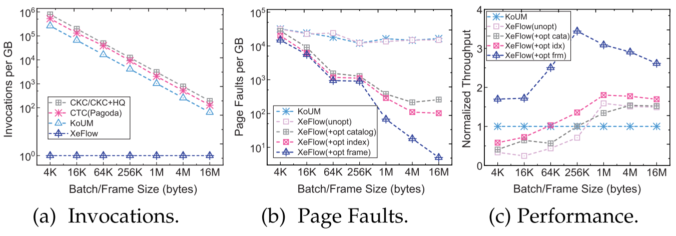


Fig. 10. Detailed profiling.

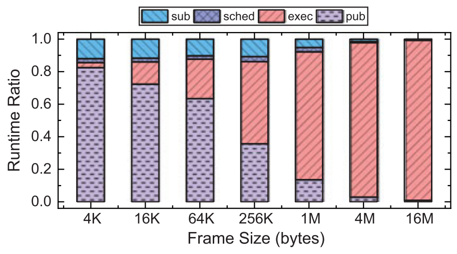


Fig. 11. The runtime distribution of XeFlow.

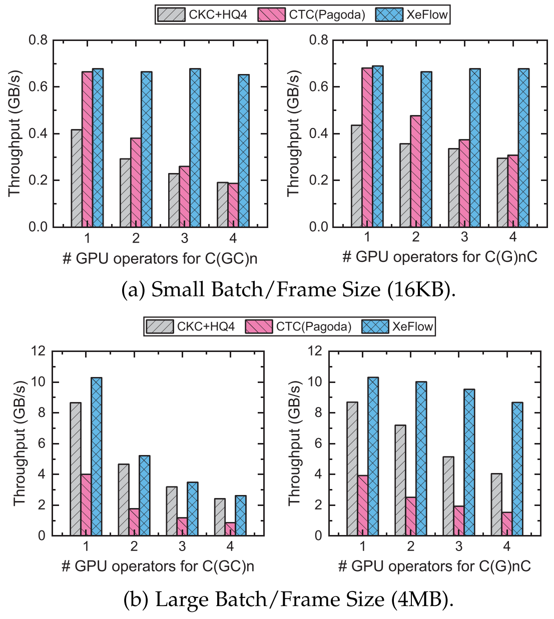
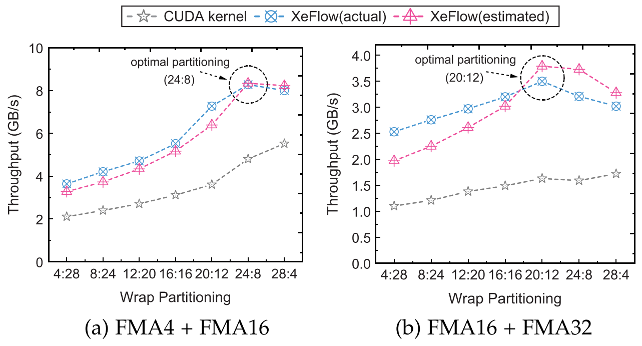


Fig. 12. The scalability of different topologies.

TABLE 2
The Tuning Time (milliseconds) for Varying Workloads and the Numbers of Samples

# samples	FMA4	FMA8	FMA16	FMA32	FMA64
800	2.64	3.19	5.72	6.88	9.26
1600	8.95	9.38	10.71	12.66	17.69
3200	17.82	18.74	20.28	25.54	35.20
6400	33.68	35.79	42.37	53.91	81.91
12800	68.85	74.55	78.47	95.64	149.79



(a) FMA4 + FMA16 (b) FMA16 + FMA32

Fig. 13. The performance of heterogeneous workloads.

4:28 to 28:4). Then, we run concurrent CUDA kernels and XeFlow to get the actual performance. Fig. 13 shows that estimated throughput has a similar curve like the actual value. Two optimal partitioning cases, 24:8 in Fig. 13a and 20:12 in Fig. 13b, show up to $4.1\times$ and $3.2\times$ over concurrent CUDA kernels, along with $2.0\times$ and $1.4\times$ speedups over the worst partitioning case of XeFlow itself.

6.3 Application Evaluation: Data Encoding

The first application is "data encoding" that is ubiquitously used in data centers. For example, erasure coding helps to tolerate storage error, and AES encryption is employed to ensure data privacy. Owing to massive calculations, CPU-only designs may incur poor performance. To accelerate such workloads with XeFlow, we develop such a prototype topology illustrated in Fig. 16 that implements data encoding on the GPU operator. To avoid the impact of different encoding implementation, we use hand-written code rather than third-party libraries. Besides, we pre-load data to the ramdisk to bypass the impact of disk I/O.

We simulate erasure encoding with the RAID-6 algorithm in the GPU operator. Fig. 14a and 14b show the performance of erasure encoding. For very fine-grained input (e.g., 4 KB), the CPU-only version that directly accesses host memory achieves better performance than all GPU versions. XeFlow achieves $2.2\times/1.8\times/3.1\times$ throughput than CKC/CKC+4HQ/KoUM when the batch/frame size is 64 KB. For coarse-grained input, XeFlow outperforms CTC that cannot overlap transfer with computing. XeFlow provides about $1/5\sim1/2$ latency than other GPU versions or even the CPU version when GPU acceleration dominates the cost of CPU-GPU data transfer.

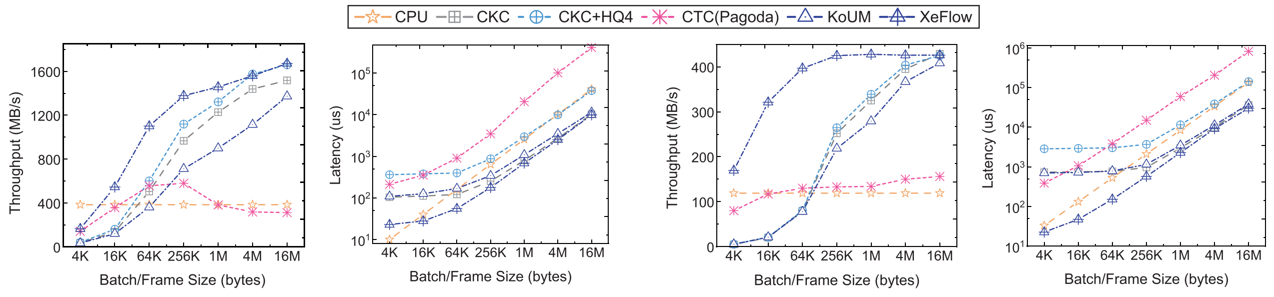
We further simulate AES encryption with *electronic code book (ECB)* mode in the GPU operator. Fig. 14c and 14d present the performance of AES encryption, where XeFlow also shows significant improvements over other methods. This experiment has a similar trend like erasure coding but has lower throughput due to more computations in AES encryption.

6.4 Application Evaluation: OLAP Query

Another application is "OLAP query" that scans the relational table by an SQL clause that consists of *select* and *filter* operators. Recent works [23], [24] employ GPU to accelerate OLAP query with the CKC workflow that needs to load the table into device memory before executing the query as a kernel. Unlike data encoding, OLAP query workloads tend to be more data-intensive. It implies that CPU-GPU transfer is the bottleneck. Thanks to fine-grained data structures, including columnar paging and lightweight indexing, XeFlow enables to skip transfer of undesired pages (see Section 4.3).

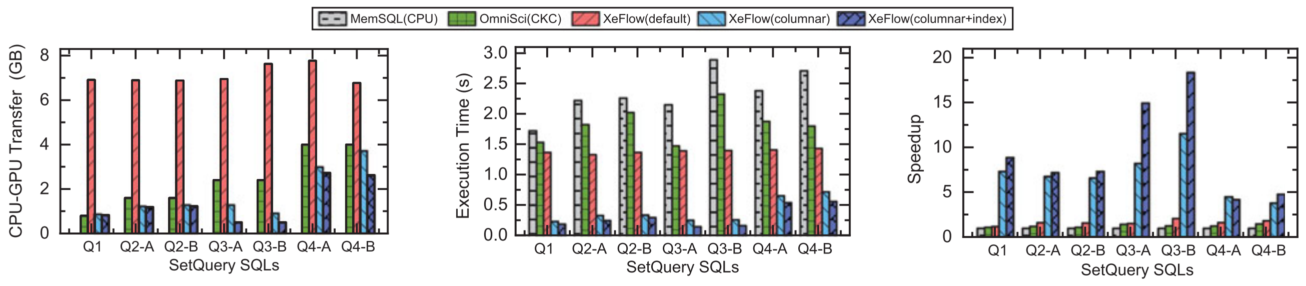
To evaluate the impact of these optimizations, we implement seven SQLs with XeFlow based on *SetQuery* [25], a synthetical OLAP benchmark. The evaluated table (i.e., 6.5 GB) consists of ten million items in 13 scalar columns. Before experiments, this table is also loaded into a ramdisk to bypass disk I/O during runtime. We also analyze other OLAP systems, including GPU-accelerated OmniSci [23] and CPU-only MemSQL [26].

Fig. 15a presents the amount of transfer between CPU and GPU sides during query execution. Technically, CPU-only MemSQL does not involve any CPU-GPU transfer. For Q1, Q2-A/B, Q3-A/B that only utilize a few columns, XeFlow reduces up to $80\%\sim85\%$ amount of CPU-GPU transfer by skipping undesired columns. Q4-A/B bring fewer $45\%\sim60\%$ reductions because more columns are scanned by the GPU side. With lightweight indexing, XeFlow shows further reduces $5\%\sim61\%$, especially for Q3-A/B with a smaller data selectivity. Restricted by coarse-grained copy, OmniSci cannot adopt these fine-grained data structures. Fig. 15b shows that the execution time follows an inverse



(a) Erasure Coding Throughput. (b) Erasure Coding Latency. (c) AES Encryption Throughput. (d) AES Encryption Latency.

Fig. 14. The performance of encoding.



(a) CPU-GPU Transfer. (b) Execution Time. (c) Normalized Speedup.

Fig. 15. The performance of OLAP query.

trend of CPU-GPU transfer. When CPU-GPU transfer dominates GPU acceleration, GPU-accelerated OmniSci is not better than CPU-only MemSQL. Thanks to fully overlapped table transfer, default XeFlow has $8\% \sim 64\%$ promotion over OmniSci based on CKC. With the combination of columnar paging and lightweight indexing, Fig. 15c illustrates that XeFlow achieves up to $4.8\times \sim 18.5\times$ speedup over MemSQL and $3.1\times \sim 10.3\times$ speedup over OmniSci.

6.5 Discussion

Even though XeFlow helps to upgrade current applications with GPU acceleration, it requires programmers to reorganize the application as a series of pipelines for data processing. Since current XeFlow does not support control operators like "if-else" and reduce operators like "aggregation", we still cannot adopt XeFlow to the application based on a more complex topology. To support these structures, the runtime system of XeFlow should be modified. Each control or reduce operator will incur a global synchronization before starting the next operator, which cannot be executed in a streamlined fashion.

7 Related Work

Considering the rapid development of hardware, the architecture of CPU-GPU applications is still an open issue. This section briefly concludes the relevant research.

Data Transfer within Discrete CPU-GPU Systems. [7] models the performance of CPU-GPU applications and presents the significant impact of data transfer. To simplify data management in CPU-GPU systems, ADSM [27] enables the CPU side to access GPU data transparently, but not vice versa. GPUfs [28] enables GPU to manipulate file data with POSIX-like API. Gensys [29] allows GPU to control memory allocation and file access by system calls. Gdev [30] supports oversubscribed device memory by swapping data between CPU and GPU sides. GPUnet [31] abstracts transfer primitives in the CPU-GPU system. Note that the above methods employ software invocations to perform inter-processor copy, while XeFlow solves this issue via hardware page fault with optimized coherency. HiWayLib [16] avoids redundant transfer when the areas of accessed data are overlapped and regular, while XeFlow skips the transfer of data that will not be computed by GPU.

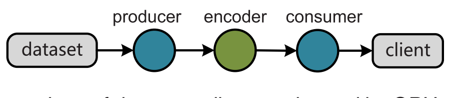


Fig. 16. The topology of data encoding accelerated by GPU.

CPU-GPU Programming Model. CUDA 10.0 provides the notion of "graph" to launch multiple GPU kernels with a single call. However, it does not change the CKC workflow when input data lies in the host memory. PTask [2] supports POSIX pipe with CPU and GPU, and Dandelion [4] uses it to execute compiled programs. XKaapi [3] runs CPU-GPU pipelines with work stealing. Although these works use pipeline at the API level, their underlying execution still employs the CKC workflow, which is inefficient to deal with small batches. [32] enables pipeline execution on the integrated GPU by hardware mechanisms. [5], [33] execute query pipelines on the integrated GPU, while our paper employs fine-grained data structures to address the bottleneck of PCIe bandwidth on the discrete GPUs. Pagoda [9] employs lightweight tasks to substitute heavyweight kernels but does not optimize data copy especially for large batches. By minimizing runtime invocations with persistent operators and shared topics, XeFlow ensures a good performance under both fine-grained and coarse-grained workloads.

The Trend of CPU-GPU Fusion. Nowadays, the fusion of CPU and GPU has been a trend for both industry and research. UM is a key technology for CPU-GPU fusion. [34], [35] focus on the hardware design of virtual GPU memory, which are not adopted to the current products. Before NVIDIA's Pascal GPUs providing real UM, prior works of virtual GPU memory [11], [12], [36] are limited by hardware supports. RSVM [37] and ActivePointer [38] achieve software virtual memory as an alternative when GPU hardware does not support virtual memory. [39] provides basic synchronization primitives in the GPU, we extend them for the discrete CPU-GPU system and amortizes the overhead of synchronization with optimized coherency. By modifying GPU memory designs, [40] allows CPU to get a subset of results from GPU before the kernel finishing. [41], [42] maintain inter-processor coherency through hardware protocols but need to modify the design of CPU and GPU. Therefore, they can only run on the simulators, while XeFlow provides a software solution for the current products.

GPU Kernel Scheduling. [43], [44] study how to integrate multiple operators into a single kernel during the compilation time. Whippletree [13] implements intra-GPU pipelines through the task queue and persistent thread but does not take CPU-GPU pipelines into considerations. VersaPipe [14] analyses multiple types of GPU execution models, including persistent thread that enables further control within GPU. Kernelet [15] partitions GPU kernels for multiple tasks based on the characteristics of memory access. However, Kernelet requires prior assumptions of black-box GPU hardware. Poise [45] uses machine learning to optimize thread organizations of kernel but does not consider the operators within a virtualized kernel. Inspired by these works, XeFlow divides thread warps in the GPU kernel for heterogeneous operators according to their resource demands.

8 Conclusion

This paper has presented XeFlow that exploits pipeline execution on the discrete CPU-GPU platforms. Limited by hardware, prior CPU-GPU systems still follow the CKC workflow or its variants, which are only optimized for coarse-grained scenarios. With the notion of persistent operator and shared topic, XeFlow minimizes inter-processor invocations during runtime. XeFlow also alleviates the cost of involved page faults through layered memory coherency. As for heterogeneous workloads, we optimize GPU scheduling to co-utilize computing and bandwidth resources. In evaluation, XeFlow achieves significant speedup for both coarse-grained and fine-grained cases. This paper explores a novel architecture for CPU-GPU software and encourages future works on the new GPU hardware.

Acknowledgments

This research was supported by National Key Research and Development Program of China (No. 2018YFB1003400), and National Natural Science Foundation of China (No. 61772204 and 61732014).

References

[1] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines," *Proc. VLDB Endowment*, vol. 12, pp. 544–556, 2019.

[2] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 233–248.

[3] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 1299–1308.

[4] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A compiler and runtime for heterogeneous systems," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 49–68.

[5] J. Paul, J. He, and B. He, "GPL: A GPU-based pipelined query processing engine," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1935–1950.

[6] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Scaling large-data computations on multi-GPU accelerators," in *Proc. 27th Int. ACM Conf. Supercomputing*, 2013, pp. 443–454.

[7] B. Van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for CPU-GPU data transfers," in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 11–20.

[8] H. Li, D. Yu, A. Kumar, and Y.-C. Tu, "Performance modeling in CUDA streams–A means for high-throughput data processing," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 301–310.

[9] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-grained GPU resource virtualization for narrow tasks," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 221–234.

[10] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Proc. Innovative Parallel Comput.*, 2012, pp. 1–14.

[11] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on NVIDIA GPUs," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 1092–1098.

[12] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2014, pp. 1–6.

[13] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg, "Whippletree: Task-based scheduling of dynamic workloads on the GPU," *ACM Trans. Graph.*, vol. 33, 2014, Art. no. 228.

[14] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "VersaPipe: A versatile programming framework for pipelined computing on GPU," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 587–599.

[15] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1522–1532, Jun. 2014.

[16] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "HiWayLib: A software framework for enabling high performance communications for heterogeneous pipeline computations," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 153–166.

[17] A. Berson and S. J. Smith, *Data Warehousing, Data Mining, and OLAP*. New York, NY, USA: McGraw-Hill, 1997.

[18] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proc. 27th Int. Conf. Very Large Data Bases*, 2001, pp. 169–180.

[19] G. Lu, Y. J. Nam, and D. H. Du, "BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.

[20] A. V. Adinetz and D. Pleiter, "Halloc: A high-throughput dynamic memory allocator for GPGPU architectures," in *Proc. GPU Technol. Conf.*, 2014, pp. 1–59.

[21] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "ScatterAlloc: Massively parallel dynamic memory allocation for the GPU," in *Proc. Innovative Parallel Comput.*, 2012, pp. 1–10.

[22] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Uni. Illinois at Champaign, Champaign, IL, Tech. Rep. IMPACT-12–01, 2012.

[23] C. Root and T. Mostak, "MapD: A GPU-powered big data analytics and visualization platform," in *Proc. Int. Conf. Comput. Graph. Interactive Techn.*, 2016, pp. 1–2.

[24] S. Zhang, J. He, B. He, and M. Lu, "OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures," in *Proc. IEEE Symp. Very Large-Scale Data Anal. Vis. Endowment*, 2013, pp. 1374–1377.

[25] P. E. O'Neil, "The set query benchmark," The Benchmark Handbook, pp. 1–39, 1993.

[26] 2019. [Online]. Available: https://www.memsql.com/

[27] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. J. Patel, and W. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proc. 15th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 347–358.

[28] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a file system with GPUs," *ACM Trans. Comput. Syst.*, vol. 32, 2014, Art. no. 1.

[29] J. Veselỳ, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic system calls for GPUs," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 843–856.

[30] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-class GPU resource management in the operating system," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, Art. no. 37.

[31] M. Silberstein *et al.*, "GPUnet: Networking abstractions for GPU programs," *ACM Trans. Comput. Syst.*, vol. 34, 2016, Art. no. 9.

[32] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on GPUs," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, 2014, pp. 181–192.

[33] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood, "Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries," in *Proc. 11th Int. Workshop Data Manage. New Hardware*, 2015, pp. 1–8.

[34] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "GPU virtualization and scheduling methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 50, 2017, Art. no. 35.

[35] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 37–48.

[36] J. Hestness, S. W. Keckler, and D. A. Wood, "GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 87–97.

[37] F. Ji, H. Lin, and X. Ma, "RSVM: A region-based software virtual memory for GPU," in *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, 2013, pp. 269–278.

[38] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A case for software address translation on GPUs," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, pp. 84–95, 2018.

[39] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs," pp. 1–12, 2011, *arXiv:1110.4623*.

[40] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 354–365.

[41] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 339–351.

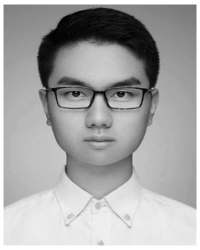
[42] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU–GPU HW cache coherence," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 494–506.

[43] C. Gregg, J. Dorn, K. M. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Proc. 4th USENIX Conf. Hot Top. Parallelism*, 2012, Art. no. 10.

[44] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded GPU," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun. and Int. Conf. Cyber Phys. Soc. Comput.*, 2010, pp. 344–350.

[45] S. Dublish, V. Nagarajan, and N. Topham, "Poise: Balancing thread-level parallelism and memory system performance in GPUs using machine learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 492–505.

Zhifang Li received the bachelor's degree from East China Normal University, Shanghai, China, in 2016, where he is currently working toward the PhD degree. His research interests include real-time analytical system, heterogenous computing system, and distributed database system.



Beicheng Peng received the bachelor's degree from East China Normal University, Shanghai, China, in 2018, where he is currently working toward his master's degree. His research interests include massively-parallel computer system, heterogenous computing, and distributed database system.

Chuliang Weng received the PhD degree from Shanghai Jiao Tong University, Shanghai, China, in 2004. He is currently a professor in East China Normal University (ECNU), Shanghai, China. Before joining ECNU, he worked at Huawei Central Research Institute and was an associate professor in Shanghai Jiao Tong University, Shanghai, China. He was also a visiting research scientist in Columbia University, New York City, New York. His research interests include parallel and distributed systems, storage systems, OS, and system security.