HiWayLib: A Software Framework for Enabling High Performance Communications for Heterogeneous Pipeline Computations

Zhen Zheng
Tsinghua University
z-zheng14@mails.tsinghua.edu.cn

Chanyoung Oh
University of Seoul
alspace11@uos.ac.kr

Jidong Zhai
Tsinghua University, BNRist
zhaijidong@tsinghua.edu.cn

Xipeng Shen
North Carolina State University
xshen5@ncsu.edu

Youngmin Yi
University of Seoul
ymyi@uos.ac.kr

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

Abstract

Pipeline is a parallel computing model underpinning a class of important applications running on CPU-GPU heterogeneous systems. A critical aspect for the efficiency of such applications is the support of communications among pipeline stages that may reside on CPU and different parts of a GPU. Existing libraries of concurrent data structures do not meet the needs, due to the massive parallelism on GPU and the complexities in CPU-GPU memory and connections. This work gives an in-depth study on the communication problem. It identifies three key issues, namely, slow and error-prone detection of the end of pipeline processing, intensive queue contentions on GPU, and cumbersome inter-device data movements. This work offers solutions to each of the issues, and integrates all together to form a unified library named HiWayLib. Experiments show that HiWayLib significantly boosts the efficiency of pipeline communications in CPU-GPU heterogeneous applications. For real-world applications, HiWayLib produces 1.22~2.13× speedups over the state-of-art implementations with little extra programming effort required.

CCS Concepts • **General and reference → Performance**; • **Computing methodologies → Parallel computing methodologies**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords pipeline communication, CPU-GPU system, contention relief, end detection, lazy copy

ACM Reference Format:
Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2019. HiWayLib: A Software Framework for Enabling High Performance Communications for Heterogeneous Pipeline Computations. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304032

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304032

1 Introduction

Pipeline execution model [5] is widely used for stream-based applications ranging from network packet processing, graph rendering, video encoding and decoding, machine learning, to image processing [1, 17, 22, 39, 49]. In a pipeline computation, an application or a task graph is implemented as a set of concurrently running stages, in which the output of a former stage serves as the input of the latter stage. This work focuses on pipeline programs running on CPU-GPU heterogeneous platforms, where, the different stages of a pipeline program run concurrently on both CPU and GPU.

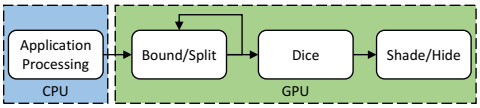


Figure 1. Reyes rendering pipeline.

Figure 1 shows an example, Reyes rendering pipeline [7], which consists of four main stages. The first stage runs on CPU and generates data items for the following stages on GPU. The second stage (object partitioning) has a recursive structure, in which, data items are produced and then fed back to the stage itself until some condition is met (e.g., object size is small enough). With pipeline programming, a ready task generated by a former stage can be processed by the next stage while the former stage is working on other task items. It helps deliver high responsiveness, and exploit both data parallelism within each stage and task parallelism between stages [54].

One of the important components of such pipelines is the mechanism for supporting communications between stages, *both within and across* devices. Inter-stage communications are needed for coordinating the computations of different stages, as well as passing task items through the pipeline.

Developing a sound and efficient communication support for pipeline computations is especially challenging on CPU-GPU heterogeneous platforms for its massive parallelism and complex and continuously changing memory systems. Massive parallelism complicates coordinations between stages, while memory complexities (e.g., weak memory coherence support) make consistency of data copies tricky to maintain.

In current practices, the communications among pipeline stages are typically implemented by the developers of a pipeline application. The ad-hoc implementations often suffer from serious performance issues.

Recent several years have seen some efforts in building pipeline programming frameworks for simplifying the development efforts. WhippleTree [45], for instance, is a framework built on C++ templates. Through it, programmers only need to specify the computations in each stage and the connections of the stages, the framework automatically creates task queues and schedulers to materialize the communication support of the pipeline. A follow-up work, VersaPipe [54], improves the GPU resource usage and task scheduling by enabling more flexible pipeline execution models. *These frameworks concentrate on computation rather than communication.* They try to maximize GPU resource utilization by the pipeline program, but use only the rudimentary mechanism (basic task queues) for inter-stage communications. Moreover, they do not provide APIs for inter-device communications as they assume all stages of a pipeline reside on GPU; all data to be used by the pipeline are simply copied from CPU to GPU before GPU code starts running.

In this work, we strive to address the problem by developing HiWayLib, a High Performance Multi-Way Communication Library for heterogeneous pipeline computations. Through three key techniques, HiWayLib overcomes the main barriers for efficient multi-way (CPU to GPU, GPU to CPU, GPU to GPU) communications in heterogeneous pipeline computations, providing an efficient solution for the development of pipeline programs on CPU-GPU systems.

We have designed the three key techniques in HiWayLib respectively for overcoming the following three main barriers for effective communication support.

(1) **Cumbersome inter-device data movements**. Communications between CPU and GPU are subject to lower bandwidth and longer latency than intra-device communications. Recent GPU architectures support Unified Memory technique in which a single address space is provided to the distributed memories in a CPU-GPU heterogeneous system. Although this technique makes programming of cross-device memory access easier, it does not show good performance for the frequent and simultaneous cross-device access patterns that can be found in communication of stages in a pipeline [26]. Some common techniques such as prefetching [28] may mitigate the problem to a certain degree, but inter-device data movements still remain a frequent bottleneck in pipeline programs.

HiWayLib employs a novel technique named *Lazy Reference-Based Scheme* to mitigate the problem. It is based on an observation that in many pipelined programs (e.g., image processing), the data between tasks often have overlaps. Instead of moving the data of each task separately, *Lazy Reference-Based Scheme* avoids repeated movements of the overlapped data by replacing the task-based data copy with a region-based lazy data copy, and hence significantly reduces the data movement overhead.

(2) **End detection of pipeline processing**. It is tricky to determine when a pipeline program finishes processing all tasks in an input stream. With recursive structure in a pipeline (e.g., Figure 1), stages can generate new data items dynamically; as a result, the total data item count is unknown ahead of time. For immediate detection of the end of pipeline processing, a solution may need to track the status of all task items, no matter whether it is in the queue/buffer or on the fly, or whether it is on CPU or GPU. The tracking overhead can be large. If the pipeline program runs on multiple devices, the status tracking and checking would incur extra cross-device communications.

HiWayLib tackles the problem with a novel technique named *Late Triggered Inter-Stage Tracking*. By appending special markers to a stream of inputs, the technique triggers status tracking and checking only when the execution is getting close to the finish. It hence can still immediately report the end of pipelined processing while avoiding most unnecessary pipeline status checking overhead.

(3) **Intensive contentions on communication data structures on GPU**. On GPU, each stage of a pipeline may have many threads running, whose accesses to shared communication structures (e.g., task queues) may cause severe access contention and performance hazards.

HiWayLib employs a technique named *Bi-Layer Contention Relief* to alleviate the problem. The technique uses a two-level delegation scheme to minimize the number of concurrent accesses to the communication data structures without delaying task executions.

As mentioned, for the complexities in hardware and pipeline, manually implementing effective communications is challenging for general programmers. Existing pipeline programming frameworks are subject to soundness and efficiency problems. Although our three proposed techniques help with performance, they add even more complexities to the already involved implementations of pipeline programs. HiWayLib frees programmers from those concerns by integrating the proposed techniques together and offering a set of uniform simple APIs for different heterogeneous platforms.

Experiments show that HiWayLib significantly boosts the efficiency of pipeline communications in CPU-GPU heterogeneous applications. For real applications, HiWayLib produces 1.22~2.13× speedups over the state-of-art implementations with little extra programming effort required.

2 Background

This section offers background knowledge on CPU-GPU systems.

Memory Systems On a CPU system with a discrete GPU, the bandwidth of the CPU-GPU connection is usually much lower than the memory bandwidth in GPU (e.g., 16GB/s of PCIe-3 ×16 versus 700GB/s of the main memory on Pascal GPU). On IBM Power-9 systems with NVIDIA GPUs, the connection is through NVLink, which can provide much higher bandwidth than PCIe does. But the bandwidth of inter-device communication still remains several times lower than that of main memory on GPU devices [10].

On previous generations of CPU-GPU systems, the two kinds of processing devices have separate address spaces; data need to be explicitly copied between them through APIs for processing. Recent GPU architectures support a cross-device memory access technique called Unified Memory (UM) [32]. With UM technique, CPU and GPU share an address space; one variable can be accessed by different devices without explicit memory copy. The underlying architecture automatically copies data on demand at memory page level.

UM technique is built upon the hardware-based detection of memory page faults between devices. When a device tries to access a variable that is not on its main memory, a page fault happens and the page that this variable resides on gets automatically copied from the other device to this device. While there are heuristic algorithms to make page migration efficient, programs still suffer when it needs to frequently access data on the same page from both CPU and GPU. Such cases are particular common on task queues connecting two pipeline stages residing on two different devices. One device enqueues new tasks frequently while the other device dequeues from the same queue simultaneously. The corresponding memory pages may migrate back and forth between the devices frequently.

With the UM technology, memory coherence between the last-level caches of CPU and GPU becomes supported. Also, system-wide (across CPU and GPU) atomic operations become possible through a solution that combines the new hardware features and the system software support. Both features give certain conveniences to the implementation of pipeline applications. However, the usage of both features is subject to large time overhead.

The old explicit data copy-based method is still applicable on new GPUs—in fact, for its frequently better performance over UM, the explicit copy method remains popular in modern GPU programs.

GPU Execution Models Threads in a CUDA program are organized as warps and thread blocks. One warp contains 32 threads. All these 32 threads are issued at the same time and execute the same instruction in lock-step mode if there is no branch in the program. With warp shuffle operation, threads of a warp can exchange data with each other directly without going through shared (or global) memory, which makes interchanging data among threads efficient. One thread block contains one or more warps depending on the configuration of the CUDA function. Different warps are issued independently, and some functions are provided to synchronize all warps in a block. In some recent works, persistent thread technique is used for flexible task scheduling on GPU. With this technique, every thread stays alive throughout the execution of a kernel function. The function typically contains a loop, through which, the threads continuously fetch new data items from a task queue and execute the corresponding procedures.

It is worth noting that the focus of this work is on communications. Finding the best placements of pipeline stages on different devices in a system is a problem orthogonal to this work. We assume that the placement has been decided through some approach. Our following discussions are based on heterogeneous systems with CPU and NVIDIA GPUs, but the general principles behind the approach could be applied to other discrete heterogeneous platforms.

3 Design and Implementation

This section presents the challenge, design, and implementation details of HiWayLib. The design considers both efficiency and correctness for communication on heterogeneous systems.

Our description concentrates on the four advanced features of HiWayLib, which respectively address the four main barriers existing work faces in supporting efficient pipeline executions. Section 3.2 explains how HiWayLib solves the problem of cumbersome inter-device data movements, Section 3.3 describes how HiWayLib timely and efficiently detects the end of pipeline processing, Section 3.4 explains how HiWayLib addresses the problem of intensive queue contentions on GPU, and Section 3.5 describes the overall HiWayLib API for programming productivity and some considerations in the library implementation for ensuring the soundness of the communication support in the presence of GPU memory complexities. Before presenting these features, we first describe the basic ring-buffer based communication mechanism of HiWayLib, as understanding it is the prerequisite for comprehending the subsections on the advanced features of HiWayLib.

3.1 Basic Communication Scheme

The basic communication scheme in HiWayLib is a ring-buffer based scheme for inter-device communications.

Different stages of a pipeline program can be put onto different devices on a CPU-GPU heterogeneous system. When two adjacent stages in a pipeline program run on different devices, inter-device communications are necessary. As discussed in Section 2, although recent GPUs provide unified memory to ease the programming of communication between CPU and GPU, frequent accesses of shared data between CPU and GPU still suffer large performance penalty from page-faults. We employ a batched data movement approach for efficiency. The basic idea is to store the output of a former stage into a temporary buffer, and then transfer them to another device in batch. With this method, the communication may better utilize Direct Memory Access (DMA) to mitigate the transfer overhead between devices.

Specifically, HiWayLib implements this basic scheme for inter-device communications as follows. A queue is attached with a consumer stage and a buffer is attached with a producer stage. For the buffer, HiWayLib uses a ring-buffer to store data items needed to be transferred between adjacent stages. A ring-buffer is a circular queue data structure whose head and tail are connected. When the number of data items exceeds a predefined threshold in the buffer, all these data need to be moved into the queue of consumer device. The buffer is non-blocking. While moving data into the queue, new data items can be inserted to the buffer simultaneously until it is full. Beside the threshold, the buffer will be also flushed if a timeout occurs. When the queue or buffer is full, the producer retries the data insertion until a success.

To guarantee that the ring-buffer is thread safe, each enqueue/dequeue operation updates some counters of the buffer—such as the front/end index and the number of items—atomically, and get a position to write/read data items. Also, the target queue on another device maintains a same set of counters in the same way. When the buffer prepares to flush data to the target queue on another device, a function named *enqPrep* is invoked to update the *front* counter of the target queue by the number of flushed data items and get the address to write data back, and then mark the corresponding queue area as "busy" until all flushed data have been written back to the queue. Only the queue area marked "busy" is not allowed to be accessed by other threads; other areas of the queue are free for update.

Figure 2 shows the hand-shaking process of batched data movement. Each device launches several threads (2 threads on CPU and a one-warp-kernel on GPU that uses persistent threads [3]) as *monitors* to maintain the buffer status. They are the main part of HiWayLib runtime. Each thread for a stage checks whether the number of data items in the buffer exceeds the threshold after enqueuing, and the monitor checks whether timeout occurs for each stage in a round

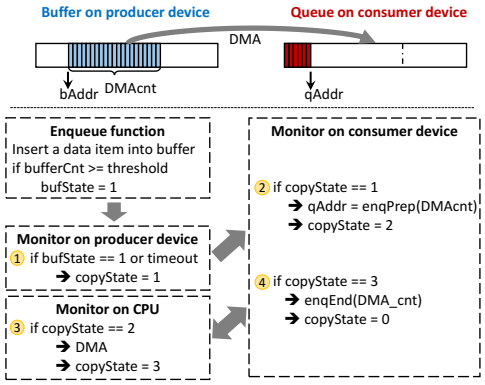


Figure 2. Hand-shaking process of batched data movement. Variables *bAddr*, *DMAcnt*, *qAddr* and *copyState* are unified memory variables that can be accessed by all devices.

robin fashion. If a timeout occurs, a per-device global flag (*bufState*) is set. When the monitor on device *Producer* finds that *bufState* is updated, it will set flag *copyState* (step 1 in Figure 2) and ask the monitor on device *Consumer* to invoke function *enqPrep* and get the write-back address of the queue (step 2). The monitor on *Producer* then writes the buffer address and the number of flushed data items to specific system-wide global variables (*bAddr* and *DMAcnt*), and the monitor on *Consumer* will store the write-back address of the queue into a specific system-wide global variable (*qAddr*) and raise a system-wide global flag *copyState* to inform the monitor on CPU to execute DMA functions with *bAddr*, *qAddr* and *DMAcnt* (step 3). After the DMA operation, *copyState* is reset (step 4).

We note that in HiWayLib, the pipeline stages on GPU use persistent threads [3], in which, only a limited number of thread blocks are launched for a GPU kernel such that they can all be active on GPU. And every thread lives permanently on GPU (by running a while loop), continuously trying to grab new tasks to process. This model avoids frequent kernel launching overhead which is essential for high responsiveness of pipeline applications [45, 54]. A problem with this model is how to efficiently detect the end of all tasks and exit quickly. It is called an *end detection* problem, which we will discuss more in Section 3.3.

With the basic communication mechanism explained, we are ready to move on to the advanced features of HiWayLib where the main novelty of this work resides. These features respectively address the main issues faced by existing pipeline programming frameworks. The solutions together pave the way for a streamlined support of efficient pipeline computations on CPU-GPU systems.

3.2 *Lazy Reference-Based Scheme*

Due to the limited inter-device bandwidth and relatively long latency, inter-device data movements create a major bottleneck frequently seen in CPU-GPU pipeline computations.

This part describes a novel technique named *Lazy Reference-Based Scheme* to mitigate the problem. It is based on an observation that in many pipelined programs (e.g., image processing [2, 33]), the data used between tasks often overlap. *Lazy Reference-Based Scheme* avoids repeated movements of the overlapped data by replacing the default *task-based* data copy with a new lazy reference-based scheme, along with some changes to the implementations of task queues.

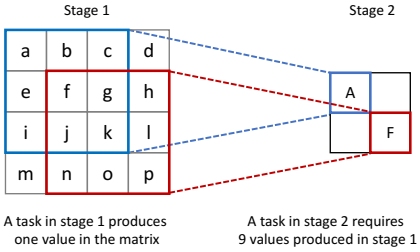


Figure 3. Inter-task data overlaps in a stencil pipeline.

We explain the observation with a stencil pipeline program. Figure 3 depicts the dependence between two stages of a stencil pipeline program. Each task in stage 1 produces one value in the left-side matrix. Each task in stage 2 requires 9 values from that matrix. The values required by task *A* and task *F* in stage 2 are partially overlapped (values *f, g, j, k* produced by stage 1).

Existing pipeline frameworks (e.g., [45, 54]) do not consider such overlappings. They use a *task-based copy paradigm*, which copies the data needed by each individual task into a *task queue* separately without considering the relations between tasks. For the example in Figure 3, the blue tile will be copied into the *task queue* as part of the task *A* of stage 2, and the red tile will be also copied into the *task queue* as part of the task *F*, causing values (*f, g, j, k*) being copied twice unnecessarily.

Lazy Reference-Based Scheme deals with the overlapping between tasks on two principles. First, it uses references rather than data content when forming a task. It means that the definition of a task in the *task queue* contains only a reference to the start location (i.e., the index in matrix) of data the task requires. As the stencil size is given (through API in Section 3.5), with the start location, the range of all data the task requires can be determined. For clarity, in the following we use *target values* to refer to the actual values of data needed by a task. For instance, for task A in Figure 3, its task content will be (0,0). From it, the corresponding stencil to process can be determined as the range from (0,0) to (2,2) given that the stencil size is 3 by 3. In our solution, we use the relative location as the reference as it gives good portability, allowing different stages on different devices to locate the data easily. This principle is used for both inter-device and intra-device communication for stencil pipeline programs.

The second principle is that *Lazy Reference-Based Scheme* copies data in a lazy manner in a *region-based* rather than *task-based* manner to capitalize on inter-task relations. It

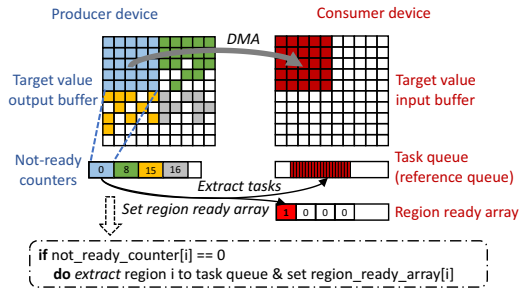


Figure 4. *Lazy Reference-Based Scheme.* The range size for a lazy copy in this example is set to 5×5.

means that *target values* will not be copied onto the device of the next stage immediately when they are produced, but will be stored into a *target value buffer* (on the device where the data are produced) temporarily and a scoreboard is marked to indicate their readiness. HiWayLib runtime periodically examines the scoreboard and copies a whole region of data—which we call *lazy copy region*. A *lazy copy region* is a set of adjacent data items that are copied together. Its size could be larger than the range of data one consumer task requires. This *region-based* copy scheme copies a region of data only once, regardless of how many consumer tasks require the data. It hence avoids the duplicated data movements happening in the default *task-based* scheme.

The implementation of *Lazy Reference-Based Scheme* involves changes to the scheme of data movements as well as the design of task queues. Figure 4 shows the implementation details. On the producer side, two data structures are created. The *target value output buffer* is the holder of actual values produced by the producer for the next stage. The *not-ready counters* serve as the scoreboard with each recording the number of data items in each *lazy copy region* that has not yet been produced. On the consumer device, three data structures are created. The *target value input buffer* receives the values copied from the *target value output buffer* of the producer. The *task queue* holds the tasks to be processed in the consumer stage. As mentioned earlier in this section, in this scheme, the definition of a task in the *task queue* contains only a reference to the start location of the data the task requires; so the content of the *task queue* is simply data references. The *region ready array* serves as the indicator to tell whether the corresponding *lazy copy region* has already been copied to the consumer's *target value output buffer*.

As producer puts produced data into *target value output buffer*, it updates the *not-ready counters* accordingly. When HiWayLib runtime periodically examines *not-ready counters*, it copies the regions whose *not-ready counters* have turned 0 and then sets the counters to -1. Meanwhile, it enqueues into consumer's *task queue* the reference to every data item contained in the just copied *lazy copy regions*; each of the references corresponds to the start of one stencil yet to process and hence a task of the consumer. When the consumer dequeues its *task queue*, based on the reference and the *region*

ready array, it checks whether all data in the corresponding stencil have been copied to its *target value input buffer*. If so, it processes the task, otherwise, it re-enqueues the reference to the task queue for later processing.

HiWayLib provides a set of APIs (described in Section 3.5) to allocate areas for all above buffers and queues. Users just need to provide task data type, matrix size, stencil size, and lazy region size through the APIs. Buffers are recycled when they become useless to save memory space.

With the above lazy copy approach, not only many redundant data movements are avoided, but also the data locality becomes better as adjacent data items are transferred together. Our current implementation is mostly designed for stencil processing pipeline programs as they are the most common pipeline programs with such data overlaps, but the basic idea can be extended to other pipeline application whose tasks have overlapped content.

3.3 Late Triggered Inter-Stage Tracking

Besides inter-device data movements, another main source of delay for pipeline computations is the detection of when the whole pipeline processing on a stream of inputs has finished. We call this problem *end detection problem*. Delays in the detection could affect the follow-up operations and hence increase the latency of a pipeline program.

One of the main challenges for *end detection* is that sometimes the total number of task items needed to process is unknown beforehand. A stage may dynamically create many task items; and with recursive structures in a pipeline (e.g., the back edge in Figure 1), a later stage may generate and enqueue tasks to an earlier stage in the pipeline. These dynamic complexities make simple task counting insufficient for *end detection*.

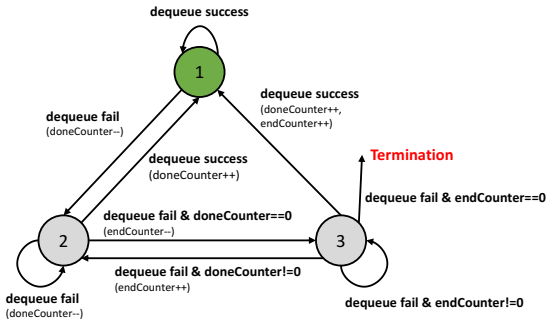


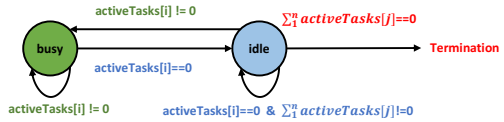
Figure 5. The FSM in WhippleTree for end detection.

A previous pipeline framework for GPU, WhippleTree [45], for instance, develops a delicate and complicated finite state machine (FSM) to check the ending of a pipeline (Figure 5). Each GPU thread block maintains two counters (*doneCounter* and *endCounter*) whose initial values equal the total number of thread blocks launched. (As WhippleTree uses persistent threads, the total block count is exactly the number of thread blocks that the program can execute concurrently on GPU.) As the FSM shows, a thread block may be in one of three states before termination, and depending on the state of a thread block, its two counters get updated accordingly when task dequeuing happens. A thread block exits when dequeue fails and its *endCounter* reaches 0. The program terminates after all blocks exit. WhippleTree works well when all stages of a pipeline are put into a single kernel. But if stages are mapped into multiple kernels (e.g., via CUDA Streams) and inter-kernel communications use queues, the above method does not work properly. A scenario is that when the program starts, a previous kernel has not yet produced any data item for a following kernel. Thus the following kernel cannot fetch any data item and the counters will drop to zero quickly, causing the kernel to terminate prematurely [45].

Some other implementations of pipeline (e.g., VersaPipe [54]) simply use time-out for end detection. They start a timer when all task queues become empty and terminate the pipeline when the timer times out after a period of time. The problem with the scheme is that it cannot guarantee that the processing is truly over; tasks coming after the timeout cannot get processed. The implementations hence usually set the timeout a long time (e.g., 10 times of a single task length in VersaPipe), causing a long delay without completely avoiding the risk of premature terminations. (Our basic scheme in Section 3.1 uses this method.)

Moreover, none of the previous solutions are designed for handling inter-device pipelines.

To address these problems, we propose a novel method, called *Late Triggered Inter-Stage Tracking*, which consists of two key techniques: inter-stage status tracking, and late triggering. We first explain inter-stage status tracking scheme, and then explain how late triggering helps with the efficiency and correctness.



(a) Finite state machine for termination checking in HiWayLib

Operations on device i		incurred operations to activeTasks[i]
enqueue	→	activeTasks[i]++
insert to buffer	→	activeTasks[i]++
task executes and ends	→	activeTasks[i]--
DMA to another device done	→	activeTasks[i] -= #tasks transferred by DMA
DMA from another device	→	activeTasks[i] += #tasks transferred by DMA

(b) Mechanism to update activeTasks[i] on device i

Figure 6. The FSM for end detection in HiWayLib.

Inter-Stage Tracking The essential feature of *inter-stage tracking* to address the limitations of previous designs lies in its co-operations among different stages on different devices for status tracking. We first explain the idea in the case where the pipeline has continuous inputs, which means there is no gap between the arrival times of two adjacent input items. The non-continuous scenario will be discussed in *Late Triggering* part.

Figure 6 illustrates the scheme with a finite state machine. Note that the scheme uses a counter for each device (rather

than each stage). The counter (*activeTasks[i]*) records the number of active tasks on device *i*. In this scheme, a pipeline stage can be in one of two states before termination: *busy* and *idle* (all stages on the same device are in the same state at a given moment). When counter *activeTasks[i]>0*, the stages on the *i*th device are all *busy*. They turn to *idle* state when the counter becomes 0. When counters on all devices drop to 0, all stages on all devices terminate.

Figure 6(b) shows how the counters are updated. The counter on a device increases by 1 when there is a task enqueue operation on this device. When a task on that device finishes, the counter decreases by 1. There are some special considerations in our design for inter-device operations. When a task item is inserted into a buffer on device *i* that is intended to be transferred to another device, *activeTasks[i]* also increases by 1. When the buffer is transferred to the other device (through DMA), *activeTasks[i]* deducts by the number of task items in that buffer. When DMA transfers a buffer to device *i*, *activeTasks[i]* increases by the number of task items contained in the buffer. This design ensures correct tracking of inter-device task transfers which is essential for the soundness of the end detection.

The correctness of the *Inter-Stage Tracking* scheme is straightforward to prove. ① With the method, a pipeline will not terminate when there are still unfinished tasks, because *activeTasks[i]* tracks the tasks that are not yet finished on device *i*. If there are still unfinished tasks on any device, the counter on that device must be greater than 0 and the pipeline cannot terminate. ② With the method, a pipeline terminates when all tasks are done, because when and only when a task is generated on or sent to device *i*, its counter increases by 1. When and only when a task ends on the device, the counter decreases by 1. Hence, when all tasks end, the counters on all devices must be all 0 and the pipeline hence terminates.

Late Triggering In the basic *inter-stage tracking* scheme, the counters on all devices indicate the total number of task items that are either being processed or need to be processed. If all counters are zero, it indicates the end of the pipeline processing. A key requirement is efficiently checking whether all counters are zero; the check needs to happen atomically to avoid data races. It is tricky to do cross the boundaries between CPU and GPU. Our basic method is to use spin-locks on a unified memory variable. To guarantee the correctness, all the checking transactions and counter update operations need to compete for the same lock.

This basic method suffers from two problems. The first is that the spin-lock–based checking introduces large time overhead as all counters on all devices compete for the same global lock. The second problem is that, if the input stream of a pipeline has some "bubbles" (i.e., intervals between the arrivals of data items), all counters may drop to zeros and the pipeline would terminate prematurely.

Our *late triggering* scheme helps *inter-stage tracking* solve both problems. HiWayLib provides an interface, through which programmers can append a special "end marker" to the end of an input stream. With this marker appended, HiWayLib performs the inter-stage tracking and status checking *only when* the pipeline detects that marker. This method addresses two problems of the basic method. First, because it avoids all the checking and status updates before seeing the end marker, it avoids most of the unnecessary tracking and status checking time overhead. Second, for the same reason, it avoids premature terminations when there are "bubbles" in an input stream as it has not yet detected the end marker. And after the input end marker is seen, the definition of an end marker means that there will be no "bubbles" ever coming in the input stream. The *inter-stage tracking* scheme has no problem in faithfully tracking active tasks that have already arrived at devices as the earlier part of this section has already shown.

3.4 Bi-Layer Contention Relief

Intensive contention on communication data structures (e.g, task queues), is another main challenge for efficient pipeline computations. This problem becomes even harder on GPU as a large number of threads need to access the same task queue concurrently. One common practice to reduce contention is to split one single task queue into multiple sub-queues and each sub-queue handles partial queue operations [8, 9]. But this method can introduce extra scheduling overhead to keep load balance among different sub-queues.

To mitigate such contention, we propose a technique called *Bi-Layer Contention Relief*. Our approach adopts a two-level delegation scheme to minimize the number of concurrent accesses to the communication data structures without delaying task executions. The basic idea is to select a delegator from a number of threads to access the shared data structure for those threads. Conventional methods [35, 54] aggregate threads in either warp level or thread block level, which can lead to block-level synchronization. However, our approach features a distinctive two-level design. The first level leverages *warp-level shuffle* operations for efficiency, and the second level employs an *auto-grouping mechanism* to avoid risky costly synchronizations.

Figure 7 illustrates the basic idea of our approach. The following discussion uses enqueue operations to describe our approach, but the same mechanism applies to dequeue operations. The first layer is within a single warp. As threads in a warp always execute in a lock-step mode, when one thread does an enqueue operation, all the other threads in the same branch within the same warp will also do enqueue operations. The first thread in this branch within the warp will be selected as a sub-delegator. In the second layer, each sub-delegator acquires a starting address of a memory region in the queue, which is used to store the data that all the threads need to enqueue. To get corresponding offset in

the region for each thread within the same warp, we take advantage of warp shuffle operations [31], which makes the communications between threads within a warp very efficient. And then all the threads can write their data into the queue concurrently.

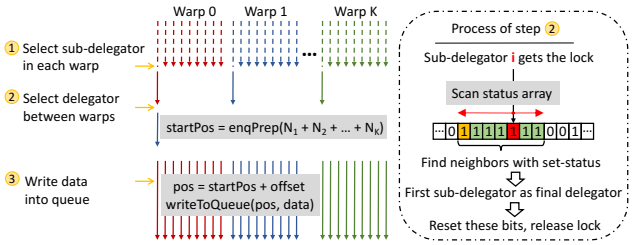


Figure 7. Delegator-based enqueue approach to reduce contention. Function *enqPrep* is to get the start position in the queue to write the data of all threads.

The second layer is within a thread block. For a block, if some of its warps need to enqueue at a similar time, our approach selects a delegator to update queue counters for all of them. There are two differences from the first-level delegation: Different warps do not execute in a lock-step mode, and there are no shuffle instructions across warps. To aggregate operations of different warps, one option is to insert block-level barrier and update queue counters once for the whole blocks. However, the inserted barrier could cause dead locks. Consider a case where thread x waits for a flag to be set to true by thread y which belongs to a different warp but in the same thread block. If the barrier is inserted before thread y sets the flag, a dead lock would happen.

To address this problem, we propose an auto-grouping scheme. The core idea is to group as many warps as possible each time (not necessary all the warps in a block). Even if the final delegator may not perform queue operations on behalf of all warps, block synchronization can be effectively avoided. Specifically, each thread block maintains a data strucutre called *status array* with size of 32 (current maximum warp count in a thread block) and a lock for the *status array* on shared memory. One *status* corresponds to one warp. When a warp needs to enqueue some data, its sub-delegator sets the corresponding status in the array and tries to acquire the lock for the array if its status stays as *set* (value of 1). If it acquires the lock successfully, it checks its adjacent status through bi-direction scanning and stops once it encounters unset status on both directions. All the sub-delegators corresponding to the adjacent set status form a group. The first sub-delegator in this group is selected as the final delegator to do the queue operations for the entire group. As mentioned above, each sub-delegator then will get a starting address of a memory region in the queue to write data. The status of sub-delegators in the group is cleared after the queue operation finishes, and the acquired lock is released. In such a way, our scheme can avoid inserting block synchronizations.

3.5 HiWay Library and API

API HiWayLib provides a set of APIs that ease the adoption of those optimizations by application developers. For inter-device communication, HiWayLib supports both batched data copy method and discrete unified memory access. With simple configurations in the API, developers can choose which communication method to use.

Figure 8 shows the main interfaces of HiWayLib. The functions are C++ based. Note that we call the data transfer media a *tunnel* in the API rather than queue for the generality of the API. Even though we currently implement the *tunnel* with a queue-based data structure, other data structures (stack and heap) could be used for the implementation.

To use HiWayLib, users should firstly create tunnels between stages with *createTunnel*. If cross-device batched data copy is required, users should set up monitors for DMA control with *setupService*. *put_x* ('x' should be 'h' on CPU, and 'd' on GPU) to put data items into a tunnel. Pipeline stages can get data from a specific tunnel with *get_x*. API *endTask_x* is used to help track the number of active task items, which should be called whenever a task ends. API *isTerminate_x* will return TRUE unless the pipeline should terminate. When that happens, all monitor services will be shutdown automatically. Meanwhile, users still have the choice to shut down DMA services manually with *shutdownService*.

For stencil pipelines, *configStencilTunnel* should be called before setting monitor service. The *config* parameter includes the matrix size, stencil size, and copy region size, which should be specified by users. This function provides context for stencil tunnel and create buffers described in Section 3.2. *fetch_x* is used to fetch a data item in the *target value buffer*, while *set_x* is used to set a data item in the *target value buffer*. *isStencilReady_x* is used to check whether the data in a stencil area are all ready.

Besides above functions, users should define the end flag variable in *HiWayLib* name space. Figure 9 shows the usage of HiWayLib for a real application.

Other Soundness Considerations The techniques in Sections 3.1 and 3.3 rely on system-wide shared data structures. The cross-device coherence of the shared data is essential to guarantee the correctness. These shared data structures are defined as unified memory variables with volatile keyword. System-wide atomic functions (e.g., *atomicAdd_system*) are used to guarantee the coherence. Memory fences are used to guarantee the memory order for queue operations and the hand-shaking process in Section 3.1.

4 Evaluation

To evaluate the efficacy of HiWayLib, we use six pipeline applications on two different platforms. Each of the platforms consists of two Intel Xeon E5-2620 8-core CPUs, to which a GPU is connected via PCIe. One platform is equipped with

Context construction
- *createTunnel*<Type>(int& id, TunnelDirection direc, bool isBatched)
  Create tunnel with specified direction and data movement scheme. Get tunnel ID.

Control flow and monitor management
- *isTerminate_h*()
  Check whether the pipeline should terminate on CPU side. *'isTerminate_d'* for GPU side. *'XXX_d'* will be the function on GPU for the following functions in the same form.
- *endTask_h*(int id)
  Should be called when a task on CPU side ends.
- *setupService*()
  Set up monitors for DMA control.
- *shutdownService*()
  Shut down monitors.

Data management
- *put_h*<Type>(Type data, int id)
  Put data into specific tunnel from CPU side.
- *get_h*<Type>(Type *data, int id)
  Get data from specific tunnel on CPU side.

Stencil pipeline
- *configStencilTunnel*<StencilType>(int id, StencilConfig config)
  Setup stencil pipeline context and create input matrix for stencil algorithm.
- *fetch_h*<StencilType>(int id, int x, int y, StencilType* content)
  Fetch data from stencil matrix with tunnel id and index in matrix.
- *set_h*<StencilType>(int id, int x, int y, StencilType content)
  Set data in stencil matrix with tunnel id and index in matrix.
- *isStencilReady_h*(int id, int x, int y)
  Assume point (x, y) is the middle point in the stencil, check whether all values in this stencil area are ready.

Figure 8. HiWayLib APIs.

NVIDIA GTX 1080 GPU and the other NVIDIA Tesla V100 GPU.

The evaluated pipeline applications are shown in Table 1. They present various characteristics. The stage count of these workloads ranges from 2 to 5. The pipeline structures of the workloads include linear, loop, and recursion. (Linear pipeline structure is the simplest, whereas recursion pipelines have recursive executions and loop pipelines have iterations among different stages.) The communication modes between stages include intra-device, CPU-to-GPU, and GPU-to-CPU data transfer. `Gene Alignment` and `Evolutionary Program` are high performance CPU-GPU pipeline programs previously manually developed as part of Hetero-Mark [46]; `Rasterization` and `Reyes` are ported from the work of Piko [37] using a similar optimized pipeline scheme as used in Hetero-Mark to utilize both CPU and GPU; `Face detection` is ported from a real-world application developed by Oh and others [33]; `Stencil` is a representative of stencil programs, in which CPU generates input values and GPU executes the five-point stencil computations. As how to best partition a program into CPU and GPU parts is not the focus of this work, we use simple trial-and-error to select a partition which gives the baseline version the highest performance.

Baseline for Comparison Previous general pipeline programming frameworks are either for GPU only [54] or CPU only [15, 16, 47]. Hetero-Mark [46] is a work trying to efficiently utilize both CPU and GPU for a pipeline computation. Although it is not a general programming framework, its scheme of implementation represents the state of the art in the field for constructing CPU-GPU pipeline programs. The left part of Figure 9 outlines the scheme. The input data set is divided into many chunks before entering the loop in line 12. These chunks are processed in the loop successively. In each iteration, the CPU master thread uses `cudaMemcpy` to copy data of a chunk to GPU, launches the GPU kernel (one stage of the pipeline). When the GPU kernel finishes processing the chunk (ensured by a synchronization barrier on line 15), results are copied to CPU through `cudaMemcpy`. The CPU master thread forks a child CPU thread to do the next stage of processing for that chunk, while the master thread itself moves on to the next iteration to move the next chunk's data to GPU and launch the next GPU kernel to process the next data chunk. CPU and GPU can hence process different chunks concurrently, forming a pipeline. As this scheme use no persistent threads (nor task queues), there is no need for *end detection* of all tasks. Pipeline ends after all function call ends.

Table 1. Various pipeline applications used for evaluation.

Applications	Abbr	Description	Stage Count	Pipeline Structure
Gene Alignment	GA	Bioinformatics	2	Linear
Evolutionary Program	EP	Genetic Algorithm	5	Loop
Reyes	RE	Reyes Rendering	3	Recursion
Rasterization	RA	Image Rasterization	3	Loop
Face Detection	FD	LBP Face Detection	5	Recursion
Stencil	ST	5-Point Stencil	2	Linear

```
1.  producerGPU (input, output) {
2.    // each thread generate 'data' with 'input'
3.    output[thread_index] = data;
4.  }
5.
6.  consumerCPU (input_h) {
7.    for(auto data: input_h) {
8.      process(data); }
9.  }
10.
11. main () {
12.   while (start < maxLength) {
13.     cudaMemcpy(input[start], chunk[start]);
14.     producerGPU(&input[start], output_d);
15.     synchronize();
16.     cudaMemcpy(input_h, output_d);
17.     thread_create(consumerCPU, input_h);
18.     // thread_create is non-blocking
19.     start++;
20.   }
21.   thread_join_all();
22. }
```

a) Baseline implementation

```
1.  producerGPU (input, tunnelid) {
2.    // each thread generate 'data' with 'input'
3.    put_d<>(data, tunnelid);
4.    if(isLastThread) {
5.      put_d<>(HiWayLib::endFlag_d, tunnelid); }
6.  }
7.
8.  consumerCPU(tunnelid){
9.    while(!isTerminate_h()) {
10.     if(get_h<>(&data, tunnelid)) {
11.       process(data);
12.       endTask_h(tunnelid); } }
13. }
14.
15. main () {
16.   // initialize HiWayLib::endFlag_d
17.   createTunnel<>(id, GPU_TO_CPU, true);
18.   setupService();
19.   thread_create(consumerCPU, id);
20.   while (start < maxLength) {
21.     producerGPU(&input[start], id);
22.     start++; }
23.   thread_join();
24. }
```

b) Implementation with HiWayLib API

Figure 9. The pseudo-code of `Gene Alignment` in Hetero-mark (baseline) and in HiWayLib API.

The previous work [46] reports promising performance by this scheme. The first four benchmarks in Table 1 all use that scheme. The other two benchmarks feature stencil computations which do not fit that scheme; the CPU-GPU communications in their baseline versions are implemented with our basic scheme described in Section 3.1. In this scheme, the use of persistent threads and task queues require *end detection*. But because the total numbers of tasks are known

beforehand for these two benchmarks, they just use simple task counting which is sufficient.

4.1 Overall Results

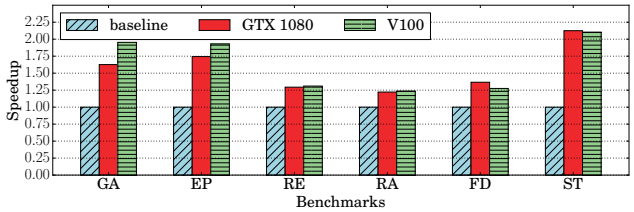


Figure 10. Overall results on different platforms.

Figure 10 reports the overall performance of all the applications we have evaluated on the two platforms.

All timing results are the average of five repeated runs. The execution time is measured on a CPU side, starting from the beginning of the first pipeline stage and ending just after the last pipeline stage finishes (with *cudaDeviceSynchronize*). It includes all the pipeline stage executions, CPU controls, and memory copies covered in that entire code range. All results have been normalized by the performance of the baseline implementation.

In general, the results show that, for all the applications, HiWayLib has better performance than the baseline implementation. On the GTX 1080 platform, HiWayLib has achieved up to 2.1× better performance than the baseline. On average, HiWayLib is 1.56× better than the baseline on GTX 1080, and 1.65× on V100. The speedups on V100 are overall slightly higher because V100 computes faster, leaving communications a more important performance factor. The performance gains of HiWayLib come from the contention relief, efficient termination checking, and the reduced redundant copy in inter-device communication. We explain the details with time breakdowns next.

4.2 Detailed Results

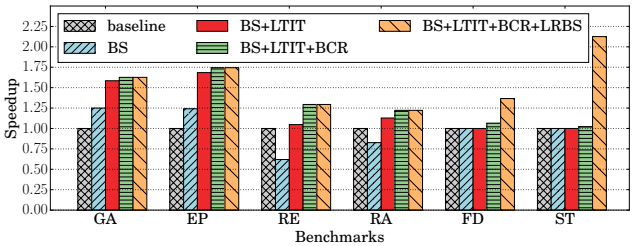


Figure 11. Breakdown results on GTX 1080 platform. Abbr: BS(Basic Scheme), LTIT(Late Triggered Inter-Stage Tracking), BCR(Bi-Layer Contention Relief), LRBS(Lazy Reference-Based Scheme)

Figure 11 shows the time breakdown. We first enable the communication system with the method described in section 3.1, which we call *basic scheme* version. Then, we add in the proposed techniques one by one in this order: *Late Triggered Inter-Stage Tracking* (LTIT), *Bi-Layer Contention Relief* (BCR), and *Lazy Reference-Based Scheme* (LRBS). Figure 11 reports the measures on GTX 1080 platform. The increased speedups show the contribution of BS, LTIT, BCR, and LRBS. Due to the different natures of the programs, these optimizations have different impact on them. We elaborate them next.

Gene Alignment and Evolutionary Program Both come from Hetero-Mark. `Gene Alignment` is an important bioinformatics algorithm which has two stages. The first stage is performed on GPU and the second on CPU. The input data set has a target length of 65536 and query length of 1024.

A problem of the baseline implementation comes from the synchronization barriers and data copies between the devices. HiWayLib enables fine-grained pipeline parallelism through its efficient asynchronous inter-device communications and task queues. There is no barrier between the producer and the consumer, and the producer will not be blocked when transferring data to the consumer. That is the main reason for the significant speedups BS has over the baseline, 64.4ms versus 80.5ms. It is worth noting that BS (for its use of persistent threads and task queues) actually adds some extra delays at *end detection* compared to the baseline version (which needs no *end detection* for the simplicity of the scheme). The evidence is shown by the extra speedups when the better *end detection* method LTIT is applied to BS, reducing the time from 64.4ms to 50.8ms. Such an effect is seen on the `Evolution Program` and also `Reyes` and `Rasterization`.

Queue contention reduction has little effect on this benchmark as there are only two stages and the queue contention between them is not a major issue.

`Evolutionary Program` implements genetic algorithm, which has five consecutive stages. The second and fifth stages run on GPU, while the others on CPU. Thus, the implementation contains both CPU to GPU and GPU to CPU data transfer. The population size is 2048. The baseline pipeline implementation taken in Hetero-Mark is effective in exploiting the pipeline parallelism among different populations, but not the pipeline parallelism within the same population. With the improved CPU-GPU communication scheme, the HiWayLib implementation exploits both. Results on GTX 1080 show that the baseline takes 44.3ms to execute the pipeline, while the HiWayLib version takes 25.4ms. As shown in Figure 11, all techniques in HiWayLib contribute to the speedup except LRBS because LRBS is applied to only stencil programs.

Reyes and Rasterization Both are for image rendering, but of different algorithms. `Reyes` [7] contains four stages. Only the first stage runs on CPU, which prepares data for the subsequent 3 stages. The rendering result of `Reyes` is a teapot in a resolution of 1280×720 pixels. `Rasterization` contains three stages. Only the first stage runs on CPU to prepare data for subsequent stages. It generates 100 pictures of cube in 1024×768 resolutions.

The performance observations and analysis on these two are similar. We take `Reyes` as the example for discussion. The baseline, which is optimized with the approach in Hetero-Mark, realizes the pipeline between CPU and GPU stages for different images. But it fails to exploit the potential of pipeline parallelism among the primitives of the same image With HiWayLib APIs, both image level and primitive level pipeline parallelism are realized with only about 10 extra lines of code. HiWayLib does not copy all the primitives of an image to GPU at once, but enqueues each primitive individually. A loop in GPU kernels continuously dequeues and processes primitives. Results show that the baseline takes 4.4ms, while HiWayLib version takes 3.4ms.

As shown in Figure 11, the batched copy scheme shows worse performance than the baseline. It is because that the batched copy overhead is large compared to the short execution time of `Reyes` itself. *Late Triggered Inter-Stage Tracking* and *Bi-Layer Contention Relief* boost the performance by reducing the overhead significantly.

For both `Reyes` and `Rasterization`, the execution times of the CPU functions are larger than the execution times of the GPU, making the GPU time nearly overlapped with the CPU time. Thus the speedup of GPU kernels on V100 does not help increase the overall speedup much compared to the case on GTX 1080.

Face Detection and Stencil Both of them have stencil computation patterns. `Face Detection` is a real-world application [33] that finds location and size of faces in a given image. It consists of 5 stages; the fourth stage has stencil dependency and runs on CPU, while others run on GPU. The input is an image in 1280×720 resolution.

As shown in Figure 10, HiWayLib shows 1.37× better performance than the baseline. The baseline does not employ *Lazy Reference-Based Scheme*. It invokes a task as soon as its input pixels are available. A task consumes 3×3 stencil tiles, which may overlap with the ones belonging to other tasks. Since the baseline copies the overlapped pixels redundantly, it shows a poor performance. With *Lazy Reference-Based Scheme*, the redundant data transfer is removed and each data item will be transferred only once.

`Stencil` is a 5-point stencil workload with two pipeline stages. The CPU stage generates the initial value of the matrix, and GPU executes the stencil process. As shown in Figure 10, the HiWayLib version shows 2.13× better performance than the baseline. Similar to `Face Detection`, the speedup mainly comes from the removal of redundant data copy between CPU and GPU stages. In the stencil program, each task reads its four adjacent values. In the baseline implementation, each data in the matrix will be copied five times. In contrast, each data is copied only once in HiWayLib.

4.3 Communication Overhead

We discuss the communication overhead introduced by HiWayLib and give some insights on how HiWayLib boots pipeline programs performance. We draw on `Evolutionary Program` as the example for our discussion, which has the most complicated communication pattern in the six evaluated applications, including both CPU-to-GPU and GPU-to-CPU data transfer.

We measure the execution time of the benchmark after removing all computation logic in both CPU and GPU tasks. The result shows the time taken by the communications and some control flows on CPU. The time is 10.5ms for the baseline version and 14.1ms for the HiWayLib version. Considering that they have the same control flow on CPU, HiWayLib introduces about 3.6ms extra communication overhead. The time is a small portion of the overall running time (25.4ms) that has been reported in section 4.2. The influence of the overhead is largely hidden as the pipeline communications overlap with task processing. In the baseline implementation, the barrier between stages causes some ready tasks to accumulate on the producer side. HiWayLib runtime transfers ready tasks to the consumer side in a more timely manner, helping reduce the latency. The performance boost that comes from the better overlapped computation with communication and the more performant computation pipelining with HiWayLib outweighs the effects of the communication overhead, and leads to the overall significant speedups.

4.4 Programming Effort

We show the programming effort of optimizing a pipeline program using HiWayLib with `Gene Alignment` application. Figure 9 shows the pseudo-code of the baseline implementation and HiWayLib implementation for this program.

Figure 9(a) shows the baseline implementation in Heteromark. Figure 9(b) shows the implementation with HiWayLib API. There are three major modifications compared with the baseline version. First, the producer does not place the output data into an array, but enqueues them to the queue using a HiWayLib API. Second, there is no barrier or explicit CUDA memory copy. The consumer obtains the input with a dequeue API provided in HiWayLib. Third, the consumer is not called for each output section of the producer, but called only once. There is a loop in the consumer to check whether it has terminated.

This sample shows that about 10 lines of modification is made to the baseline implementation to make use of HiWayLib. In fact, for all applications we have evaluated, only tens of lines are required to optimize the original pipeline implementation with HiWayLib.

4.5 Unified Memory

With default configuration, HiWayLib adopts the batched data copy scheme (Section 3.1). HiWayLib also supports cross-device memory access through Unified Memory. We evaluate it with two configurations. Results show that the benchmarks we evaluate achieve an average speedup of 1.31× with batched data copy scheme than with Unified

Memory scheme. The reason is that the frequent and concurrent queue accesses between two different devices lead to severe page fault overhead with Unified Memory.

5 Related Work

There are some previous promising studies optimizing the communication and data movements between CPU and GPU for general programs [20, 21, 34]. They use compiler to identify some unnecessary data copy operations in a program and transform the code to avoid them. For example, Pai and others [34] have proposed a compiler-based method to avoid transferring data to GPU if GPU's copy of the data is not stale, and avoid transferring data from GPU to CPU if the data are not needed by CPU or are private GPU-only data. Although these techniques could in principle benefit pipeline programs, the intensive use of concurrent data structures (e.g., queues) and framework APIs in pipeline programs could make them challenging for compiler tools. HiWayLib is complementary to the previous studies in that it focuses on a different set of communication optimizations that are special to pipeline programs.

Kumar et al. [25] provides a solution (a middle layer) to programming difficulties of data management posed by the many platform-specific intricacies on heterogeneous platforms. Ramashekar et al. [40], Wang et al. [51] and Pichai et al. [38] study memory management and data transfer on CPU-GPU system. Kim et al. [23] propose a memory network architecture to simplify and speedup cross-device communication. NCCL [30] provides multi-GPU and multi-node collective communication primitives for NVIDIA GPUs. GPUDirect [42] enables remote GPU to transfer data through InfiniBand without involving CPU. All these studies focus on the CPU-GPU communication for general applications, but not for pipeline communications where efficient pipeline concurrency control is of keen interest.

Some studies give some insights for the pipeline communication between CPU and GPU. Yeh et al. [52] propose a technique to manage cross device task transfer with tables on both devices. Van et al. [50] study the performance of explicit memory copy and direct access with mapped memory with a performance model. Hestness et al. [19] evaluate the fine-grained CPU-GPU communication with asynchronous memory copy and implicit data copy with Unified Memory for a variety of workloads. Some works [29, 36] study task parallelism and use basic data copy approach for cross-device communication. Sun et al. [46] study the main programming patterns for CPU-GPU applications and use task queues for pipeline workloads on integrated CPU-GPU systems.

Some prior work focuses on task scheduling of pipeline programs on GPU only. Besides Whippletree [45] and VersaPipe [54] that the paper has discussed earlier, Tzeng et al. [48] study task scheduling with persistent-thread [3] technique and queue structure. None of these systems focus on inter-device pipeline communication.

There are some prior research on task scheduling of the pipeline programs on heterogeneous systems. They all use the explicit memory copy approach for communication. Bodin et al. [6] schedule pipeline tasks on embedded heterogeneous systems with synchronous data flow models. Augonnet et al. [4] propose five main task scheduling policies on heterogeneous systems. Gautier et al. [13, 14] design a runtime system for data-flow task programming on multi-CPU and multi-GPU architectures, which supports a data-flow task model and a locality-aware work stealing scheduler. Schor et al. [41] develop a framework for task scheduling of pipeline applications on heterogeneous platforms, which uses package transfer scheme for cross-device data transfer. Many other studies [11, 24, 27, 43, 44, 53] work on task partition and task placement.

Some researchers have studied the contention problem for queue structures. Fatourou et al. [12] and Hendler et al. [18] propose *combining* technique on CPU, where fine-grained synchronization operations are combined by a thread into a coarser-grained operation. Pai et al. [35] study the contention relief technique on GPU, in which threads are aggregated at the level of either warp or block, and additional synchronization is introduced for block level aggregation.

6 Conclusion

In this paper, we present an in-depth study on the communication problem for pipeline programs on heterogeneous systems. To address several key issues in this important topic, such as slow and error-prone detection of the end of pipeline processing, intensive queue contentions on GPU, cumbersome inter-device data movements, we propose a series of novel techniques, including *Lazy Reference-Based Scheme*, *Late Triggered Inter-Stage Tracking*, and *Bi-Layer Contention Relief*. Finally, we integrate the above techniques together into a unified library HiWayLib, which significantly improves the efficiency of pipeline communications in six CPU-GPU heterogeneous applications.

Acknowledgments

The authors would like to thank all the anonymous reviewers whose feedback are helpful for improving the final version of the paper. This work is partially supported by the National Key R&D Program of China (Grant No. 2017YFB1003103), National Natural Science Foundation of China (Grant No. 61722208). This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. CCF-1455404, CCF-1525609, CNS-1717425, CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF. This work is partially supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (No. 2018R1D1A1B07050463). (The corresponding author at Tsinghua University is Jidong Zhai.)

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[2] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. 1984. Pyramid methods in image processing. *RCA engineer* 29, 6 (1984), 33–41.

[3] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the conference on high performance graphics 2009*. ACM, 145–149.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[5] Christian Bienia and Kai Li. 2010. Characteristics of workloads using the pipeline programming model. In *International Symposium on Computer Architecture*. Springer, 161–171.

[6] Bruno Bodin, Luigi Nardi, Paul HJ Kelly, and Michael FP OâĂŹBoyle. 2016. Diplomat: Mapping of Multi-kernel Applications Using a Static Dataflow Abstraction. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*. IEEE, 241–250.

[7] Robert L Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes image rendering architecture. In *ACM SIGGRAPH Computer Graphics*, Vol. 21. ACM, 95–102.

[8] Sivarama P Dandamudi. 1997. Reducing run queue contention in shared memory multiprocessors. *Computer* 30, 3 (1997), 82–89.

[9] Sivarama P. Dandamudi and Philip S. P. Cheng. 1995. A hierarchical task queue organization for shared-memory multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 6, 1 (1995), 1–16.

[10] John Danskin and Denis Foley. 2016. Pascal GPU with NVLink. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*. IEEE, 1–24.

[11] Kapil Dev, Xin Zhan, and Sherief Reda. 2016. Power-aware characterization and mapping of workloads on cpu-gpu processors. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–2.

[12] Panagiota Fatourou and Nikolaos D Kallimanis. 2012. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 257–266.

[13] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. 2007. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 15–23.

[14] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. 2013. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 1299–1308.

[15] John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 43–52.

[16] Michael I Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 151–162.

[17] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM, 195–206.

[18] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 355–364.

[19] Joel Hestness, Stephen W Keckler, and David A Wood. 2015. GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors. In *2015 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 87–97.

[20] Thomas B Jablin, James A Jablin, Prakash Prabhu, Feng Liu, and David I August. 2012. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 165–174.

[21] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. 2011. Automatic CPU-GPU communication management and optimization. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 142–151.

[22] Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. 2016. Automatically exploiting implicit Pipeline Parallelism from multiple dependent kernels for GPUs. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 339–350.

[23] Gwangsun Kim, Minseok Lee, Jiyun Jeong, and John Kim. 2014. Multi-GPU system design with memory networks. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 484–495.

[24] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. 2013. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 149–160.

[25] Tushar Kumar, Aravind Natarajan, Wenjia Ruan, Mario Badr, Dario S Gracia, and Calin Cascaval. 2017. Abstract Representation of Shared Data for Heterogeneous Computing. In *The 30th International Workshop on Languages and Compilers for Parallel Computing*. Springer 2017.

[26] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in cuda. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 1–6.

[27] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John. 2015. Data partitioning strategies for graph workloads on heterogeneous clusters. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807632

[28] Hung-Fu Li, Tyng-Yeu Liang, and Yu-Jie Lin. 2016. An OpenMP Programming Toolkit for Hybrid CPU/GPU Clusters Based on Software Unified Memory. *Journal of Information Science and Engineering* 32, 3 (2016), 517–539.

[29] Joao Vicente Ferreira Lima, Thierry Gautier, Nicolas Maillard, and Vincent Danjean. 2012. Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs. In *24rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 75–82.

[30] NVIDIA Corporation. [n. d.]. NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl.

[31] NVIDIA Corporation. [n. d.]. NVIDIA Warp Shuffle. https://devblogs.nvidia.com/using-cuda-warp-level-primitives.

[32] NVIDIA Corporation. [n. d.]. Unified Memory Programming. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[33] Chanyoung Oh, Saehanseul Yi, and Youngmin Yi. 2015. Real-time face detection in Full HD images exploiting both embedded CPU and GPU. In *Multimedia and Expo (ICME), 2015 IEEE International Conference on*. IEEE, 1–6.

[34] Sreepathi Pai, R Govindarajan, and Matthew J Thazhuthaveetil. 2012. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 33–42.

[35] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 1–19.

[36] Sankaralingam Panneerselvam and Michael Swift. 2016. Rinnegan: Efficient resource use in heterogeneous architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 373–386.

[37] Anjul Patney, Stanley Tzeng, Kerry A Seitz Jr, and John D Owens. 2015. Piko: a framework for authoring programmable graphics pipelines. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 147.

[38] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. *ACM SIGPLAN Notices* 49, 4 (2014), 743–758.

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[40] Thejas Ramashekar and Uday Bondhugula. 2013. Automatic data allocation and buffer management for multi-GPU machines. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 60.

[41] Lars Schor, Andreas Tretter, Tobias Scherer, and Lothar Thiele. 2013. Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*. IEEE, 41–50.

[42] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R Trott, Greg Scantlen, and Paul S Crozier. 2011. The development of Mellanox/NVIDIA GPUDirect over InfiniBandâĂŤa new model for GPU to GPU communications. *Computer Science-Research and Development* 26, 3-4 (2011), 267–273.

[43] Jie Shen, Ana Lucia Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. 2014. Improving performance by matching imbalanced workloads with heterogeneous platforms. In *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 241–250.

[44] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John. 2016. Proxy-Guided Load Balancing of Graph Processing Workloads on Heterogeneous Clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*. 77–86. https://doi.org/10.1109/ICPP.2016.16

[45] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: task-based scheduling of dynamic workloads on the GPU. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 228.

[46] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–10.

[47] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. 2007. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 356–369.

[48] Stanley Tzeng, Brandon Lloyd, and John D Owens. 2012. A GPU task-parallel model with dependency resolution. *Computer* 8 (2012), 34–41.

[49] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. 2009. Software pipelined execution of stream programs on GPUs. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 200–209.

[50] Ben Van Werkhoven, Jason Maassen, Frank J Seinstra, and Henri E Bal. 2014. Performance models for CPU-GPU data transfers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 11–20.

[51] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. 2014. GDM: device memory management for gpgpu computing. *ACM SIGMETRICS Performance Evaluation Review* 42, 1 (2014), 533–545.

[52] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G Rogers. 2017. Pagoda: Fine-grained GPU resource virtualization for narrow tasks. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 221–234.

[53] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. 2017. FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 27–38.

[54] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. Versapipe: a versatile programming framework for pipelined computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 587–599.