AdaptChain: Adaptive Data Sharing and Synchronization for NFV Systems on Heterogeneous Architectures

Kai Zhang, Jiahui Hong, Zhengying He, Yinan Jing, *Member, IEEE*, and X. Sean Wang

***Abstract*—In a Network Function Virtualization (NFV) system, network functions (NFs) are implemented on general-purpose hardware, including CPU, GPU, and FPGA. Studies have shown that there is no one-size-fits-all processor, as each processor demonstrates performance advantages to implement certain types of NFs. With more general-purpose processors such as GPUs being deployed in data center servers, the best practice to build a high-performance NFV service chain should employ available heterogeneous processors. However, current NFV systems fail to utilize these processors for acceleration. This is because, due to separate memory spaces, data synchronization is demanded to guarantee correctness, which can incur non-trivial overhead and result in low performance. This paper proposes AdaptChain, a data management facility that enables adaptive data sharing and synchronization for hybrid NFV systems on heterogeneous architectures. AdaptChain shares the host and device memory among NFs in a service chain. With adaptive synchronization plan generation and NF code adaptation, AdaptChain exploits three classes of opportunities to reduce the amount of synchronized data while guaranteeing correctness. Experimental results show that AdaptChain improves the overall throughput by up to 3.2× and reduces the latency by up to 52%.**

***Index Terms*—Adaptive synchronization, GPU, heterogeneous architecture, network function virtualization.**

I. Introduction

NETWORK Function Virtualization (NFV) is a network architecture concept that virtualizes network functions (NFs) to provide agile software implementation, deployment, and management. Studies show that both CPU and Graphics Processing Unit (GPU) are good at building and accelerating certain types of NFs [1]. By leveraging the massive number of cores in GPU, NFs with compute-intensive algorithms can benefit from the parallel computational capacity of GPU [2], [3], [4], [5]. For instance, compared with optimized CPU-based implementations, SHA-1 and 2048-bit RSA decryption show 2.3× and 4.1× performance improvements on GPU, respectively [1]. Instead, for memory-intensive algorithms such as IPv4 table lookup and Poly1305, CPU outperforms GPU (with PCIe data transfer) with techniques such as SIMD and prefetching [1], [6]. As each processor can achieve higher performance in implementing specific network algorithms, a high-performance NFV system should be able to utilize the most beneficial processors for each NF. Therefore, a service chain should be able to connect NFs that are implemented on different processors.

Avoiding or reducing expensive data movements is a common practice in building high-performance service chains [7], [8]. When each NF chooses the most appropriate processor, a service chain may consist of both CPU-based NFs and GPU-based NFs, which we call a hybrid NFV service chain. Different from a pure CPU-based service chain, the main overhead of a hybrid service chain comes from the data transfers between the host memory and the GPU device memory. For packet processing in GPU, packets must be batched in the host memory and transferred into the GPU memory through a PCIe bus, which incurs high costs [9], [10], [11]. Moreover, if an NF writes packet data in the GPU, packets must be transferred back, so that the following NFs can read the latest data in the host memory. For specific algorithms such as IPv4 table lookup, ChaCha, and SHA-256, PCIe data transfer can result in 2.7×, 3.6×, and 6.0× performance degradation, respectively [1]. With multiple NFs in a service chain, data sharing has been an effective technique to avoid data movements. For instance, with packets being shared in the host memory, the packet movement overhead between NFs can be significantly alleviated [7], [12], [13]. However, for a hybrid service chain with NFs implemented on both CPU and accelerators, data sharing cannot be directly applied because NFs write data in two or more separate memory spaces. When an NF writes a packet field in either the host or GPU memory, data in the two memory spaces become inconsistent. Consequently, synchronization is demanded to guarantee correctness. With continuous incoming traffic, synchronizations would be performed frequently and simultaneously by multiple NFs in a hybrid service chain. Besides the data transfer overheads, we find that intensive synchronizations would result in PCIe conflicts that postpone other data transfers and subsequent GPU kernel executions. Moreover, performing synchronization also involves packet parsing, batching, and data copying, which consume large amounts of CPU computational resources. As a result, the involved synchronization overhead would offset the benefits from the heterogeneous processor architectures and result in

Manuscript received 14 December 2023; revised 10 April 2024; accepted 8 May 2024. Date of publication 13 May 2024; date of current version 24 May 2024. Recommended for acceptance by M. Si. *(Corresponding author: Kai Zhang.)*

The authors are with the School of Computer Science, Fudan University, Shanghai 200433, China (e-mail: zhangk@fudan.edu.cn; hongjh20@fudan.edu.cn; zhenying@fudan.edu.cn; jingyn@fudan.edu.cn; xywangCS@fudan.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3400594

low overall performance. To make things worse, Unified Memory [14] cannot alleviate the overhead because its coarse-grained *page* data migration cannot reduce the amount of transferred data. This blocks the adoption of heterogeneous processors in an NFV system since it cannot achieve the expected performance acceleration.

In this paper, we propose AdaptChain, a data management facility that enables adaptive data sharing and synchronization for hybrid service chains on heterogeneous architectures. To minimize the synchronization costs, AdaptChain exploits three classes of opportunities to reduce the amount of synchronized data while guaranteeing correctness, i.e., avoiding unnecessary synchronization, fine-grained synchronization, and global optimization with transfer pushdown. By exploiting the optimization opportunities, a synchronization plan is adaptively generated for a service chain to reduce its data movement costs. With dynamic code adaptation, NFs in the service chain can read inputs and write outputs that are specified by the plan. Moreover, to support efficient synchronization, a system framework with a low-overhead packet state tracking scheme is designed. With the proposed set of techniques, AdaptChain guarantees the data consistency between the host and GPU memory while dramatically alleviating the synchronization overhead.

The main contributions of this paper are as follows.

- We identify the synchronization overhead for NFV systems on heterogeneous architectures and analyze the opportunities to alleviate the overhead.
- We propose AdaptChain, a data management facility for NFV systems on heterogeneous architectures with adaptive synchronization plan generation.
- We design and implement an NFV system prototype, with a lightweight packet state tracking scheme that efficiently applies AdaptChain to reduce data transfer costs.

Our evaluation results show that AdaptChain is capable of enhancing the overall throughput by up to 3.2× and reducing the latency by up to 52% for hybrid service chains with complex functionalities.

II. Background and Motivation

A. Data Movement and Sharing

In an NFV system, NFs form a service chain to deliver customized network functionalities. To leverage the architectural advantages of heterogeneous processors to build a high-performance NFV system, each NF in a service chain should be implemented and accelerated with an appropriate processor that meets its computational features. For an NFV system with hybrid NF implementations on processors such as CPU and GPU, there are two kinds of data movements in a service chain: 1) *data movement between NFs (in the host memory):* If packets are copied between NFs, it will incur substantial performance penalties [7]. State-of-the-art NFV systems address the problem in two main ways, i.e., allocating a shared memory region and passing pointers among NFs [7], [8], or discarding virtualization to avoid data movements between virtual machines [12], [13], [15]. These efforts can reap notable performance improvements. 2) *PCIe data transfer for GPU processing (between the host memory and GPU device memory):* GPU-accelerated NFs need to transfer packets to the GPU memory for processing. The expensive PCIe data transfers may result in up to 8× performance degradation for GPU-accelerated NFs [1]. Specifically, for IPv4 table lookup and Poly1305, GPU implementations without data transfers achieve 2.0× and 2.8× higher performance than CPU-based implementations. However, data transfers degrade their performance to be 23.8% and 27.5% lower than that of CPU-based implementations [1].

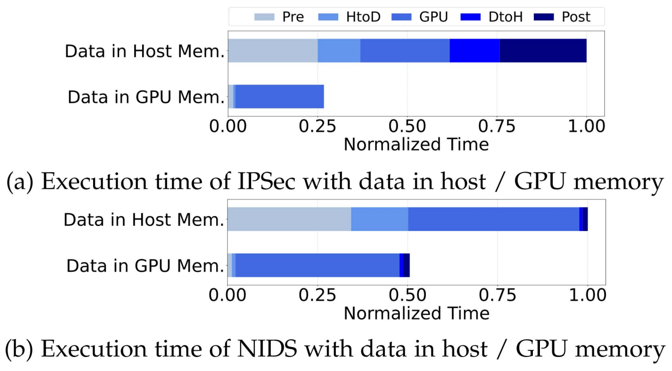


Fig. 1. Evaluation of PCIe data transfer overhead.

For a GPU-accelerated NF, if packet data have already been stored and shared in the GPU device memory, data transfers can be avoided to improve performance. Fig. 1 compares the normalized processing time of an IPSec gateway and NIDS with data residing in the host or GPU memory. To utilize GPU acceleration, an NF preprocesses packets with parsing and batching (Pre), copies data from the host memory to the GPU device memory (HtoD), launches GPU kernel processing (GPU), and transfers packets back (DtoH), which are post-processed and sent out (Post). For a single batch of packets, HtoD and DtoH data transfers take 25.8% and 16.8% of the overall processing time in IPSec and NIDS, respectively. If packets have been stored in the GPU memory, NFs only need to transfer pointers of packets in the GPU memory; thus, the HtoD and DtoH time are dramatically reduced to only 2.5% and 4.9% of the normalized time, respectively. Besides, it avoids the expensive batching and memory copying operations in preprocessing and post-processing, which take 49.2% of the time in IPSec and 35.6% in NIDS. With data being shared in the GPU memory, the CPU costs in the two NFs are reduced to only 6.3% and 7.1% when normalized to the baseline. For different batches of packets, these processing stages are executed on CPU and GPU concurrently. However, data transfers still add overheads and postpone kernel executions in NFs, which significantly degrades performance. Overall, processing data in the GPU memory can reduce the processing time by up to 77.1% and 58.9% for IPSec and NIDS, which demonstrates that synchronizations play a key role for enhancing the performance of a hybrid service chain.

The CPU and the integrated GPU in a coupled CPU-GPU architecture share the host memory, which avoids data movements between the two processors. Due to the limited chip size of desktop CPUs, the performance of an integrated GPU is much

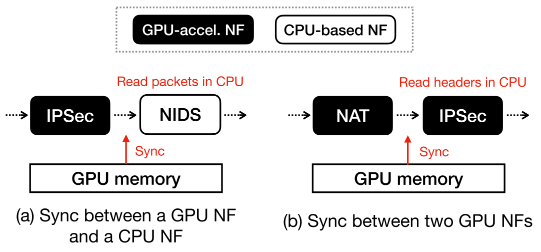


Fig. 2. Demanded synchronizations with shared GPU memory.

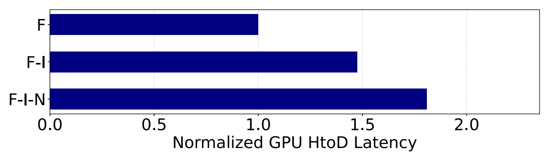


Fig. 3. The HtoD data transfer time of firewall with different degrees of conflicts.

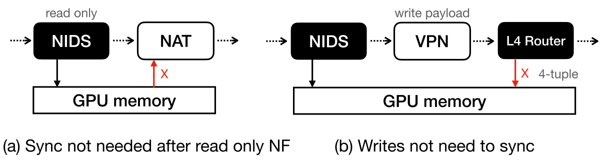


Fig. 4. Synchronizations that can be avoided.

lower than that of a discrete GPU, which is incompetent to process tens of gigabits of network traffic [1], [16]. However, its price is orders of magnitude higher than desktop processors. Therefore, this paper focuses on heterogeneous architectures with discrete GPUs, where future PCIe updating would further unleash their potential.

B. Synchronization and the Overhead

Data sharing has demonstrated its potential to enhance NFV efficiency, but it requires data synchronization for a service chain with hybrid NF implementations.

Demanded synchronizations in a hybrid service chain: An ideal case for data sharing is that GPU-accelerated NFs read and write data directly in the GPU memory while CPU-based NFs access data in the host memory. However, if an NF modifies packets in either the host or GPU memory, data in the two memory spaces becomes inconsistent. As a result, synchronization needs to be performed to guarantee the correctness of NF functionalities, or the following NF will access the wrong data in the other memory. Fig. 2 demonstrates two typical cases. (1) In Fig. 2(a), an IPSec gateway decrypts packets in the GPU memory while the following CPU-based NIDS performs pattern matching. Thus, synchronization is required for NIDS to read the decrypted packet data. (2) Fig. 2(b) shows a case where synchronization is also performed between two GPU-accelerated NFs. Because GPU is less efficient at stateful operations for its architectural characteristics, tasks such as maintaining session states are usually performed on the CPU. Therefore, session information is stored and managed in the host memory for the GPU-based NFs, and a GPU-accelerated IPSec gateway retrieves session states such as AES keys in the CPU [4]. After the NAT modifies the destination IPs and ports in the GPU memory, it has to synchronize its modified information to the host memory so that the following GPU-based IPSec can read the correct IPs and ports.

The overhead of synchronization: Besides the expensive PCIe transfer costs, the performance would be further influenced when multiple NFs perform synchronizations. There are *coexistence overheads* for CPU-based NFs [17], [18], and the new coexistence overhead of a hybrid service chain comes from PCIe data transfer conflicts. Since there are only one or two Host-to-Device (HtoD) and Device-to-Host (DtoH) DMA engines in a commodity off-the-shelf GPU, there will be conflicts when multiple NFs transfer data simultaneously. Consequently, other data transfers and the subsequent kernel executions would be queued and postponed if DMA engines are occupied [19]. Fig. 3 shows the performance degradation caused by PCIe data transfer conflicts. *F* denotes the HtoD data transfer latency of a firewall when it runs alone, and *F-I* shows that the HtoD latency rises by 47.5% when co-running with an IPSec gateway. For a service chain with three NFs (*F-I-N*), i.e., firewall, IPSec, and NIDS, the HtoD transfer latency of the firewall rises by 80.8%. The figure illustrates the performance impact of PCIe conflicts, especially for building complex service chains. With continuous incoming network traffic, NFs need to perform simultaneous synchronizations frequently, resulting in notable performance penalties. NVLink provides much higher interconnection performance, but it can only be used for data transfers between GPUs.

III. Opportunities and Challenges to Alleviate Synchronization Overhead

A. Opportunities to Reduce Synchronized Data

We identify that, in many scenarios, synchronizations in a hybrid service chain can be avoided or optimized to reduce the amount of transferred data. We classify the optimization opportunities into three main classes.

1) Avoiding unnecessary synchronization: Synchronizations are not always necessary for adjacent NFs that work on different processors. Fig. 4 demonstrates two cases where explicit data synchronizations are not needed. In Fig. 4(a), the CPU-based NAT can directly access packets in the host memory because NIDS does not modify packet data in the GPU memory. Fig. 4(b) shows another case where packets have already been transferred to the GPU memory by NIDS. With a CPU-based VPN that reads and writes packet payloads in the host memory [20], the GPU-based L4 router does not need to transfer the source and destination IPs and ports. This is because the VPN does not modify these fields, so the GPU memory already contains the latest data.

2) Fine-grained synchronization: NFs read and write different packet fields [21]. Only transferring needed but modified fields can save considerable amounts of transferred data. Fig. 5 shows

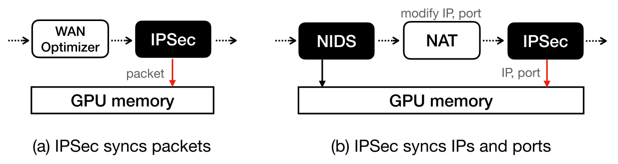


Fig. 5. Only synchronizing modified packet fields.

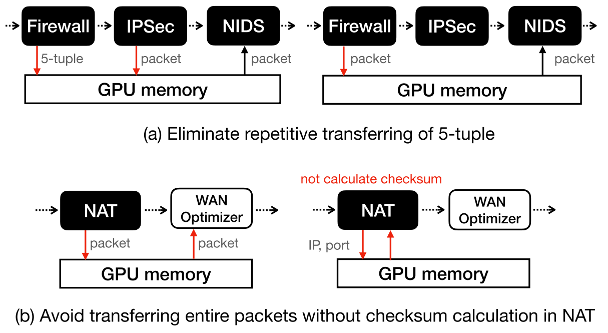


Fig. 6. Transfer pushdown to reduce synchronized data.

a case where the data transferred by an IPSec gateway can be significantly reduced in different scenarios. In Fig. 5(a), a WAN optimizer compresses packet payloads and writes packet lengths in the header. Therefore, the following IPSec gateway needs to transfer all packet data into the GPU memory for encryption. Differently, in Fig. 5(b), the first NIDS transfers packets into the GPU memory, while the second CPU-based NAT modifies IPs and ports in the host memory. Instead of transferring all packet data, the third IPsec gateway only needs to transfer IPs and ports into the GPU memory, as other fields transferred by NIDS stay unchanged. With an average packet size of 1 KB, only transferring IPs and ports can reduce more than 98% of transferred data.

3) Global optimization: For a service chain consisting of multiple NFs, evaluating the global synchronization cost can further reduce the overall transferred data. In Fig. 6(a), a GPU-based firewall transfers 5-tuple (protocol and source and destination IPs and ports) for packet filtering while the following IPSec gateway transfers entire packets to the GPU memory. Therefore, if the firewall transfers all packet data and shares them with IPSec and NIDS, the repetitive transferring of 5-tuple can be avoided. Fig. 6(b) demonstrates another service chain that consists of a GPU-accelerated NAT and a CPU-based WAN optimizer. There may be redundant logic in NFs when being composed as a service chain [22]. Since the GPU-accelerated NAT modifies both the IP addresses and ports, entire packets are transferred to the GPU memory to recalculate TCP checksums. Checksum is useful for detecting packet errors when being transferred through the network. In an NFV system, since multiple NFs are deployed on the same server to form a service chain, such checksums become unnecessary, especially considering the huge overhead incurred. Therefore, while the following WAN optimizer compresses packet payloads and recalculates the TCP checksums, we consider the checksum calculation in NAT as redundant. A more efficient way is to let the NAT only transfer IPs and ports without calculating checksums, which not only reduces data transfer overhead but also saves substantial CPU cycles.

B. Unified Memory Cannot Help

Unified Memory, available in NVIDIA Pascal or newer GPUs, provides a *managed* memory space in which both CPU and GPU see a coherent memory image within the same address space. It simplifies GPU programming by eliminating the need for explicit data copying in programs. With a page fault mechanism, Unified Memory supports on-demand page migration and GPU memory oversubscription [23]. However, Unified Memory cannot help address the synchronization overhead for a hybrid CPU-GPU NFV system. The main reason lies in the coarse granularity of its page migration mechanism. The minimum data migration size usually equals the page size in the operating system, such as 4 KB in x86 or 64 KB in Power [23]. The maximum Ethernet frame is 1518 bytes, which is much less than the minimum data migration size. Consequently, if a four-byte destination IP address is modified, Unified Memory would synchronize the entire packet and all the other data on the page. With packets being batched and continuously stored in a shared buffer, each page in the buffer will become dirty with writes to any packet field. As a result, packet data in the entire buffer would be synchronized. Therefore, Unified Memory cannot alleviate the synchronization overhead because it cannot support fine-grained synchronization. Heterogeneous Memory Management (HMM) is an abstraction that further eases programming, with which GPUs can directly access host-allocated memory. Like Unified Memory, HMM faces the same problem and cannot address the synchronization overhead.

C. Challenges for Efficient Synchronization

Without Unified Memory, a software system should be designed to exploit the three classes of optimizations. However, implementing a synchronization scheme that can handle various service chains faces the following challenges.

1) Demanding a specific synchronization plan for each service chain: An NFV system should be extensible to build various service chains with different NF compositions and implementations. Moreover, as shown in Fig. 6, a locally optimal plan by considering only adjacent NFs may be suboptimal for the service chain. Therefore, the synchronization plan should be highly adaptive, which may require searching the space of all possible plans for a service chain.

2) Fixed NF implementation cannot meet different synchronization plans: To adapt to different service chains, an NF should be flexible to synchronize different packet fields. As shown in Fig. 5, an IPSec gateway may transfer either all packet data to the GPU memory or only IPs and ports in some cases. Therefore, each NF should be able to transfer designated packet fields and read them in specified memory locations. Simply implementing an NF as several versions, with each handling a particular input and output case, however, requires substantial development efforts, and the code is also hard to maintain.

Algorithm 1: Synchronization Plan Generation PLANGEN (i, Plan), where i=0, plan.CV=0xFF..F and Other Plan Fields are Initialized as 0 or Empty in the First Call.

```
PLANGEN $i, plan$
1  if $plan.cost > mincost$ then return;
2  if $i$==$N$ then (//End of the service chain)
3    mincost = plan.cost;
4    RECORDPLAN (plan);
5    return;
6  if $plan.GSB == 1$ && $plan.CV$ & $CR_i \neq CR_i$ then
7    return;
8  if $NF_i.proc == CPU$ then (// CPU-based NF)
9    if $plan.GSB == 1$ then
10     $plan.CV| = CW_i$;
11     $plan.GV\& =\sim CW_i$;
12   PLANGEN (i+1, plan);
13 else (//GPU-accelerated NF)
14   for $allpossible\, P_{H2D}, P_{D2H}, OutGSB$ do
15     if $(plan.GV|P_{H2D})\&GR_i \neq GR_i$ then
16       continue;
17     plan.GSB = OutGSB;
18     if $plan.GSB == 1$ then
19       $plan.CV = (plan.CV\& \sim GW_i)|P_{D2H}$;
20         $plan.GV| = (GW_i|P_{H2D})$;
21     UPDATEPLAN (&plan, $P_{H2D}$, $P_{D2H}$);
22     PLANGEN (i+1, plan);
23 return;
```

3) Lacking effective and efficient system support: With packets being processed by different CPU threads and GPU kernels in NFs, adjacent packets may be processed in a different order. When synchronizing batched packets from the GPU memory, the system needs to make sure all the transferred packets in the batch have completed processing. It requires the system to track the status of every packet, which incurs non-trivial overhead. Therefore, efficient system support is demanded to guarantee both correctness and performance of the system.

Overall, the above challenges make it difficult to exploit the opportunities to reduce data synchronization costs for hybrid NFV service chains.

IV. ADAPTIVE SYNCHRONIZATION IN ADAPTCHAIN

We propose AdaptChain, a data management facility that adaptively optimizes the synchronization costs for a hybrid NFV service chain. In this section, we describe the main techniques in AdaptChain on a CPU-GPU heterogeneous architecture.

A. Synchronization Plan Generation

The solution of AdaptChain is to use the read-write behaviors of NFs to discover the potential data-sharing opportunities between NFs. According to the read-write behaviors, AdaptChain can know which modified fields by one processor will not be read by the other processor in the following NF, which helps avoid unnecessary synchronizations. AdaptChain proposes an algorithm to enumerates all possible synchronization plans for a service chain, and derive the one with the lowest overhead.

Fine-grained Synchronization: AdaptChain adopts fine-grained synchronization, which only transfers needed data to minimize its overhead. To know the data inconsistency in the two memory spaces, each NF needs to correctly specify the read and write behaviors in involved processors with read-write hints. The hints used by NFs designate which packet fields are read or written by CPU and GPU, which can be as small as a two-byte TCP port. By recording dirty packet fields written by the CPU and GPU, the system knows which memory keeps the latest data, thus, the packet fields to be synchronized can be derived accordingly. In AdaptChain, GPU-accelerated NFs are in charge of performing synchronizations, while a CPU-based NF always sees the latest version of its needed packet fields. The reason is that if a CPU-based NF performs synchronization, it needs to include GPU supports in the source code and launch an additional GPU kernel to read needed packet fields, incurring non-trivial overhead. Based on the mechanism, the H2D and D2H synchronization plans for each GPU-based NF are enumerated and evaluated to generate an efficient plan for a service chain.

To enable fine-grained synchronization, a CPU-based NF uses two 32-bit vectors to represent the CPU behavior on packet fields, i.e., CPU read (CR) and CPU write (CW). For instance, setting the 5th bit of CR to 1 means that this NF would read the destination IP address in the CPU. For a GPU-accelerated NF, besides the CR vector, a GPU read (GR) vector is used to indicate the packet fields to read in the GPU, and a GPU write (GW) vector specifies the modified fields. CW is not used for a GPU-accelerated NF, while a P_{D2H} vector reflects the synchronized packet fields from the GPU memory to the host memory.

Synchronization Plan Generation: To derive a synchronization plan, AdaptChain enumerates all possible plans with dynamic pruning to enhance efficiency. The plan generation algorithm is shown in Algorithm 1. In the process, two 32-bit vectors, i.e., CPU View (CV) and GPU View (GV), are maintained to denote the current view of packet fields in the shared host and GPU memory, respectively. A bit in CV or GV is set as 1 when the corresponding memory keeps the latest packet field, or 0 otherwise. The views are updated with read-write hints and the synchronization plan of each NF. We create a GPU shared buffer (GSB) to share the packets in the GPU memory among NFs (details in Section 5.1). Since it may needs at least one PCIe transfer for all packets, we decide whether to enable GSB according to the overall costs of a service chain.

For a CPU-based NF, which does not perform synchronization, CV and GV are updated in lines 10-11. Differently, a GPU-based NF needs to consider different synchronization plans. Because AdaptChain adopts global optimization, the scheme generates the plan that transfers the dirty fields for the current NF and enumerates possible plans that synchronize fields to be accessed by following NFs. Packet fields to be synchronized in the host-to-device direction are recorded in a vector called P_{H2D}, which is enumerated in Table I. There are two P_{H2D} plans if GSB is enabled in the previous NF and two plans if it equals 0. When the GSB is enabled (Input GSB = 1), an

TABLE I
P_{H2D} ENUMERATION

	Input GSB	P_{H2D}	Explain	H2D GSB
1	1	$(GV\&GR_i) \oplus GR_i$	needed fields	1
2	0/1	0xFF...F	entire packet	1
3	0	GR_i	direct use	0

TABLE II
P_{D2H} ENUMERATION

	H2D GSB	P_{D2H}	Explain	Out GSB
1	1	$((GV \oplus CV)\&GV\|GW_i)$ $\&(CR_{i+1}\|...\|CR_{i+k})$	needed fields	1
2	1	$(GV \oplus CV)\&GV\|GW_i$	last GPU NF	0
3	1	0xFF...F	entire packet	1
4	0	GW_i	direct use	0

NF may transfer the needed fields for the GPU kernel where the GSB does not contain the latest data (the 1st plan). With a plan, the corresponding bit in the generated vector is 0 if a field does not need to be synchronized. AdaptChain only needs to transfer the fields that are marked as 1 from the host memory to the GPU memory. With the plan, the NF transfers the needed fields and the pointer of the corresponding packets to the GPU and merges the fields into the packets stored in the GSB. If $P_{H2D} = 0$, which means the GSB already contains all needed data, the NF only transfers pointers of packets in the GSB as the input of the GPU kernel. When large payloads need to be synchronized, transferring and merging them with the packets stored in the GSB incurs high costs; the second plan can be chosen to synchronize entire packets, which saves the merging costs. When the GSB is disabled (Input GSB = 0), an NF may choose to transfer entire packets to enable GSB (the 2nd plan), or only transfer GR_i fields to perform GPU processing (the 3 rd plan). In the second plan, the GSB status is changed to enabled.

With the GSB from the P_{H2D} as the input (H2D GSB), Table II enumerates possible plans of device-to-host synchronizations, which is denoted as P_{D2H}. When the GSB is enabled, the first plan counts all fields to be read in the following CPU-based NFs, where NF$_{i+k}$ is the first GPU-accelerated NF after NF$_i$. It means that a GPU-accelerated NF performs D2H synchronizations so that its subsequent CPU-based NFs and the next GPU-accelerated NF can read correct data in the host memory. The last GPU-accelerated NF should synchronize all dirty fields in the GSB (the 2nd plan). The third plan synchronizes all packet data back to the host memory. If the P_{H2D} chooses not to enable GSB, the P_{D2H} can only select the 4th plan to synchronize the fields written by the GPU kernel. For each possible P_{H2D} and P_{D2H} combination, CV and GV are updated accordingly in lines 19-20.

Pruning for Fast Search: For a GPU-accelerated NF, there are a total of six synchronization plans if the GSB equals 1 (2 $P_{H2D}\times$ 3 P_{D2H}) and four plans if it equals 0 (3 if H2D enables GSB + 1 if not). With k GPU-accelerated NFs, there are a total of $(4 + 6^{k-1})$ plans for a service chain. To accelerate the plan generation process, the algorithm prunes two types of branches. 1) *Branches that get incorrect results:* When the condition in line 6 is met, it means NF_i will read obsolete data in the host memory, and the branch is pruned. In lines 15-16, a branch is also pruned if an NF reads obsolete data in the GPU memory. 2) *Costly branches:* During the search, we record the amount of H2D and D2H transferred data and calculate the overall synchronization cost (line 21). We also maintain the plan with the lowest overhead. If the cost of a branch has surpassed the cost of the current optimal plan, the branch is directly pruned before reaching the leaf node (line 1). The pruning helps accelerate the search process for complex service chains. When the procedure reaches the end of the service chain, a plan with a lower cost is recorded and updated (lines 2-4).

For a synchronization plan, we estimate the costs as three parts: 1) The PCIe data transfer cost $C_P = D_{H2D}/S_{H2D} + D_{D2H}/S_{D2H}$, where D_{H2D} and D_{D2H} denotes the amount of transferred data for each packet in the two directions, and S_{H2D} and S_{D2H} denotes the corresponding PCIe transfer speed with batched data. C_P estimates the PCIe data transfer time for a packet. 2) H2D batching cost $C_H = D_{H2D}/S_H$, where S_H denotes the host memory copying speed. 3) D2H batching cost $C_D = D_{D2H}/S_G$, where S_G denotes the GPU memory copying speed. With a synchronization plan of an NF being added to the service chain, the overall costs $C \mathrel{+}= (C_P + C_H + C_G)$. When batching and copying data is too expensive, NFs can transfer entire packets to avoid the cost. This is the reason why an NF may choose the second P_{H2D}, and the third P_{D2H} plan to copy consecutively stored packets even if not all packet fields are modified.

B. Dynamic NF Code Adaptation

After plan generation, the input and output of the GPU kernel and the CPU code of each GPU-accelerated NF are changed to execute the derived synchronization plan, while the code of CPU-based NFs stays unchanged. The following parts of code in an NF need to adapt to the plan. 1) *The preprocessing stage in the CPU:* As specified in P_{H2D}, the preprocessing stage batches specified data fields or collects pointers to the duplicated packets in the shared GPU memory. 2) *The GPU kernel:* With different input sources, a kernel should be modified to access data in either the transferred buffer or the GSB. Besides, the kernel should also write output in the specified location for D2H transferring. 3) *The post-processing stage in the CPU:* Based on the P_{D2H} vector of an NF, packet fields may need to be transferred from the GPU memory and used to update packets in the shared host memory. These three parts are tightly correlated, where the preprocessing stage provides the input for a kernel, and the post-processing stage handles its output. Correspondingly, the data transfer and kernel launch calls also need to adapt to the plan.

To minimize the code complexity while allowing flexibility, AdaptChain dynamically adapts an NF to a plan with code templates and conditional compilation. In the preprocessing stage, for fields in the IP or TCP/UDP header, a packet is first parsed to locate the start of the header. Then batching is performed to fetch the targeted fields and store them in a buffer with a code template: *batch(header + field_offset, buffer, field_length)*. A

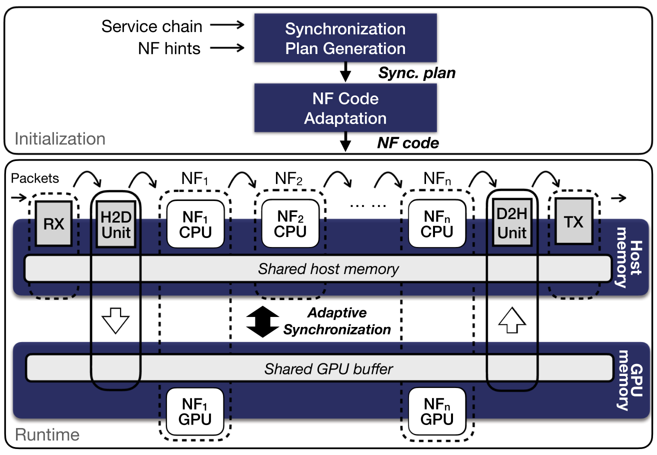


Fig. 7. The system framework.

mapping table is stored to map a fixed-length field in the plan vector to its corresponding offset in the header and its length. For variable-length payload, its offset and length are read and calculated at run time. In the post-processing stage, the same approach is adopted to update fields in the shared host memory.

For the GPU kernel, the code for two types of input are generated, i.e., in a transferred buffer or the GSB. We extract demanded packet fields in the kernel and build flexible code templates with macros in two steps. Firstly, each field is associated with a buffer name for separate transferring and added to the parameters of a kernel function. The buffer name is protected by a macro, which is only enabled when the plan specifies. Secondly, in the first lines of a kernel, a demanded packet field is read into its corresponding variable from either the transferred buffer or the GSB. Implemented with macros, each variable selects an input source with conditional compilation. By using fixed variables to access packet fields, the core algorithm of a kernel stays unchanged. At last, the GPU kernel writes the fields to be synchronized into a buffer with the same approach. In this way, the NFV service chain can be deployed to perform specified synchronizations.

V. System Implementation

A. System Framework

Fig. 7 illustrates the frame of the system prototype, which consists of two main parts, i.e., initialization and runtime. In the initialization, the system takes the service chain graph and read-write hints from NFs as the input. According to the information, a synchronization plan is derived for the service chain. Based on the plan, the system uses conditional compilation to enable specified input and output memory locations and perform corresponding data transfers.

At runtime, the hypervisor layer consists of five major functional components: *RX*, *TX*, *H2D Unit*, *D2H Unit*, and *Execution Proxy*. *Execution Proxy* is in charge of GPU virtualization. It uses API remoting to virtualize GPUs, where a shared GPU execution context is created for all NFs so that their kernels can share the GPU memory space and execute simultaneously. It receives requests from NFs and performs kernel execution or data transfer operations with CUDA driver APIs. The implementation of the mechanism is similar to that in G-NET [19].

Because the PCIe data transfer throughput of a small memory region is orders of magnitude lower than that of a large continuous one, packets have to be batched for transferring between the host memory and the GPU memory. In the system prototype, it shares the host memory among NFs to pass packets, like NetVM [7] and G-NET. In the GPU memory, we create a GPU shared buffer (GSB) to share packets among NFs. GSB is implemented as a first-in-first-out ring buffer, where incoming packets are directly transferred to the tail of the buffer, overwriting obsolete packets circularly. The detailed implementation is similar to that of Gaviss [24]. For a service chain with a designed maximum system throughput V and an allowed maximum latency T, the shared buffer size can be set as $V \cdot T$. For instance, with $V = 100$ Gbps and $T = 1$ ms, the shared buffer is calculated as 12.5 MB. The buffer size guarantees that the overwritten packets have already been processed by all NFs, so their memory can be reused safely.

A *RX* module is deployed at the beginning of the service chain to receive packets and store them in the shared host memory. Ahead of the first NF that uses the GSB, the system deploys a *H2D Unit* to batch and transfer packets to it. Each packet is associated with a packet descriptor, which records both its address in the shared host memory and the duplication in the GSB. Pointers to packet descriptors are passed among NFs with queues to reduce data movement costs, and NFs can batch and transfer GPU pointers to access the corresponding packets in the shared GPU buffer. When entire packets need to be synchronized from the GPU device memory (the third P_{D2H} plan is selected), a *D2H Unit* is deployed after the NF to help correctly perform the synchronization (details in Section V-B). After packets flow through the service chain, a *TX* module sends the processed packets out through NICs.

B. Lightweight Packet State Tracking

When entire packets need to be synchronized from the shared GPU memory (the 3 rd plan in P_{D2H}), continuously stored packets in the GSB are transferred in a batch for higher PCIe efficiency. However, as discussed in Section III-B, with packets being dispatched to multiple CPU threads in an NF for processing, adjacent packets in the GSB may have an unpredictable processing order. Therefore, the D2H Unit needs to confirm that the continuously stored packets in a memory region have completed processing before being synchronized in a batch. However, maintaining the states of every packet introduces huge overheads and results in low performance.

To address the issue, Fig. 8 demonstrates the mechanism in AdaptChain, where all transferred packets are guaranteed to be correctly processed with little influence on the system performance. AdaptChain divides the GSB as *blocks*, where a block is large enough to perform efficient PCIe data transfer in the D2H Unit. A block is a group of continuously stored packets in the shared GPU buffer, which is further divided into *units*. NFs use the unit as the basic piece in packet forwarding and processing,

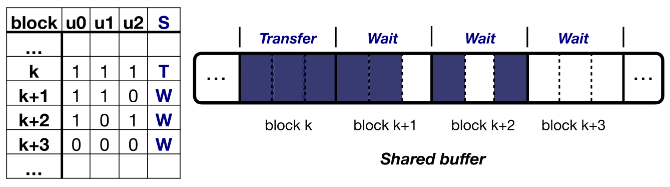
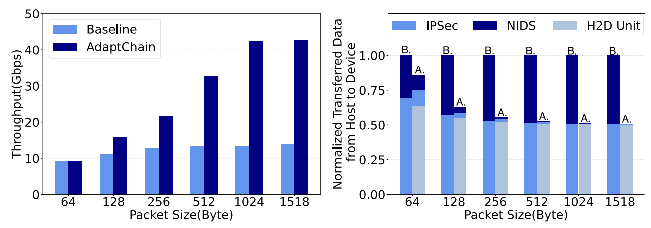


Fig. 8. Packet state tracking with blocks.



(a) Throughput improvement (b) HtoD data reduction

Fig. 9. Performance of C1.

i.e., all the packets in a unit are processed and enqueued together. With blocking queues to pass batched packets, when an NF or the TX module receives a packet of a unit, it is guaranteed that all packets in the unit have been received in one dequeue operation. Therefore, as shown in the figure, the D2H Unit tracks the units of each block in a state table. Only when all units in a block have arrived, the entire block can be synchronized (e.g., block k in the figure).

In AdaptChain, we derive an efficient block size B through evaluating the PCIe performance. To keep a consistent state tracking cost, we keep a fixed number of units ((denoted as k)) in a block. The number of packets to be stored in a unit is calculated as $s = \lceil B/P/k \rceil$, where P is the average packet size. Adopting unit and block is critical for guaranteeing correctness while alleviating state tracking overhead. The deployment of a D2H Unit would cause extra overheads, such as packet pointer passing with queues and delays for state tracking. Therefore, a penalty cost α is added for the third P_{D2H} plan in Table II. To optimize the performance, when the D2H Unit is placed at the end of a service chain, it is merged with the TX module to reduce extra enqueue and dequeue operations. Note that the state tracking scheme supports service chains with no branches.

VI. Experiments

A. Experimental Setup

Platform: We evaluate the system performance on a server with an Intel Xeon E5-2695 v4 CPU (2.10 GHz, 18 physical cores) and 128 GB main memory (DDR4, 2400MT/s, 16 GB $\times$ 8). The server is equipped with a dual-port Mellanox ConnextX-5 100GbE NIC for network I/O. An NVIDIA Titan X (Pascal) GPU with 28 streaming multiprocessors (3584 CUDA cores in total) is installed. The server runs on Ubuntu 20.04 with Linux kernel 4.15.0-43-generic. Docker 18.09.3 is used as our virtualization platform, with each NF running as a Docker instance. The GPU kernels of NFs are compiled and executed with NVIDIA CUDA Toolkit 11.1. The system and NFs are compiled with GCC 7.5.0. In the evaluation, we implement a network traffic generator with DPDK, which can consistently generate synthesized traffic by up to 100 Gbps. The traffic generator runs on a separate server, which is connected to the server running AdaptChain with a 100 Gbps optical fiber.

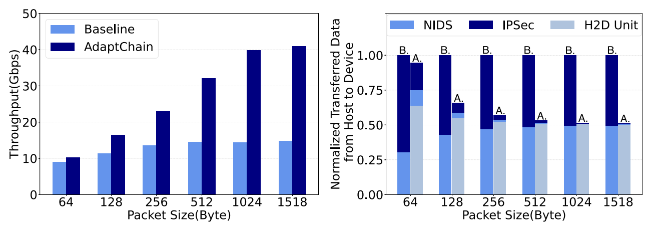
System Configuration: The hypervisor layer of AdaptChain is deployed as a system process, including RX ($\times$4 threads), H2D Unit ($\times$4 threads), Execution Proxy ($\times$1 thread), D2H Unit ($\times$5 threads), TX ($\times$4 threads), and a statistic thread. Each GPU-accelerated NF takes three threads in the server, with two threads for packet processing (DP) and one for controlling its GPU execution (CP), including submitting GPU kernels and data transfer requests. A CUDA stream is created for each DP thread, so that they can concurrently transfer packet to the GPU memory and launch GPU kernels. We use DPDK 18.11 to implement high-performance packet I/O and queue operations. In packet state tracking, the block size is set as 1 MB, and a unit contains 256 packets.

NFs and Service Chains: We implement seven NFs in the experimental evaluation: (a) *Router* is an IPv4 router that performs DIR-24-8-BASIC lookup algorithm [25] in packet routing; (b) *Firewall* performs packet filtering with bit vector linear search on the 5-tuple of packets [26]; (c) *IPsec gateway* performs packet encryption with HMAC-SHA1 and AES-128 (CTR mode) algorithms; (d) *NIDS* performs deep packet inspection with Aho-Corasick algorithm [27] consisting of 147 rules. (e) *NAT* performs network address translation, and (f) *Load Balancer (LB)* maps and distributes flows to NF instances. (g) *Monitor* performs statistics on network traffic. We build CPU-based and GPU-accelerated implementations of the NFs to compose different service chains. Since we cannot get the code of all state-of-the-art implementations for the algorithms, we implement the NFs by ourselves. Please note that faster NF implementations would benefit more from our techniques because the synchronization cost would take a larger portion of the overall processing time.

Baseline and Evaluation: We implement a baseline system based on G-NET [19] for performance comparison. G-NET is the pioneering work on GPU-based NFV systems. It shares the GPU computational resources among NFs to enhance the performance. In the baseline system, we adopt G-NET for GPU-based NFs in the hybrid service chain. Gaviss [24] enables memory sharing among GPU-based NFs. For C1-C3, a mix of CPU-based NFs and GPU-based NFs can barely utilize the advantages of GPU memory sharing. C4 and C5 consist of up to three consecutive GPU-based NFs, and these NFs share the GPU memory in the same way as Gaviss.

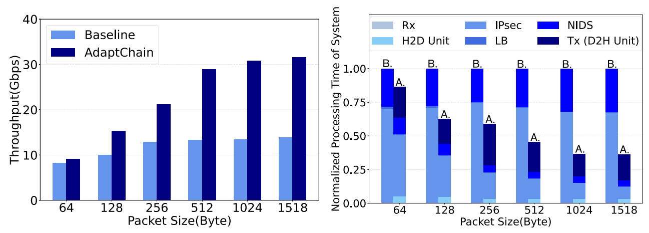
B. Performance Improvement With Adaptive Fine-Grained Synchronization

In Fig. 9, we evaluate the performance improvement from AdaptChain with fine-grained synchronization. Three service chains are used to evaluate different scenarios that lead to different synchronization plans: 1) C1: Firewall(C) - IPSec(G)



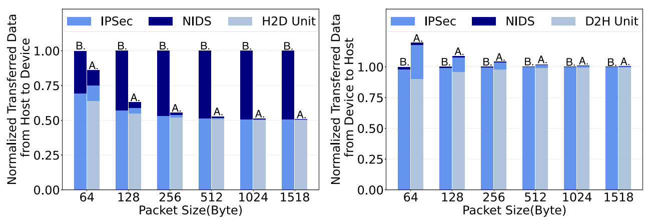
(a) Throughput improvement (b) HtoD data reduction

Fig. 10. Performance of C2.



(a) Throughput improvement (b) Time breakdown

Fig. 11. Performance of C3.



(a) HtoD data reduction (b) DtoH data reduction

Fig. 12. Transferred data reduction of C3.

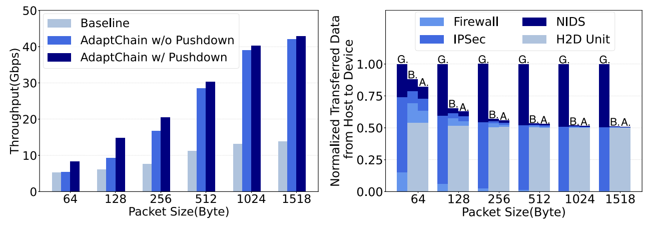
- NIDS(G); 2) C2: NIDS(G) - NAT(C) - IPSec(G); 3) C3: IPSec(G) - LB(C) - NIDS(G).

The *C* or *G* in the parentheses denotes the running processor of the NF, which is CPU or GPU. In C1, the firewall is implemented on the CPU while the IPSec gateway and NIDS are accelerated by GPU. In the baseline system, IPSec transfers entire packets because packet headers also need to be decrypted, while NIDS transfers packet payload to perform deep packet inspection (DPI). In AdaptChain, the GSB is enabled after the firewall, and packets are batched and copied to the shared GPU buffer by the H2D Unit. Then, packet descriptors with their GPU memory pointers are passed to the IPSec gateway and NIDS to utilize the shared memory. As shown in Fig. 9(a), AdaptChain improves the overall throughput by up to 3.2×. In particular, AdaptChain reduces more transferred data and achieves much higher improvement for workloads with larger packets. Fig. 9(b) demonstrates that the normalized transferred data size in the HtoD direction is reduced by up to 49.1% for the larger 1518-byte workload and 13.9% for the 64-byte workload. Take a 64-byte Ethernet frame as an example, the application layer payload is only 18 bytes for a UDP packet. Therefore, to perform DPI in an NIDS, the transferred data is reduced from an 18-byte payload and a 4-byte offset to an 8-byte GPU memory pointer, i.e., a 63.6% reduction. Instead, for a 1518-byte packet, the transferred payload is 1472 bytes, which is reduced by almost 99.5% with an 8-byte pointer.

For C2, the NAT is implemented on CPU while the NIDS and IPSec gateway are accelerated by GPU. The CPU-based NAT modifies IP addresses and ports in the host memory. In the baseline, the last NF, i.e., the IPSec gateway, has to transfer packets to the GPU memory again because the host memory keeps the latest data. Instead of transferring all packet data, the IPSec gateway in AdaptChain reuses most of the packet data being shared by NIDS in the GPU memory, and only transfers modified fields with adaptive synchronization. As shown in Fig. 10(a), adaptive synchronization improves the overall performance by 14.1%–2.8×. Fig. 10(b) shows that the HtoD transferred data is reduced by 5.6-48.9% by adaptive synchronization. Specifically, the H2D transferred data for the IPSec gateway is reduced by 72.0–99.1%.

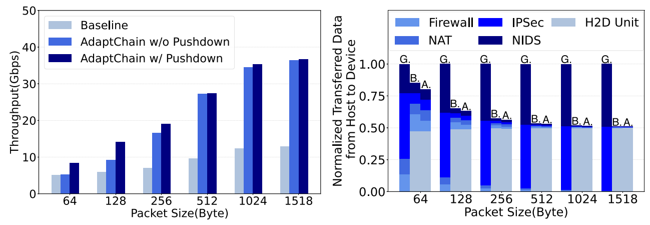
C3 demonstrates a different scenario that adaptively synchronizes packets in the H2D direction. The GPU-accelerated IPSec gateway needs all packet data in the GPU memory for decryption. AdaptChain enables GSB before IPSec, and the H2D Unit batches and transfers packets into it. IPSec adaptively synchronizes decrypted IP addresses and ports from the GPU device memory to the host memory so that the following load balancer can read correct data. The GPU-accelerated NIDS can directly access the shared payloads in the GPU memory, which have been decrypted by IPSec. In the end, both the host memory and GPU memory keep part of the latest data, and the D2H unit performs synchronization from the device memory to the host memory. Because the D2H Unit is placed after the last NF, it is merged with TX to reduce data movement overhead. Fig. 11(a) demonstrates that adaptive synchronization enhances the overall performance by 10%-2.3×. Correspondingly, as shown in Fig. 12(a), AdaptChain reduces the amount of transferred data in the HtoD direction by 13.9-49.1%. Because AdaptChain demands extra synchronizations to transfer decrypted packet fields to the host memory in IPSec, the DtoH transferred data is slightly higher than that of the baseline system (Fig. 12(b)).

To investigate the performance improvement in AdaptChain, we measure the average packet processing time in all stages of C3. Fig. 11(b) shows the execution time of a batch of 1024 packets in NFs and data plane modules of AdaptChain. The system throughput is maintained at 10 Gbps. For each packet size, there are a pair of bars to compare the baseline system (B) and AdaptChain (A), where the processing time is normalized to the baseline. For the baseline, the RX and TX modules implemented with DPDK take only a tiny fraction of the overall processing time. Around 99.8% of time is taken by NFs, especially for GPU-accelerated NIDS and IPSec gateway. Differently, in AdaptChain, packets are batched and transferred to GPU in the H2D Unit, which takes 5.9–8.9% normalized time, and synchronized back in the D2H Unit, which takes 26.5-52.3% normalized time (D2H Unit and TX). The overall execution time of IPSec, Load Balancer, and NIDS takes 38.7-67.3% normalized time, which is 41.7-85.9% lower than that of the



(a) Throughput improvement (b) H2D data reduction

Fig. 13. Performance of C4.



(a) Throughput improvement (b) H2D data reduction

Fig. 14. Performance of C5.

baseline. Overall, AdaptChain reduces the average execution time by 13.6-63.7%. This figure demonstrates that, after moving main data transfers from NFs to the H2D Unit and D2H Unit, the data transfer overhead of the system is dramatically alleviated with adaptive synchronization.

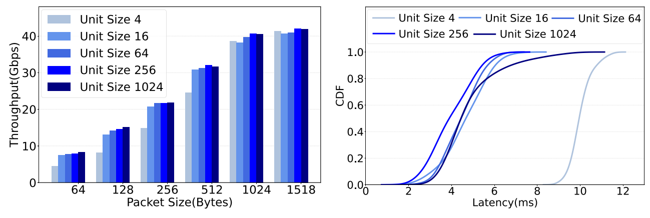
For these service chains, the reduced data transfer size approaches 50%, while the performance improvements are up to 3.2×. The main reason is that adaptive data sharing and synchronization 1) reduces the amount of transferred data, 2) mitigates PCIe data transfer conflicts, and 3) alleviates the batching overhead in CPU in certain NFs. It allows the CPU to process more packets, leading to a much higher overall throughput. It is worth noting that our approach can achieve even higher performance than running each NF alone because the adaptive data synchronization can reduce the average amount of transferred data for the NFs.

C. *Performance Improvement With Adaptive Global Optimization*

The following two service chains benefit from the global optimization in AdaptChain. In this subsection, we evaluate the performance improvements brought by global optimization with two service chains: 1) C4: Firewall(G) - IPSec(G) - NIDS(G) - Router(C); 2) C5: Firewall(G)- Monitor(C) - NAT(G) - IPSec(G) - NIDS(G).

For C4, the firewall in the baseline system copies the 5-tuple of each packet into the GPU memory, while IPSec and NIDS need to copy all packets into the GPU memory. Without global optimization, H2D Unit transfers packet data to the GSB before IPSec, and data are shared by IPSec and NIDS. Since the firewall only demands the five-tuple, it runs independently without memory sharing between other NFs. Since AdaptChain adopts a similar GPU memory sharing scheme as Gaviss, AdaptChain w/o Pushdown in the figures can represent Gaviss's perofrmance for comparison. With global optimization, the system pushes down the shared data transfer before the firewall and shares data with all three GPU-accelerated NFs. Therefore, the transferring of 5-tuple in the firewall is avoided, which only needs to transfer a packet pointer. Fig. 13 shows that, before global optimization, the system reduces the HtoD data transfer by 11.8-48.9% and improves the performance by 3.1%-3.0×. With transfer pushdown, the performance can be further enhanced by 1.8-58.4%.

C5 demonstrates another scenario. Without global optimization rules in the plan generation, the H2D Unit is placed before NAT, which shares data with IPSec and NIDS. Instead, with



(a) Throughput impact (b) Latency impact

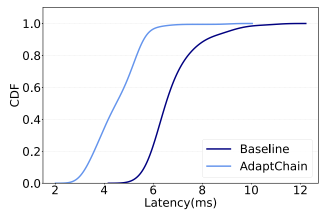
Fig. 15. Performance impact by unit size.

global optimization, the system enables GSB before the firewall rather than NAT, which avoids the transferring of 5-tuple in the firewall. Fig. 14 shows that compared with the plan without global optimization, transfer pushdown further reduces the HtoD data transfer by 0.3-6.0% and improves the performance by 0.7-58.3%. Compared with the baseline system, AdaptChain reduces the transferred data by up to 49.0% and improves the throughput by 2.8× in C5.

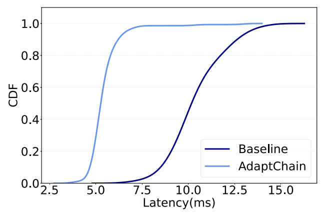
C4 and C5 show that transfer pushdown has a better effect on workloads with small packets. Specifically, for 64-byte packets, the system reduces an additional 6.3% transferred data and reaches 58.4% throughput improvement in C4, while it reduces an additional 6.0% transferred data and achieves 58.3% throughput improvement in C5. This is because the per-packet processing overheads are comparatively high for small packets. Transfer pushdown avoids the packet parsing and batching costs in the firewall by only transferring GPU pointers. Therefore, lots of CPU resources are saved to process more packets.

D. *Performance Impact From Unit Size*

Fig. 15(a) evaluates the performance impact from unit size with C4. In the experiment, the block size is fixed and set as 1 MB, which is sufficient for efficient PCIe transfer. Therefore, different unit size leads to a different number of units in a block. Unit size denotes the number of packets in a block, which defines the granularity of data transfer. Therefore, it influences the throughput of the D2H Unit. For instance, with a unit size of 16 and a packet size of 1024 bytes, a block contains 64 units. With a unit size of 4 and packet size of 64 bytes, instead, a block contains 4096 units. As shown in the figure, system throughput stays stable with unit sizes between 16 and 1024. With a unit size of 4, however, the throughput can be up to 40% lower than that with a unit size of 64. The reason is that, for workloads with



(a) C1



(b) C4

Fig. 16. Latency comparison on C1 and C4.

smaller packets or smaller unit sizes, the system needs to handle more units. As a result, the D2H Unit becomes a bottleneck when managing the states of too many units.

Fig. 15(b) compares the system latency with five different unit sizes. With a large unit size, the large granularity of batched packet passing in NFs would also influence system latency. For unit sizes between 16 and 1024 with an input speed of 10 Gbps, the 50th percentile latency increases from 3.8 ms to 4.6 ms, while the 95th percentile latency increases from 5.5 ms to 8.0 ms. However, with a unit size of 4, the 50th and 95th percentile latency dramatically increases to 9.9 ms and 11.1 ms, respectively. This is because, when the unit size is very small, there will be a large number of units in a block, which makes the D2H Unit spend a significant amount of time checking unit states. Besides, when the D2H Unit becomes a bottleneck, it makes packets blocked in the queues between NFs and system modules, which results in higher system latency. With a larger unit size, system latency also increases because more packets need to be batched when being passed and processed in the service chain. In AdaptChain, we choose a unit size of 256 as it helps maintain high throughput with only limited effect on the latency.

E. Evaluation of Latency

Besides the improvements in throughput, the reduction of transferred data also brought a reduction in overall processing latency. Fig. 16(a) and (b) demonstrate the latency improvements of AdaptChain over the baseline system on C1 and C4, respectively. For a fair comparison, the latencies of the two systems are evaluated with the same input network speed, i.e., 10 Gbps. For C1, the 50th percentile latency of the baseline is 6.4 ms, while AdaptChain achieves 4.5 ms latency, which is 30% lower. The 95th percentile latency of AdaptChain is 5.6 ms, which is 39% lower than that of the baseline system (9.2 ms).

C4 consists of four NFs with three running on GPU. Therefore, it has a higher degree of conflict compared with C1. In the baseline system, the 50th percentile latency is 10.2 ms, and the 95th percentile latency is 13.2 ms. The 50th and 95th percentile latency of AdaptChain are 5.2 ms and 6.4 ms, leading to 49% and 52% reduction over the baseline, respectively. It demonstrates that, for C4, which demands a higher amount of synchronized data, the approach of AdaptChain is more effective in reducing the packet processing latency. This is because, with less amount of synchronized data, the degree of PCIe conflicts is significantly alleviated.

VII. Related Work

A series of systems are proposed to build NFV systems on commodity CPUs [7], [8], [12], [13], [28], [29], [30]. NetVM [7] is a CPU-based NFV system that utilizes data sharing in the host memory. As its follow-up work, Microboxes [31] eliminates redundant TCP processing in a service chain to enhance performance. G-NET [19] and Grus [32] are NFV systems that utilize GPUs to enhance system performance. However, both face the inefficiencies described in Section II-A. Gaviss [24] shares GPU device memory among GPU-accelerated NFs. Different from AdaptChain, Gaviss does not use adaptive synchronization but directly transfers all packets to the GPU memory in the first NF, and the rest of the NFs directly process packets in the GPU memory.

VIII. Conclusion

Aiming at solving the data synchronization overhead in hybrid NFV service chains on heterogeneous architectures, we propose AdaptChain, a data management facility for adaptive data sharing and synchronization in NFV systems. AdaptChain adaptively generates the synchronization plan for a service chain and changes the corresponding NF code to deploy the plan. With an efficient system framework with lightweight packet state tracking, the correctness of the service chain is guaranteed with enhanced system performance. Our evaluation results show that AdaptChain improves the throughput of a hybrid service chain by up to 3.2× and reduces the latency by up to 52%. With AdaptChain, existing NFV systems can effectively utilize heterogeneous architectures to achieve higher performance.

References

[1] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as packet processing accelerator," in *Proc. 14th USENIX Symp. Networked Syst. Des. Implementation*, 2017, pp. 83–96.

[2] M. A. Jamshed et al., "Kargus: A highly-scalable software-based intrusion detection system," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 317–328.

[3] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: A multi-parallel intrusion detection architecture," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 297–308.

[4] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Symp. Networked Syst. Des. Implementation*, 2011, pp. 1–14.

[5] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 195–206.

[6] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing," in *Proc. 12th USENIX Symp. Networked Syst. Des. Implementation*, 2015, pp. 409–423.

[7] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. 11th USENIX Symp. Networked Syst. Des. Implementation*, 2014, pp. 445–458.

[8] V. Jain et al., "L25gc: A low latency 5 g core network based on high-performance NFV platforms," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 143–157, doi: 10.1145/3544216.3544267.

[9] D. Tarjan, K. Skadron, and P. Micikevicius, "The art of performance tuning for cuda and manycore architectures," in *Proc. Birds-of-a-Feather Session at SC*, 2009. Accessed: May 18, 2024. [Online]. Available: https://www.cs.virginia.edu/∼skadron/Papers/cuda_tuning_bof_sc09_final.pdf

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

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

[12] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Maguire Jr., "Metron: NFV service chains at the true speed of the underlying hardware," in *Proc. 15th USENIX Symp. Networked Syst. Des. Implementation*, 2018, pp. 171–186.

[13] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the v out of NFV," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 203–216.

[14] Nvidia, "Cuda C programming guide," Release 12.4, 2023. Accessed: May 18, 2024. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[15] H. Li, Y. Dang, G. Sun, G. Liu, D. Shan, and P. Zhang, "LemonNFV: Consolidating heterogeneous network functions at line speed," in *Proc. 20th USENIX Symp. Netw. Syst. Des. Implementation*, 2023, pp. 1451–1468. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/li-hao

[16] K. Zhang, J. Hu, B. He, and B. Hua, "DIDO: Dynamic pipelines for in-memory key-value stores on coupled cpu-gpu architectures," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 671–682.

[17] Y. Hu and T. Li, "Enabling efficient network service function chain deployment on heterogeneous server platform," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 27–39.

[18] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proc. Annu. Conf. ACM Special Int. Group on Data Commun. Appl., Technol., Architectures, Protoc. Comput. Commun.*, 2020, pp. 270–282.

[19] K. Zhang et al., "G-NET: Effective GPU sharing in NFV systems," in *Proc. 15th USENIX Symp. Networked Syst. Des. Implementation*, 2018, pp. 187–200.

[20] M. Feilner, *OpenVPN: Building and Integrating Virtual Private Networks*. Birmingham, U.K.: Packt, 2006.

[21] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 43–56, doi: 10.1145/3098822.3098826.

[22] B. Deng and W. Wu, "NFReducer: Redundant logic elimination for network functions with runtime configurations," in *Proc. IEEE INFOCOM - Conf. Comput. Commun.*, 2021, pp. 1–10.

[23] N. Sakharnykh, "Maximizing unified memory performance in CUDA," 2017, [Online]. Available: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/

[24] L. Guo, K. Zhang, and X. S. Wang, "Gaviss : Boosting the performance of GPU-accelerated NFV systems via data sharing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4472–4483, Dec. 2022.

[25] P. Gupta and N. McKeown, "Packet classification on multiple fields," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 147–160, 1999.

[26] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM Computer Commun. Rev.*, 1998, pp. 203–214.

[27] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[28] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 511–524.

[29] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr., and D. Kostić, "SNF: Synthesizing high performance NFV service chains," in *Proc. PeerJ Comput. Sci.*, Nov. 2016, vol. 2, p. e98, doi: 10.7717/peer-j-cs.98.

[30] M. Reininger et al., "Safely bringing network function virtualization to Tor," in *Proc. ACM SIGCOMM 2021 Conf.*, 2021, pp. 821–835, doi: 10.1145/3452296.3472919.

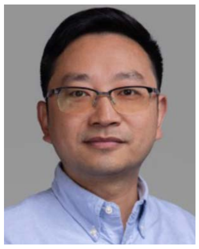
[31] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 504–517.

[32] Z. Zheng et al., "Grus: Enabling latency SLOS for GPU-accelerated NFV systems," in *Proc. IEEE 26th Int. Conf. Netw. Protoc.*, 2018, pp. 154–164.

Kai Zhang received the PhD degree with the University of Science and Technology of China in 2016. He is currently an associate professor with the School of Computer Science, Fudan University. He was a research fellow with the National University of Singapore and a visiting scholar with The Ohio State University. His main research interests include parallel and distributed computing and database systems.

Jiahui Hong received the master's degree from the School of Computer Science, Fudan University in 2023. His research interests include networked systems, general-purpose GPU computing, and high-performance computing.

Zhengying He received the BS, MS, and PhD degrees in computer science from the Harbin Institute of Technology, China, in 1998, 2000, and 2006 respectively. He is currently an associate professor with the School of Computer Science, Fudan University. His research interests include keywords search on structured data, query processing on RDFdata, and Big Data.



Yinan Jing (Member, IEEE) received the PhD degree in computer science from Fudan University, Shanghai, China in 2007. He is an associate professor with the School of Computer Science, Fudan University. He was also a visiting scholar with the Department of Computer Science, University of Southern California. His research interests include Big Data analytics, database systems, spatial and temporal data management, and data security and privacy. He is a member of ACM.

X. Sean Wang received the PhD degree in computer science from the University of Southern California. He is currently a distinguished professor with the School of Computer Science, Fudan University, Shanghai, China. Before joining Fudan University in 2011, he was the dorothean chair professor with the University of Vermont. His research interests include data systems and data security. He is the fellow of CCF and the member of ACM.