Journal of Parallel and Distributed Computing 133 (2019) 51–62

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Efficient low-latency packet processing using On-GPU Thread-Data Remapping

Huanxin Lin *, Cho-Li Wang

Department of Computer Science, The University of Hong Kong, Hong Kong

HIGHLIGHTS

- On-GPU Thread-Data Remapping accelerates routing without relying on CPU.
- Without CPU processing, packets are transferred with lower latency.
- Frequent synchronization in loops impairs GPU memory latency hiding.

ARTICLE INFO

Article history:
Received 31 July 2018
Received in revised form 23 May 2019
Accepted 18 June 2019
Available online xxxx

Keywords:
Packet processing
Software router
GPU control flow divergence
SIMD

ABSTRACT

Graphics processing units are widely-used for packet processing acceleration in both physical and virtual networks. However, real-life packets come in highly-divergent sizes, causing severe GPU control flow divergence. Previous solutions rely on CPU preprocessing to reduce divergence, but it forbids the more efficient NIC–GPU packet streaming as packet batches have to stop completely at host machine. To fully utilize both GPU and PCIe resources, we propose Blink as a GPU modular software router. Instead of CPU pre-processing, the Blink router uses On-GPU Thread-Data Remapping to reduce divergence, and our novel Cross-Iteration Thread Event Signaling mechanism filters unnecessary inter-thread synchronization, doubling the performance gain achieved by traditional solution. Serving as a TCP/IP router with Deep Packet Inspection (DPI) firewall, Blink can sustain processing throughput of 31.5 GBit/s over a PCIe bandwidth of 32 GBit/s. Given a certain bandwidth, Blink reduces processing latency at least by half compared with other works.

1. Introduction

As the Internet of things comes into being, the demand for computer networks with higher speed and complexity is getting stronger. On the other hand, the demand for rich router functionality also grows from basic routing and forwarding to traffic reduction [3], security [4], *etc*. Apart from physical networks, virtual networks also require efficient packet processing that gives throughputs of at least 10 Gbit/s without putting much computation burden on the hosting server [25].

Given the massive parallel nature of packet processing, the need for advanced networks drives rapid development of GPU-accelerated routing [17,22,25,27]. The GPU platform is well-known for energy efficiency, and has been successful in accommodating a wide range of packet processing tasks, which includes both memory-intensive tasks like IP-lookup [10] and compute-intensive tasks like encryption [11]. GPU processing has been incorporated into commercial routers as feature [16,24].

However, at least two issues hinder GPU from further accelerating packet processing. First, real-life packets come in divergent sizes, which decelerates size-sensitive tasks on GPU [11,26]. In a network traffic trace captured by the *equinix-sanjose* monitor under CAIDA [5], maximum packet size is 1518 bytes, but the average is only 606 bytes. Each packet is inspected by a GPU thread with a loop on packet size, and all threads may have to compute max-size looping workload due to SIMD execution. Such significant control flow divergence accounts for the 60% computation wastage [8,27]. Second, both network interface card (NIC) and GPU are connected to host CPU via the PCIe bus. In existing solutions, network packets received from the NIC are first copied by CPU to GPU's DMA buffer, and then transferred to GPU for processing. Instead of being transferred in a pipelining fashion, packet batches come to a full stop at the main memory of host machine. Data flow between the two PCIe-connected devices is thus explicitly broken into NIC-host transfer and host-GPU transfer. With a two-phase transfer, per-packet processing latency is increased and the CPU involvement may become the bottleneck [14,27].

Previous solutions are stuck in a dilemma created by the two issues. On one hand, control flow divergence forces certain

* Corresponding author.
E-mail addresses: hxlin@cs.hku.hk (H. Lin), clwang@cs.hku.hk (C.-L. Wang).

https://doi.org/10.1016/j.jpdc.2019.06.009
0743-7315/

processing operations to rely on CPU computation [12,17,22]. In a more recent work by Vasiliadis et al. [27] where more GPU processing is employed, divergence reduction still requires CPU preprocessing: packets are first grouped into batches based on a few size ranges, and then further radix-sorted on GPU. Their method is a compromise to the fact that it may take forever to gather a same-size-packet batch which is large enough to fully occupy GPU computation resources. The number of radix bits has to be configured optimally to achieve a balance between sorting overhead and residual divergence. On the other hand, it is CPU computation that limits throughput and increases processing latency. Host-GPU transfer is postponed until CPU finishes its work, otherwise direct device-to-device transfer can be implemented in various ways [14,20]. In fact, Vasiliadis et al. proposed to let NIC and GPU share the same DMA buffer so that packets are streamed on the fly between devices, but they have to deactivate this feature when CPU preprocessing is necessary. To the best of our knowledge, other existing GPU-accelerated routing solutions also involve CPU as the transfer middleman. As a result, performance of current solutions is limited by either control flow divergence or two-phase packet transfer. There is a need for GPU-runtime divergence solution without CPU preprocessing.

Proposed recently, On-GPU Thread-Data Remapping (TDR) [15] is a promising candidate for resolving the dilemma, but some design challenges still remain. On-GPU TDR allows threads to exchange data sets and thus the work states during kernel execution, so it can be utilized to regroup threads that are still in the loop into fewer running wavefronts (OpenCL term) at GPU-runtime. However, since threads do not exit the loop in every iteration, each round of TDR may not always provide performance gain. What is worse, it introduces inter-wavefront synchronization, and frequent synchronization weakens GPU-native latency hiding as runnable wavefronts are blocked until peers catch up. Besides TDR orchestration, non-coalesced memory access is another major issue that has not been addressed. Extra memory transactions are needed as threads access cache lines scattered in different packets, which are unfortunately stored in the slow global memory. The access slowdown becomes more severe in the case of full-packet inspection.

Considering all the above issues, we propose Blink as a software router that provides efficient low-latency packet processing. Instead of reducing divergence with CPU preprocessing, we apply On-GPU TDR and design a mechanism to trigger TDR only when it is beneficial. Packets are batched and processed on a first-come-first-serve basis, and a prefetching scheme that makes use of shared memory is incorporated to reduce memory overhead.

The contributions of this work are:

- We propose Blink as a GPU modular software router, that fully adopts pure-GPU packet processing and implements direct NIC–GPU packet streaming. Established processing pipeline sustains throughput close to total PCIe bandwidth and maintains stable low latency.
- To reduce loop-carried divergence, we propose *Cross-Iteration Thread-Event Signaling* (*CITES*) as a mechanism that optimizes On-GPU TDR in loops. *CITES* orchestrates On-GPU TDR with atomic flags, and protects GPU native latency hiding from the harm of frequent unnecessary synchronization.
- To reduce memory overhead, we propose a memory prefetching scheme for the packet access pattern. Without decreasing GPU occupancy, packets are prefetched from global memory to shared memory using coalesced accesses. To avoid explicit synchronization, prefetching is hidden in parallel with normal processing.

Evaluation is conducted on an NVIDIA GTX 980 GPU. For size-sensitive operations on the CAIDA traffic trace, On-GPU TDR achieves a speedup of 1.2, while *CITES* further boosts it to 1.9. The final achieved speedup is 2.1 with memory prefetching scheme activated, exceeding the highest speedup of 1.4 in previous work [27]. For TCP/IP router with DPI firewall, which is also evaluated in two other works [22,27], module capsuling effectively squeezes away 8% of processing time. When the GPU is connected to NIC via 32 Gbit/s PCIe bandwidth, a throughput of 31.5 Gbit/s is reached for the above-mentioned router, reducing latency at least by half compared with the other works.

The remainder of this paper is organized as follows. Section 2 provides background information and features motivation experiments. Section 3 shows design and implementation corresponding to each objective. Section 4 evaluates Blink and compares with two other works. Finally, Section 5 is related work, and last comes conclusion.

2. Background and motivation

This section reviews the basics of GPU computing and packet processing. Our motivation experiments show that control flow divergence is the key obstacle for GPU-accelerated routing.

2.1. GPU control flow divergence

GPU platform features the Single Instruction Multiple Data execution model. Hardware compute cores are grouped into Streaming Multiprocessors (SM, NVIDIA term). Instructions are issued to each SM and then executed by every member core, so that high parallelism is achieved with low energy consumption.

OpenCL [21] is the GPU programming model used in this work, and here is a brief introduction. Codes to be executed by GPU are called *kernels*, which are written in a syntax similar to the C language. Kernels are launched to GPU and executed by threads, or *work-items*, in the specified problem space. Work-items are grouped into *wavefronts*, and wavefront members are scheduled onto the SM together to execute the kernel codes in lockstep.

GPU has the following three-tier memory hierarchy.

Private memory. Each work-item possesses a piece of the fastest exclusive private memory.

Shared memory. Each workgroup has shared memory that can be accessed and synchronized by its members.

Global memory. The largest and slowest piece of memory is global memory, which can be accessed by all work-items.

In GPGPU development, control flow divergence is a well-known cause for performance loss. It occurs when threads in a wavefront go on different execution paths at conditional statements. Since wavefront members must execute the same instruction at any time, every work-item executes all the paths but only keeps the results corresponding to its path. Such computation wastage lowers parallelism and efficiency, until normal execution resumes at the convergence point right after the conditional statements. In fact, control flow divergence can be divided into two categories and we stick to the definitions below.

- *Branch divergence* happens at branch statements. Each wavefront needs to execute all branch paths taken by its member threads. An if-else branch can halve computation efficiency, and nested branches can cause performance loss exponentially.
- *Loop divergence* happens in loops. Every loop has a looping condition that determines whether a thread enters the next iteration. Threads that exit the loop earlier cannot proceed with execution until all peers in the same wavefront exit and reach the convergence point right after the loop. Thus, each wavefront runs the loop for the largest number of iterations needed by its members, resulting in computation wastage.

Table 1
CFE of modules while processing the CAIDA trace.

Module	Divergence Type	CFE
Classifier	Branch	95.7%
IPLookup	Branch	95.9%
IDSMatcher	Loop	39.5%

Control flow efficiency (CFE) is a metric for severeness of control flow divergence, and we utilize it to quantify the divergence in packet processing. By definition, CFE is the percentage of executed thread instructions that are not masked off due to either branch divergence or loop divergence. It equals 100% when execution is non-divergent. In the next subsection, we will introduce common packet processing practices and measure their CFE.

2.2. Packet processing on GPU

Network packets come from various protocols and layers, but processing operations share similar elements. Common router setups are comprised of some or all of the following representative processing modules.

- *Classifier* provides the most basic routing utilities: packet decoding, classification and fast operations such as TTL decrement. This module alone serves as an SDN-like switch.
- *IPLookup* implements the forwarding rules. With the addition of this module, a basic TCP/IP router is built.
- *IDSMatcher* checks whether the packet payload matches any intrusion–detection signature. This module adds a DPI firewall, which enhances the router up to the practical standard [22].

All these modules are implemented and demonstrated in both previous GPU modular routers, Snap [22] and GASPP [27]. Packets are gathered into batches to undergo processing by corresponding GPU threads. Both routers report high efficiency for the first two modules, but disagree on the last one.

The different computation efficiency for IDSMatcher is caused by the different evaluation settings. As a matter of fact, both routers implement the Aho–Corasick algorithm [2] that has a loop on packet size, but Snap uses fixed-size packets for evaluation. On the other hand, GASPP uses the real-life CAIDA trace for evaluation and gets challenged by severe loop divergence in IDSMatcher.

As part of the motivation experiment, we implement all three modules and test them with the CAIDA trace. We measure the CFE on NVIDIA GTX 980, as shown in Table 1. The low CFE of 39.5% reveals severe loop divergence in IDSMatcher, indicating that execution time is at least doubled. Given that IDSMatcher is one order of magnitude more time-consuming than the other two modules, loop divergence is a significant bottleneck for GPU packet processing.

Optimizations are proposed in GASPP to reduce loop divergence, but the peak achieved speedup is only 1.4 for the same trace. Packets are first grouped into different batches by CPU based on a few size ranges. After packets are transferred onto GPU, radix sort is used to generate a thread-packet redirection array, which incurs both computation and I/O overheads. It is difficult to predict which number of radix bits achieves the best balance between sorting overhead and residual divergence. A trade-off is thus formed between performance gain and low overhead, putting a limit on the speedup.

We aim to design a lightweight loop divergence solution without relying on CPU. Besides IDSMatcher, encryption is also a family of size-sensitive tasks, which can be even more compute-intensive. However, there is still not a standard practice regarding connectionless integrity and data origin authentication on GPU [27]. In this work, we thus focus on IDSMatcher and the loop divergence issue that all size-sensitive tasks are subject to. Our techniques can be applied on the extended modules in the future.

2.3. Apply on-GPU TDR on loop divergence

Aiming to bypass CPU as transfer middleman, we turn to the new technique of On-GPU Thread-Data Remapping [15] for solution. It has proved effective in branch divergence reduction, where threads in the same wavefront are remapped to work states corresponding to the same execution path.

In the case of loop divergence, looping condition can be viewed as a two-path branch that is computed in every iteration. One path is to exit the loop and stay idle until all peers in the wavefront finish. The other is to stay in the loop, and the path length is at least one iteration.

As shown in Fig. 1, our TDR strategy is to gather exiting threads. Each time a thread is ready to exit the loop, its work state is remapped to the remaining thread with the largest ID, so that the latter thread exits the loop instead. Consequently, wavefronts of idle threads are filled up as soon as possible, and they can proceed with post-loop computation without being dragged due to loop divergence.

Since threads may exit the loop in any iteration, we first make a naive attempt to perform On-GPU TDR in every iteration, applying the Head-or-Tail algorithm [15] in the IDSMatcher loop. As a result, per-iteration TDR more than doubles CFE to 96.1%, almost totally eliminating computation wastage due to loop divergence. However, the execution time is only reduced by 20%, suggesting that per-iteration TDR does not only affect instruction computation.

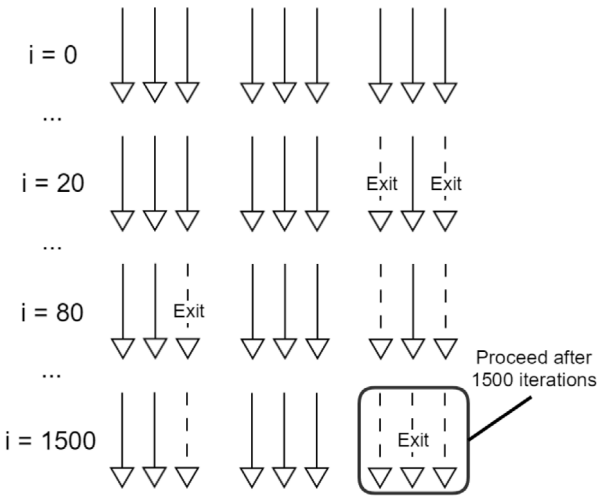
2.4. Problem of per-iteration TDR: GPU scheduling

After investigation, we believe the huge gap between CFE and throughput improvement is caused by the fact that the synchronization required by On-GPU TDR impairs flexibility of GPU wavefront scheduling.

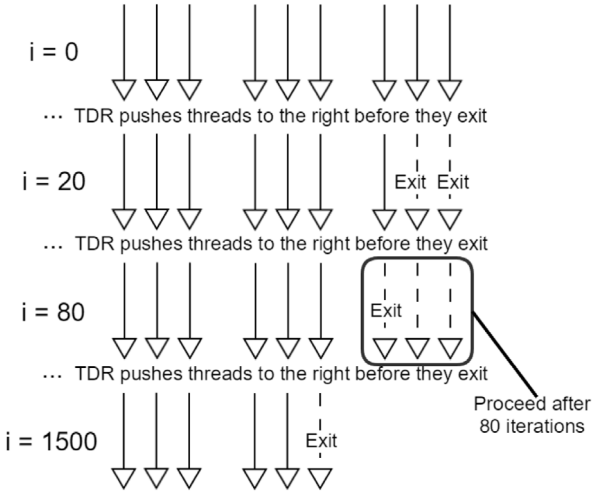
GPU latency hiding relies on instant context switching between wavefronts, which is important because memory accesses can cost hundreds of cycles [9]. Usually tens of wavefronts are scheduled onto the same SM, and a fixed number of wavefronts occupy the compute cores at a time. A wavefront keeps executing until it encounters a memory miss. With negligible delay, one wavefront in the ready status is scheduled to execute. Therefore, memory accesses are hidden by overlapping with execution of other wavefronts. Fewer accesses are needed in total because newly brought-in memory segments may contain data requested by multiple wavefronts.

Given such a scheduling policy, wavefronts are naturally executing in different iterations, but per-iteration TDR breaks native scheduling and thus weakens latency hiding. On-GPU TDR imposes synchronization on threads in order for them to exchange data correctly. In GPU architecture, wavefront synchronization is implemented as a barrier function. Wavefronts are blocked at the function until all peers reach the barrier. In per-iteration TDR, one barrier function is put into each iteration, which may block wavefronts even in the ready status. As a result, fewer wavefronts can be scheduled to execute when memory miss occurs.

Fig. 2 shows a simplified example to illustrate the way that per-iteration synchronization weakens latency hiding. Suppose a memory read by either W1 or W2 brings in data that support two iterations of execution by both wavefronts. In normal execution,

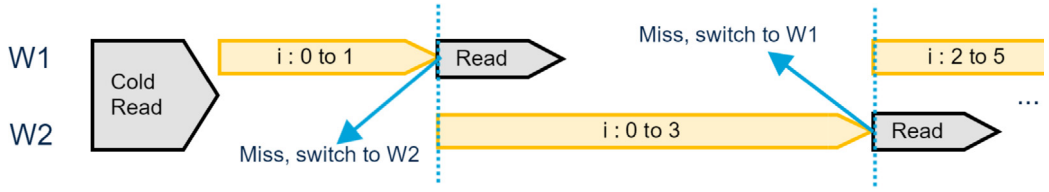


(a) Without TDR, all wavefronts need to stay in loop for 1500 iterations.



(b) TDR frees up a wavefront ASAP (80 iterations).

Fig. 1. An example of three wavefronts executing a thread-divergent loop without and with On-GPU TDR.



(a) In normal loop execution, memory reads are hidden well.

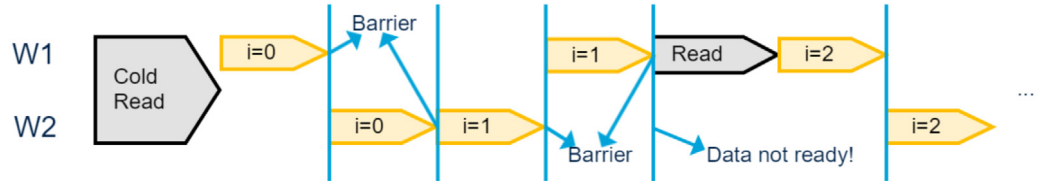


Fig. 2. Effect of per-iteration synchronization on the scheduling of two wavefronts (W1 and W2). A read by either wavefront brings in data for two iterations.

when one wavefront issues memory read, the other wavefront is scheduled instantly and benefits from the hidden read as well. For per-iteration synchronization, however, wavefronts are stopped by barrier when they could have continued further to trigger the next read.

2.5. Design objectives

To fully utilize the power of GPU for packet processing, our solution is designed to address the following issues.

Loop divergence. Per-iteration synchronization or TDR is useful for loop-carried regular branch divergence, but it is not suitable for solving loop divergence. Threads do not exit the loop in every iteration, so some synchronization brings nothing but performance loss. It can be remedied by only conducting TDR when at least one thread is about to exit. However, the required inter-thread communication should also be achieved without synchronization.

Non-coalesced global memory access. Each GPU wavefront needs to access cache lines scattered across different packets. Previous solutions use fixed-size frames to contain packets, which further increases memory transactions due to unused bytes in the frames. Apart from preserving latency hiding while applying On-GPU TDR, it is also critical to prevent such slow accesses from becoming the bottleneck.

Module interconnectivity. On-GPU TDR provides in-kernel loop divergence reduction, which does not require explicit storage of global information like a thread-packet redirection array. Therefore, intermediate result I/O between modules and kernel relaunch can be further reduced. On the other hand, threads terminated in an earlier module leave holes in wavefronts, which becomes a new source of divergence to be taken care of.

NIC–GPU packet streaming. To achieve lowest processing latency, packets should be streamed between NIC and GPU on the fly. Statistics on packet sizes can be collected during packet batching to facilitate the processing on GPU. As data are transferred in 4-KB pages, memory overheads should be gathered in a page with access-efficient layout.

3. Design and implementation

This section presents details of design and implementation corresponding to each objective.

3.1. System overview

Fig. 3 shows the system overview. In this work, the packet data flow is orchestrated by a modified netmap module [18], and the processing is all conducted on GPU.

Incoming packets are gathered by netmap on a first-come-first-serve basis and streamed to GPU as a batch, minimizing the processing latency. During the batching process, netmap also collects information to be put into the batch header, which is used later for loop divergence reduction and memory prefetching. The header contents will be explained in Section 3.5.

Required memory is also minimized as packets are stored consecutively, unlike the fixed-frame design in previous solutions [22,27]. This design is critical for memory prefetching as well as efficient transfer, and details will be given also in Section 3.5.

With a configurable number of packets, each transferred batch triggers one-stop GPU processing. Processed batches are transferred back to netmap, and NIC takes care of final transmission of the packets. Detailed approaches to each design objective are presented in following subsections.

3.2. Cross-iteration thread-event signaling (CITES)

We propose CITES as a mechanism that signals all other threads in a workgroup when a specified event happens to one of them. For inter-thread communication, it uses atomically-managed flags instead of synchronization, and thus preserves GPU latency hiding.

When CITES is applied in the loop divergence scenario, wavefronts are allowed to execute at own paces as usual, with cross-iteration TDR and synchronization happening upon the exit-signal from any thread. As a matter of fact, cross-iteration TDR is not only feasible but also correct. If there is an iterator variable, it can be treated as a normal private memory variable, and exchanged as part of the work states. It is true that the iterator may become uneven within a wavefront, but it does not give rise to divergence because the threads are still sharing the same program counter value and executing the same instructions.

As shown in Listing 1, the codes of CITES and TDR are appended to the end of a given loop body. CITES utilizes the keyword *continue* to skip unnecessary computation on two occasions.

The first is when the number of alive threads has reduced to a threshold (Line 7), which is derived from the batch header. In general cases, threshold should be the larger value between the number of max-size packets and the wavefront size (32 for NVIDIA). On one hand, when the remaining packets are all of the same size, loop divergence becomes negligible. Meanwhile, no matter how threads are shuffled within the last wavefront, they have to wait for the thread with the most iterations.

Listing 1: Example Codes of CITES

```
1  ...
2  __local int exit_flag;
3  ...
4  while( loop_condition ){
5    IDSMatcher( &loop_condition );
6
7    if(noOfAliveThreads() <= THRESHOLD)
8      continue;
9
10   if( !loop_condition )
11     atomic_inc( &exit_flag );
12
13   if(atomic_add(&exit_flag,0) == 0)
14     continue;
15
16   barrier();
17   reset( &exit_flag );
18   on_GPU_TDR();
19 }
```

The second is when no thread has signaled that it is leaving the loop. Each thread first checks whether it needs to leave, and raises the flag if yes (Line 11). Then at Line 13, each thread accesses the most updated flag by atomically adding 0 to it. If the flag is unset, TDR is skipped. In this part CITES can benefit from warp-aggregated atomics [1], but it is only supported for CUDA.

With CITES, synchronization and TDR happen only when the flag is raised. Alive threads execute until they are blocked by the barrier function at Line 16, and then the flag will be reset (Line 17).

In Blink, CITES is used in the IDSMatcher module to condense alive threads into fewer wavefronts. Nonetheless, it is also useful for solving general loop-carried branch divergence. For example, there must be a rarely-taken path for a branch that is not divergent in every iteration. We can treat taking that path as an event, and use CITES to track its occurrence and then orchestrate On-GPU TDR in a similar fashion.

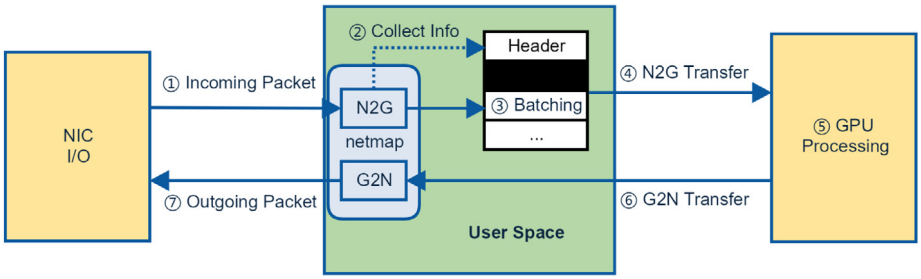


Fig. 3. System Overview of Blink.

3.3. Memory prefetching

To improve memory performance, we use one wavefront in each workgroup to prefetch some packets from global memory to shared memory with coalesced accesses. Acceleration comes from the fact that coalesced accesses improve locality and that shared memory is a much faster storage medium than global memory. This technique is useful for processing modules that require packet payload traversal.

Our consecutive packet storage layout increases effectiveness of memory prefetching. In previous solutions, packets are contained in fixed frames (*e.g.* 1536 bytes in [27]). Valid data are thus scattered sparsely in the global memory, so even coalesced global memory accesses cannot improve locality by much. In Blink, however, packets are stored consecutively, and the batch header lists the cut-off number for packets to be prefetched, which we denote as N. The first N packets assigned to each workgroup are prefetched into shared memory, and will be consumed by threads with intra-workgroup ID smaller than N before and after each round of TDR.

N should first be limited by the shared memory size, as excessive usage leads to drop in GPU occupancy and thus significant slowdown. In recent NVIDIA architectures, each SM has 96 KB of shared memory and hosts up to 64 wavefronts. When workgroup size is 256 as in this work, each workgroup can use 12 KB. With the average packet size of 606 bytes, each workgroup can prefetch 20 packets on average.

Such an average number inspired us to further limit N to be no larger than wavefront size (32), so that prefetching can be performed by only the first wavefront correctly. It assures that the prefetcher and consumer threads are all from the same wavefront, so that an initial synchronization is not necessary. In parallel with the prefetching, the other wavefronts perform packet processing normally instead of waiting idly due to barrier function. Since On-GPU TDR comes with synchronization, threads remapped to ID smaller than N can also access prefetched packets in shared memory correctly.

The actual N for each workgroup is determined and recorded during packet batching. A counter that restarts for every 256 (workgroup size) packets is utilized to accumulate the total packet size for the workgroup. Once the counter exceeds occupancy threshold or 32 packets have arrived, N is determined and written into a batch header array together with the accumulated size.

With these designs, memory prefetching is hidden in the normal packet processing, which improves memory performance with coalesced accesses and faster storage.

3.4. Module capsuling

In this work, packet processing modules are implemented using module capsuling mechanism to remove intermediate result storage and to safeguard efficiency and correctness.

Overall, interconnectivity between modules is improved due to On-GPU TDR. In face of control flow divergence, previous work has to keep modules separated. When threads finish early due to rules like TTL expiration, the following module must be re-launched so that alive threads are not mixed with finished threads in wavefronts. The new thread-data mapping has to be stored into global memory like the thread-packet redirection array in GASPP. However, On-GPU TDR takes effect during kernel execution, and thus allows modules to be combined while also reducing divergence.

To remove overheads in kernel launching and intermediate result I/O, modules are capsuled in two dimensions.

3.4.1. Data dimension

Memory resource contention is relieved as modules are generated into a single kernel. Previous intermediate result storage in between module kernels consumes global memory, and the slow access lowers processing throughput. Now, however, private memory variables are reused in different modules, and the information they carry is passed through modules with much smaller cost.

For example, the Classifier module is responsible for packet decoding. With module capsuling, later modules do not need to read information from global memory. Classifier is also simplified to perform minimum decoding for its own purpose, because later modules can easily pick up the progress to further decode, which enhances the extensibility of modular routers.

On the other hand, Classifier also benefits from module capsuling. As Classifier only looks at packet headers, memory prefetching brings no merit to the standalone module. With module capsuling, global memory accesses are more condensed, and memory optimizations take effect on all modules.

3.4.2. Control flow dimension

Computation efficiency decreases due to control flow divergence across modules. As mentioned earlier, processing can end early due to rules like TTL decrement. Snap points out that it is wasteful to spawn threads in later modules for such packets, but it is even worse not to because of management overheads [22]. However, alive threads can be capsuled into fewer wavefronts, and thus allows threads to terminate processing at any module.

Control flow flexibility is especially critical for divergent modules like IDSMatcher. Threads that are terminated in earlier modules cannot revive to join synchronization or TDR, which may undermine the efficiency and correctness of the loop divergence reduction mechanism.

Module capsuling saves resources in both dimensions, maximizing the efficiency of GPU packet processing.

3.5. NIC–GPU packet streaming

Bypassing the middleman role of CPU, our pipelining design implements packet streaming between NIC and GPU, and further eliminates GPU idling by overlapping the two-way transfer of incoming and processed batches.

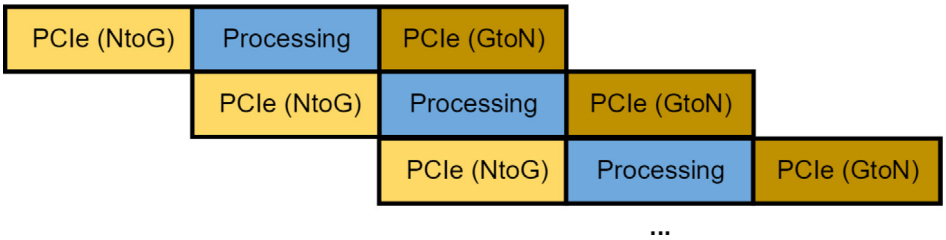


Fig. 4. Ideal pipelining when processing speed matches PCIe bandwidth. Simultaneous transfer on two directions is supported by PCIe.

Table 2
Hardware Setups for Snap, GASPP and our work.

Solution	Snap	GASPP	Snap*, GASPP* & Blink
GPU	NVIDIA TESLA C2070	2 * NVIDIA GTX 480	NVIDIA GTX 980
GPU GFLOPS	1030	2690	4981
GPU Memory Bandwidth (GB/s)	144	354.8	224.4
GPU PCIe Bandwidth (Gbit/s)	64	128	64 / 32
CPU	Intel Core i7–930	2 * Intel Xeon E5520	Intel Core i7-3820
GPU + CPU Energy Cost (W)	238 + 130 = 368	500 + 160 = 660	165 + 130 = 295
NIC		2 * Intel 82599EB, each with dual 10 GbE ports	
NIC PCIe Bandwidth (Gbit/s)		64	

Since packet I/O is very fast at the NIC side, matching PCIe bandwidth with GPU throughput can realize perfect usage of all resources. Fig. 4 depicts the ideal pipelining effect, and more details are presented below.

Batch header and payload. As mentioned in Section 3.1, the batch header includes information useful for CITES and memory prefetching:

- Maximum packet size in the batch, and the number of such packets;
- Number of packets to prefetch for each workgroup, and the total size.

The rest of header is an important offset array. For efficient memory access and storage, packets are 8-byte padded and then stored consecutively. An integer array is thus needed to mark the starting position of each packet. Actual packet size can be decoded from the packet itself.

With this straightforward payload mechanism, each batch is filled up as soon as possible, as opposed to the fixed-frame design with size ranges in previous works.

Data layout on GPU. Each batch is streamed to GPU where there are four batch slots. In the ideal scenario three slots suffice: one is receiving a new batch, one is being processed and one is sending out. The fourth slot is added to increase robustness. Once a batch is accommodated, the packet processing kernel is launched. For management simplicity, the slots are used in a round-robin fashion. State-of-the-art GPUs have enough memory for tens of batch slots, so the design can be easily extended for traffic that is more unstable.

Packet dropping. Considering the low dropping rate, we agree with GASPP that dropped packets should still be transferred back. Otherwise a larger overhead will be incurred by multiple PCIe transfers or data reordering. Instead of being dropped on GPU, the packet is labeled with a negated offset in the batch header, and will not be sent out by NIC.

Memory overhead. Blink incurs memory overhead in packet padding and batch header. On average, each packet consumes 1.8 bytes of padding, plus 4 bytes in the offset array. Every 256 packets use 8 bytes for memory prefetching. Each batch uses 8 bytes for information on max-size packets.

Since packet batches are streamed to GPU as 4-KB memory pages, we store header as a separate page for best memory alignment. A full-page header can support a batch of one million packets, while the typical batch size is thousands of packets. Compared to the average packet size of 606 bytes, total memory overhead is smaller than 1%, and has been deducted from the reported throughputs.

4. Evaluation

Experiments are conducted to evaluate individual designs and the overall efficiency.

4.1. Methodology

Hardware setups of Snap, GASPP and our platform are listed in Table 2. As Snap is open-sourced, it is evaluated on our platform to comply with the same divergence setting, which we denote as Snap*. On the other hand, we implement the key features of GASPP and evaluate it on the same platform as well, which we denote as GASPP*. All the listed CPUs support PCIe 2.0 with each lane providing 4 Gbit/s of bandwidth. Our GPU can be tested with both 16-lane and 8-lane PCIe connections. Tests are conducted for three modules implemented in OpenCL with configurations closer to those of Snap.

- Classifier uses the ACL1_10K filter set from ClassBench [23] as major classification rules.
- IPLookup uses a routing table dump from *routeview.org*, and works on a radix tree.
- IDSMatcher uses the rules for MySQL, Apache, Webapps and PHP from Snort [19]. Same as Snap and GASPP, our solution implements the Aho–Corasick algorithm [2]. For efficient memory access, packet payloads are accessed in 8-byte chunks, using the *ulong* data type.

With traffic generated based on the CAIDA trace [5], three stages of evaluation are performed: IDSMatcher, all modules and Blink. GPU-runtime performance statistics are collected using the command-line profiler from the toolkit of CUDA 7.5 on Ubuntu 16.04.

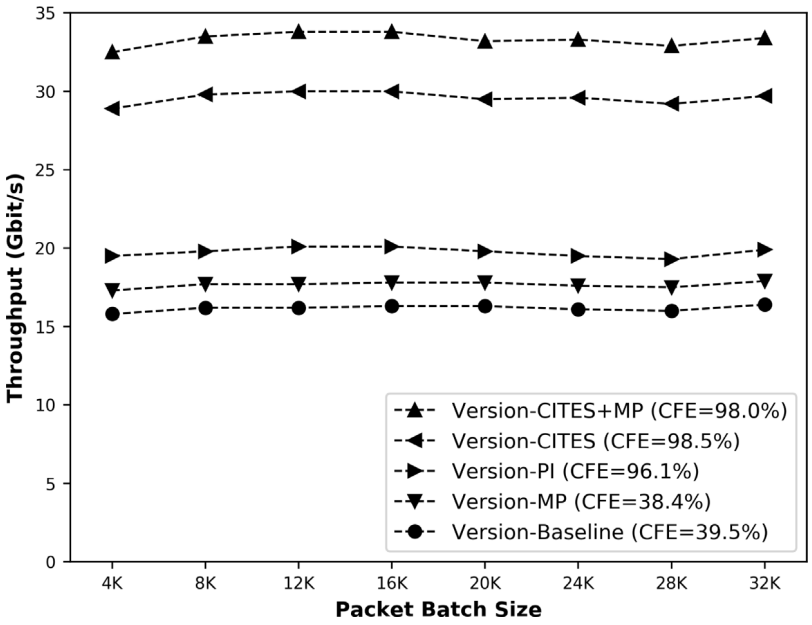


Fig. 5. Performance of five IDSMatcher versions: baseline, MP (Memory Prefetching), PI (Per-Iteration TDR), CITES and CITE+MP.

Table 3
Profiled statistics per workgroup (256 threads).

Version	Baseline	MP	PI	CITES	CITES+MP
Speedup	1	1.1	1.2	1.9	2.1
inst_executed	12,206	12,390	7355	5488	5556
active_cycles	65,795	63,447	50,924	48,361	47,316
active_warps	2104,205	2,078,732	654,288	656,246	649,003
L2$_misses	5774	5609	5634	5457	5305

4.2. IDSMatcher

This subsection evaluates CITES and memory prefetching on IDSMatcher. Fig. 5 and Table 3 compare performance of five IDSMatcher versions: baseline, PI (Per-Iteration TDR), MP (memory prefetching), CITES and CITES+MP. Average speedup for Version-CITES compared with baseline is 1.9, which is much closer to the CFE improvement than in the case of Version-PI. As expected, memory prefetching further boosts speedup to 2.1, which exceeds the speedup of 1.4 in GASPP. Peak throughput of IDSMatcher module is raised to 33.8 Gbit/s.

Two metrics in Table 3 may need extra clarification. By the definition from the profiler, *active_cycles* is the number of cycles in which at least one wavefront is executing, while *active_warps* is the accumulated number of executing wavefronts per cycle. The key findings are discussed below.

Consistent throughput. All the sustained throughputs are fairly consistent over time. Obviously batch execution helps smoothen the total amount of computation each time, while the consistency is also related to the common property of network traffic. As a case in point, 38% of packets in the CAIDA trace are larger than 1 KB, which are the actual traffic payloads. In general, it is very unlikely for consecutive hundreds of packets to all be larger than 1 KB, or to all be small handshake packets. Loop divergence thus consistently occurs. In addition, as described in Section 3.2 our reduction solution is smartly deactivated when the remaining threads all correspond to same-size packets. When this feature is taken away, the throughputs do decrease a little with more noticeable variance, and the fluctuation is still within 1 Gbit/s.

Hidden memory prefetching. Apart from the proposed design, we also tested a naive scheme where all the threads perform memory prefetching and then synchronize at a barrier function. With the same prefetch size, the number of executed instructions is unchanged, so there is no noticeable difference in profiled counters. However, the explicit synchronization increases 3% of execution time on average.

Influence on CFE. When CITES is applied, CFE is improved to 98.5%. This originates from the fact that On-GPU TDR introduces small branch divergence in itself. In Version-PI, only around 20% of the TDR rounds are meaningful. CITES eliminates TDR rounds where no thread is actually leaving the loop, and thus leads to higher CFE. On the other hand, memory prefetching also leads to different behavior for threads with IDs smaller than N, which causes a small drop in CFE.

Enhanced latency hiding. As expected, performance gain after applying CITES stems from better latency hiding. Comparing Version-PI and Version-CITES, there is no significant change in the number of cache misses. Although the number of executed instructions is reduced by 25% as less TDR is performed, these reduced instructions are lightweight ones in the sense that they do not involve global memory access. Therefore, the reduction is not on the computation bottleneck and does not affect *active_cycles* by much. CITES enables the same amount of memory latency to be better hidden in the same amount of active cycles, cutting the execution time almost by half.

Relieved loop divergence. In all the versions with TDR (PI, CITES and CITES+MP), the *active_cycles* metric decreases by around 25%, while the *active_warps* reduces drastically by 69%. This confirms the severeness of loop divergence in IDSMatcher. Without TDR, alive threads are mixed with idle threads in wavefronts. Although each wavefront requests fewer data, the total data volume is unchanged. As a result, it becomes more likely for multiple wavefronts to wait for the same memory read, increasing the ratio of *active_warps* over *active_cycles*. After all, TDR improves the computation efficiency and thus decreases *active_cycles*, saving execution time.

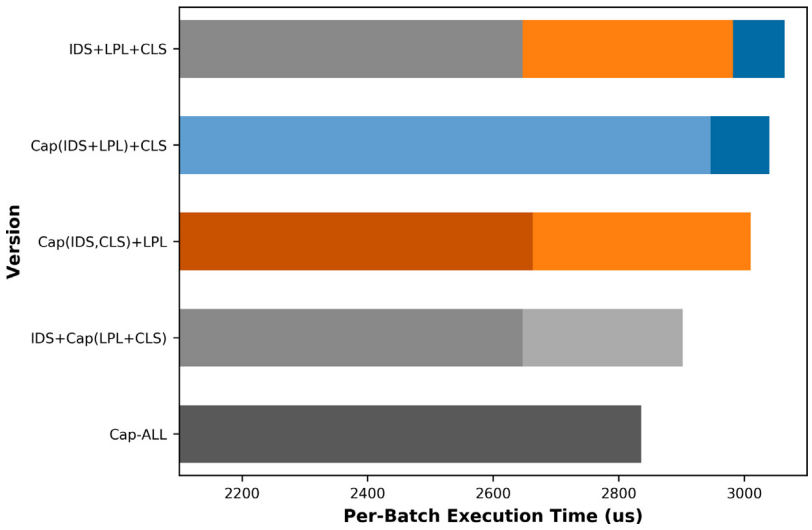


Fig. 6. Execution time for different capsuling (Cap) of IDSMatcher (IDS), IPLookup (IPL) and Classifier (CLS).

Discussion on packet batch size. As shown in Fig. 5, performance of all five versions scales well with different batch sizes. It is also confirmed for the other two modules.

Optimal batch size should be GPU-specific. Our GPU card has 2048 cores and allows 8192 threads to be scheduled on SM at a time. Smaller batches cannot fill up the GPU to maintain best latency hiding, which is why throughput of size 4 K falls a bit short compared with other sizes.

On the other hand, larger batch means higher latency, covering processing and round-trip transfer. In the following evaluation, default batch size for the results is 16 K, and IDSMatcher has both CITES and MP activated.

4.3. All modules

Results of different capsuling settings are shown in Fig. 6. Module capsuling saves time from kernel launching and memory I/O, and squeezes away 8% of execution time when applied on all three modules (*Cap-ALL*).

As expected, major time saving comes from replacing expensive global memory access with fast reuse of registers. When capsuling is applied on the Classifier module, it spends less time writing information into global memory, and recipient module also spends less time reading. IPLookup makes use of more decoded information (*e.g.* IP addresses) than IDSMatcher (*e.g.* packet length), so more time is saved for *Cap(IPL+CLS)*, which is even longer than the standalone execution time of Classifier. Thus, for routers without DPI firewall, module capsuling can reduce 36% of GPU processing time.

Despite the fact that no information is reused between IDSMatcher and IPLookup, their capsuling also provides time saving. Part of it is the basic saving of kernel launching overhead. On the other hand, IPLookup also benefits from the memory prefetching scheme in IDSMatcher, as memory reading and writing is done more condensely during single kernel execution.

4.4. Blink

Fig. 7 shows the performance of Snap*, GASPP*, Blink-64 with GPU connected to 16 PCIe lanes (64 GBit/s), and Blink-32 with GPU connected to 8 lanes (32 GBit/s). Blink makes full use of PCIe capability, and achieves higher processing throughputs even with fewer PCIe connection lanes. In both settings, Blink sustains throughputs of 31.5 GBit/s for full processing (CLS+IPL+IDS). Compared with other solutions over the same bandwidth, Blink reduces processing latency at least by half.

Snap* corresponds to the results of Snap measured on our platform. The original processing throughputs reported by Snap [22] are close to 40 GBit/s (maximum NIC capability), but their evaluation methodology is different from GASPP and our work. On the other hand, the evaluation of GASPP* confirms the original results reported in [27], with a slightly larger per-batch processing latency due to the change from dual-GPU setting to single-GPU, which occupies less PCIe resources.

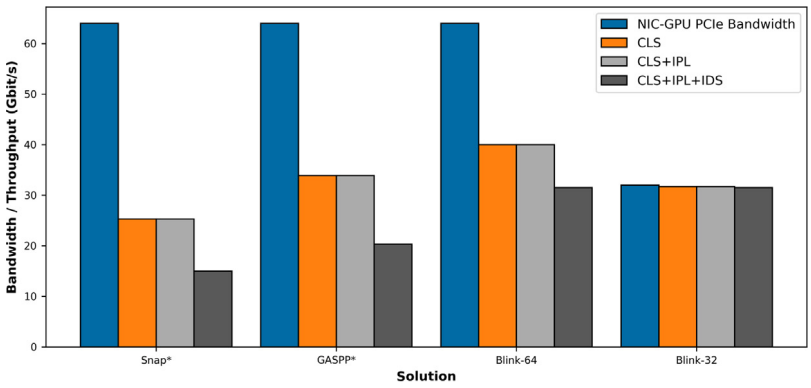
4.4.1. Throughput

For both Snap* and GASPP*, the processing throughputs are much lower than NIC–GPU PCIe bandwidths. Without IDSMatcher, GPU processing is much faster, and data transfer becomes the bottleneck. Unused space in the fixed-size frames impairs transfer efficiency significantly. The introduction of IDSMatcher, however, makes GPU processing the bottleneck and limits throughput under 21 GBit/s in both cases, despite the effort of GASPP* in loop divergence reduction. This is because IDSMatcher processing time is a magnitude longer than the other two modules and challenges the computing power of GPU, which is showcased in Section 4.3.

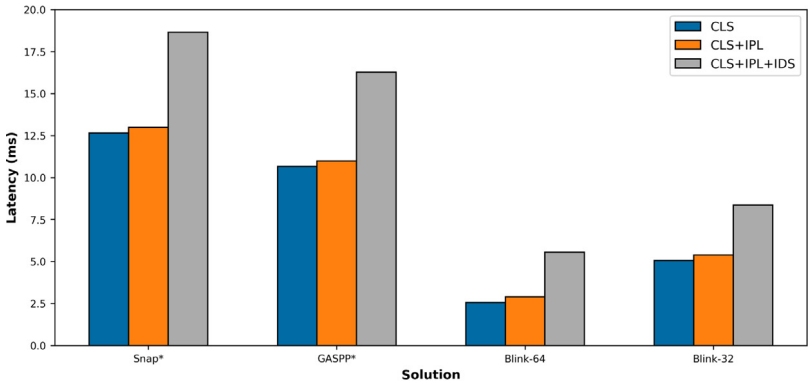
For both Blink-64 and Blink-32, the low memory overhead leads to a much higher PCIe utilization when IDSMatcher is not included. Maximum NIC throughput (40 GBit/s) is reached in the case of Blink-64. When IDSMatcher is added, the processing throughput is 31.5 Gbit/s, which is also sustained for Blink-32. With the extra 8 PCIe lanes, one more GPU can be added to Blink-32 to further increase throughput easily.

4.4.2. Latency

Here latency refers to the end-to-end processing latency per batch, from the arrival of the first packet to the outgoing of the last one. Thus it is also the worst-case latency for any packet. Round-trip data transfer accounts for the majority of processing latency, especially when IDSMatcher is not included. In terms of data transfer scheme, Snap makes use of traditional two-phase transfer, as opposed to NIC–GPU packet streaming implemented in Blink. GASPP* uses a hybrid scheme as a compromise to the need for CPU preprocessing. As shown in Fig. 7(b), GASPP* sustains a smaller latency than Snap. Although GASPP* adds CPU preprocessing and extra sorting on GPU, its transfer efficiency is higher as a gradient of frame sizes is utilized. Blink further reduces latency by compact batching and module capsuling.



(a) Bandwidth and Throughputs



(b) Latency

Fig. 7. Performance of four solutions, including two settings of Blink with different PCIe connections.

We would like to cast a closer look at the impact of IDSMatcher on the latency. For Snap*, IDSMatcher incurs an extra latency of 5.7 ms, which is essentially the module execution time before loop divergence reduction. For GASPP*, the extra latency is 5.3 ms, which is only reduced by 0.4 ms. The module execution time is actually reduced to 4.2 ms, but the preprocessing costs 1.1 ms. For Blink, the extra latency is much smaller at 2.6 ms. In comparison to the GASPP solution, our solution using On-GPU TDR not only gives better speedup, but also is more suitable for cases where latency is prioritized.

Our evaluation demonstrates that Blink can utilize PCIe bandwidth to almost the fullest extent, and that Blink achieves a much smaller processing latency given a certain NIC–GPU bandwidth. Balancing of NIC and GPU capability as well as detailed hardware selection is not in the scope of this paper, but a brief discussion will be given in the next section.

5. Related work

5.1. GPU packet processing

Light has been shed on packet processing ever since the birth of computer network. In 2000, the Click router demonstrated the necessity of modularized routing [13]. Due to the massive parallel nature of packet processing, GPU quickly became a popular accelerator for individual modules [10,11,17,25,26].

In 2015, Kalia et al. [12] questioned the low resource efficiency in GPU packet processing solutions at the time, pointing out that GPU parallelism should be further exploited. By addressing the loop divergence and transfer issues, GASPP [27] revitalizes GPU as an accelerator. Our work further pinpoints the low efficiency due to loop divergence, non-coalesced global memory accesses, inter-module I/O, and two-phase packet transfer.

In Section 4, Blink has been evaluated and compared against the other two GPU modular software routers, Snap [22] and GASPP [27]. Snap was the pioneer to demonstrate the effectiveness of such routers, and GASPP continued to integrate features like stateful packet processing. Techniques proposed in this work can also be incorporated into their solutions.

Table 4
Performance of different hardware for 40GbE networks (throughput maximized)

Hardware	CPU	GPU		FPGA
Solution	G-Opt [12]	Single-GPU Blink	Dual-GPU Blink	AccelNet [6]
Throughput (GBit/s)	39.8	31.5	39.6	31.0
Latency (ms)	3.1	5.5	2.1	1.1
Energy cost (Watt)	780	295	460	260

5.2. GPU vs. other hardware

Apart from GPU, two major options of packet processing hardware are CPU and FPGA. Table 4 lists the performance of key solutions for TCP/IP router with DPI firewall in 40 Gbit/s Ethernet (40GbE) networks. All data are obtained with settings that maximize the throughput. More details will be discussed below.

Among these hardware options, CPU provides the best programming extensibility but incurs the highest energy cost. G-Opt [12] is a state-of-the-art solution, which mimics the memory latency hiding mechanism of GPU to maximize the processing throughput. On the other hand, special-purpose hardware FPGA

incurs the smallest energy cost and latency, but comes with the hardest deployment and maintenance. AccelNet [6] is a feature of Microsoft Azure that makes use of FPGA packet processing. Its FPGA hardware supports 40GbE network, but the network capacity of Azure VM is limited at 32 Gbit/s.

Compared with CPU and FPGA, GPU is in a balanced position, as it is optimized for general low-cost parallel computing. However, it is throughput-oriented by nature, so the packets have to be batched to ensure efficient processing, which leads to extra processing latency. Blink advances both the cost effectiveness and the processing latency of GPU modular router to the next level. Solving the loop divergence issue strengthens the advantage in energy efficiency over the CPU, while direct packet streaming cuts the processing latency to be on par with the other two options.

The processing speed of Blink scales well in the dual-GPU setting, while a trade-off between latency and energy cost is revealed. When the energy budget is adequate, adding extra GPU takes up more PCIe resources and lowers the latency. In this particular case, each GPU only needs to sustain a throughput of 20 Gbit/s, allowing us to cut the batch size to 8192 packets. The latency can be further reduced to 1.1 ms by setting batch size to 4096, but the throughput will start to tremble due to low GPU occupancy. After all, an optimal setting is subject to different budgets and priorities, but the above-mentioned techniques can be applied in different cases.

5.3. Control flow divergence

To the best of our knowledge, there are very few solutions specifically targeting loop divergence. Nevertheless, some solutions targeting branch divergence may also have effect on loop divergence. There are two categories of divergence solutions, namely hardware and software.

Fung et al. [7] were the first to address control flow divergence, and laid the foundation for hardware approaches. They proposed to compact threads running the same instruction into fewer wavefronts at GPU-runtime. Hardware solutions may be effective on loop divergence by compacting alive threads. However, the achieved divergence reduction is less flexible and thus incomplete, as each thread is bounded by its relative position in a wavefront due to energy considerations.

Thread-data remapping is the most widely-used software approach [28]. Apart from On-GPU TDR which is used in this work, traditional on-CPU TDR is also relevant. In fact, GASPP has inexplicitly applied TDR by sorting according to packet size. As mentioned earlier, CPU preprocessing lowers packet transfer efficiency and thus limits throughput. The required mapping information storage also introduces considerable overheads.

6. Conclusion

In this paper, we propose Blink as a GPU modular software router for efficient low-latency packet processing. Unlike traditional GPU routers, Blink reduces loop divergence by means of pure on-GPU techniques, allowing the full implementation of NIC–GPU packet streaming. Therefore, Blink makes full use of both GPU computational power and PCIe bandwidth at the same time. Evaluation is conducted on NVIDIA GTX 980, and some key findings are listed below.

- Our novel CITES mechanism works well with On-GPU Thread-Data Remapping, and achieves a speedup of 1.9 on the IDSMatcher module.
- Our memory prefetching scheme reduces non-coalesced global memory accesses, further boosting the speedup to 2.1 for IDSMatcher.
- Module capsuling successfully reduces overhead of kernel launching and intermediate result I/O, squeezing away 8% of processing time serving as a TCP/IP router with DPI firewall.
- Blink sustains throughputs higher than 31.5 GBit/s with PCIe bandwidth of 32 GBit/s, and the processing latency is much smaller than the other works.

In the future, we will seek to incorporate stateful processing features. We hope our techniques can benefit GPU computation not limited to packet processing, and possibly inspire new hardware designs involving NIC and GPU.

Acknowledgment

This work was supported by Hong Kong RGC GRF [106160098].

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.06.009.

References

[1] A. Adinets, CUDA Pro Tip: Optimized filtering with warp-aggregated atomics - <2018-04-22>, 2014, https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.
[2] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.
[3] A. Anand, A. Gupta, A. Akella, S. Seshan, S. Shenker, Packet caches on routers: the implications of universal redundant traffic elimination, ACM SIGCOMM Comput. Commun. Rev. 38 (4) (2008) 219–230.
[4] T. Anderson, T. Roscoe, D. Wetherall, Preventing internet denial-of-service with capabilities, ACM SIGCOMM Comput. Commun. Rev. 34 (1) (2004) 39–44.
[5] CAIDA, The CAIDA UCSD Anonymized Internet Traces 2011 - <2018-03-29>, 2011, URL http://www.caida.org/data/passive/passive_2011_dataset.xml.
[6] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al., Azure accelerated networking: SmartNICs in the public cloud, in: 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18), 2018, pp. 51–66.
[7] W.W. Fung, I. Sham, G. Yuan, T.M. Aamodt, Dynamic warp formation and scheduling for efficient GPU control flow, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2007, pp. 407–420.
[8] Y. Go, M.A. Jamshed, Y. Moon, C. Hwang, K. Park, APUNet: Revitalizing GPU as packet processing accelerator, in: NSDI, 2017, pp. 83–96.
[9] N.G. GTX, 980: Featuring Maxwell, the most advanced GPU ever made, White paper, NVIDIA Corporation, 2014.
[10] S. Han, K. Jang, K. Park, S. Moon, PacketShader: a GPU-accelerated software router, ACM SIGCOMM Comput. Commun. Rev. 40 (4) (2010) 195–206.
[11] K. Jang, S. Han, S. Han, S.B. Moon, K. Park, SSLShader: Cheap SSL acceleration with commodity processors, in: NSDI, 2011.
[12] A. Kalia, D. Zhou, M. Kaminsky, D.G. Andersen, Raising the bar for using GPUs in software packet processing, in: NSDI, 2015, pp. 409–423.
[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, M.F. Kaashoek, The Click modular router, ACM Trans. Comput. Syst. 18 (3) (2000) 263–297.
[14] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, S. Swanson, Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics, Proc. VLDB Endow. 9 (14) (2016) 1647–1658.
[15] H. Lin, C.-L. Wang, H. Liu, On-GPU thread-data remapping for branch divergence reduction, ACM Trans. Archit. Code Opt. 15 (3) (2018) 39.
[16] B. Pfaff, J. Pettit, T. Koponen, E.J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al., The Design and Implementation of Open vSwitch, in: NSDI, vol. 15, 2015, pp. 117–130.
[17] E.G. Renart, E.Z. Zhang, B. Nath, Towards a gpu sdn controller, in: Networked Systems (NetSys), 2015 International Conference and Workshops on, IEEE, 2015, pp. 1–5.
[18] L. Rizzo, Netmap: a novel framework for fast packet I/O, in: 21st USENIX Security Symposium (USENIX Security 12), 2012, pp. 101–112.
[19] M. Roesch, et al., Snort: Lightweight intrusion detection for networks, in: Lisa, vol. 99, 1999, pp. 229–238.

[20] D. Rossetti, S.C. Team, GPUDIRECT: Integrating the GPU with a network interface, in: GPU Technology Conference, 2015.
[21] J.E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, Comput. Sci. Eng. 12 (3) (2010) 66–73.
[22] W. Sun, R. Ricci, Fast and flexible: parallel packet processing with GPUs and click, in: Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on, IEEE, 2013, pp. 25–35.
[23] D.E. Taylor, J.S. Turner, Classbench: A packet classification benchmark, IEEE/ACM Trans. Netw. 15 (3) (2007) 499–511.
[24] J. Tseng, R. Wang, J. Tsai, Y. Wang, T.-Y.C. Tai, Accelerating open vSwitch with integrated GPU, in: Proceedings of the Workshop on Kernel-Bypass Networks, ACM, 2017, pp. 7–12.
[25] M. Varvello, R. Laufer, F. Zhang, T. Lakshman, Multilayer packet classification with graphics processing units, IEEE/ACM Trans. Netw. 24 (5) (2016) 2728–2741.
[26] G. Vasiliadis, S. Antonatos, M. Polychronakis, E.P. Markatos, S. Ioannidis, Gnort: High performance network intrusion detection using graphics processors, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2008, pp. 116–134.
[27] G. Vasiliadis, L. Koromilas, M. Polychronakis, S. Ioannidis, Design and implementation of a stateful network packet processing framework for GPUs, IEEE/ACM Trans. Netw. 25 (1) (2017) 610–623.
[28] E.Z. Zhang, Y. Jiang, Z. Guo, K. Tian, X. Shen, On-the-fly elimination of dynamic irregularities for GPU computing, ACM SIGARCH Comput. Archit. News 39 (1) (2011) 369–380.

Huanxin Lin received his B.Eng. degree in Computer Engineering from the University of Hong Kong in 2014. He is currently a PhD candidate at the Department of Computer Science of the University of Hong Kong. His research is focused on tackling GPU control flow divergence, so as to enable efficient conditional execution on GPU. He is an advocate of GPGPU and would like to accelerate more applications with GPU.

Professor Cho-Li Wang received his B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1985. He obtained his M.S. and Ph.D. degrees in Computer Engineering from University of Southern California in 1990 and 1995 respectively. He is currently a professor at the Department of Computer Science of the University of Hong Kong. Professor Wang's research interests include parallel architecture, software systems for Cluster and Grid computing, and virtualization techniques for Cloud computing.