Enabling Efficient Network Service Function Chain Deployment on Heterogeneous Server Platform

Yang Hu
Department of Electrical and Computer Engineering
The University of Texas at Dallas
Richardson, TX, USA
yang.hu4@utdallas.edu

Tao Li
Department of Electrical and Computer Engineering
University of Florida
Gainesville, FL, USA
taoli@ece.ufl.edu

***Abstract*—Network Function Virtualization (NFV) aims to run software-implemented network functions on general hardware such as Commodity Off-the-Shelf (COTS) servers to trade the application-specific performance with generality and re-configurability. Nevertheless, with the wide adoption of general accelerator such as GPU, the researchers seek to boost the performance of software-based network functions while trying to maintain the reusability and programmability in the meantime. The Service Function Chain (SFC) is a key enabler of service flexibility of NFV. The network functions stitch into a chain to provide differentiated services to multi-tenants. However, our characterization results show that existing heterogeneous packet processing frameworks do not handle NFV SFC well since two new overheads, the aggregated processing overheads and co-existence interference overheads, are introduced by SFC.**

Motivated by our characterization, we propose NFCompass, a runtime framework that employs SFC re-organization technique and graph-partition based task scheduling technique to conquer the two challenges brought by SFC. By re-organizing the SFC components, the length and complexity of processing paths are reduced and the aggregated overheads are mitigated. By applying the graph-partition based task allocation, better load balance is achieved and the data transfer overheads are considerably reduced.

Keywords-NFV; GPU; Service-Function-Chain; Graph-Partition

I. INTRODUCTION

Modern network operators and data center vendors are continuing to enrich their network functions (NFs) to provide various additional services, such as TCP optimization, packet encryption/decryption, NAT, video transcoding, etc. for their tenants. Traditional network function enrichments are achieved by deploying hardware based middleboxes on the network traffic's path, which requires costly capital and operational expenditures and is incapable of accommodating swift on-demand service provisioning.

To tackle the challenges of cost and flexibility of service provisioning, Service Providers (SPs) such as AT&T, Verizon, and China Mobile propose to run network functions as virtual machines or containers on Commercial Off-The-Shelf (COTS) servers, hence replacing expensive and inflexible hardware-based middleboxes with software-based Virtualized Network Functions (VNFs). This revolution is called Network Function Virtualization (NFV) [1]–[3].

The core benefit of NFV is to trade the application-specific performance (hard-coded middlebox) with generality and re-configurability (software-implemented virtual middlebox), since many NFs, such as firewalls, intrusion detection system, and load balancers process packets based on whole packet payloads while not only headers. This will bring additional computation loads to physical servers.

With researchers are always in hot pursuit of performance, many hardware acceleration techniques such as multi-core CPU [4], network processor [5][6], GPU[7]–[10], FPGA [11]–[13], etc., have been developed to boost the performance of software-based network functions while trying to maintain the reusability and programmability in the meantime. Offloading the network functions to hardware accelerators such as GPUs can benefit network functions that involve intensive parallel lookups into large data structures, such as route lookup [8], deep packet inspection [14], and encryption [15].

A key enabler of the service flexibility of NFV is the Service Function Chain (SFC), where the network traffic traverses a sequence of NFs to provide differentiated services for multiple tenants. For example, a parent control service calls for an additional content filtering middlebox on the service path for target traffic flows.

Though existing heterogeneous hardware-accelerated packet processing frameworks [16], [17] have well explored the resource handling issues and may perform well for certain network functions, the adoption of SFC can pose several new challenges to heterogeneous packet processing system. This can result from two root causes.

First, SFC can impose aggregated processing overheads. These aggregated overheads are resulted from complicatedly interconnected network functions. Our characterization demonstrates that the aggregated overheads consist of packet re-organization overheads and offloading overheads. The packet re-organization could be caused by the batch split between packet processing elements such as Click modules and the packet re-organizing in the stateful processing. The offloading overheads originate from the frequent GPU kernel launch and tear down and various offloading ratio for different NFs. With the increase of the length of SFC, the aggregated packet re-organization overheads significantly reduce the system throughputs, and the aggregated offloading overheads even can offset the GPU acceleration benefits.

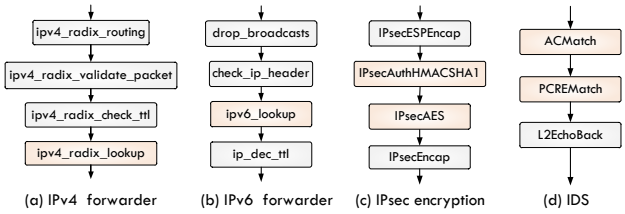

Figure 1. Samples of Click module configurations

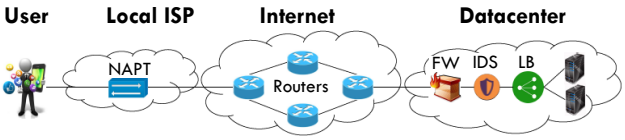

Figure 2. A typical service function chain in telecommunication data centers. The user traffic will be processed by firewall, DPI and load balancer sequentially.

Second, deploying SFC on heterogeneous platform also brings the resource interference. By characterizing the performance of three typical software network functions (IPv4/v6 forwarder, IPsec gateway, and Deep Packet Inspection) on GPU enabled COTS server, we observe challenges of *running a single NF* and *co-scheduling/ running of multiple NFs*. Different from the traditional applications on which the prior works of heterogeneous system focus, the performance of a single network function is collaboratively impacted by *packet batch sizes*, *traffic characteristics*, *co-running network functions*, and *task offloading ratios between CPU and accelerators*. In the NFV environment with varying traffics, the optimal configurations for network function task mappings can deviate significantly. Existing work either is incapable of handling the fast-switching network traffics[18]–[20], or relies on inefficient ad-hoc and manual optimizations [7], [9], [17].

Our characterizations motivate us to design a framework that is able to efficiently run service function chain on heterogeneous platform. It employs two novel techniques to cooperatively ease the task mapping of network functions. The first technique is a two-level SFC optimization that parallelizes available NFs at SFC level and synthesizes NFs at NF level. This is motivated by the aggregated processing overheads caused by long SFC. Parallelizing available NFs can help to reduce the length of processing path and reduce the traffic latency. In addition, synthesizing NFs in each sequential SFC by eliminating redundant NF elements and re-ordering certain read and write elements can further simplify the SFC and save the compute resource. The second technique is a Directed Acyclic Graph (DAG) based tasking model that minimizes data transfer across different memory space, and meanwhile maintains a load balance between processors. Our graph-partition algorithm partitions the optimized SFC deployment graph into sub-graphs, which uses the node execution time and data transfer time as node weights and edge weights in a data-flow graph. Our experimental results show that NFCompass can achieve 60% more throughput improvement, and about 1.4X to 9X lower than state-of-the-art packet processing framework when running real world service function chain.

Our key contributions in this work are as follows.

- To the best of our knowledge, this is the first work that characterizes running SFC on heterogeneous platform. We are the first to summarize the two main overheads caused by SFC, *aggregated processing overheads* and *co-existence interference overheads*.
- Our experiences show that software network functions can perform extremely diverse under different configurations. The optimal configuration is collaboratively determined by *packet batch sizes*, *traffic characteristics*, *co-running network functions*, and *task offloading ratios between CPU and accelerators*.
- We propose NFCompass, a runtime framework. It solves the issues of SFC deployment on heterogeneous platform by exploiting two novel techniques, the two-level SFC re-organization and graph-based task scheduling.

The rest of this paper is organized as follows. Section 2 describes essential backgrounds of NFV SFC and GPU offloading. Section 3 characterizes running the SFC on heterogeneous platform and summarizes two critical overheads. Section 4 introduces our main design. Section 5 reports our validation and evaluation results and Section 6 concludes this paper.

II. BACKGROUNDS AND MOTIVATION

In this section, we briefly introduce the essential concepts in NFV, including Click modular router and service function chaining. We also describe existing heterogeneous packet processing acceleration implementations.

A. Network Function Virtualization

NFV implements network functions as software appliances instead of hardware devices. Traditional network functions such as routers and firewalls require specialized devices for different functionalities, and each needs to be individually deployed. NFV enables service providers to run NFs on commodity off the shelf servers, thus easing deployment of services from different vendors. NFV promises to greatly improve the flexibility with which services can be deployed and modified, while lowering costs.

Many of the NFV platforms take the advantage of state-of-the-art I/O libraries such as Intel DPDK [21] and Netmap [22]. Such kinds of data plane acceleration techniques mitigate the kernel network stack packet processing overheads through lock optimization, zero-copy and kernel bypass. However, these libraries only focus on packet processing between NIC and userspace applications. The data movement optimization between network function processing elements is still absent.

1) Packet Processing Style of Network Functions

We introduce Click modular router [23], a framework that is widely used to model the middlebox abstraction, as shown in Figure 1. In Click, the basic packet processing components are defined as *elements*, which may generate, process, or receive packets. Developers can construct their

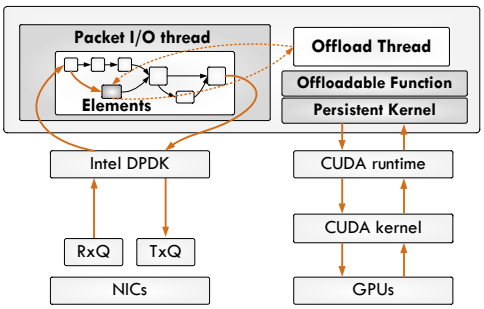

Figure 3. A high-level packet processing architecture.

own network functions by connecting a series of elements as pipeline structures.

2) Service Function Chain in NFV

The traditional network function processing is implemented by letting the network traffics traverse a series of hardware-based proprietary middleboxes. The advent of NFV improves the total cost of ownership by altering the hardware-based middleboxes into software-based virtualized network functions. To provide the on-demand scaling and provisioning of network services, the Services Function Chaining should also be implemented in NFV. A typical example of service function chain is shown in Figure 2. A packet may traverse a firewall, an Intrusion Detection System (IDS), and a load balancer to reach its final destination. A number of challenges arise when addressing the design of a SFC system. The *reduced throughput* and *increased latency* caused by the increasing length of SFC may hurt the quality of service of network traffics.

B. Heterogeneous NFV Acceleration

Recently the researchers resort to the heterogeneous accelerators such as general-purpose GPUs (e.g. Intel Xeon Phi coprocessor [24]) and many-core processors (e.g. Tilera [6]) to augment the packet processing leveraging the data parallelism. Recent approach also explores the opportunity to use FPGA to accelerate software NFs [11]. Prior works have shown the potential of using GPUs as packet processing accelerators [7], [8], [14], [15], [17]. In general, GPU can transparently hide the 60-200ns of latency required to retrieve data from main memory, thus speeding up the entire packet processing elements pipeline.

1) GPU Offloading Model for Packet Processing

We first present a packet processing model based on Receive Side Steering (RSS). A high-level model is shown in Figure 3. It has two critical threads: packet I/O threads and offload threads. We adopt a pipelining model in our setup where each thread is affinitized to an individual CPU core. The packet I/O threads reads incoming packet batches using Intel DPDK [21]. The IO loops of the packet I/O threads synchronously receive packet batches from the NIC RX queue, check the validity of each packet, and inform GPU offload thread or discard. In this paper, we use the batch size of 32 and 64 packets. The IO loop also polls offload completion notifications from the offload thread. Offload threads manage communications with accelerators such as GPUs. As packet I/O threads send offload tasks containing

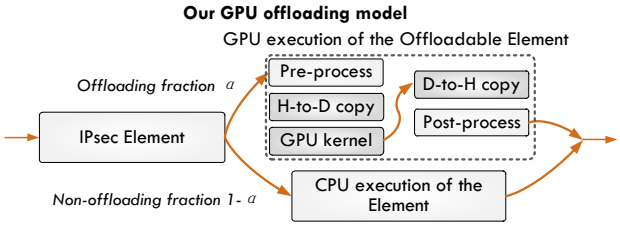

Figure 4. The offloading acceleration of an IPSec.

TABLE I. PLATFORM CONFIGURATIONS

Item	Value
COTS system	SuperMicro 8048B, 4-socket NUMA
Processor	Intel Xeon E7-4809 v2, 1.9GHz (IvyBridge) 6 physical cores (12 Threads)/socket 12MB L3 cache for each socket 64KB L1 cache and 256KB L2 cache for each core
Memory	64GB, DDR3 for each socket, 256GB in total
GPU	2x Nvidia Titan X, 3072 CUDA cores, 336.5GB/s
NIC	Intel X540 10GBase-T, Mellanox 40GB SFP+ Associate with socket 0 and 4

packet batches and element information, they execute them on a pool of command queues for the configured GPUs.

The biggest challenge of heterogeneous packet processing is to hide the packet offloading details. In this work, we use a modified offloading mechanism introduced in [9]. The offloadable Click elements employ CPU-side functions, GPU-side functions, and the corresponding input/output data formats. Figure 4 shows a data offloading process of the IPsec, the data offloading process runs the accelerator-side functions including preprocessing of the input data, host-to-device data copies, kernel execution, device-to-host data copies, and post-processing of the output data. We employ a persistent kernel design for GPU code.

III. CHALLENGES OF DEPLOYING SFC ON HETEROGENEOUS SERVER PLATFORM

In this section we explore two root causes that incur the performance issues of deploying SFC on heterogeneous server platform, the *aggregated processing overheads* and the *co-existence interference*. Our methodology is to identify the quantitative performance impacts of these two root causes by running both *single network function* and *multiple chained network functions* on GPU-equipped COTS server platform. These root causes motivate our design of re-organizing SFC process pattern and DAG-based task scheduling in Section 4.

A. Experimental Setup

1) Platform

Our physical platform configuration is shown in Table 1. Our heterogeneous platform equips two Nvidia Titan X. The system uses four Intel X520 SPF+ 10 Gigabit Ethernet NICs divided into two groups and are associated with two NUMA nodes respectively. We use Ubuntu Linux 14.04 and DPDK 16.04, and NVIDIA CUDA 8 as GPU framework.

2) Network Function Workloads

We choose three packet processing network functions, that present both compute-intensive and memory-intensive behavior to fully test the heterogeneous platform.

IPv4/IPv6 Packet Forwarding. IP packet forwarder is the simplest yet the most deployed service. Upon receiving a packet, the forwarder uses IP forwarding lookup table to rewrite the destination IP address for this packet and transmits it. The IP forwarding table lookup could be memory-intensive operation. The IPv4 table lookup takes two memory accesses and IPv6 table lookup takes up to 7 memory lookups. The hashing in IPv6 also makes it compute-intensive since binary search should be performed for every destination address.

IPsec Encryption. The Internet Protocol Security (IPsec) is an encryption protocol suite that is widely used by Internet applications, VPN, and P2P communications to secure the IP traffics. Since intensive encryption and hashing operations are performed, IPsec is highly computation-intensive. The frequent packet payload copy also makes it an IO-intensive application. We use HMAC-SHA1 to authenticate the packets and AES-128CTR to encrypt them in our IPsec encryption. We implement it to exploit AES-NI for faster computation AES in recent CPU models.

Deep Packet Inspection. Deep packet inspection (DPI) is an essential security approach that is adopted in network traffic processing applications such as network intrusion detection systems (IDS) [25], traffic classification [26], and Web application firewalls [27]. These applications perform pattern matching to select flows or packets for stateful inspections. For the string matching we use Aho-Corasick algorithm [28] that is implemented in Snap [17]. For the regular expression we use a Deterministic Finite Automata (DFA) implementation. DPI tends to exhaust the compute and memory resource due to its heavy pattern matching operations. The host-to-device packet copy operations also make it IO-intensive.

3) Test Traffics

In this paper, we use network intensive micro-benchmark Netperf [29] to generate traffic loads. Unless otherwise specified, we use a randomly generated IP traffic with UDP payloads and offer 40 Gbps load from two separate packet generator machines, 80 Gbps in total. For IPv6 router application, we use IPv6 headers and IPv4 headers for other cases.

B. Impacts of Aggregated Processing Overheads

To demonstrate the impacts of aggregated processing overheads, we first evaluate the performance of network traffics and quantitatively analyze the packet processing overheads on a single network function setup. We then run chained network functions on heterogeneous platform and vary the length of SFC to compare the performances with their counterparts running on general COTS platform. Our experiment results show that the benefits of GPU acceleration could be offset by the aggregated overheads with the increase of NFs.

1) Aggregated Packet Re-organization Overheads

To guarantee the functional correctness and improve the efficiency, existing packet processing frameworks involve several approaches that maintain the packet sequence and process packets in batch. However, the parallelism of GPU processing does not support packet order preservation well.

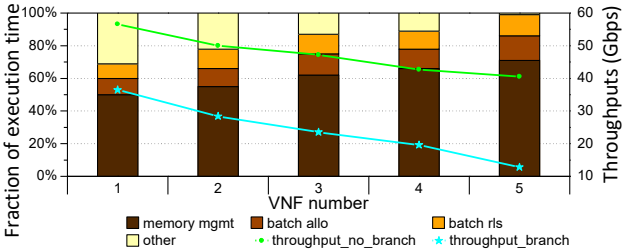

Figure 5. Throughputs and overheads fractions of w and w/o batching split.

a) Re-organization Caused by Branching

Modern heterogeneous framework employs batching method [9] to increase packet processing throughput and reduce the per-packet overheads. However, the batching method is not intrinsically compatible with the Click module's element abstraction. Since the output packets from one element may be diverted to different Click elements, a larger batch of packets have to be re-organized and pushed to different elements as smaller batches. Such re-organizing incurs extensive memory operation and batch management overheads. We demonstrate this in Figure 5. We design a simple branch test element. We test the throughputs of with_split and without_split by running the branch test elements as service chain. The results show that the throughput of with_split drops from 36.5Gbps to 15.8Gbps

In addition, the branch condition in Click also incurs the control-flow divergence on the GPU-based platform when block-level parallelism is employed, where the blocks of threads are cooperatively executing packet processing. When the packets that are belonged to different traffic flows are scheduled in the same warp, they may take different execution paths and can cause control-flow divergence. With the increase of SFC length, the aggregated idle condition can consume a significant amount of processing time.

b) Re-organization Caused by Stateful Processing

It is necessary to maintain stateful processing for traffics in intrusion detection system and traffic classification. The stateful processing ensures the in-order processing of packet in the same connection. To guarantee the stateful processing, the incoming packets are buffered and then offloaded to heterogeneous acceleration hardware. The processed data is finally resembled as traffic stream for further process. Such buffering-based approach requires a large amount of memory budget and may significantly increase the latency of traffics.

2) Aggregated Offloading Overheads

For discrete GPU platform, the data transfers among NIC, CPU memory, and GPU memory are one of the main overheads and have been discussed in prior work [7], [9], [17]. Several solutions such as shared buffer for NIC and GPU [7], analogous element for reduced memory copy [17], and partial offloading [9] have been proposed. We characterize three typical NFs under different offload ratios,

as shown in Figure 6. Our experiment results show that the best offloading ratios varies for different NFs.

Note that in the GPU acceleration of IPsec encryption, offloading all tasks to GPU does not yield the best performance. Figure 6 shows that offloading 70% of input packets to GPU while processing the rest packets on CPU can yield the best performance. This indicates that offloading all workloads to GPU can saturate the computing resource. The reason is that the un-optimized framework employs frequent small Click element kernel launch and teardown. This will incur extensive packet data loading and thread synchronizations. If we choose small batch size to mitigate the overheads of element branch, the number of kernel calls will be further increased and the kernel re-establish overheads could be aggravated.

Running NFs as a SFC can pose extra challenges to handle the aggregated offloading overheads. Identifying and configuring the best offload ratio for each NF could be troublesome, while a one-size-fits-all offload ratio may harm the SFC performance. We conduct an experiment to demonstrate how the acceleration benefit is offset along with the increase of SFC length. In this experiment, we test four cases, single IPsec (A), IPsec + IPv4 forwarding (B), Firewall + IPv4 forwarding + IPsec (C), IPv4 forwarding + IPsec + IDS (D). We report three offloading ratios: CPU only, GPU only, and 70% offload to GPU. As shown in Figure 7, the same offload ratio cannot always keep the consistent performance in different scenarios.

3) Findings

In this subsection, we can learn that both packet re-organization overheads and offloading overheads are caused by the complexity of Click elements and the redundancy of long service function chain. This motivates us to devise a framework that is able to reduce the complexity and redundancy of SFC at both NF granularity and element granularity.

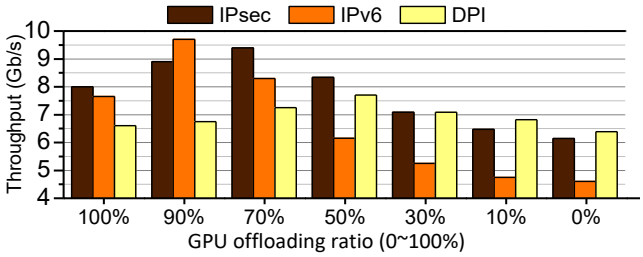

Figure 6. Performance variation by the fraction of offloading.

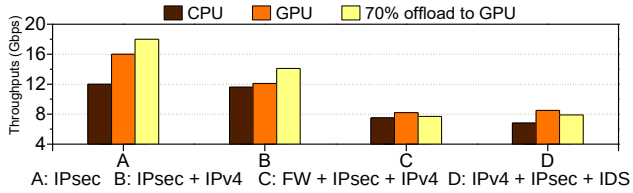

Figure 7. Acceleration is offset with the increase of SFC length.

C. *Impacts of Co-existence Interference*

To demonstrate the impacts of co-existence interference, we first explore the challenges of running single network function on GPU-based server. We investigate the performance and architectural behaviors under impacts of various *packet batch sizes*, *NF traffic patterns*, and *co-running traffic flows*. We draw several observations and thus motivating our design.

Batch Size: The throughput usually improves with the increase of batch size though it also relates to the packet size. The maximum throughput should be achieved by setting up a different batch size for different NFs. Large batch sizes are good for memory intensive NFs such as the IPv4 forwarder. Small batch size is better for compute intensive applications such as IDS and IPSec. The bigger batch size may lead to higher cache miss rate for CPU. In Figure 8 (d), we can observe that a CPU throughput drop occurs to DPI when the batch size is larger than 256 packets. Worse, increasing the

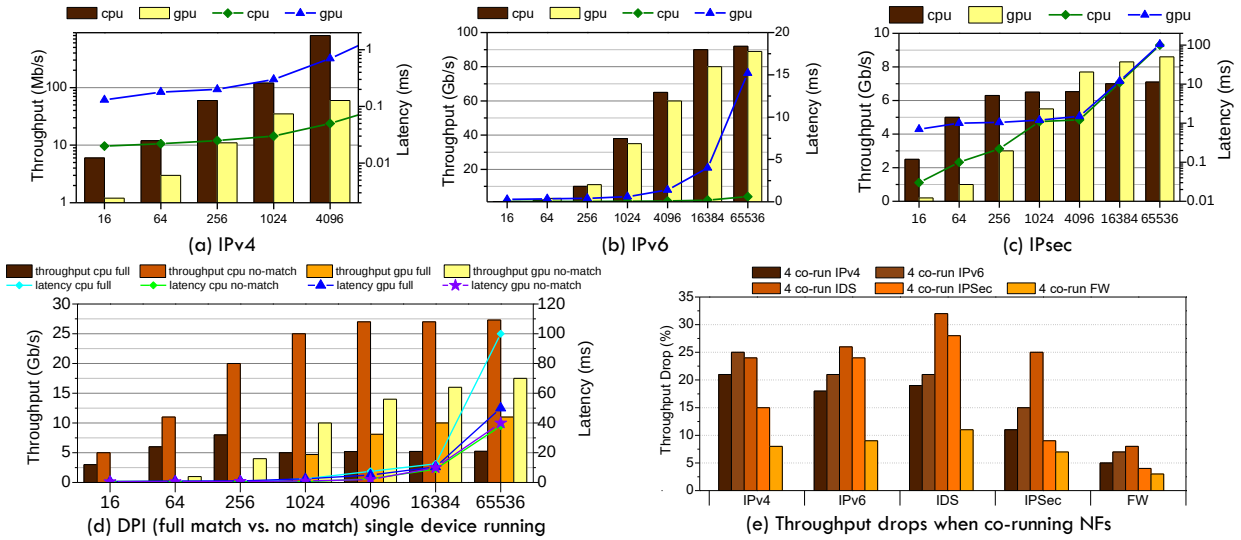

Figure 8. Characterization of network functions with various packet batch sizes, incoming traffic patterns, and co-running NFs; DPI has various traffic characteristics (full match v.s. no match);

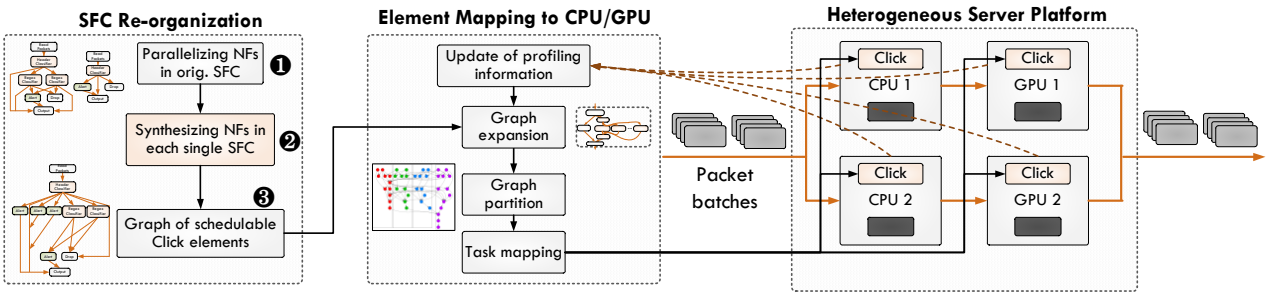

Figure 9. Flowchart of NFCompass runtime framework

batch size after this threshold will only lead to latency increase.

NF Traffic Patterns: The intrinsic traffic patterns also impact the throughput and latency of network functions. For example, the performance of DPI is strongly determined by the input traffic flows. As shown in Figure 8 (d) and (e), the CPU/GPU throughputs of no-match are significantly higher (4X~5X) than the throughputs of full-match. DPI/IDS rely on Aho-Corasick multi-string pattern matching algorithm. This algorithm scans the incoming packet payloads and looks up DFA table to match any string patterns. This memory intensive application can lead to at least five memory accesses. Hence, an incoming packet flow with low-match profile will reduce the cache and memory access and reduce the cache contention with other applications, while this will convert to an increased throughput.

Co-running Traffic Flows: We further investigate how co-running traffic flows affect the system throughputs. We measure the throughput drops of five typical NFs when another four NFs are co-running with them, as shown in Figure 8 (e). We can observe that IDS is the most exclusive application, with the highest average performance drop as 22.2%. In contrast, firewall is the least sensitive application when co-runs with other NFs. On CPU platform, the bottleneck of co-running NFs is the cache. If an NF causes a high cache hit number during the solo run, there is a high possibility that it will be suffered by the high throughput drop in the co-run. On GPU platform, the main bottleneck is that the co-run incurs frequent kernel launch and context switch. Therefore, the GPU acceleration for a complicated SFC with various NFs is not beneficial.

IV. Re-organizing the Service Function Chaining Processing Patterns

We propose NFCompass, a service function chain acceleration framework that targets heterogeneous COTS platform. NFCompass exploits two novel designs to guarantee the best latency for network traffics. First, NFCompass adopts a two-level SFC re-organization technology to reduce the redundancy and complexity of SFC. The aggregated packet processing overheads could be significantly mitigated by parallelizing the NFs at the SFC level and re-building the NF elements at element level. Second, NFCompass adopts a DAG-based task allocation scheme to ensure that all processing resources (CPU and GPU in our case) are balanced, the offloading overheads are minimized, and the overall packet processing latency is minimized. To avoid the issue of frequent kernel launch mentioned in Section 3, NFCompass employs a persistent GPU kernel design. The core idea is to keep a portion of GPU threads continuously running to process the input network packet stream.

TABLE II. NF Actions on Packet

	HDR/PL Rd	HDR/PL Write	Add/Rm bits	Drop
Probe	Y/N	N/N	N	N
IDS	Y/Y	N/N	N	Y
Firewall	Y/N	N/N	N	N
NAT	Y/N	Y/N	N	N
LB	Y/N	N/N	N	N
WAN Optimization	Y/Y	Y/Y	Y	Y
Proxy	Y/Y	N/Y	N	N

TABLE III. NF Parallelization Criteria

	HDR/PL Read	HDR/PL Write	Drop
HDR/PL Read	√	* √	√
HDR/PL Wrt	×	* √	√
Drop	N/A	N/A	√

The NF operations in the column are performed on the former NF in the SFC and the operations in the top row are performed on the later NF. Two parallelizable NFs are marked with "√".

A. System Overview

We present the system architecture in Figure 9. The NFCompass runtime consists of three critical components, the SFC orchestrator, the NF synthesizer, and task allocator. The core idea of NFCompass is to model the network processing elements in the service chains as a dataflow graph. Based on current data-flow graph, the SFC orchestrator analyzes the order-dependency of NFs in a SFC and examines if certain NFs could be processed in parallel. Then the NF synthesizer analyzes each parallelized SFC to remove the redundant elements and re-build the optimized NFs in each SFC. Finally the task allocator exploits a graph-partitioning based scheme to meet the various compute budget and real-time demands.

B. Exploring Parallelism and Composability of Service Function Chain

1) Exploring Parallelism at SFC level

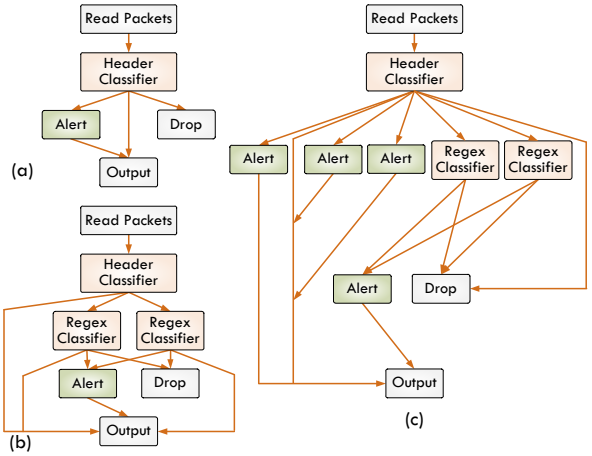

Figure 10. A sample of synthesizing new NFs.

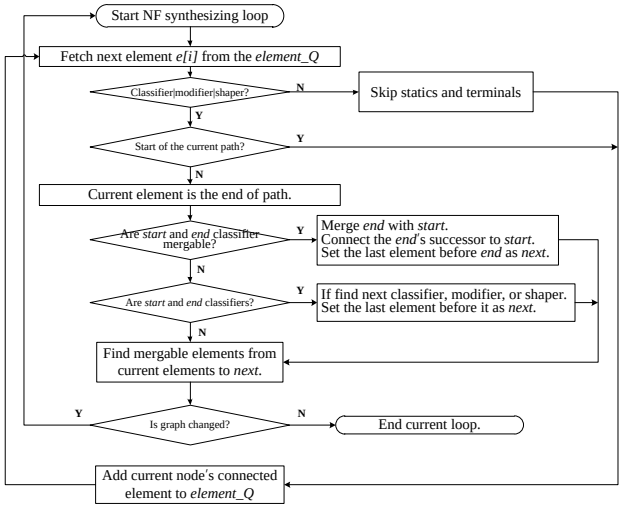

Figure 11. The decision-making workflow of merging.

We have shown that the increasing length of SFC can significantly increase the traffic latency in Section 3. An intuitive approach is to explore the parallelism of packet processing of SFC. We define two NFs are parallelizable if they do not have dependency with each other. Network traffics could be duplicated and input to parallelizable NFs for processing. For example, whether a packet is processed by IDS system or WAN proxy does not affect the output functional correctness of the other NF. So IDS and WANproxy are parallelizable.

Parallelizing network functions calls for the packet action independency between NFs. To evaluate the dependency of NFs in a SFC, we consider following criteria. First, the read and write operations of different NFs can impact the dependency of them. Normally the packet header or payload could be overwritten by some NFs. For example, the Network Address Translation (NAT) always changes the packet header. Second, a packet drop in an NF can also affect the correctness or efficiency of the SFC. If a packet could be dropped by a former NF, the operation based on that packet could be emitted on the latter NF. We investigate the packet actions of most typical NFs and list them in Table 2.

Similar to the hazards in the instruction pipeline, different NFs can have four situations in which read or write operations are conducted sequentially. Read after read, read after write, write after read, and write after write. NFs with RAW and WAW dependency cannot be parallelized to keep the packet data coherency, while the RAR and WAR cases are safely parallelized. We show the criteria of whether two consequential NFs can be parallelized in Table 3. Note that in the WAW and WAR cases, we need to further locate the changed fields in header and payloads. There exist certain cases in which only header or payloads are changed while the payloads or header of this packet will be read or written in the latter NF. Under such circumstance, the WAW and WAR cases are safely to be parallelized.

After analyzing the SFC order dependency, the SFC orchestrator needs to duplicate input traffics to several parallelizable NFs. It just creates the copy of network packets and distributes them. The merge function is designed base on classical exclusive or logic. The xor operation is leveraged to identify the difference of processed packets from parallel NFs. The original packet will be xor-ed to each output packet to get the modified bits. All the modified results from each NF will be or-ed to generate an aggregated modified result. This aggregated result will be xor-ed with the original packet again to get the final output of parallel NFs.

2) Reducing the Redundancy at Element Level

Having solved the long chain issue at SFC level, we continue to explore within the NFs and would like to further eliminate the redundancy at the NF element level. Many NFs adopt similar steps to process the packets [30]. We summarize following source of redundancy in existing Click module based NF. First, multiple NFs in a SFC incur multiple network I/O operations. It would be beneficial to combine the elements in multiple NFs into a synthesized NF. Second, a packet drop occurs at the later elements may unnecessarily consume computing resource. Third, some general elements may be used multiple times in each NF, such as IP lookup element that reads IP address. Combining the NFs at element level helps to control the redundancy of these kinds of elements by placing it at former stage and only using it once. Fourth, some elements overwrite the same field of a packet.

We design our processing graph engine using a directed acyclic graph (DAG) based scheme. The goal is to eliminate the four sources of redundancy. As shown in Figure 10, a simple SFC consists of a fire wall and an IDS system. In this case the redundant header classifier element is desired to remove and two graphs could be merged into a synthesized graph. The principles to merge redundant graph are listed as follows. First, the packet processing path must not be modified in the merged graph compared to in the individual graph. Second, the stateful processing must be guaranteed for each packet. The alert or log operation should be executed in the same packet state compared to in the individual graph. Our NF synthesizing algorithm makes the rule for changing

the order of elements and removing elements according to each element's traffic class. For example, to keep the correctness of classification, the classifiers are not allowed to move across modifiers or shapers. Based on the original NF connection order, the NF synthesizer first parses the Click module DAG of each NF and generates a processing tree by concatenating them together. The depth from root to leaf equals to the length of processing path in the NF. The synthesizer then re-orders and de-duplicates elements. The decision making flow is shown in Figure 11.

C. Graph Partitioning-based Task Allocation

Our initial input workloads graph is the synthesized Click element graph. An example is shown in Figure 10 (c). The task allocator runtime is responsible for mapping the SFC element dataflow graph to CPU-GPU heterogeneous platform. The mapping runtime aims to maximize the system throughput while minimizing the data movement among CPUs and GPUs. Existing works [31][32] employ the queue-based scheduling method which fails to leverage the global dataflow dependency. We choose a graph partitioning based algorithm to solve this issue in this work.

1) Fine-grained Element Graph Generation

The original element graph generated by our SFC reorganization technique only includes the essential Click elements in the SFC. Note that the offloadable Click elements could be either executed on CPU or GPU or both as discussed in Section 3.2. Therefore, it is hard to assign the appropriate weight to a single element to sufficiently represent the offloading scenarios with various load-balance ratios. For example, the cases of no offloading, full offloading, and different offloading ratios present different weights for an element with certain function. Moreover, the combinations of different offloading configurations can further complicate the problem size. These uncertainties challenge the graph partitioning stage since it is impossible to generate all partition results for each offloading ratio and different offloading combinations.

To accommodate the number and construction of Click elements graph to the graph partitioning phase, we employ a fine-grained element expansion scheme for the offloadable elements, as shown in Figure 12. Our goal is to incorporate full potential offloading configurations in the graph. The key idea is to create virtual instances of real element, where each virtual instance represents a portion of offloaded task (offload ratio increases as δ =10% in our design) or CPU-side task of original element so that the following partitioning phase can directly partition elements to different processors. Note that this method results in a dependent task allocation weights for CPU-side element. The weights must be accommodated correspondingly in the graph partition stage.

To conquer the challenge of packet reordering as discussed in Section 3.2.1, we leverage the design of `GPUCompletionQueue` element in Snap [17] that only releases a batch until all packets in this batch are processed.

2) Runtime Profiling

To build accurate graph for Click element allocation, the system characteristics are needed as the weights of the

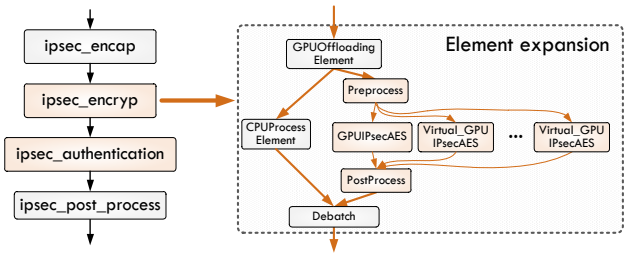

Figure 12. Fine-grained element expansion for graph partition.

original element graph, which only shows dependency and connectivity information. As we studied in Section 3, the various processing capability of heterogeneous processors and the network traffic characteristics synergistically impact the performance of SFC deployment. In particular, the fast-changing network services can generate diverse network traffic types and processing requirements with different data-dependency. So we choose a run-time plus offline profiling to collect two types of system operating information: the traffic-related statistics and the performance-related statistics. The traffic-related statistics depict the network traffic intensity and distribution among current Click element graph. We measure this by sampling the next element destination of packets at each element over time (1000/10K packets in our case). By collecting the packet flow distribution on each edge, we can obtain the time-dependent traffic intensities on each edge, and the utilization of each element. The offline profiling collects the processing rates (packets/second) of all Click elements on CPU and GPU under various input traffic intensities (packet-per-second: 1K~14M, increased by 10K; packet sizes: 64B~1500B, increased by 64B). It also collects the data transfer overheads for various traffics. The traffic intensity information will be factored in the processing time on each element and be used as the weights for graph partitioning. NFCompass uses a dictionary to store the profiling information and indexed by vertex ID and edge ID. This dictionary provides performance guides to scheduler.

3) Mapping Elements to Heterogeneous Processors

Assigning the packet processing elements to appropriate processing units (CPU or GPU) while preserving the system throughput is NP hard. We employ two graph partition-based algorithms to find the best tradeoff between practicality and accuracy. The Max-Flow/Min-Cut (MFMC) is widely used to model flow-based clustering problems [33] to find the graph partitions with the least inter-cluster communication costs. This feature intuitively complies with our task mapping goals of maximizing resource utilization and throughput while minimizing communication costs. Our first graph partitioning algorithm is implemented as a modified Kernighan-Lin (KL) Algorithm [34] using METIS [35]. The core idea of our algorithm is: Given an initial graph $G = G_1 + G_2$, where G_1 and G_2 are initial partitioned graphs. The algorithm iteratively swaps X, and Y, which are two subsets of elements that belong to G_1, and G_2, respectively, and then examines the gain function determined by the removed edges and balanced tasks between two graphs.

We also design a light-weight and highly scalable naïve graph partitioning scheme to cope with the challenge when

the complexity of system increases, such as extreme diverse traffics and complicated SFCs are presented. This algorithm is a seed-based agglomerative node clustering. It starts with single element graphs with seed elements. In our design we select a random GPU element and a CPU element in each SFC as the seed vertices for two initial graphs. With *n* SFCs we have *2n* initial graphs. The algorithm then merges two graphs at each step by choosing two vertices with lowest communication overheads. The complexity of this algorithm is *O*(*klogk*), where *k* is the edge number of global graph. This light-weight partition may result in unbalanced throughput on different processing units. We still need to apply the dynamic task adaption.

V. EVALUATION

In this section, we evaluate NFCompass by breaking down the performance improvements (throughputs and latency) gained by our SFC re-organization technique and graph-partition based task allocation technique. We then demonstrate the overall performance gain brought by NFCompass compared to baseline CPU-only and current GPU-based baselines.

A. Evaluation Setup

Our experimental platform is configured as described in Section 3. We implement NFs as Docker containers [33] and run them on dedicated CPU core. We also install Nvidia driver [36] to enable the GPU-offload support. To effectively evaluate NFCompass, we choose a DPDK-based packet generator that runs on a client server. It produces up to 40 Gbps network traffic that contains packets with various types of lengths, such as fixed lengths, uniform random lengths, and lengths that follow a specific network traffic distribution pattern.

B. Effectivenss of SFC Re-organizing Technology

We first demonstrate the effectiveness of SFC re-organization technology (SFC parallelization and SFC merging) by evaluating the throughput improvement and latency reduction on both CPU and GPU platform. Towards this goal, we explore the performance of SFCs under various complexity and parallelism configurations on both traditional CPU-only platform and GPU-only platform. We disable our graph-partition based task allocation in this part.

1) Effectiveness of SFC Paralleling Technique

We first evaluate effectiveness of our SFC parallelization technique. Towards this goal, we compose three simple sequential service function chains with different NF types respectively (firewall, IPSec, or IDS). Each SFC consists of four identical NF, as shown in Figure 13 (a). By applying SFC parallelization, the sequential SFC could be re-organized into two SFC structures with different parallelism degrees, as shown in Figure 13 (b) and (c). We can observe that the effective length of SFC configuration (*a*) is 4, while the length of configuration (*b*) and (*c*) are reduced to 1 and 2 respectively. Note that the IDS and IPSec have higher complexity than firewall since they involve complicated forwarding rules and intensive computation of pattern matching.

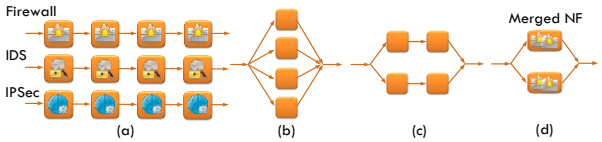

Figure 13. A sample of synthesizing new NFs.

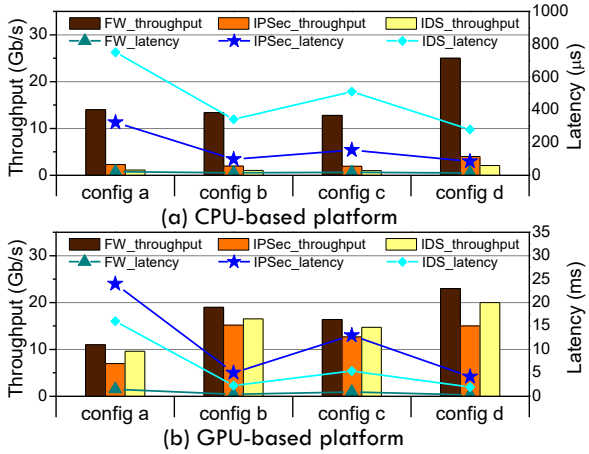

Figure 14. A sample of synthesizing new NFs.

We choose TCP stream with 64B packet as test load for the SFCs and evaluate the throughput and latency. To obtain the accurate throughput, the rules of firewall are modified to never drop packets. We measure the average packet traveling time from its arrival to test machine NIC until its departure from the out NIC.

Evaluation results shown in Figure 14 demonstrate several key benefits. First, our SFC parallelization technique can effectively reduce the packet latency for both CPU-based and GPU-based platforms, and maintain the system throughput in a reasonable range (with less than 10% throughput reduction in all configurations).

Second, for NFs with higher computation complexity such as IDS and IPSec, the SFC parallelization can bring more significant latency reduction. For example, the maximum latency reduction for a simple NF such as firewall is 24% on CPU. In contrast, for complicated NF such as IDS who involves intensive pattern matching computation, the maximum latency reduction is 54% on CPU platform.

Moreover, SFC parallelization benefits GPU-based NFs more significantly than CPU-based NFs in terms of latency. We can observe that the maximum SFC latency of configuration *a* on GPU-only platform is around 24 ms, while it is reduced to 5 ms in configuration *b*. The biggest improvement is 79% while the biggest improvement on CPU is 54%. We can also note that the throughputs of IPSec and IDS gained by GPU-only platform are considerably higher than the throughputs of CPU-based platform.

2) Effectiveness of NF Synthesizing Technique

We then evaluate the effectiveness of our NF synthesizing/merging technique. We test this case by applying the NF merging to SFC configuration *c*, and obtain the synthesized SFC structure as shown in Figure 13 (d). With NF synthesizing technique, the two pipelined NFs are merged into a single NF. According to the throughput and

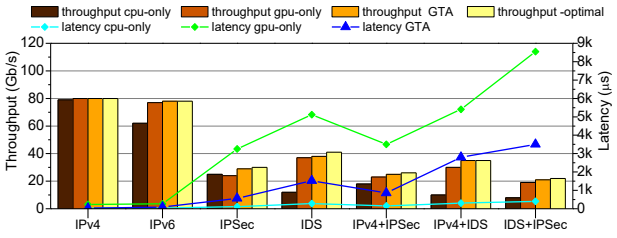

Figure 15. Throughputs comparison between GTA, CPU-only, GPU-only and Optimal under various combinations of network functions and packet sizes.

latency results shown in Figure 14, we have following observations.

First, we can observe that the NF synthesizing can achieve nearly the similar throughput and latency compared to SFC parallelization. This could be derived by comparing configuration *b* and *d*. The equivalent lengths of SFC are all 1 in these two scenarios.

Second, we can note that the latency in configuration *d* is lower than configuration *b* by 12%~18% on CPU and by 14%~30% on GPU. This could be resulted by the inefficient SFC branching operations (packet copying at the start of SFC branch and packet merging at the end of SFC branch). The latency benefit brought by the short SFC is offset by the packet copying/merging. Designing an optimized packet and memory management scheme will be our future work.

Third, we can observe that the configuration *d* achieves higher throughput than configuration *b* on both CPU (86%~100%) and GPU based platforms (13%~21%).

To sum up, the NF synthesizing technique is well complement to the SFC parallelization technique. A joint application of these two techniques such as in configuration *d* can result in the best performance for SFCs.

C. *Effectiveness of Graph-based Task Allocation*

We demonstrate that how NFCompass handles the task balance between CPU and GPU under various NF setups. Figure 15 reports the throughputs and the corresponding latency of Graph-based Task Allocation scheme (GTA), compared with the CPU-only, GPU-only cases and optimal offloading fractions obtained by manually exhaustive searches. We use IPv4, IPv6, IPSec, IDS, and their combinations as our test NFs.

We use the DPDK-based traffic generator and choose IMIX (Internet Packet Mix) that resembles the real-world traffic in the distribution of packet lengths and defined by Intel. It consists of 61.22% of 64B packets, 23.47% of 536-byte packets, and 15.31% of 1360-byte packets [37].

We can note that GTA can achieve more than 90% of the maximum possible throughput in all scenarios, and maintain the latency lower than 4ms. Another key finding is that GTA gains higher throughput than both CPU-only and GPU-only for all NF setups except for IPv4. Note that the latency of GTA for IPv4 is equal to the latency of CPU-only. This indicates that GTA does not offload tasks to GPU at all for IPv4.

We also observe that GTA achieves better performance improvement for SFC setups than for single NFs. We define a metric as (GTA_throughput - best-effort_throughput)/ best-effort_throughput, where best-effort_throughput indicates the better performance improvement gained by CPU or GPU. We can note that the average performance gain is 5% for single NF. It increases to 16% for SFCs. This indicates that GTA can handle the complicated NF tasks well.

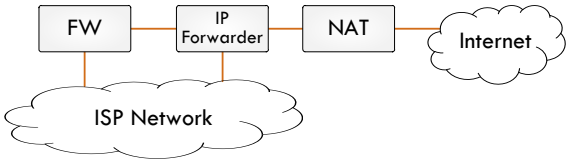

Figure 16. Test service function chain.

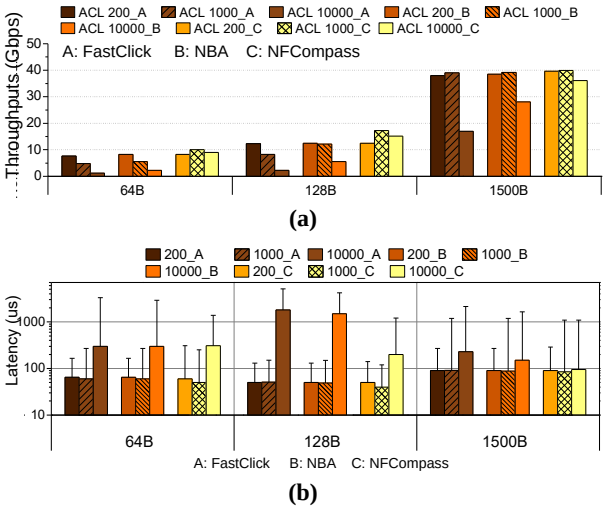

Figure 17. Performance comparison between NFCompass and baselines

D. *Validation of NFCompass Using Real Service Function Chain*

In this part we validate the effectiveness of NFCompass using a representative service function chain.

The processing network function chain in the evaluation is shown in Figure 16. It consists of a firewall (FW), a router, and a NAT. The FW may contain several thousands of rules in the Access Control List (ACL) which may consume significant computer resource. We use three real ACLs [38] in our FW. We generate ACL containing 200, 1000, and 10000 rules. The IP router forwards packets to ISP's domain or Internet. For traffics to Internet, the NAT network function performs source and destination NAT. We use three different packet sizes in our test traffics (64B, 128B, 1500B). We choose FastClick [39] and NBA [9] as our baseline systems.

Figure 17 shows that when executing as a single FastClick instance, the small ACL achieves fairly good performance. All three test systems achieve nearly the same performance. The packet latencies are also well controlled within 100us. However, when we look into bigger ACLs with 1000 and 10000 rules, the classification tree becomes huge since the traffic complexity among FW, router and NAT. At this time, the FastClick comes across severe

performance issue. The throughput drops about 38% to 84% for 1000 rules and 10000 rules respectively. Its latency on ACL 10000 is more than an order of magnitude higher than ACL 200 latency. The NBA also does not perform well in the large ACL scenarios. The throughput drops about 32% to 73% for 1000 rules and 10000 rules respectively. The latency on ACL 10000 is 6.7X higher than ACL 200 latency.

Thanks to our NF synthesizing technique, NFCompass maintains its high throughput and low latency. In ACL 1000 case, the throughput is nearly same to the ACL 200 throughput. NFCompass also achieves much better latency than its counterparts. It performs about 1.4X to 9X lower average latency and 2.9X to 4.3X lower variance latency.

VI. Related Work

GPUs and dataflow graph: PTask [32] proposes a GPU based task mapping framework for complex dataflow graphs. However, the packet processing system cannot directly leverage its Unix pipe-based data processing model. The input data is processed by its processing element and is converted into new streams of output data. In contrast, the packet data is processed by a series of processing elements as a pipeline, where the data modification, annotation, and drop are conducted. NFCompass provides the new design insights for GPU offloadable task allocation using light-weight graph partition. Flextream [40] is a compilation framework for the SDF model that dynamically adapts applications to target architectures in the face of changing availability of FPGA, GPU, or CPU resources.

Packet processing acceleration: GPUs provide a substantial performance boost to many network-related workloads. PacketShader [8] demonstrates the feasibility of 40 Gbps on a single PC with optimized packet I/O processing and GPU offloading. MIDeA [41], SSLShader [15], and Kargus [14] all exploit GPU to accelerate network applications, such as SSL (Secure Sockets Layer). Snap [17] proposes a packet batch processing framework for GPU. GASPP [7] proposes a stateful approach to GPU-oriented packet processing. GPUnet [42] is a socket abstraction for GPUs, which allows GPU programs to control RX/TX with remote machines. Snap [17] is a packet processing framework that adds a set of extensions to Click to integrate GPU elements. APUNet [43] and PIPSEA [44] propose to accelerate packet processing such as IP forwarder and IPSec on APU. Furthermore, [45] and [46] propose the dynamic core allocation and task mapping schemes for network traffics based on network processors. NFCompass is different from prior works since we use a fine-grained task partition scheme for task offloading. We treat CPU and GPU equally and only offload tasks to GPU when it is beneficial to the performance.

Load balancing system for heterogeneous platform: Qilin [19] employs an offline training and dynamic compilation at run time to fulfill the adaptive workload mapping. It provides an API that is compatible for both CPU and GPU. PetaBricks [47], an implicitly parallel language and compiler, uses an empirical auto-tuning approach to search the space of possible implementations at installation time to construct poly-algorithms that combine many different algorithmic techniques to obtain better performance. StarPU [31] is a task scheduling framework for heterogeneous systems. It uses heterogeneous earliest finish time (HEFT) scheduling algorithm and automatically calibrates its performance model by observing task completion times. The disadvantage of these approaches is that they have been designed for applications that take as input constant streaming data and as a consequence, they adapt very slowly when the input data stream varies. Our system targets a very different workload, network traffic, which stresses not only CPU, but also I/O, depending on the traffic composition. Koromilas et al. [16] and NBA [9] tackle scheduling problem of network packet processing workloads on GPUs. Differently from above work, our graph-based algorithm can better handle the systems with complex interdependencies in the network function service chains.

NFV acceleration and network function optimization: A myriad of prior work addresses the NFV acceleration in terms of single NF acceleration [11], [13], [48] and NFV acceleration framework [49]–[56]. ParaBox [57] proposes initial NFV service chain parallelization technique and NFP [58] elaborates this technique in the same time with our work. NFCompass augments these contributions by proposing a comprehensive SFC framework for heterogeneous platform.

VII. Conclusions

In this paper, we study the typical software based network functions that are widely used in modern Network Function Virtualization environment. In particular, we characterize the running of service function chains on modern GPU-accelerated COTS server platform to identify the performance bottlenecks. Our characterization experiences show that the performance of SFC is mainly restricted by two root causes, the aggregated processing overheads, and the co-existence interference overheads. To tackle these two issues, we propose NFCompass, a runtime support for high-performance network function service chaining on heterogeneous COTS server platform. NFCompass innovatively leverages a two-level SFC re-organization technique and a graph partition-based scheduling scheme to conquer the above two overheads respectively. Our experimental results show that NFCompass can achieve 60% more throughput improvement, and about 1.4X to 9X lower than state-of-the-art packet processing framework when running real world service function chain.

Acknowledgment

We thank all the anonymous reviewers for the invaluable and insightful comments to make this paper better. This work is supported in part by NSF grants 1527535, 1423090, 1320100, 1117261, 0937869, 0916384, 0845721 (CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multicore Computing Awards, and by three IBM Faculty Awards.

References

[1] C. Cui, H. Deng, D. Telekom, U. Michel, and H. Damker, “Network functions virtualisation: An introduction, benefits, enablers,

challenges and call for action," Netw. Funct. Virtualisation – Introd. White Pap., no. 1, pp. 1–16, 2012.

[2] ETSI ISG NFV, "Network Functions Virtualisation (NFV): Architectural Framework," 2013.

[3] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network Virtualization in Multi-tenant Datacenters," Proc. 11th USENIX Symp. Networked Syst. Des. Implement. (NSDI 14), pp. 203–216, 2014.

[4] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker, "SoftFlow: A Middlebox Architecture for Open vSwitch," in 2016 USENIX Annual Technical Conference (USENIX ATC 16), 2016, pp. 15–28.

[5] Cavium, "Cavium Networks OCTEON II processors." [Online]. Available: http://cavium.com/.

[6] Mellanox Technologies, "NPU & Multicore Processors." [Online]. Available: http://www.mellanox.com/page/npu_multicore_overview.

[7] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-accelerated stateful packet processing framework," USENIX Annu. Tech. Conf., pp. 321–332, 2014.

[8] S. Han and K. Park, "PacketShader : A GPU-Accelerated Software Router," Internetworking Res. Exp., vol. 40, no. 4, pp. 195–206, 2010.

[9] J. Kim, K. Jang, K. Lee, S. Ma, K. Shim, and S. Moon, "NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors," Eur. Conf. Comput. Syst., 2015.

[10] S. Kim, S. Huh, Y. Hu, X. Zhang, A. Wated, M. Silberstein, and T. Israel, "GPUnet : Networking Abstractions for GPU Programs This paper is included in the Proceedings of the Operating Systems Design and Implementation . GPUnet : Networking Abstractions for GPU Programs," 2014.

[11] B. Li, K. Tan, L. (Larry) Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware," in Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference, 2016, pp. 1–14.

[12] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat, "Chimpp: A Click-based Programming and Simulation Environment for Reconfigurable Networking Hardware," in Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2010, p. 36:1--36:10.

[13] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable Router Architecture for Experimental Research," in Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow, 2008, pp. 1–7.

[14] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: A Highly-scalable Software-based Intrusion Detection System," in Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012, pp. 317–328.

[15] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors," in Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, 2011, pp. 1–14.

[16] L. Koromilas, "Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures," Proc. tenth ACM/IEEE Symp. Archit. Netw. Commun. Syst., pp. 207–218, 2014.

[17] W. Sun and R. Ricci, "Fast and flexible: Parallel packet processing with GPUs and click," ANCS 2013 - Proc. 9th ACM/IEEE Symp. Archit. Netw. Commun. Syst., pp. 25–35, 2013.

[18] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability," in Proceedings of the ACM International Conference on Computing Frontiers, 2013, p. 21:1--21:10.

[19] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009, pp. 45–55.

[20] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs," in Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, 2011, pp. 277–288.

[21] D. Intel, "Data Plane Development Kit," URL http//dpdk. org.

[22] L. Rizzo, "NetMap: A novel framework for fast packet I/ O," in USENIX'12, 2012, pp. 101--112.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," ACM Trans. Comput. Syst., vol. 18, no. 3, pp. 263–297, Aug. 2000.

[24] J. Jeffers and J. Reinders, Intel Xeon Phi coprocessor high-performance programming. Newnes, 2013.

[25] "Snort Intrusion Detection System." [Online]. Available: https://snort.org.

[26] "Configuring nDPI for custom protocol detection." [Online]. Available: http://www.ntop.org/ndpi/configuring-ndpi-forcustom-.

[27] "ShoreWall iptable." [Online]. Available: http://www.shorewall.org/.

[28] 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.

[29] R. Jones, "NetPerf: a network performance benchmark," Inf. Networks Div. Hewlett-Packard Co., 1996.

[30] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in Proceedings of the 2016 ACM SIGCOMM Conference, 2016, pp. 511–524.

[31] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," Concurr. Comput. Pr. Exper., vol. 23, no. 2, pp. 187–198, Feb. 2011.

[32] C. J. Rossbach, J. Currey, and M. Silberstein, "PTask: Operating System Abstractions To Manage GPUs as Compute Devices," ACM Symp. Oper. Syst. Princ., pp. 1–16, 2011.

[33] Y. Hu, M. Song, and T. Li, "Towards 'Full Containerization' in Containerized Network Function Virtualization," in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 467–481.

[34] B. W. Kerninghan and S. Lin, "An efficient heuristic procedure for partitioning graphs," Bell Syst. Tech. J., vol. 49, no. 2, pp. 291–307, 1970.

[35] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," SIAM J. Sci. Comput., vol. 20, no. 1, pp. 359–392, Dec. 1998.

[36] Nvidia, "Docker Engine Utility for NVIDIA GPUs." [Online]. Available: https://github.com/NVIDIA/nvidia-docker.

[37] A. Hoban, "Using Intel AES New Instructions and PCLMULQDQ to Significantly Improve IPSec Performance on Linux," 2010.

[38] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," IEEE/ACM Trans. Netw., vol. 15, no. 3, pp. 499–511, Jun. 2007.

[39] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing," in Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2015, pp. 5–16.

[40] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," Parallel Archit. Compil. Tech. - Conf. Proceedings, PACT, pp. 214–223, 2009.

[41] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: A Multi-parallel Intrusion Detection Architecture," in Proceedings of the 18th

ACM Conference on Computer and Communications Security, 2011, pp. 297–308.

[42] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, “GPUnet: Networking Abstractions for GPU Programs,” in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 201–216.

[43] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, K. Park, and M. Jamshed, “APUNet: Revitalizing GPU as Packet Processing Accelerator,” Nsdi, pp. 83–96, 2017.

[44] J. Park, W. Jung, G. Jo, I. Lee, and J. Lee, “PIPSEA: A Practical IPsec Gateway on Embedded APUs,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 1255–1267.

[45] M. F. Iqbal, J. Holt, J. H. Ryoo, G. De Veciana, and L. K. John, “Dynamic Core Allocation and Packet Scheduling in Multicore Network Processors,” IEEE Trans. Comput., vol. 65, no. 12, pp. 3646–3660, 2016.

[46] Q. Wu and T. Wolf, “Runtime task allocation in multicore packet processing systems,” IEEE Trans. Parallel Distrib. Syst., vol. 23, no. 10, pp. 1934–1943, 2012.

[47] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, “Portable Performance on Heterogeneous Architectures,” SIGPLAN Not., vol. 48, no. 4, pp. 431–444, Mar. 2013.

[48] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, “NetBricks: Taking the V out of {NFV},” in 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 203–216.

[49] J. Hwang, K. K. Ramakrishnan, and T. Wood, “NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms,” Proc. 11th USENIX Symp. Networked Syst. Des. Implement. (NSDI 14), vol. 12, no. 1, pp. 445–458, 2014.

[50] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, and G. Todeschi, “OpenNetVM : A Platform for High Performance Network Service Chains,” pp. 26–31, 2016.

[51] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Huici, and B. Felipe, “ClickOS and the Art of Network Function Virtualization,” 11th USENIX Symp. Networked Syst. Des. Implement. (NSDI 14), pp. 459--473, 2014.

[52] Y. Hu and T. Li, “Towards Efficient Server Architecture for Virtualized Network Function Deployment: Implications and Implementations,” in Proceedings of the 49th International Symposium on Microarchitecture - MICRO-49, 2016.

[53] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, “E2: A Framework for NFV Applications,” in Proceedings of the 25th Symposium on Operating Systems Principles, 2015, pp. 121–136.

[54] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, “Design and Implementation of a Consolidated Middlebox Architecture,” in Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), 2012, pp. 323–336.

[55] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, “IX: A Protected Dataplane Operating System for High Throughput and Low Latency,” Proc. 11th USENIX Conf. Oper. Syst. Des. Implement., pp. 49–65, 2014.

[56] S. Peter, T. Anderson, and T. Roscoe, “Arrakis: The Operating System as Control Plane,” Proc. 11th USENIX Conf. Oper. Syst. Des. Implement., vol. 38, no. 4, pp. 44–47, 2014.

[57] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, “ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining,” in Proceedings of the Symposium on SDN Research, 2017, pp. 143–149.

[58] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, “NFP: Enabling Network Function Parallelism in NFV,” in Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2017, pp. 43–56.