FlowShader: a Generalized Framework for GPU-accelerated VNF Flow Processing

Xiaodong Yi*, Junjie Wang*, Jingpu Duan†, Wei Bai‡, Chuan Wu*, Yongqiang Xiong‡, Dongsu Han§
*The University of Hong Kong,†Southern University of Science and Technology
Email: *{xdyi, jjwang2,cwu}@cs.hku.hk,†duanjp@sustc.edu.cn
‡Microsoft Research,§Korea Advanced Institute of Science and Technology
Email: ‡baiwei0427@gmail.com,‡yongqiang.xiong@microsoft.com,§dongsu.han@gmail.com

***Abstract*—GPU acceleration has been widely investigated for packet processing in virtual network functions (NFs), but not for L7 flow-processing NFs. In L7 NFs, reassembled TCP messages of the same flow should be processed in order in the same processing thread, and the uneven sizes among flows pose a major challenge for full realization of GPU's parallel computation power.**

To exploit GPUs for L7 NF processing, this paper presents *FlowShader*, a GPU acceleration framework to achieve both high generality and throughput even under skewed flow size distributions. We carefully design an efficient scheduling algorithm that fully exploits available GPU and CPU capacities; in particular, we dispatch large flows which seriously break up the size balance to CPU and the rest of flows to GPU. Furthermore, *FlowShader* allows similar NF logic (as CPU-based NFs) to run on individual threads in a GPU, which is more generalized and easy to take on as compared to redesigning an NF for operation parallelism on GPU. We implemented a number of L7 flow processing NFs based on *FlowShader*. Evaluations are conducted under both synthetic and real-world traffic traces and results show that the throughput achieved by *FlowShader* is up to 6x that of the CPU-only baseline and 3x of the GPU-only design.

I. Introduction

Virtual network functions (VNFs) have been advocated for processing network traffic in many domains (e.g., telecom's networks, the WAN) [1], [2], due to their deployment agility and low management costs. Many NFs deployed over the WAN perform L7 flow-processing , such as stream-processing Intrusion Detection/Prevention System (IDS/IPS) [3]–[6], L7 load balancer [7], stateful firewall [8]–[10], Web Security Gateway (WSG) [11], [12], and Web Application Firewall (WAF) [13], [14]. With these NFs, messages of the same flow must be processed in order, to guarantee the correctness of the processing logic; after processing each message, states associated with the flow may be updated, in which case the change must be reflected before subsequent messages are processed. For example, after a state machine-based IDS analyzes one message in a flow, a state is updated and the new state must be used when processing the next message in this flow. Similarly, a WAF may record the frequency of each keyword in a flow, which must be updated after processing each message.

To enhance the VNF throughput, advanced, general-purpose hardware devices have been exploited, such as FPGA [15] and GPU [16]. In this paper, we focus on GPU acceleration because GPUs are more widely available [17]–[19] and easier to program (more similar to CPU programming) than FPGAs. With thousands of parallel processing units, GPUs are well known for high computation capacity and have been shown to significantly improve packet processing speeds as compared to using CPUs [16], [20]–[23].

We propose *FlowShader*, a GPU acceleration framework for L7 VNFs that maximizes flow processing throughput by fully exploiting available GPU and CPU capacities. *FlowShader* makes the following design choices which combined represent a unique solution in the design space of hardware-accelerated VNFs.

We use GPU to accelerate L7 flow-processing NFs for WAN traffic. Most existing work studies GPU acceleration of packet-processing, stateless NFs, where packets can be processed in any order and no flow states are maintained [16], [20], [21]. L7 NFs pose important additional requirements on GPU-accelerated NF design: 1) the processing unit for most L7 NFs is reassembled message rather than an individual packet; 2) in each batch of messages to be fed into a GPU for parallel processing, messages belonging to the same flow must be sent to the same GPU thread(s) for serial processing; and 3) a new GPU kernel for processing a flow can only be launched after a previous kernel on the same flow is completed and flow states are updated.

We pursue a general, flow-parallel acceleration framework. Existing work on GPU-accelerated flow processing (e.g., SSLShader [22]) mostly adopts a strong scaling approach. They design a specific GPU parallel execution algorithm to achieve operation-level parallelism within the NF logic and use multiple GPU threads to run different parallel operations for processing one flow. Adopting such a strong scaling approach lays a high customization requirement for migrating NFs from CPU to GPU, as careful redesign of the code for operation parallelism is necessary for each NF. Instead, we advocate flow-level parallelism in which similar NF logic (as that running on a CPU) is executed on the GPU, and each GPU thread runs the entire processing logic of an NF to sequentially process

This work was supported in part by MSRA Collaborative Research Grant, grants from Hong Kong RGC under the contracts HKU 17204715, 17225516, C7036-15G (CRF), and the project "PCL Future Regional Network Facilities for Large-scale Experiments and Applications (PCL2018KP001)".

978-1-7281-2700-2/19/$31.00 2019 © IEEE

messages within a flow. This weak scaling approach makes it much easier to migrate various existing NFs to GPUs.

▷ *FlowShader* provides a generalized acceleration architecture for L7 flow-processing NFs. With flow-level parallelism on GPUs, developers do not need to redesign custom GPU-side NF logic for different NFs, but can directly port any CPU-side NF processing logic to the GPU side with minor modifications.

▷ A set of APIs are provided to further ease the job of developers on NF implementation, enabling a clear separation of architecture design and NF implementation. Using the APIs, developers only need to implement the core processing logic of the NFs, without handling when and where the NFs should be invoked in the system.

We resolve the flow data size imbalance issue to maximize GPU efficiency. An important issue may result due to flow-level parallelism. Specifically, the size of messages belonging to different flows in the same batch, to be processed by different GPU threads, can be significantly unbalanced. This can significantly undermine the parallel processing power of a GPU. This issue has not been posed and not resolved in existing studies (e.g., GASPP [23]).

▷ The core of *FlowShader* is a carefully designed flow scheduling algorithm to maximally utilize available capacities of CPUs and GPUs for throughput maximization. It efficiently dispatches flows of different data sizes to different CPU cores and the GPU in real time, based on detailed system modeling and periodical processing time prediction. The idea is to dispatch large flows which seriously break up the size balance to CPU and the rest of flows to GPU, in order to maximize GPU parallelism and minimize CPU waiting time.

In summary, *FlowShader* is developer friendly and ensures high flow-processing performance. We implemented a number of L7 NFs based on *FlowShader*: an IDS, a L7 load balancer, a WSG and a WAF. Evaluation results using both synthetic and real-world traffic traces show that *FlowShader* achieves up to 6x throughput as compared to the CPU-only baseline, 3x to the GPU-only solution, and 2.3x to approaches adopting fixed-ratio flow partition between CPU and GPU. The message processing latency incurred with *FlowShader* is acceptable, and much smaller than a GPU-only approach.

II. Motivation and Challenge

A. GPU Background

There are several types of GPUs by different providers such as NVIDIA [24], AMD [25] and Intel [26]. We use NVIDIA's discrete GPUs [27] given their popularity in the market. In NVIDIA's representative Pascal architecture [28], a GPU has several graphics processing clusters which consist of multiple streaming multiprocessors (SMs). Each SM has multiple stream processors (SPs); each SP can run a GPU thread (also called a CUDA core). The basic execution unit of an SM is a warp, a group of 32 threads with Single Instruction Multiple Threads (SIMT) architecture. The processing time of one warp depends on the slowest thread in the warp. A *kernel* is a program that can be concurrently run on multiple GPU threads. For example, a NVIDIA Tesla P100 GPU has 6 graphics processing clusters, 56 SMs, 64 SPs, and 2 warp schedulers in each SM.

Compared with a CPU, one GPU's overall computation power is much stronger because of its thousands of cores. However, a single CPU core is much more powerful than a single GPU core. Batching is widely adopted when running GPU programs: the CPU gathers data into a *batch* and copies the batch to GPU's memory; after the GPU has processed the data, the CPU copies the data back to its memory.

B. Flow Processing versus Packet Processing

In flow-based NFs (such as the stream snort [29], an L7 load balancer or WAF), the basic processing unit is TCP session (namely, flow) between clients and servers, instead of packets. In such NFs, network packets in the same flow are first processed by TCP stack, and further reassembled into SSL/HTTP messages (representing application level requests from clients or response from the server), using high-performance pre-processors such as Linux's TCP stack (mTCP [30] or F-Stack [31]), OpenSSL, Apache/Nginx, etc.; the NF core only processes the reassembled messages.

In GPU-accelerated NF architectures for packet-processing [16], [32], there are no per-flow states and their management; packets can be fed into a GPU in any order and processed by different GPU threads. Moreover, most of these NFs only process packet headers rather than whole packets. For example, an IPv4 router checks destination IP address in IP header, and a packet-based firewall processes the five tuples [33]. The processing time of each packet does not depend much on the packet size.

A straightforward extension of existing architectures to flow processing is to batch messages for GPU processing, however, the message size may vary significantly. In this paper, we refer to a flow as a *large flow* or a *small flow* according to this data size of the flow in the current batch. L7 NFs typically need to process the entire content in the messages, e.g., a WAF checks the entire message to see whether it violates the predefined rules. Therefore, processing time of different flows by their respective GPU threads may differ significantly. Especially, GPU processing time of each batch is largely decided by the largest flow in the batch; most other GPU threads would have finished processing and wait for the thread processing this largest flow, resulting in severe underutilization of many GPU threads.

We show how flow data size imbalance affects GPU processing throughput, by implementing a few flow-processing NFs using a GPU acceleration design as in GPUNFV [34]. There are 2000 flows going through each NF, and the batch contains 1000 messages (the optimal batch size maximizing the NFs' throughput based on our tests). We vary the *ratio* of the data size of the largest flow over the average data size of all flows in each batch. The average results are shown in Fig. 1. The throughput drops significantly with the increase of the skewness ratio.

Therefore, imbalance of message size is an important challenge in exploiting GPU for flow processing.

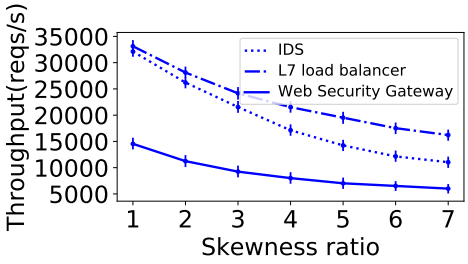


Figure 1: Throughput at diff. flow data size imbalance levels

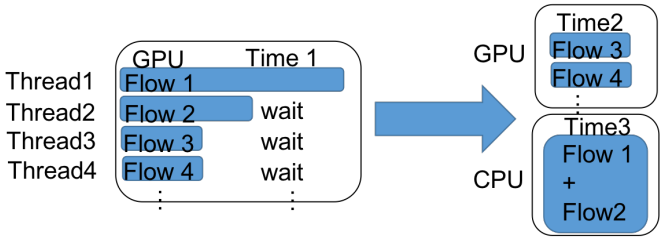


Figure 2: *FlowShader* dispatches large flows to CPU to address the flow size imbalance. Our flow scheduling algorithm tries to find an optimal partition to minimize max(Time2,Time3)

C. Design Rationale

We pursue a general GPU acceleration framework for reverse-proxy style of NF flow processing, achieving high throughput.

First, we adopt weak scaling to run one GPU thread to process one flow using the entire NF processing logic (Sec. III). We further provide a set of APIs for both CPU and GPU programming of an NF, for developers to implement core processing logic and specify flow states (Sec. V). Existing GPU acceleration architectures such as SSLShader [22] utilize strong scaling on a GPU, e.g., using multiple GPU threads to replace a *for* loop in CPU-based implementation. Such design can fully utilize parallel computation power of a GPU; however, the NF processing logic needs to be redesigned for each individual NF, and only computation intensive NFs benefit most. With our weak scaling approach, any flow-processing NF can benefit from GPU parallel processing regardless of whether it is computation intensive or merely carries out simple processing of small messages.

Second, to handle flow size imbalance, *FlowShader* exploits available capacities of CPU cores for flow processing, in addition to GPU processing. As shown in Fig. 2, our key idea is to dispatch large flows which seriously break up the flow size balance to CPU and the rest of flows to GPU. In this way, we can fully utilize the high computation power of a CPU core, efficient parallelism among GPU cores and otherwise-idling time of a CPU thread. To this end, we need to design an efficient flow scheduling algorithm among GPU and CPU cores for throughput maximization.

A well-known downside of using GPU is its additional latency due to the need of batching and copying messages between CPU and GPU memory. We target L7 NFs for WAN flow processing such as WSG and WAF, and the latency introduced by batching and GPU memory copy is much smaller as compared to the end-to-end delay experience by flows.

III. *FlowShader* Design

A. Architecture

We focus on NF processing in the reverse-proxy manner, (as in SSLShader [22]) and widely adopted in real-world L7 NFs (e.g., L7 load balancer, WSG, WAF); connections from the clients are terminated at the NF and re-established with the servers after NF processing is done. *FlowShader* is built on top of a TCP/IP stack to get the reassembled messages. We consider the representative system architecture [16], [22] that consists of multiple CPU cores and one GPU. We will discuss the extension to multiple GPUs in Sec. VII. *FlowShader* creates a thread for each CPU core. We set the thread affinity to reduce the overhead of cache misses. With Receiver Side Scaling (RSS), *FlowShader* guarantees that packets from a connection is always processed by a specific CPU core.

As shown in Fig. 3, each CPU thread of *FlowShader* has 4 major functional modules: *flow manager*, *flow operator*, *batch manager* and *flow scheduler* (inside batch manager). The processing logic of each thread can be divided into the following four steps.

Buffering: The *flow manager* keeps polling reassembled messages from the TCP/IP stack and stores the messages to the corresponding *flow operators*. A *flow operator* is uniquely identified by five-tuple. It has a per-flow message queue and a state storage. It is created when a new connection is established and destroyed when the connection is terminated.

Dispatching: The *batch manager* only processes messages when the number of unprocessed messages in message queues of all flow operators reaches a threshold or a timer times out. We use two thresholds, the *unprocessed message threshold* and the *timeout threshold*, to provide consistent good performance in various scenarios. With the unprocessed message threshold, we can batch many messages for GPU to process at each time, thus fully utilizing the GPU's computation power. This ensures high throughput at high traffic loads. However, at low loads, the unprocessed message threshold may cause large latency. To address this, we introduce the timeout threshold to bound the maximum batching time.

The *batch manager* invokes the *flow scheduler* to dispatch the unprocessed messages to the pinned CPU core and shared GPU card for processing. To guarantee processing consistency and accuracy, all the messages of a flow must be processed on the same CPU or GPU thread according to the arrival order. We develop a flow scheduling algorithm to resolve flow data size imbalance issue (more details in Sec. IV).

Processing: *FlowShader* uses both CPU and GPU to process messages. For messages dispatched to GPU, we copy them into a message batch and their flow states into a state batch (Fig. 4). Then the batch manager copies message batch and state batch into GPU's memory and launches the corresponding GPU NF kernel to process these messages (more details in Sec. III-C). The total number of launched GPU threads equals the number of flows included in the message batch.

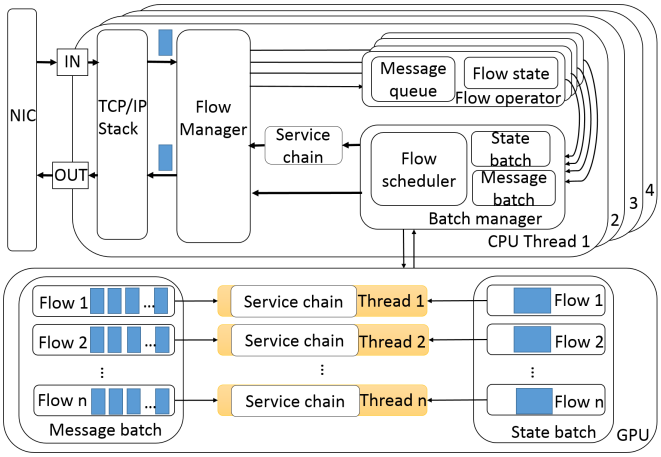


Figure 3: Architecture of *FlowShader*

For messages dispatched to CPU, the batch manager directly calls the corresponding CPU NF function to process them. Unlike GPU, the CPU NF function directly fetches messages and flow states from flow operators without the extra data copy.

Post Processing: After GPU processes the messages, the *batch manager* fetches processed messages and state updates from the GPU's device memory, updates the states in corresponding flow operators, and then forwards processed messages to the flow manager. Messages processed by CPU are directly sent back to the flow manager. The flow manager forwards these messages out of the system (e.g., drop or send to a backend server).

B. Dynamic Threshold Adjustment

The batch manager uses two thresholds, unprocessed message threshold and timeout threshold, to invoke message dispatching and processing. How to set the two thresholds is crucial. The timeout threshold can be set according to users' latency requirement. For example, Google observes that 100 to 400 ms latency slowdown reduces the number of searches by 0.2% to 0.6% [35]. Hence, we recommend to set the timeout threshold no larger than 100ms.

For the unprocessed message threshold, *FlowShader* adopts a dynamic adjustment strategy to find an approximately optimal threshold. We start with a relatively small threshold (e.g., 32 in the evaluation) and set the current threshold adjustment direction to $plus$. We periodically collect the overall flow processing throughput and define the throughput variation $r = \frac{Throughput_{curr} - Throughput_{prev}}{Throughput_{prev}}$.

We use a constant value r_thresh to control the adjustment direction. r_thresh is a positive number. If $r > r\_thresh$, it means that we receive a positive feedback in current adjustment direction. Hence, the threshold will be changed in the current direction with a *step size*. If $r < -r\_thresh$, it means that we receive a negative feedback. Hence, the direction will be set to the opposite ($plus$ becomes $minus$, or $minus$ becomes $plus$) and the threshold will be changed in the new direction with a step size. Otherwise, we keep the threshold unchanged. We note that a too large r_thresh is insensitive to detect the throughput feedback; a too small r_thresh is too sensitive to system noises, resulting in direction oscillations. We choose r_thresh to 0.5% in our evaluation and find it works well. We set step size to 32, which is the GPU warp size.

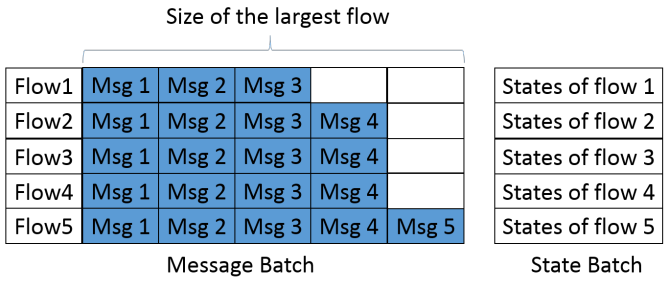


Figure 4: Structure of message batch and state batch

C. Batch Design

Fig. 4 gives an illustration of the batch. The message batch is a two-dimensional array, where the size of the first dimension is the number of flows, and the size of the second dimension is the maximum number of messages in a flow in the batch. An unprocessed message is copied to the corresponding position (m, n) in the two-dimensional array, where m is the flow index and n is the message index inside the flow. The state batch is a one-dimensional array and each object in the array contains one or several states of the respective flow.

We launch GPU threads such that all the messages in flow m are processed by GPU thread m. In this way, the GPU threads can easily fetch messages and states from the message and state batches according to thread IDs. We refer to the total number of messages in the message batch as the message *batch size* in this paper. The message batch size is no larger than the unprocessed message threshold.

D. Overlapping GPU and CPU Processing

To overlap GPU and CPU processing, we use asynchronous `cudaMemcpyAsync` to transfer data to GPU, launch kernel function, and then use it to transfer results back to host memory. The function returns immediately without waiting for corresponding GPU operations' completion. Indeed, we just issue three operation requests on GPU. After that, we run NF functions on CPU to process messages. At this moment, both GPU and CPU are processing messages. After CPU processing completes, we check whether GPU has finished all operations. If not, we can let CPU polls the new arrival messages instead of staying idle.

E. Kernel Fusion for NF Chain

In most of existing GPU accelerated NF solutions [16], [21], [36], each NF is implemented as a separate GPU kernel function. Hence, to traverse a NF chain, they may launch GPU kernel functions for multiple times. This causes high kernel launch delay and data transfer overhead. To address this problem, we adopt kernel fusion technique [37]. Kernel fusion merges multiple consecutively executed small kernels into a large kernel function. In *FlowShader*, we implement the processing logic of a NF service chain in a single GPU kernel function.

IV. Flow Scheduling

A. Rationale

The goal of flow scheduling is to resolve the imbalance flow data size within GPU. There exist some GPU/CPU load balancing algorithms in previous studies, but they focus on different purposes, so they cannot be utilized in our scenario: (1) Dispatch workload to GPU only when CPU is overloaded, such as opportunistic offloading in SSLShader [22] and dynamic offloading in Kargus [38]. To fully exploit expensive GPUs, *FlowShader* adopts GPU for flow processing as the first choice, and only uses the CPU core only if it has spare capacity (normally, it is responsible for fetching messages from TCP stack, and becomes idle when waiting for GPU processing), to assist in processing large flows.
(2) Allocate packets for CPU or GPU processing according to optimally tuned portions in all packets, or task queue sizes maintained for each device [32] [39] [40]. Such approaches are feasible where packet order is not distinguished, but not suitable for flow allocation where flow sizes may vary, and large and small flows have their respective ideal devices for processing. In addition, we retain processing time balance among GPU threads and CPU threads, by carefully allocating flows to individual threads, while existing approaches do not consider fine-grained inefficiency within each device.

B. Model

To design a good strategy for dividing flows between GPU and CPU threads, we segment CPU-side and GPU-side processing time in detail. In each round, the GPU time is mainly due to memory copy (for copying message batch and state batch between the CPU memory and the GPU memory), and parallel flow processing with GPU threads. Fig. 5 shows our measurements of GPU processing completion time and memory copy time (per-batch average) when the number of flows in each message batch varies, where we send flows consisting of the same HTTP requests of 128 bytes each to an NF running on a NVIDIA Tesla P100 GPU. We observe that the memory copy time increases linearly with the concurrent flow number. Since the message rate of the flows is the same, a larger concurrent flow number indicates larger size of the message batch. We have $T_{copy} = k_1 \times S_g + b_1$, where S_g is total size of the message batch copied to GPU, and k_1 and b_1 depend on the NF (if message size of flows passing through each NF differs).

For GPU processing time, we see a staircase curve: when the concurrent flow number is in a certain range, the completion time varies little. As we know, each SM in the GPU can run one thread block at each time [28]. The total number of thread blocks used by one CPU thread in a GPU is decided by the total number of GPU threads it runs on the GPU (equivalent to the number of flows sent to the GPU), N_{flows}, and the number of GPU threads in one thread block, N_{block} (e.g., 64 as we set when launching a kernel in our evaluation). We can approximately calculate the number of thread blocks to be processed by one SM (sequentially), if the CPU thread uses all N_{sm} SMs in the GPU, as $R = \lceil (N_{flows}/N_{block})/N_{sm} \rceil$.When different numbers of flows are sent to the GPU, if they lead to the same R value, we know that roughly the same number of thread blocks are processed by each SM, and hence similar GPU processing completion time results.

Fig. 6 further shows GPU processing completion time, when the size of individual messages in the flows varies. The processing time increases roughly proportionally to the increase of message size.

Therefore, we model GPU's flow processing time as $T_{g_{process}} = k_2 \times R \times M$, where M is the total data size of the largest flow in the message batch, and k_2 differs according to different NFs running on the GPU and how many CPU threads share the GPU concurrently. When the GPU is shared by multiple CPU threads, by default the SMs are evenly allocated to the CPU threads, and hence the relation $T_{g\_process} \propto R$ still holds, just with a different k_2.

The total GPU time, T_{GPU}, is:

$$\begin{aligned} T_{GPU} &= T_{g\_process} + 2T_{copy} \\ &= k_2 \lceil (N_{flows}/N_{block})/N_{sm} \rceil M + 2k_1 S_g + 2b_1. \end{aligned} \tag{1}$$

Here, factor 2 is due to copying message and state batches from the CPU memory to the GPU memory and then back after GPU has finished processing.

In each round, the CPU time mainly consists of three parts: message receiving time $T_{receive}$, batching time T_{batch}, and flow processing time $T_{c\_process}$. Based on our experiments, $T_{receive}$ is linear with the unprocessed message threshold (m), T_{batch} is linear with the size of the message batch for GPU processing (S_g), and $T_{c\_process}$ is linear with the overall data size of all flows that the CPU thread processes (S_c). Therefore, the total CPU time, T_{CPU}, is:

$$\begin{aligned} T_{CPU} &= T_{receive} + T_{batch} + T_{c\_process} \\ &= k_3 m + k_4 S_g + k_5 S_c + b_2. \end{aligned} \tag{2}$$

In each CPU thread, the flow scheduler allocates received flows to be processed by this CPU thread and the GPU, respectively, such that the processing completion time in each batching round, $T_{total} = \max\{T_{CPU}, T_{GPU}\}$, is minimized. To formulate this problem, let binary variable x_i represent which device the ith flow is allocated to: $x_i = 0$ if it is allocated to the CPU and $x_i = 1$ if allocated to GPU. Let $size_i$ denote the message size of the ith flow. We can formulate an 0-1 integer linear program(ILP):

$$\min_x \ \max\{T_{GPU}, T_{CPU}\}$$

$$s.t. \begin{cases} T_{CPU} = k_3 m + k_4 \sum_{i=1}^{m} (x_i * size_i) + k_5 \sum_{i=1}^{m} ((1 - x_i) * size_i) + b_2 \\ T_{GPU} = k_2 \left\lceil \frac{(\sum_{i=1}^{m} x_i)}{N_{block}})/N_{sm} \right\rceil * \max_{i \in [1,m]} \{size_i * x_i\} + 2k_1 \sum_{i=1}^{m} (x_i * size_i) \\ x_i \in \{0, 1\} \ \forall \ i = 1, 2, ..., m. \end{cases}$$

As we know, ILP is proved to be NP complete as some known NP complete problems such as 3-SAT and vector cover problem can be reduced to ILP [41], [42]. Although some special cases of ILP can be solved in polynomial time [42],

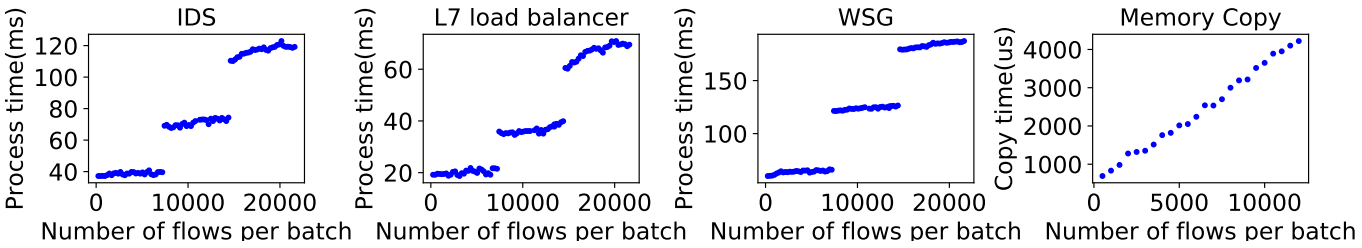


Figure 5: GPU memory copy time and processing time

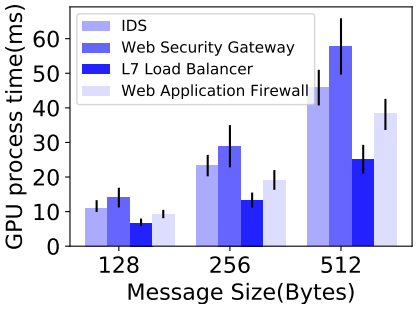


Figure 6: GPU processing time with diff. message sizes

[43], it is difficult to verify whether a specific ILP instance can be solved in polynomial time.

C. Flow Scheduling Algorithm

We propose a heuristic algorithm to solve the problem in a short time. Our flow scheduling algorithm (Alg. 1) takes as input an array $A[]$, containing overall data sizes of the flows in ascending order. For example, $A[] = [3, 4, 4, 4, 5]$ for the example in Fig. 4 (assuming each message's size is 1). The algorithm returns a partition index i in array $A[]$, such that all flows in the index range of $[0, i]$ are sent to GPU for processing, and all flows in index range $[i + 1, N_{flows} - 1]$ are processed by the CPU core, where N_{flows} is the total number of flows in the current batch. The algorithm goes by repeatedly partitioning the flows to two sets corresponding to indices $[0, i - 1]$ and $[i, N_{flows} - 1]$ in $A[]$, starting from $i = N_{flows} - 1$, and compute T_{total} with the partitions. We decrement i by 1 each time, as long as T_{total} computed with the obtained partition is decreasing, and stop at an i when its T_{total} is no smaller than that at $i + 1$.

We can easily see that the time complexity of the algorithm is $O(N_{flows} \log N_{flows})$ (due to flow sorting). We know that T_{CPU} is monotonically decreasing on i and T_{GPU} is non-decreasing with i. Hence, with the decrease of i starting from $N_{flows} - 1$, T_{total} is non-increasing at first and then increases. So there is no need for the algorithm to traverse the entire list $A[]$; it can stop when T_{total} is not longer decreasing. Since the number of flows processed by GPU should always be much larger than the number of flows processed by a CPU thread and large flows go to the CPU, we start the partition search from the largest flow, to reduce the iteration times.

Algorithm 1 Flow Scheduling Algorithm

function SCHEDULE_FLOWS(A[])
  $S_c = 0; N_{flows} = A.size$;
  $N_{msgs} = \sum_{i=0}^{N_{flows}} A[i]$;
  $M = A[N_{flows} - 1]$;
  for $i = N_{flows} - 1$ to 0 **do**
    $S_c += A[i]$;
    $S_g = N_{msgs} - S_c$;
    $T_{GPU} = k_2 \left\lceil \frac{N_{pkts}/N_{block}}{N_{sm}} \right\rceil M + 2k_1 S_g + 2b_1$;
    $T_{CPU} = k_3 m + k_4 S_g + k_5 S_c + b_2$;
    $T_{total} = \max(T_{GPU}, T_{CPU})$;
    if i == A.size-1 **then**
      $T_{min} = T_{total}$;
    else
      if $T_{total} < T_{min}$ **then**
        $T_{min} = T_{total}$;
      else
        return i;
      end if
    end if
  end for
end function

The coefficients appearing in our model, i.e., k_i's and b_i's, are estimated and calibrated periodically. Every 200 rounds (as in our experiments), we profile CPU and GPU execution times, $T_{receive}$, T_{batch}, $T_{c\_process}$, T_{copy} and $T_{g\_process}$. Then we use linear regression techniques to fit the collected data and calculate the coefficients.

Discussions. We do not consider dividing one large flow in a batch into several portions, delaying processing of some portions to later batching rounds, for evening out flow sizes. There are two reasons: First, the latency experienced by messages in the flow would be much larger, as compared to offloading their processing to CPU in this round; Second, NF code needs to be specifically modified to support processing pieces of incomplete messages, especially in cases that the large flow contains only 1-2 very large messages in the batch.

V. API FOR IMPLEMENTING NF

We separate NF processing logic from the main management framework in *FlowShader*, by providing a set of APIs for developers to implement the NF. With these APIs, the NF processing logic can be shared between the CPU side and the GPU side.

On the GPU side, an NF class and a state class need to be defined. In the state class, the operator = should be overwritten to allow easy copying of the states into GPU shared memory. In the NF class, the function `nf_logic` takes an extra parameter `info_for_gpu`, which is sent to the GPU when the NF is initialized. The processing logic implemented in `nf_logic` is same as the NF on the CPU side, while the code should be programmed with the CUDA syntax.

```
1. class NF1_flow_state{
2. public:
3.   //define the member variables.
4.   __device__ NF1_flow_state& operator=
5.     (const NF1_flow_state& s) {
6.     // overwrite "=" for easy copying states to GPU shared memory.
7.       return this;
8.   }
9. };
10. struct NF1_gpu_info;
11. class NF1{
12. public:
13.   __device__ static void nf_logic
14.     (void *msg, NF1_flow_state* state,NF1_gpu_info* info_for_gpu){
15.     //implement the core processing logic here.
16.   }
17. };
18. typedef class NF1 NF;
19. typedef class NF1_flow_state nf_flow_state;
20. typedef class NF1_gpu_info Infos;
```

Figure 7: APIs for implementing GPU-side NF

On the CPU side, the steps to implement the NF are very similar to those in Fig. 7. The developers can just refer to the code in GPU side and remove the components related with GPU.

VI. Experiments

We evaluate *FlowShader* using testbed experiments. We summarize our key findings as follows:

Throughput: The system throughput achieved by *FlowShader* is up to 6x that of the CPU-only baseline, 3x of the GPU-only solution, and 2.3x of the fixed-ratio flow partition approach; the throughput also remains stable under high skewness of the flow data sizes.

Latency: The average latency introduced by *FlowShader* is around 20ms, which is much smaller than the GPU-only approach. The worst case (100ms) is bounded by the timeout threshold. We think these latency results are acceptable in WAN environment as the based latency from the client to the nearest data center is around $20 \sim 200$ms in CDN [44]–[47]. Furthermore, deploying more NFs in one service chain does not increase latency because the messages only need to be buffered and transferred once.

Adaptiveness: Using its dynamic threshold adjustment strategy, *FlowShader* is able to identify the optimal unprocessed message threshold value, achieving the highest system throughput, within seconds.

A. Experimental Setup

Implementation. *FlowShader* is built using about 2000 lines of C++ code on top of Linux kernel TCP/IP stack with the libevent library [48]. *FlowShader* works as a reverse proxy: it intercepts and terminates client connections, processes requests from clients received over these connections, and then sets up new connections with the backend servers and forwards requests to them over the new connections; on the reverse path, it receives response messages from backend servers, and then forwards the responses to corresponding clients. This is a standard design adopted by L7 NFs [22], to take actions (e.g., drop malicious traffic) before forwarding flows to destinations. We implement four NFs on *FlowShader*.

▷ **Intrusion Detection System.** The stream-processing IDS (916 LoC) utilizes Aho-Corasick algorithm to examine the messages of each flow; after processing a message belonging to a flow, the state produced by the state machine should be stored as flow state for next message's processing.

▷ **L7 load balancer.** L7 load balancer (624 LoC) dispatches flows to different backend Web servers according to their URIs. The destination Web servers are different for different types of URIs, as in the case of Application Gateway of Microsoft Azure [49]. The load balancer routes the traffic based on the incoming URI, such as static web pages, image files or video files.

▷ **Web Security Gateway.** Traffic encryption is applied by various protocols and services, e.g., SSL, VPN, and IPsec, for secure communication. In the WSG (1415 LoC), we use AES-CBC algorithm to decrypt the messages from clients and forward the decrypted messages to backend servers, and then encrypt the response messages received from backend servers to clients.

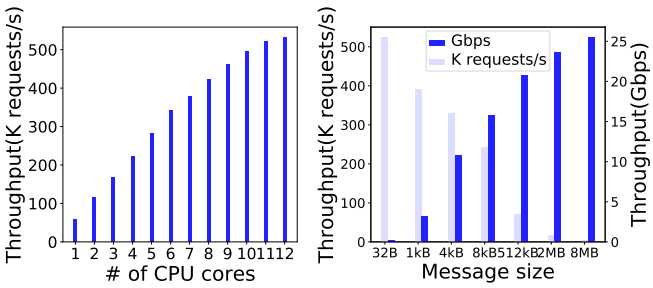
▷ **Web Application Firewall.** The WAF (1577 LoC) examines the HTTP POST requests: it uses regular expression matching to check if there are any javascript, shell or powershell keywords or build in function names in the message body; if so, it raises alerts.

Testbed. We build a testbed with 8 machines connected to a 100GbE switch. *FlowShader* is running on a machine equipped with two Intel Xeon processors E5-2630 v4 (20 physical cores in total), a NVIDIA Tesla P100 GPU (56 SMs and 64 CUDA cores per SM) and a Mellanox CX4 100GbE NIC. The OS is 64 bit Ubuntu 16.04. Another machine runs backend server services, e.g., Web servers. The rest 6 machines run clients to generate requests to *FlowShader*.

Traffic Generator. We develop a traffic generator to send HTTP GET or POST requests to *FlowShader*, and GET requests are sent in all cases other than those involving the WAF. The clients run traffic generators to produce flows with different message sizes. For each connection, the client sends the next request to *FlowShader* once it has received response to the previous request. The traffic generator runs on each client using multiple CPU threads.

We produce flows using both synthetic patterns and real-world traces. In controlled experiments, the size of the messages and the proportions of flows with the same message size are produced following a zipf-like distribution which is common for real-world network traffic [50]–[53], with message sizes in the range of 32 bytes to 8M bytes. We also replay traffic traces gathered by thousands of WAFs deployed in one of the largest public cloud provider: there are 1 million connections; message sizes among the connections varying from 7 bytes to 82MB, with an average message size of about 4KB. The clients change the volume of flows to *FlowShader* according to the trace.

Schemes Compared. We implement the following baselines:



(a) 32-byte messages (b) different message sizes

Figure 8: Throughput of *FlowShader* without NF processing.

(i) CPU-only baseline where only the CPU cores are used for flow processing and GPU is not used; (ii) GPU-only solution where all flows are dispatched to the GPU, and each flow is processed by one thread; (iii) fixed-ratio flow partition between CPU and GPU, where we order all flows according to their data sizes, and $x\%$ larger flows are sent to CPU and $(1-x)\%$ smaller flows are sent to GPU. We utilize the group prefetch strategy introduced in G-opt [54] to optimize the CPU-side implementation in all cases.

In all experiments, we set the timeout threshold to 100 ms, r_thresh to 0.5% and step size to 32. Each experiment lasts for some minutes. We repeat each experiment for 10 times and show the average results.

Throughput. We first evaluate *FlowShader*'s 'bare metal' performance without running any NF processing logic: the CPU threads copy flow messages to the GPU, the GPU threads directly return without any processing, and then the CPUs copy messages back and send them out. We send 2,000 flows to each CPU core. In Fig. 8a, the flows has a small message size of 32 bytes; with small message sizes, *FlowShader* needs to frequently retrieve/dispatch messages from/to the TCP/IP stack, with high overhead. The largest throughput (in K requests/s) is achieved when *FlowShader* launches 12 threads using 12 CPU cores to share the GPU (throughput becomes stable with additional cores). Fig. 8b shows the throughput under different flow message sizes when the number of CPU cores is 12. With larger message size, the number of requests processed per second decreases, but the overall number of bits processed per second increases.

Next, we evaluate *FlowShader* when it runs different NFs and service chains. In each experiment, the CPU cores run the same NF or service chain. In the rest of the controlled experiments, the message size is the same 1KB among the flows, unless stated otherwise. Fig. 9 and Fig. 10 show that *FlowShader* consistently outperforms the CPU-only baseline with different numbers of CPU cores in use, when *FlowShader* runs individual NFs or service chains, respectively. For example, *FlowShader* achieves 5-6x throughput of the baseline with 3-6 CPU cores for WAF. In our experiments, we observe that the CPU-side optimization as in G-opt [54] does not bring significant performance improvement. The reason may lie in that G-opt has little impact on compute-intensive operations, as also observed in [21].

Fig. 12 further shows the throughput when each CPU core concurrently runs multiple service chains. In this experiment, we send 2000 flows to each service chain in each CPU core. "SCx" corresponds to the respective service chain in Fig. 10. Compared with Fig. 10, the overall throughput is in general between those achieved by running only the fastest chain and running only the slowest chain.

Effectiveness of Flow Scheduling. We now generate flows with uneven data sizes to evaluate *FlowShader*. We use 1 CPU core and 5,000 flows in this experiment. The *skewness ratio* of flow data sizes in a batch, is defined to be to ratio of data size of the largest flow over the average data size of all flows (the same as in Fig. 1). We compare with the GPU-only approach and a fixed-ratio flow partition strategy where 80% (or 90%) smaller flows go to GPU and 20% (or 10%) larger flows go to CPU. Fig. 11 shows that the performance of GPU-only approach degrades seriously with the increase of the skewness ratio; the performance is quite stable with *FlowShader* which dispatches flows strategically to both GPU and CPU (3x throughput of the GPU-only approach at skewness ratio of 7 for most NFs). For fixed-ratio allocation, the performance is worse than *FlowShader* (*FlowShader* achieves 2.3x throughput at skewness ratio of 7 for WSG), and even worse than the GPU-only approach when the skewness ratio is small. When the skewness ratio is 1, *FlowShader* performs close to the GPU-only approach, with slightly smaller throughout in case of some NFs, which is due to the overhead of running our flow scheduling algorithm.

The second row in Fig. 11 illustrates the data size unevenness ratio in the GPU. With *FlowShader*, the ratio is always below 2, demonstrating relative data size balance among those flows. This ratio achieved by the fixed-ratio partition is even lower than *FlowShader* in some cases, so that we see relatively balanced flow data sizes in the GPU are necessary but not sufficient to achieve the best performance, and the number of flows dispatched to GPU and CPU, respectively, matters too.

We next evaluate the system throughput and message latency with different unprocessed message thresholds (hence different message batch sizes). We use 1 CPU core and 5,000 flows, with a flow data size skewness ratio of 5. We disable the dynamic threshold adjustment in *FlowShader* and manually set the unprocessed message threshold in each experiment.

Throughput with different Message Threshold. Fig. 13 shows that the throughput is larger with the increase of the threshold at first, and after a peak value, it becomes stable or even drops a bit if the threshold continues to grow. The drop could be due to the more-than-linear increase of overhead when more messages/flows are processed, e.g., for sorting the flows, etc.

Latency. We compute the latency of messages from when a message is received in the message batch (after it is reassembled by the TCP/IP stack) to when the message batch is forwarded out. Fig. 14 shows this value that 90% messages are smaller than. The latency increases linearly with the unprocessed message threshold, but remains at around 20 ms when the system throughput peaks (at a threshold value of about 1024).

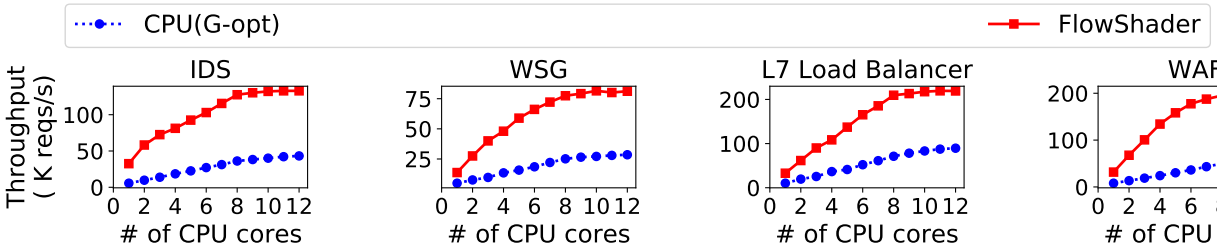


Figure 9: Throughput of NFs compared with CPU-only baseline

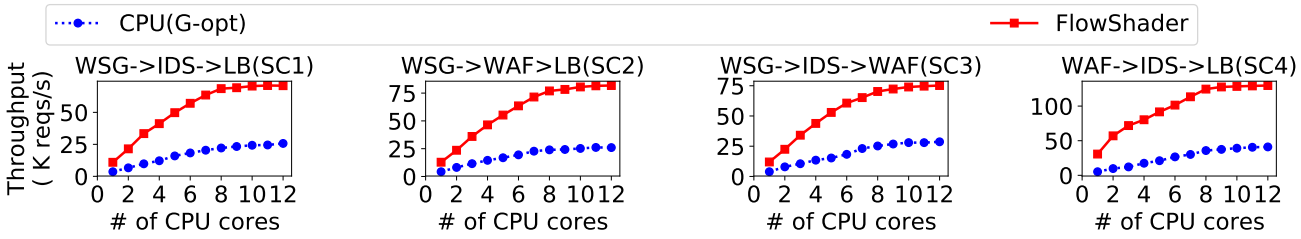


Figure 10: Throughput of service chains compared with CPU-only baseline

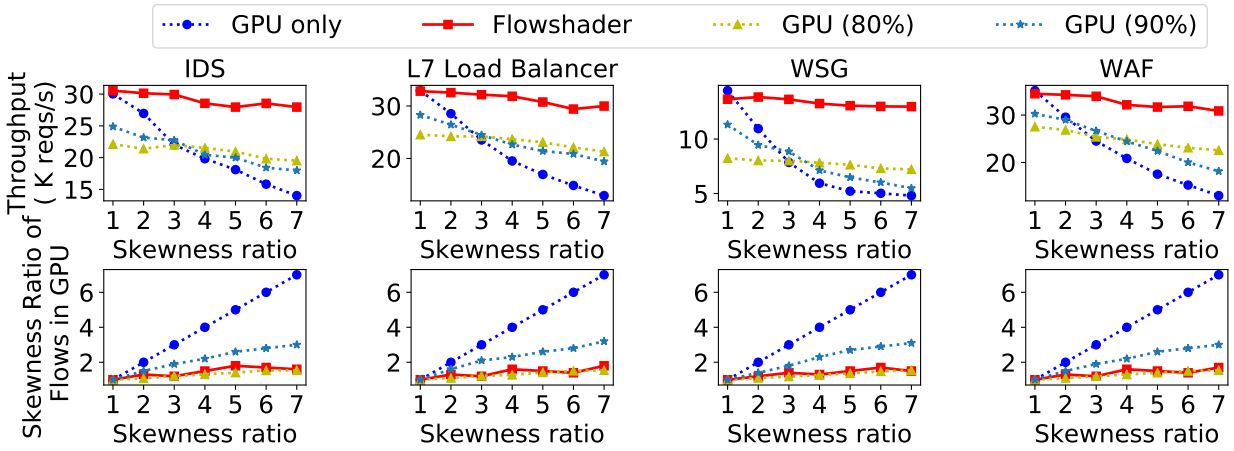


Figure 11: Throughput with/without our flow scheduling algorithm

We have also evaluated message latencies incurred by GPU-only and fixed-ratio partition strategies, and found that the latencies experienced by *FlowShader* are approximately half of theirs.

Dynamic Threshold Adjustment. We further evaluate the dynamic adjustment of unprocessed message threshold in *FlowShader*. Initially, the threshold is 32 (the number of threads in a GPU warp); each time we adjust the threshold by 32 and $r\_thresh = 0.5\%$. Under the same settings as in previous set of experiments, Fig. 15 shows the overall system throughput achieved over time. We can see that the throughput improves with the threshold adjustment and stabilizes at the maximum within 20-30 seconds.

Evaluation with Real-world Traffic Traces. We now evaluate *FlowShader* with traffic produced following the real traces. Fig. 16 shows that *FlowShader* always outperforms CPU-only and GPU-only baselines.

We next inspect average message latency experienced by this realistic traffic with *FlowShader* and with GPU-only baseline, when WAF is run in the systems. In Fig. 17, we see that *FlowShader*'s average message latency with the realistic traffic is about 26ms, much lower than that experienced by the GPU-only baseline. The overhead of running the flow scheduling algorithm is ignorable (only about 50μs). The GPU processing time roughly equals the sum of the message receiving/batching time and CPU processing time.

We further evaluate the influence of timeout threshold on the average message latency in *FlowShader*. We sample flows from the real traffic traces and vary the concurrent number of flows in each message batch. In Fig. 18, we see that when the number of concurrent flows exceeds the unprocessed message threshold, the timer threshold has no effect on latency. When the concurrent flows are less, timers are triggered, and the latency is bounded by the timeout threshold.

VII. Related Work

Flow-processing NFV Frameworks. Most existing work on NFV focus on stateless, packet-processing network functions [55]–[59]. A few recent studies investigate stateful flow-processing NFs, e.g., StatelessNF [60], S6 [61], Flurries [62]. Built on mTCP [30], mOS [63] proposes a unified interface for stateful middleboxes. *FlowShader* can also be built on

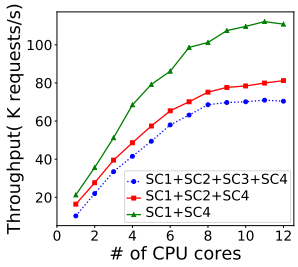


Figure 12: Throughput with multiple service chains

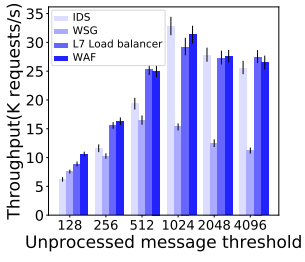


Figure 13: Throughput with diff. message thresholds

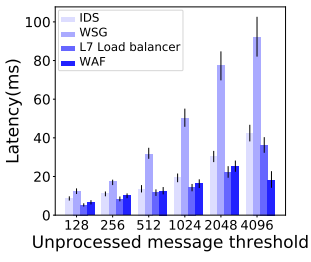


Figure 14: Latency with diff. message thresholds

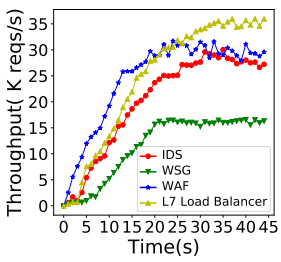


Figure 15: Throughput with dynamic adjustment strategy

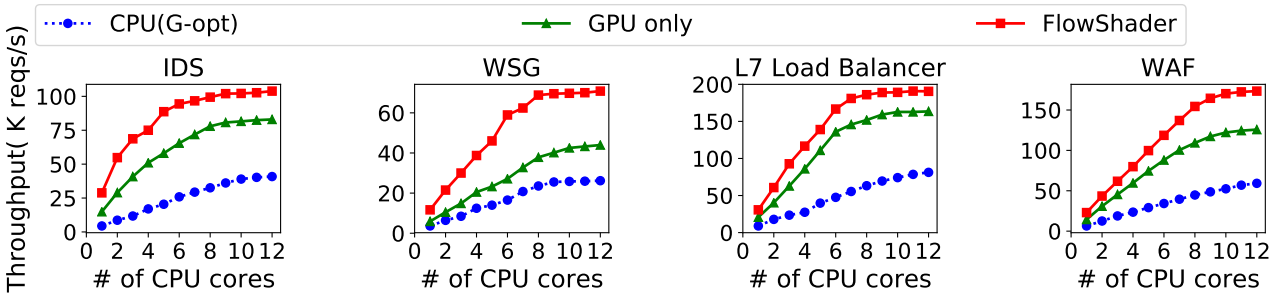


Figure 16: Throughput comparison using real traffic traces

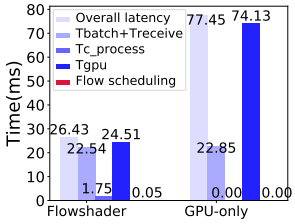


Figure 17: WAF's latency using real traffic traces

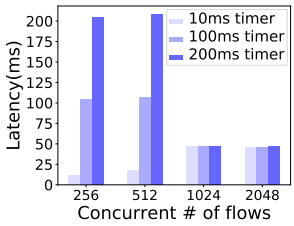


Figure 18: WSG's latency with diff. timeout thresholds

top of user space TCP/IP stacks such as mTCP, F-Stack [31] and SeaStar [64]. We choose to directly use linux TCP/IP stack because it is most widely used among the candidates for supporting a large number of concurrent connections.

GPU acceleration of NFs. A number of studies exploit GPUs to accelerate packet processing [65], [66]. PacketShader [16] implements GPU-based IPsec gateway and software router. Snap [36] builds software routers exploiting GPUs, based on the Click modular router [20]. Kargus [38] is a high-performance IDS and employs GPUs for pattern matching. Gnort [67] offloads pattern matching in Snort [3] to GPU. GSwitch [68] built a GPU accelerated software switch. G-NET [69] focuses on GPU Sharing among multiple NFs running on different processes. NBA [32] adopts optimally tuned portions to allocate packets to CPUs and GPUs for processing. APUNet [21] utilizes integrated GPU to avoid data transfer bottleneck in PCIe communication between CPU and GPU.

For flow processing acceleration using GPUs, we have extensively discussed SSLShader [22]in the introduction and motivation sections. The scenario and approaches are both different from *FlowShader*. For GASPP [23], it does not consider the important flow size imbalance problem.

VIII. Discussions

State management. In *FlowShader*, the flow states are stored locally. It can be easily extended to support state-of-the-art state management approaches to handle shared states, such as remote state server proposed in StatelessNF [60] and DHT-based distributed state storage in S6 [61]. A communication function for contacting remote state storage can be added in our flow operators. The communication time, for contacting remote state storage, is part of the CPU time, and our flow scheduling algorithm can readily take it into consideration when scheduling flows to CPU and GPU.

Extension to multiple GPUs. When there are multiple GPUs, the batch manager in each CPU thread can decide which GPU to launch the NF kernel on and dispatch flows to. It can implement a two-level flow dispatching algorithm: (i) Decide which GPU(s) to use to process the current batch of messages based on some multi-device load balancing algorithms [39] [40] [32]. (ii) Once the GPU is selected, the batch manager can run our proposed flow scheduling algorithm to partition flows among its CPU thread and the GPU.

IX. Conclusion

We have presented *FlowShader*, a GPU acceleration architecture for L7 flow-processing NFs. *FlowShader* is general, that any NF can be implemented in the framework using the same set of processing logic on both GPU and CPU, without redesigning custom NF logic for operation-level parallelism on GPU. *FlowShader* is efficient: our carefully designed flow scheduling algorithm enables significant throughput improvement as compared to CPU-only, GPU-only and hybrid approaches with fixed-ratio flow partition between CPU and GPU, as well as latency reduction as compared to GPU-only and fixed-ratio flow partition solutions.

REFERENCES

[1] "Large-scale adoption of nfv and sdn by telecom providers within two years," http://www.telco.com/blog/large-scale-adoption-of-nfv-and-sdn-by-telecom-providers-within-two-years/, 2017.

[2] "Virtual network services - wan optimization," http://www.verizonenterprise.com/products/networking/sdn-nfv/virtual-network-services/vns-wan-optimization/, 2018.

[3] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks," in *Proc. of the 13th USENIX Conference on System Administration (LISA'99)*, 1999.

[4] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Elsevier Computer networks*, vol. 51, no. 12, pp. 3448–3470, 2007.

[5] G. Gu, R. Perdisci, J. Zhang, W. Lee *et al.*, "Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection," in *Proc. of the USENIX Security Symposium*, 2008.

[6] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in *Proc. of the International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2004.

[7] Z. Shan, C. Lin, D. C. Marinescu, and Y. Yang, "Modeling and performance analysis of qos-aware load balancing of web-server clusters," *Elsevier Computer Networks*, vol. 40, no. 2, pp. 235–256, 2002.

[8] M. G. Gouda and A. X. Liu, "A model of stateful firewalls and its properties," in *Proc. of the 2005 IEEE International Conference on Dependable Systems and Networks (DSN'05)*, 2005.

[9] A. Lahmadi and O. Festor, "Secsip: a stateful firewall for sip-based networks," in *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, 2009.

[10] F. Cuppens, N. Cuppens-Boulahia, J. Garcia-Alfaro, T. Moataz, and X. Rimasson, "Handling stateful firewall anomalies," in *Proc. of IFIP International Information Security Conference*, 2012.

[11] W.-H. Kao, "Security gateway utilizing ssl protocol protection and related method," 2006.

[12] A. Shimbo, A. Inoue, M. Ishiyama, and T. Okamoto, "Packet authentication and packet encryption/decryption scheme for security gateway," 2000.

[13] M. Becher, *Web application firewalls*. VDM Verlag, 2007.

[14] E. Fong and V. Okun, "Web application scanners: definitions and functions," in *Proc. of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007.

[15] B. Li, K. Tan, L. L. 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 *Proc. of ACM SIGCOMM*, 2016.

[16] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *Proc. of ACM SIGCOMM*, 2010.

[17] "Azure windows vm sizes - gpu," https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu, 2018.

[18] "Amazon ec2 elastic gpus," https://aws.amazon.com/ec2/elastic-gpus/, 2018.

[19] "Gpus on google cloud," https://cloud.google.com/gpu/, 2018.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[21] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: Revitalizing GPU as packet processing accelerator," in *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017.

[22] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "Sslshader: Cheap ssl acceleration with commodity processors," in *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011.

[23] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "Gaspp: A gpu-accelerated stateful packet processing framework," in *Proc. of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[24] "NVIDIA," http://www.nvidia.com/page/home.html, 2018.

[25] "AMD," https://www.amd.com/en, 2018.

[26] "Intel graphics technology," https://www.intel.com/content/www/us/en/architecture-and-technology/visual-technology/graphics-overview.html, 2018.

[27] "Discrete GPU," https://www.quora.com/What-is-a-discrete-GPU, 2018.

[28] "Nvidia pascal architecture," https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/, 2018.

[29] "Readme.stream5," https://www.snort.org/faq/readme-stream5, 2018.

[30] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems," in *Proc. of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[31] "Fstack," http://www.f-stack.org, 2018.

[32] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "Nba (network balancing act): a high-performance packet processing framework for heterogeneous processors," in *Proc. of the Tenth European Conference on Computer Systems (EuroSys'15)*, 2015.

[33] A. Mayer, A. Wool, and E. Ziskind, "Fang: A firewall analysis engine," in *Proc. of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[34] X. Yi, J. Duan, and C. Wu, "Gpunfv: a gpu-accelerated nfv system," in *Proc. of the First Asia-Pacific Workshop on Networking (APNet)*, 2017.

[35] J. Brutlag, "Speed matters for google web search," https://services.google.com/fh/files/blogs/google-delayexp.pdf, 2009.

[36] W. Sun and R. Ricci, "Fast and flexible: Parallel packet processing with gpus and click," in *Proc. of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'13)*, 2013.

[37] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *Proc. of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*, 2010.

[38] 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 *Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[39] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-gpu systems," in *Proc. of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[40] L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," in *Proc. of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'14)*, 2014.

[41] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, 1972.

[42] "Integer programming," https://en.wikipedia.org/wiki/Integer-programming, 2018.

[43] J. Edmonds and E. L. Johnson, "Matching: A well-solved class of integer linear programs," in *Combinatorial OptimizationEureka, You Shrink!*, 2003.

[44] S. Choy, B. Wong, G. Simon, and C. Rosenberg, "A hybrid edge-cloud architecture for reducing on-demand gaming latency," *Proc. of the Multimedia Systems*, 2014.

[45] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, and J. Padhye, "Analyzing the performance of an anycast cdn," in *Proc. of the Internet Measurement Conference*, 2015.

[46] X. Fan, E. Katz-Bassett, and J. Heidemann, "Assessing affinity between users and cdn sites," in *Proc. of the International Workshop on Traffic Monitoring and Analysis*, 2015.

[47] R. Singh, A. Dunna, and P. Gill, "Characterizing the deployment and performance of multi-cdns," in *Proc. of the Internet Measurement Conference*, 2018.

[48] "Libevent - an event notification library," http://libevent.org, 2003.

[49] "Application gateway of Microsoft Azure," https://docs.microsoft.com/en-us/azure/application-gateway/application-gateway-introduction, 2018.

[50] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging zipf's law for traffic offloading," 2012.

[51] L. Durbeck, J. G. Tront, and N. J. Macias, "Energy efficiency of zipf traffic distributions within facebook's data center fabric architecture," in *Proc. of Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*, 2015.

[52] B. Blaszczyszyn and A. Giovanidis, "Optimal geographic caching in cellular networks," in *Proc. of Communications (ICC), 2015 IEEE International Conference on*, 2015.

[53] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[54] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing," in *Proc. of the 12th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI'15), 2015.

[55] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *Proc. of ACM SIGCOMM*, 2015.

[56] J. Hwang, K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," in *Proc. of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, 2014.

[57] J. Fan, Z. Ye, C. Guan, X. Gao, K. Ren, and C. Qiao, "Grep: Guaranteeing reliability with enhanced protection in nfv," in *Proc. of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015.

[58] C. Sun, J. Bi, Z. Zheng, and H. Hu, "Sla-nfv: an sla-aware high performance framework for network function virtualization," in *Proc. of ACM SIGCOMM*, 2016.

[59] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv," in *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[60] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NDSI'17)*, 2017.

[61] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.

[62] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *Proc. of the 12th ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT'16)*, 2016.

[63] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: a reusable networking stack for flow monitoring middleboxes," in *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017.

[64] "Seastar," http://seastar.io/, 2018.

[65] K. Kang and Y. S. Deng, "Scalable packet classification via gpu metaprogramming," in *Proc. of Design, Automation & Test in Europe (DATE'11)*, 2011.

[66] J. Tseng, R. Wang, J. Tsai, S. Edupuganti, A. W. Min, S. Woo, S. Junkins, and T.-Y. C. Tai, "Exploiting integrated gpus for network packet processing workloads," in *Proc. of the 2nd IEEE Conference on Network Softwarization (NetSoft'16)*, 2016.

[67] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Proc. of International Workshop on Recent Advances in Intrusion Detection*, 2008.

[68] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman, "Multilayer packet classification with graphics processing units," *Proc. of IEEE/ACM Transactions on Networking*, 2016.

[69] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective GPU sharing in NFV systems," in *Proc. of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.