Hummingbird: Leveraging Heterogeneous System Architecture for deploying dynamic NFV chains

Avinash Kumar Chaurasia*, Bhaskaran Raman*, Praveen Kumar Gupta†
Omkar Prabhu†, Shashank P†, Anshuj Garg*

*Department of Computer Science and Engineering
Indian Institute of Technology Bombay
{avinashk, br, anshujgarg}@cse.iitb.ac.in

†Department of Computer Science and Engineering
National Institute of Technology Surathkal
{pvgupta24, omkarsp31, shashankp5424}@gmail.com

***Abstract*—Network Function Virtualization has gained traction as a network function deployment alternative due to its flexibility and cost benefits. The telecommunication (telecom) operators and infrastructure providers are looking for high throughput, low latency NFV deployment model to avail the benefits of NFV. Moreover, NFV is one of the core technology for the next-generation communication network such as 5G. Furthermore, telecom operators employ groups of network functions(NFs) that process packets in linear order so that the output of one NF becomes an input for another, thus forming the network function chain (NFC). However, these NFCs should be flexible, as all telecom packets do not necessarily need to be processed by the same set of NFs. It has been earlier shown that GPU increases the throughput of NFV chains. To the best of our knowledge, none of the GPU-based frameworks supports dynamic NFV chains. Furthermore, discrete GPUs are expensive and consume a fair amount of energy. This paper presents the design and evaluation of Hummingbird, a framework to support high throughput, dynamically routed NFV chain on Heterogeneous System Architecture (HSA). Though HSAs are affordable and power-efficient, they lack high throughput GPU-CPU synchronization. Furthermore, current technology does not provide a zero-copy mechanism for network IO between GPU and NIC for HSAs. Hummingbird addressed those challenges. As per our knowledge, this is the first such framework that provides high throughput dynamic NFV chaining, with NFs chained across GPU and CPU and designed in conformance to OpenCL 2.0 standard. Hummingbird achieves 6x throughput per-core and 3.5x throughput per unit of energy consumption compared to state-of-the-art NFV deployment framework G-net, which uses powerful and costly discrete GPU.**

***Index Terms*—Network function virtualization, network function chain, APU, GPU, HSA**

I. Introduction

Network Function Virtualization (NFV) [1] is a paradigm that decouples Network Functions (NFs) from traditional proprietary hardware appliances (ASICs) such as firewalls, proxies, routers, etc. NFV enables the programming and deployment of network function over the general-purpose off-the-shelf hardware like CPU, GPU, FPGA, etc. Virtualization of NFs over general-purpose compute units has proven to be instrumental in reducing the OPeration EXpenses (OPEX) and CAPital EXpenditure (CAPEX)) [1]. Furthermore, software-based NFs are easier to debug and can be scaled on-demand.

Though NFVs offer flexibility and cost-effectiveness, achieving a computing speed comparable to ASICs are challenging with software-based Network Functions. One of the reasons for the slow performance of NFV is the overheads with the operating system (OS) TCP/IP stack. Every packet must go through a TCP/IP stack in traditional network processing. It incurs a considerable overhead on the reception of each packet [2]. The main problem with the traditional TCP/IP stack is that it allocates and deallocates buffer per packet. As part of the packet reception process, this procedure alone consumes more than 50% of the CPU cycle [2]. To alleviate this, Intel developed a novel Software Development Kit (SDK) called data plane development kit (DPDK) [3]. DPDK bypasses the TCP/IP stack and copies the packets directly in the userspace. Hence enabling a fast path for packet processing and facilitating line-rate processing, i.e., DPDK accelerated applications can process the packet at the speed of NIC card [4].

Despite the ability of DPDK to provide line-rate packets, the compute-intensive network function's performance was way below the line-rates [5], [6]. In order to achieve a packet processing speed closer to line-rates, researchers have looked into the possibility of using hardware accelerators like Graphics Processing Units (GPUs) [2], [5], [6], [7], Field Programmable Gate Array (FPGA) [8], Heterogeneous System Architecture (HSA) [9], etc. for deploying NFs.

Prior works [2], [5], [6], [7] demonstrated that GPU acceleration improves compute-intensive NF throughput manifolds. However, the discrete GPUs used by these works are expensive and consume a fair amount of energy. Another problem with the discrete GPU is that it requires data to be copied in the GPU memory from CPU memory via the PCI bus. This data copy over PCI increases end-to-end latency and lowers the throughput. Heterogeneous System Architecture(HSA) [10] solves this inherent drawback of discrete GPU. HSA combines both CPU and GPU on the same bus, having shared memory. Since memory is shared, there is no explicit requirement to copy data in the GPU memory.

The vanilla DPDK provides zero-copy [11] between NIC and CPU but does not provide zero-copy between NIC and GPU. Vanilla DPDK copies the packet directly into the application's memory, i.e., it avoids the extra copy from OS kernel memory to the user memory region, hence called zero-copy. APUnet [9] proposed an HSA based solution that modifies DPDK in order to support zero-copy between NIC and HSA-

978-1-6654-9956-9/22/$31.00 ©2022 IEEE
DOI 10.1109/CCGrid54584.2022.00060

GPU. APUnet demonstrated the efficacy of its solution by experimenting with single NFs on AMD HSA.

Though APUnet [9] solution is suitable for a single NF, in real-world deployments, multiple interconnected NFs called NF chains (NFC) are often required to provide certain network features to a network user. In the case of an NF chain, the output of one NF becomes an input to another. Broadly, there are two types of NFC: static and dynamic. In a static chain, every network flow must go through all the NFs that form the chain. In dynamic chaining, different network flows can go through different sets of NFs.

Dynamic chaining is essential for upcoming networks such as 5G [12]. 5G offers varied features for different use cases, and each of these features is enabled by an NF, and multiple such NFs are chained to provide a set of features to the users. For example, in 5G, applications such as vehicular traffic, video streaming, AR/VR, etc., require different sets of services, bandwidths, authorizations, etc., requiring different NF processing(not necessarily disjoints). APUnet [9] cannot provide such services, as any chaining is not considered while designing the APUnet. Due to its design, APUnet does not allow more than one NF execution on the GPU, which becomes a significant bottleneck for NFCs having multiple compute-intensive NFs. With only one NF on the GPU, other NFs of the NFC get offloaded to the CPU, and the CPU does not perform well for compute-intensive NFs, thus lowering the throughput of the whole chain. Furthermore, APUnet is not OpenCL [13], [14] compliant. OpenCL is an open and cross-platform framework that provides API to write applications for HSA so that applications written for one vendor platform can be executed on other vendor platforms. Though the OpenCL standard provides API, hardware vendors implement those APIs for their platforms. Since APUnet is not OpenCL compliant, there is no guarantee whether data synchronization across GPU and CPU will work on all HSA because internal hardware implementation may differ. Additionally, APUnet uses Mellanox APIs and modify DPDK for zero-copy between NIC and GPU, which tied the solution to only one vendor NIC.

This paper presents Hummingbird— an HSA based framework with dynamic NFV chaining support to address the aforementioned challenges. Hummingbird is OpenCL 2.0 [14] compliant and can easily be ported to all DPDK compatible NICs. Furthermore, Hummingbird supports spatial sharing of GPU among NFs. Spatial sharing enables Hummingbird to host multiple NFs on the GPU. Since compute-intensive NF performance gets enhanced by the GPU, allowing multiple such NFs on the GPU improves the throughput of the NFC consisting of multiple compute-intensive NFs.

Our solution also supports the dynamic movement of network packets among the NF chains based on the predefined policy for a given network flow. Hummingbird achieves this by abstracting lower-level details of forwarding and synchronization. An application NF does not need to be aware of where the next NF would be, i.e., whether the next NF would be on CPU or GPU (NFs are assigned CPU/GPU statically at the initialization of the framework). Modules for synchronization and forwarding will take care of respective lower-level details. This feature enables network providers to statically assign NFs to either compute type (CPU/GPU) based on their requirement, i.e., compute-intensive NF should be assigned GPU and non-compute-intensive NF should be assigned CPU. This performance-based individual NF assignment allows Hummingbird to achieve the best possible NFC throughput. The following are the contribution of this paper:

- We design and develop a new framework *Hummingbird* that enables the deployment of static and dynamic network function chains over HSA.
- We identify the challenges in creating a lightweight and OpenCL 2.0 compliant data synchronization mechanism for network IO across CPU NFs and GPU NFs.
- We designed and developed a persistent GPU kernel-based technique to space-share GPU among multiple NFs for throughput enhancement.
- We modify DPDK to extend its zero-copy feature to HSAs to enable efficient packet delivery between NIC and GPU.

The rest of the paper is organised as follows. Section II presents the background and motivation. Section III describes challenges and solution requirement. Section IV present the design and implementation of our solution. Section V discusses the evaluation to prove the efficacy of our work. Section VI contains related work and section VII concludes the paper.

II. Background and Motivation

A. Network Function Virtualization (NFV)

Network functions(NFs) are well-defined building blocks of any network infrastructure. They perform specific operations on network packets. NF such as IPv4 router forwards the packet based on destination IP address. Another NF, network intrusion detection system(NIDS) inspects the packet for any intrusion. All these NFs perform a specific task. Generally, network administrators install different NFs to provide multiple functionalities to network users. Since packet traverses from one NF to another NF such that the output of one NF becomes an input to another NF, it forms a chain called network function chain. Traditionally, proprietary hardware boxes called ASICs were performing these NF operations. However, the deployment of such boxes is expensive and requires specialized networking. Compared to proprietary ASIC deployment, NFV provides various benefits to network providers. NFV enables NFs to use commodity hardware which is far less expensive compared to ASICs, thus reducing CAPEX [1], [15]. Furthermore, NFVs can be configured and managed easily, reducing operational cost [1]. Additionally, virtualization offers easier addition and removal of functionalities [15].

B. Data Plane Development Kit (DPDK)

Traditionally, all the packets received by the host must go through different layers of the operating system (OS) TCP/IP stack. The stack was designed long back, and at that time,

throughput was not the main concern [16]. During the last two decades, the speed of ethernet cards has increased from a few Mbps to 10-40 Gbps. Traditional TCP/IP stack cannot operate at this speed due to inherent drawbacks such as buffer allocation and deallocation on the reception of each packet. Prior work [2] found that allocation/deallocation consumes more than 50% of the CPU cycles as part of the packet reception process. Another problem is memory copy overhead between OS kernel space and userspace. The OS kernel creates a buffer in kernel space on every packet reception. Later, that buffer is copied to userspace so that the application can consume it. To solve the problems mentioned above, Intel came up with an SDK called Data Plane Development Kit (DPDK) [3]. DPDK pre-allocates the memory; hence it does not require per-packet allocation/deallocation. Another optimization that DPDK offers is that it delivers the received packet directly in userspace, avoiding the OS kernel space to user space memory copy overhead.

C. Graphics Processing Units (GPU)

The graphics processing units (GPU) were traditionally used to accelerate graphics based compute-intensive tasks such as image processing [17], 3D visualization [18], computer vision [19], etc. Now GPUs have grown into processing more generic tasks such as accelerating high-performance computing and scientific workloads. Given the performance benefits, researchers have tried to use graphics processing units for accelerating network functions [2], [5], [6]. GPUs can broadly be classified into two types:

Discrete GPU: Discrete GPUs are connected with the system via the PCI bus. These GPUs have a high memory bandwidth like GDDR5 and thousands of computing cores. However, the CPU must transfer the data from the main memory to the GPU memory to process any data. Hence, PCI bus speed limits the performance of IO-intensive applications over GPU.

Integrated GPU (iGPU)/Heterogeneous System Architecture (HSA): Integrated GPU means both CPU and GPU are on the same die sharing the same bus. This architecture allows the CPU to share its main memory with the GPU, removing the necessity to transfer data over the PCI bus. However, an application needs a virtual memory-sharing programming model to use shared physical memory. OpenCL standard 2.0 [14] enables this type of sharing over such architectures. Accelerated Processing Unit (APU) is a recent AMD HSA architecture with integrated GPU (iGPU). APU is cost-effective and power-efficient when compared to discrete GPU. However, APU has lesser computing cores, e.g., A12-9800 has only 512 cores in comparison to 3584 cores found in NVIDIA TITAN X Pascal (refer table I). Furthermore, APUs do not have high-speed GDDR memory. Instead, they have to contend with the CPU for DDR memory access. Although APU has advantages, it poses challenges for NF chain deployment. We describe these challenges and our proposed solution to address them in the next section.

TABLE I: Comparison of GPUs and APUs

GPU type	# of cores	Power usage	Price
NVIDIA Titan RTX	4608	280W	$2499
NVIDIA TITAN X Pascal	3584	250W	$1200
AMD APU A12-9800	512	65W	$138

III. Challenges and Solution Requirements

The choice of APU as the hardware platform for our work was motivated by the architectural advantage it provides in the form of shared main memory between CPU and GPU. However, designing a high-performance NFV chaining solution over APUs is challenging. The following are the challenges in designing and implementing such a solution.

A. CPU-GPU data synchronization overhead

When GPU is employed to process packets, a batch of packets is transferred to GPU memory, and then the kernel is launched. On completion, the kernel exits(tear down). The kernel needs to be relaunched to process the next batch of packets. This repeated kernel launch for processing every batch incurs enormous overhead [20]. Though the persistent kernel [21] technique alleviates the problem of kernel launch and tears down, the data copy overhead between CPU and GPU degrades throughput significantly. OpenCL 2.0 provides a shared virtual memory (SVM) programming model that allows data sharing between CPU and GPU to solve this.

Furthermore, this SVM model also provides a mechanism to synchronize data across CPU and GPU. However, synchronization requires the usage of atomic operations over the shared data. Extensive atomic operations on a given memory address, serialize all the GPU threads (for atomic operation) that try to access it. Hence, GPU's benefit over CPU in terms of hundreds of cores, executing threads in parallel diminishes as atomic operations and data size increases. Furthermore, data processed by the GPU must be synched at the CPU (via atomic operations) before the CPU accesses the data itself. In most cases, multicore GPU can produce faster than what a CPU core can consume, resulting in synchronization at the CPU being the bottleneck. A high throughput NFV chaining framework must resolve all the bottlenecks, including synchronization overheads for optimal throughput.

B. Need for multiple persistent kernels

In an NFV chain, offloading a compute-intensive NF on a CPU reduces the whole chain's throughput. Usage of GPU for such compute-intensive NF is proven to improve the performance [5], [6], [9]. However, if the chain consists of multiple compute-intensive NFs, the chain's throughput can be severely restricted by the compute-intensive NF executing on the CPU. Earlier research works supported offloading of only one NF to GPU [9]. Though GPU can be shared among numerous NFs using non-persistent kernels, this approach requires periodic kernel launch and termination overheads. One of our key ideas is to share GPU among multiple kernels (NFs) using the persistent kernel approach. The framework benefits from sharing GPU among multiple persistent kernels. Persistent kernels [21] eliminate per kernel launch and termination overhead, whereas

sharing the GPU among multiple NFs improves the chain throughput by allowing more compute-intensive tasks on the GPU.

C. Thread divergence due to dynamic chaining

Our framework also incorporated an NFV chaining feature called dynamic chaining that allows a packet to be routed based on either headers or its content. Since our framework employs GPU heavily for NF computation, it requires that threads should progress in a lock-step [22] manner within the workgroup for better performance. However, each thread might receive different packet sizes; hence some threads finish earlier than others, resulting in out-of-sync execution. In order to enforce lock-step execution for better performance, a barrier can be used. Furthermore, NF may also have these barriers built within. Usage of barriers enforces another challenge when dynamic chaining is enabled. Due to the dynamic chaining and unpredictable incoming traffic, some threads might not receive any packets (or receive late) for processing. Under such circumstances, threads receiving the packets will process them and hit the barrier. However, threads not receiving the packet will wait for packet reception. Since these threads do not hit the barrier unless they receive the packets, the whole workgroup waits for their progression, resulting in lower throughput.

D. Efficient packet copy to GPUs

DPDK is a de-facto standard for high-speed packet processing. Optimizations such as pre-allocation of buffer and zero-copy among NIC and CPU improve the packet IO drastically [23]. However, Vanilla DPDK does not provide zero-copy between NIC, CPU, and GPU. Hence to process packets on the GPU, packets first need to be copied from DPDK accessible memory to GPU accessible memory, then perform GPU operation and finally copy it back to DPDK accessible memory so that NIC can send the packet out. These two extra copies increase the memory contention, which further reduces the throughput [9]. To solve this memory contention problem for throughput improvement, we modified DPDK to provide zero-copy between NIC, CPU, and GPU.

IV. Design and Implementation

A. Hummingbird architecture overview

Figure 1 shows the architecture of our solution *hummingbird*. The main components of *hummingbird* are modified DPDK, GPU coordinator & scheduler module, route decider module, and packet forwarding module. The following are the details of each module.

1) Modified DPDK: Modified DPDK (mDPDK) provides zero-copy among NIC, CPU, and GPU. The NIC receives the incoming packet and copies it in shared virtual memory(SVM). OpenCL 2.0 enables SVM to allow data sharing between CPU and GPU via pointers. After packet copy in SVM, NIC writes the information regarding packet pointer in its memory-mapped registers. mDPDK polls (reads periodically) these registers. If a new packet pointer information appears, mDPDK fetches the packet using these pointers and returns it to the application. Since the packet resides in the SVM, both CPU and GPU can access it. In case the application transmits the packet, mDPDK first places the packet in SVM, then writes the packet pointer information in memory-mapped NIC registers, and at last updates the NIC register to transmit the packet. NIC copies the packet from SVM to hardware queues and transmits it on the wire.

2) route decider module: The path of packets is not static and can not be predetermined due to various functionality requirements for different packets. Generally, the network administrator (admin) configures what type of packet is processed by which set of NFs. The route decider module enables this dynamic chaining feature. The module accepts admin configuration in a comma-separated text file where each line contains packet type, current NF, and next NF. During framework initialization, the module loads the configuration in the form of a table. Both CPU and GPU-based NFs consult this module before forwarding packets to the next NF. CPU cores executing CNFs(CPU NFs) consult the module directly before forwarding the packet to the next NF, whereas GNFs(GPU NFs) delegate this task to CPUSyncer belonging to *GPU coordinator & scheduler module*.

3) packet forwarding module: This module creates an NF forwarding table and initializes it per NF information. Each row in the table contains the following information: NF type (NIDS/IPSec/Router), NF compute type (CPU/GPU), the number of cores assigned, and their receive(RX) queue. The working process of the module is as follows: 1) the module consults the *route decider module* to find the next NF for the packet. 2) it retrieves the address of the RX queue from the NF forwarding table by matching with the next NF and its compute type obtained in the previous step, and 3) the module enqueues the packet in the RX queue (obtained from step 2).

4) GPU coordinator & scheduler module: This module has two parts— one executes on the GPU, and the other executes on the CPU. Both submodules cooperate and synchronize data across GPU and CPU. Hummingbird creates one logical queue per kernel thread, which the GPU uses for in place packet processing. Additionally, Hummingbird creates a mapping of GPU-based NFs (GNFs) and kernel threads such that each GNF owns a group of kernel threads. The GPU submodule passes the packet pointer to the GNF responsible for that kernel thread for each unprocessed packet in the logical queue. The submodule enforces lock-step execution (via barriers) across kernel threads on packet processing completion within a workgroup. A workgroup (warp in terms of CUDA) is a set of kernel threads that executes on the same control unit (or SM in terms of CUDA). After passing the barriers, it informs the CPU submodule via a shared atomic variable. The CPU submodule polls (read continuously in a loop) these atomic variables, fetches the processed packets and calls the *router-decider module* for further processing.

We will take an example to showcase NFV chaining over Hummingbird. Step numbers in figure 1 explain one such example. As per example, *modified DPDK* first receives the

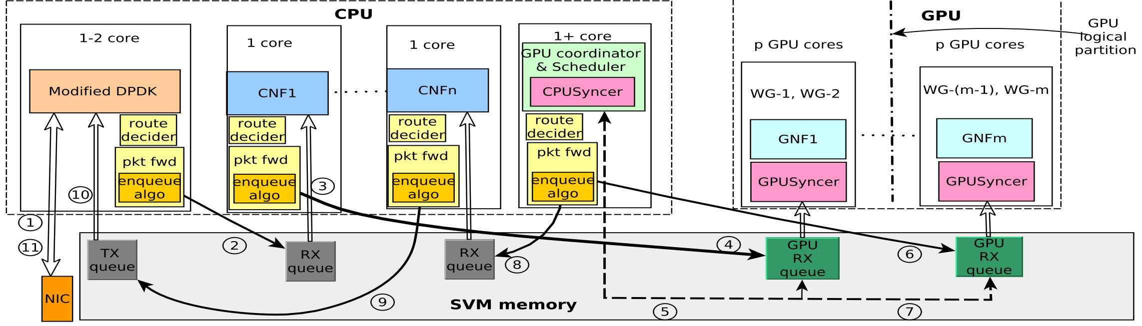


Fig. 1: Architecture of Hummingbird framework.

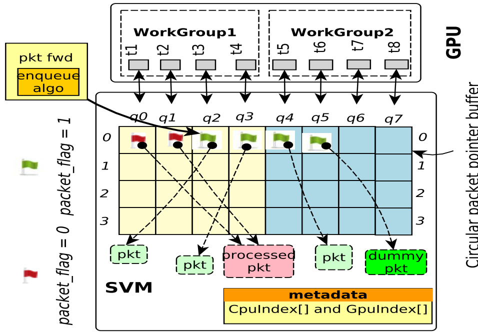


Fig. 2: Partition of a circular packet buffer into per thread queue. Where *qi-1* is assigned to *ti(thread i)*.

packet from NIC (step-1) and enqueues the packet (step-2) to *CNF1 (CPU NF)* for further processing (step-3). After processing, *CNF1* enqueues the packet for *GNF1 (GPU NF)* processing (step-4). The output of *GNF1* is synced (step-5) by the *GPU coordinator & scheduler* and packets are further enqueued (step-6) for *GNFm (GPU NF)* processing. Again *GPU coordinator & scheduler* syncs processed packets (step-7) and enqueues them for *CNFn (CPU NF)* (step-8). Next, *CNFn* enqueues the packet in the Tx queue (step-9) of *modified DPDK* and it retrieve packets from the Tx queue (step-10) and transmit them via NIC (step-11). Next few sections will explain different aspects of Hummingbird in detail.

B. Persistent thread kernel

To reduce the overhead of kernel launch/teardown for HSAs, we employed a persistent kernel, i.e., GPU thread does not terminate after processing. Instead, it loops continuously and waits for the next packet's arrival. Employing this technique means no more frequent kernel launch and tear down. *Hummingbird* employs a logical queue per GPU kernel thread for passing packet pointers to the GPU such that the CPU enqueues the packet in these queues, and each kernel thread process the packet from its queue only. However, kernel threads are hundreds in number, so managing so many queues becomes complex. In most cases, hundreds of GPU cores are faster than a CPU core; hence even a slight complexity can create a bottleneck at the CPU. To reduce the complexity, we created a single big circular queue in SVM and created a logical abstraction of the queues for each thread, as shown in figure 2. We modeled the queue as a 2D array such that each contiguous *KernSize* index forms a row where *KernSize* is the total number of kernel threads. The first cell of each row becomes the first logical queue (q0), the second cell of each row forms the second logical queue(q1), and the last cell of each row forms the last logical queue. As shown in the figure 2, each kernel thread can access packet pointers from the designated logical queue, i.e., t1 can access packet pointers from q0, t2 from q1, etc. The advantage of using a single queue is that packet enqueue operation becomes simple as packets can be enqueued at the next available index of the queue.

C. CPU-GPU synchronization and GPU logical partitioning

With the employment of persistent kernel, inherent CPU-GPU synchronization point (kernel launch and tear down) provided by the OpenCL[14] no longer exists. Hence to solve the challenge of data consistency, we leveraged the APU architecture. We have devised a new OpenCL 2.0 compliant algorithm that uses minimal atomic operations and access data consistently. Overall, we reduced thread serialization by the factor of workgroup size. The algorithm is in two parts, i.e., one for the CPU and the other for the GPU. The algorithm not only synchronizes packets across GPU and CPU but also incorporates various features for throughput improvement, such as logical partitioning (spatial sharing) of GPU to enable multiple NFs on the GPU and the ability to scale up CPU resources for the CPUSyncer. The following section discusses the design challenges of the algorithms, but first, we will describe a few keywords used in the algorithm. A workgroup is a collection of threads running on a control unit(CU) in a lock-step manner. Collection of such workgroups form a kernel size (*KernSize*), i.e., the total number of GPU kernel threads. Both workgroup size(*workGroupSize*) and the number of workgroups (*numOfWorkGroups*) are predefined and fixed at the kernel launch.

D. Addressing the design challenges

Since atomic instruction usage for synchronization between the CPU and the GPU is mandatory for the algorithm to remain

Algorithm 1 - GPUSyncer: Sync algorithm executed by each GPU thread.

```
1: localIndex ← 0
2: lastReadIndex ← 0
3: {CpuIndex and GpuIndex is atomically shared between CPU and GPU}
4: while true do
5:     while lastReadIndex = localIndex do
6:         Use only first thread of the workgroup to atomically store
           CpuIndex for the workgroup in localIndex.
7:     end while
8:     Barriers()
9:     Loop until Queue[thread_id][localIndex].packetFlag is not set
10:    NFVprocessing(Queue[thread_id][localIndex].packet)
11:    Reset Queue[thread_id][localIndex].packetFlag
12:    increment lastReadIndex
13:    atomically increment GpuIndex for the workgroup in first thread of
       the workgroup.
14:    Barriers()
15: end while
```

Algorithm 2 - CPUSyncer: Sync algorithm running on CPU

```
1: {CpuIndex and GpuIndex is atomically shared between CPU and GPU}
2: while true do
3:     for each work group do
4:         atomically store GpuIndex of the workgroup in localGpuIndex
5:         atomically store CpuIndex of the workgroup in localCpuIndex
6:         while localGpuIndex ≠ localCpuIndex do
7:             for i ← 0 to workGroupSize do
8:                 if Queue[i][localCpuIndex].packetFlag is not set then
9:                     Send Queue[i][localCpuIndex].packet to packet for-
                       warding module
10:                    Queue[i][localCpuIndex].packet ← dummy packet.
11:                end if
12:            end for
13:            increment localCpuIndex
14:            atomically increment CpuIndex of the workgroup
15:        end while
16:    end for
17: end while
```

OpenCL 2.0 compliant; hence we cannot remove it altogether. However, the higher the number of atomic operations, the worse the performance; hence we thought of reducing them. We designed GPUSyncer to reduce atomic operations by a factor of workgroup size by performing atomic operations in only the first thread of the workgroup (instead of per-thread) and then syncing it across all the threads in the workgroup via barriers. Barriers also helped in enforcing lock-step execution. Without barriers, threads would diverge at line #6 and #13 (GPUSyncer) due to conditional statement and may process stale data at line#10 (GPUSyncer). CPUSyncer also uses atomic variables per workgroup. Furthermore, each workgroup uses a separate memory address for load and store operations, thus reducing the contention among multiple workgroups. We leverage workgroup-based processing to employ multiple NFs on the GPU to enhance the performance further. Since each workgroup processes data independently of other workgroups, they can parallelly execute different instructions without a performance loss. We statically assign a set of a workgroup to each GNF (GPU NFs) during application initialization and preserve the mapping in CPUSyncer. CPUSyncer uses this information while consulting *route decider* module to find the next NF for packets.

Another advantage of using workgroup based processing is that we can scale CPUSyncer to multicore. Our experiments showed that a single core CPUSyncer becomes a bottleneck when handling small packet sizes as DPDK receives more packets at small packet sizes than at large packet sizes. A multicore distributed CPUSyncer solves this problem. To achieve scaling, we logically bind a mutually exclusive set of workgroups to each CPUSyncer core such that each CPUSyncer core only sync packets in the queues belonging to kernel threads of the assigned workgroups.

The overall working of the algorithm is as follows: GPU thread takes the packet from its queue, checks for the packet flag, and if find it set then processes it, then resets the packet flag (algorithm 1), implying GPU has processed the packet. Once all the threads within the workgroup complete the processing, they increment the *GpuIndex* corresponding to the workgroup. The *GpuIndex* and *CpuIndex* are atomically shared variable between CPUSyncer and GPUSyncer, accessed via atomic operations only. GPUSyncer uses barriers to enforce lock-step execution to ensure that all the threads within the workgroup processed the packets. On the other hand, CPUSyncer parallelly polls *GpuIndex* of each workgroup. If *GpuIndex* digresses from *CpuIndex*, then it means new packets are available to be synched. CPUSyncer accesses each abstract queue corresponding to the workgroup, retrieves the packet if the packet flag is 0 (reset), and forwards it as per policy

Though barriers have many advantages, they become a problem when the dynamic chaining feature is enabled. To understand the effect of barriers in dynamic chaining, let us consider a scenario where some GPU threads within a few workgroups might not receive data for some time. It is undoubtedly possible that a particular NF does not get enough traffic to feed all the GPU thread, especially when policy (dynamic chaining) is steering most of the flows away from the NF. In such a scenario, barriers block the work group's progress until all the threads receive a packet. This delay reduces system throughput marginally. To solve this, we introduced a dummy packet. It works as follows: 1) CPUSyncer atomically reads the shared variable for a workgroup. If *GpuIndex* and *CpuIndex* of any workgroup differs, CPUSyncer retrieve packets from their queues. 2) For each packet CPUSyncer checks whether it is a dummy or not, 3) if it is dummy then does not do anything else it forwards the packet to the next NF using packet forwarding module and 4) for each actual packet, CPUSyncer places a dummy packet at the index from where the actual packet is retrieved. However, *packet forwarding module* can rewrite these indexes with actual packets while enqueuing new packets for GNF processing. Dummy packets will ensure that all the threads of a GNF progress. If a kernel thread does not have an actual packet, it has a dummy packet to process.

E. Dynamic Chaining of NFs

packet forwarding module and *route decider module* are responsible for dynamically forwarding packets to NFs queues.

CNFs call *packet forwarding module* and pass the packet pointer to it. However, GNFs delegate this task to CPUSyncer of "GPU coordinator & scheduler", which in turn calls the *packet forwarding module*. For each packet pointer, the *packet forwarding module* asks the *route decider module* regarding the next NF. *route decider module* looks up the prepopulated table as mentioned in the section IV-A2 and retrieves the following NF RX queue. *packet forwarding module* enqueues the packet in the RX according to the section IV-A3. For enqueuing, *packet forwarding module* uses CPU-CPU synchronization (section IV-F) if the next NF is a CNF, else it uses enqueuing method mentioned in section IV-C.

F. CPU-CPU sync and Enqueue Algorithm

NFs in the chain send packets to each other via a predefined queue (steps 2, 8, and 9 in figure 1). Producer NF passes the packet pointer via this queue to consumer NF. Furthermore, dynamic chaining makes it worse as multiple NFs may be enqueuing packets to the same queue. Since mutexes and semaphores consume too many cpu-cycles, Hummingbird solves both challenges by employing a separate queue between each pair of CPU cores. The queue uses a lock-free technique [24] to avoid race conditions.

G. Zero-copy via virtual shared memory

To facilitate zero-copy, DPDK first pre-allocates memory in userspace backed by hugepages (2MB or 1GB page size) during its initialization, and then it informs the NIC to use this userspace memory for packet IO. On packet reception, NIC copies the packet via DMA (Direct memory address) in this userspace memory and writes packet pointer information in memory-mapped registers. DPDK polls these registers and accesses the packet via packet pointer information. OpenCL 2.0 mandates that the GPU can access host (CPU) memory only when it is a shared virtual memory(SVM) allocated by clSVMAlloc() API. Since the DPDK memory allocator does not pre-allocate memory in the SVM region, packets copied by the NIC are not accessible to the GPU. There are two possible approaches to solve this problem: (1) Create a separate SVM region and copy each packet to/from between DPDK and SVM region. (2) Modify the DPDK to pre-allocate the memory in SVM and inform the NIC to use this SVM memory for packet IO. Though the first option is easier to implement, we opted for the second due to its inherent performance benefits as it removes extra packet copy overhead between non-SVM and SVM regions. Challenges in DPDK modification come from the DPDK code size (170K lines of code) and lack of documentation regarding the internals of the DPDK memory model. We went through the whole codebase of the DPDK to understand the memory model of the DPDK.

Another challenge with SVM memory is that it can not use hugepages, whereas the DPDK memory model is entirely dependent on hugepages. We solved this problem by modifying the DPDK memory stack to allow SVM memory without hugepages for packet IO. However, DPDK APIs still unmaps the packet from SVM on packet reception and returns a pointer

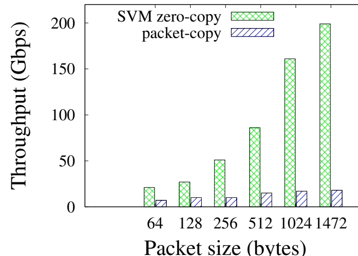


Fig. 3: Throughput: zero-copy (SVM) persistent kernel vs Packet-copy non-persistent kernel

that is only accessible to the CPU. On further investigation, we found that the DPDK calls mmap with a specific address on packet reception. It unmaps the packet from the SVM region and returns a packet pointer not accessible to the GPU due to mmap() [25] ability to replace previously mapped memory with newly mapped memory. To solve this, we further changed the packet processing stack of the DPDK to preserve the SVM memory mapping. Our modifications in the DPDK are generic and works on all DPDK compatible NICs. In comparison, APUNet [9] custom DPDK can only use Mellanox NICs.

V. Experimental Evaluation

In this section, we present the evaluation of the *hummingbird* framework. Our test setup consists of Ubuntu 14.04 (kernel 3.13.0-153) running on AMD APU Carrizo platform A12-9800 with 32GB RAM as a packet processing system. SVM features and GPU programming are enabled by OpenCL 2.0 (AMD-APP-SDK-v3.0). We Modified DPDK 18.08 for fast network IO. Our traffic generator is running on 4th gen Intel(R) Core(TM) i7-4790K (4.0 GHz) with 16 GB main memory running over Ubuntu 16.04 (kernel 4.4.0-142). Both systems are connected via 40Gbps Mellanox Connect X-4 ethernet cards (MCX414A-BCAT [26]).

A. Performance benefits of zero-copy

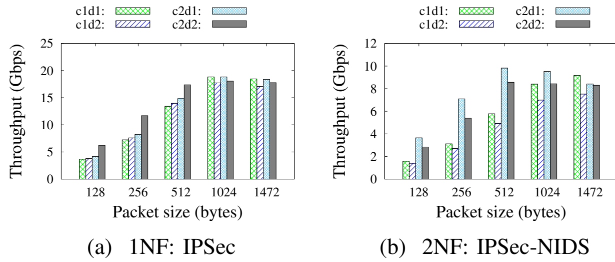
To measure the performance benefits offered by the zero-copy mechanism supported by modified DPDK, we wrote a synthetic program that measures and compares the performance metrics of zero-copy persistent kernel against traditional kernel launch without zero-copy between CPU and GPU. We observed that (refer to figure 3) zero-copy persistent kernel achieves ten times higher throughput when compared to traditional kernel launch without zero-copy. Furthermore, zero-copy persistent kernel throughput increases exponentially with an increase in packet size.

B. NFs used in NFV chaining

We have implemented IPSec and NIDS for GPU execution using OpenCL, whereas we implemented IPSec, NIDS, and

TABLE II: Algorithms used in NF implementation

NF	Algorithms
Router	DIR-24-8-BASIC [27]
NIDS	Aho-Corasick algorithm [28] (147 rules)
IPSec	AES-128 (CBC mode) [29] and HMAC-SHA1 [30]



(a) 1NF: IPSec (b) 2NF: IPSec-NIDS

Fig. 4: **Hummingbird:** NF chain performance in the various configuration of CPUSyncer and mDPDK where cxdy represents x cores for CPUSyncer and y cores for mDPDK.

Router(IP version 4) for CPU execution. Table II mentions the algorithms used for respective NF implementation. We did not implement Router over GPU because the Router is not compute-intensive, and CPU is preferable over GPU for such workloads. We reprogrammed Openssl [31] based AES-128 bit algorithm using OpenCL for IPSec GPU implementation.

C. Multicore CPUSyncer and multicore DPDK: Pros & Cons

We experimented with the various configuration on our APU. We varied CPUSyncer cores from 1 to 2 and mDPDK cores from 1 to 2 and analyzed the performance of Hummingbird in two scenarios: (1) 1NF: GPU executes a single NF, (2) 2NF: GPU executes two NF simultaneously.

1NF: We observed that multicore CPUSyncer improves the throughput in comparison to the single-core at smaller packet sizes (<1024B). Whereas for larger packets, GPU becomes the bottleneck; hence having multicore CPUSyncer does not improve or degrade the performance (c1d1 and c2d1 in the figure 4a). We also observed that mDPDK becomes the bottleneck (when CPUSyncer is not a bottleneck) at smaller packet sizes due to increased packet incoming rate. Hence, providing an extra core to mDPDK enhances the performance. mDPDK on a single core is not a bottleneck for large packets; hence adding an extra core does not improve the performance. However, it might reduce the performance due to cache bouncing [32] as packet hops across CPU cores(c2d1 and c2d2 in the figure 4a). In summary: for smaller packet sizes, c2d2 performs best, and for large packet sizes, c1d1 performs best. We performed the same experiments with NIDS as well and observed similar results.

2NF: In this experiment, we allocated GPU for two NFs and chained those two. In this scenario, having a multicore CPUSyncer enhances the throughput (c1d1 and c2d1 in figure 4b) for smaller packet sizes which is very similar to 1NF scenario. However, when two NFs parallelly utilize the same GPU, the GPU processing capacity gets divided among the NFs. Hence, GPU can not process the packets at the speed at which mDPDK is enqueuing, regardless of packet sizes. GPU's inability to process the packets at line rate results in packet drops, which exacerbates when the number of mDPDK cores increases, resulting in lower throughput. This performance can also be observed from the figure 4b where c1d2 consistently underperformed compared to c1d1. In summary, for smaller packet sizes, c2d1 performs best, and for large packet sizes, c1d1 performs best.

In subsequent sections, we are not showing and comparing performances under various configurations (c1d1, c2d2, c1d2, c2d2). Instead, we will show the best-performing configuration per packet size observed for the NF chain.

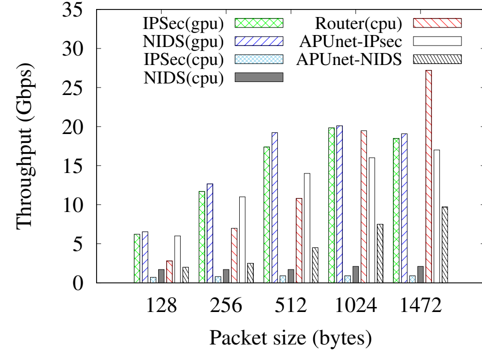


Fig. 5: NFs max throughput in our setup

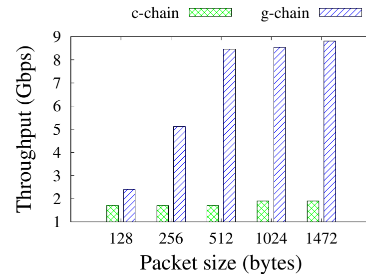


Fig. 6: Static three NF chain: Performance comparison when logical GPU partition(g-chain) is used vs not used(c-chain)

D. NFV baseline performance

This experiment aims to ascertain the maximum performance a NF can achieve in an environment where it does not have to share the resources with others. Only two NFs: IPSec [33], [34] and NIDS [28] can use GPU cores. Whereas all the other NFs can use CPU, including Router [27]. Figure 5 shows the throughput of these NFs on the respective compute. From figure 5, we can observe that the performance of NF over GPU far exceeds the performance of the identical NF on the CPU. Additionally, we measured the latency for NFs on both CPU and GPU and found that for compute-intensive NFs, the use of GPU reduces the average latency up to 5.9 times. Hence, GPU is beneficial for compute-intensive NFs deployment. Furthermore, Hummingbird's performance is better in comparison to APUnet[1]. In the rest of the paper, IPSec always executes on the GPU.

E. Static NFV chaining: Benefit of GPU partitioning

We compared two different chains to analyze the performance benefits of spatial partitioning of GPU. Both chains consist of three NFs (IPSec, NIDS, and Router), and the NF

[1]APUnet data is taken from APUnet [9] paper directly. Since APUnet code is not open-source, we could not run it on our platform to extract the results

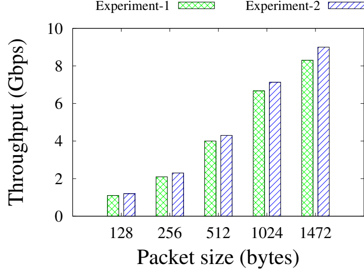
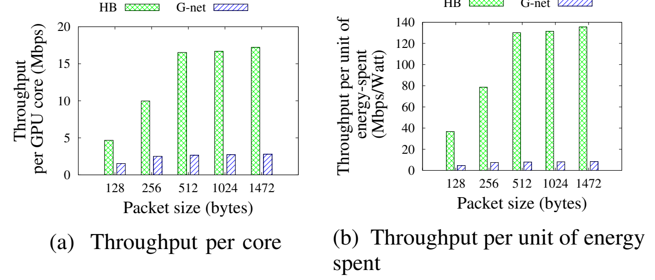


Fig. 7: Hummingbird: dynamic NFV chaining



(a) Throughput per core

(b) Throughput per unit of energy spent

Fig. 8: Hummingbird (HB) vs G-net performance

order in the chain is fixed such that packets first go to IPSec, then NIDS, and then finally Router(IPv4). Though NF order in both the chain is the same, they differ in compute type used for NIDS. One chain uses our logical GPU partition and executes IPSec and NIDS on the GPU, whereas the other chain uses GPU for IPSec only and CPU for the rest of the two NFs. Let us say the chain that uses logical GPU partition and executes two NFs on the GPU is called *g-chain* and the other chain with only IPSec on the GPU is called *c-chain*. We evaluated these two chains on the Hummingbird, and we observed that the throughput of *c-chain* almost remains constant with an increase in packet size(figure 6). In comparison, the throughput of the *g-chain* increases with an increase in packet size (as shown in figure 6). Furthermore, the throughput of the *g-chain* surpasses the *c-chain* regardless of the packet size. For example, at a packet size of 1472B, the *g-chain* achieves 8.46Gbps, whereas the *c-chain* achieves 1.9Gbps. The *c-chain* underperforms because compute-intensive NIDS becomes a bottleneck as it is using a CPU core. Whereas, if the same NIDS uses half of the GPU, we can achieve 4x performance improvement for the chain.

F. Dynamic NFV chaining over HSA

This section aims to analyze the overhead incurred due to dynamic chaining. Dynamic chaining enables dynamic routing of the packet based on a policy, i.e., different packets may follow different NF chains if the policy defines so. We defined a policy based on the IP header for dynamic chaining such that each traffic flow follows a different NF chain. Source IP, source port, destination IP, destination port, and protocol define a traffic flow. Change in any one of these five fields creates a separate traffic flow. In this set of experiments, sender is transmitting two flows: flow *f1* having *ip1* as destination IP and flow *f2* having *ip2* as destination IP. We configured Hummingbird to process the flow *f1* by the chain of IPSec(GPU)-NIDS(GPU)-IPv4(CPU) and flow *f2* by the chain of IPSec(GPU)- IPv4(CPU). To analyze the overhead of dynamic chaining caused by dummy packets, we varied the percentage of these two flows in the experiments. In first experiment, flow *f1* is 50% of the total traffic and flow *f2* is 50% of the traffic. In second experiment, flow *f1* is 93% of the total traffic and flow *f2* is 7% of the traffic. Figure 7 shows the performance of dynamic chaining for both experiments. From figure 7, we observed that the throughput of the second experiment is marginally better than the first experiment regardless of the packet size because the second experiment consists of a higher proportion of *f1* flow, thus requires fewer dummy packets. It implies that the higher number of dummy packets in the system, the lower the throughput as dummy packets do not contribute to the actual traffic but still consume CPU and GPU resources. However, the overhead is very low as the throughput gap between the two experiments is very narrow (0.7 Gbps).

G. Power and cost advantage of NFV chaining over HSA

This section highlights the performance of Hummingbird with its peer (G-net [7]). Since the GPU used by Hummingbird and G-net is different, we compare both in terms of throughput obtained per core or throughput obtained per unit of energy consumption. In terms of per-core throughput, Hummingbird outperforms G-net as shown in the figure 8a. Hummingbird achieves six times throughput per core (for 1472 byte packets) compared to G-net (figure 8a). When it comes to power consumption, Hummingbird again outperforms G-net(figure 8b). Based on our evaluation, we found that Hummingbird is 3.5 times more power-efficient for packet processing when compared to G-net. Hummingbird performance comes from various optimizations discussed in the design and implementation (section IV).

VI. Related Work

NF chaining frameworks for CPU: Click [35] was the first work to demonstrate usage of NFV for deploying a simple router. Recent high performance network processing frameworks such as NetVM [36], OpenNetVM [37] bypasses TCP/IP stack to provide line-rate processing. NetVM [36] and OpenNetVM [37] proposed a framework of dynamic NF chaining over multiple CPUs to achieve packet processing at high speed.

NF chaining frameworks using GPU: PacketShader [2] main aim was to improve throughput by accelerating it using GPUs. It leveraged the stateless nature of packet processing by software routers to execute them in parallel over GPU. SSLShader [6] leverages GPU to accelerate cryptographic computation with the help of thousands of cores. Similar to SSLShader, Kargus [5] employs GPU for accelerating intrusion detection systems, but it also balances the workload between CPU and GPU.

Tseng, Janet et al. [38] used integrated GPU to accelerate open-vswitch. However, their work focuses on open-vswitch

instead of a generic NFV framework. G-net [7] framework uses a unique GPU with HYPERQ technology to spatially share GPU among multiple NF chains. In terms of goal, G-net [7] is closer to our work, but it uses costly and power-hungry discrete GPU. Furthermore, G-net does not employ CPUs for NF processing. Though APUNet [9] employed similar HSA, it differs from us in architecture, solution, and goals. APUnet does not support NFV chain deployment. Furthermore, APUNet custom DPDK uses Mellanox APIs; hence the solution is tied to a specific vendor. On the other hand, our modified DPDK is generic and can be used with any DPDK supported NIC. Additionally, APUNet leveraged the LRU algorithm for synchronization, which is not guaranteed to work for all HSA because the solution is not OpenCL 2.0 compliant. We developed a novel OpenCL 2.0 compliant GPU-CPU cooperative synchronization mechanism for fast and consistent data access. Hummingbird additionally offers NFV chaining over HSA and incorporates various techniques such as dynamic chaining and scheduling multiple NFs on GPU to improve the performance.

VII. Conclusion

We designed and developed Hummingbird, high throughput and efficient framework for dynamic NFV chaining over APUs. Hummingbird performs well despite utilizing GPU with fewer cores (APU) that require lesser power and is affordable. Usage of such a framework in data centers by cloud providers can provide environment-friendly computing without compromising performance. However, there were many challenges while designing Hummingbird, which we solved through modified DPDK, persistent kernel, a novel distributed CPU-GPU synchronization algorithm, logical GPU partitioning, and dummy packet usage for stall-free lock-step execution. Our experimental evaluation demonstrated that these techniques helped Hummingbird achieve good performance and process more bytes per core than its contemporaries.

References

[1] ETSI, “Network Functions Virtualisation, An Introduction, Benefits, Enablers, Challenges & Call for Action. Issue 1,” http://portal.etsi.org/NFV/NFV_White_Paper.pdf, [accessed: 29-Sep-2021].

[2] S. Han, K. Jang, K. Park, and S. Moon, “Packetshader: A gpu-accelerated software router,” in *SIGCOMM ’10*. ACM, 2010.

[3] “Part 1: Architecture overview,” https://doc.dpdk.org/guides/prog_guide/overview.html, accessed: 08-March-2008.

[4] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, “Softnic: A software nic to augment hardware,” EECS Department, UCB, Tech. Rep., 2015.

[5] M. A. Jamshed, J. Lee, S. Moon, and et al., “Kargus: A highly-scalable software-based intrusion detection system,” in *CCS ’12*. ACM, 2012.

[6] K. Jang, S. Han, S. Han, and et al., “Sslshader: Cheap ssl acceleration with commodity processors,” in *NSDI’11*. USENIX Association, 2011.

[7] K. Zhang, B. He, J. Hu, and et al., “G-net: Effective gpu sharing in nfv systems,” in *NSDI’18*. USENIX Association, 2018.

[8] B. Li, K. Tan, L. L. Luo, and et al., “Clicknp: Highly flexible and high performance network processing with reconfigurable hardware,” in *SIGCOMM ’16*. ACM, 2016.

[9] Y. Go, M. Jamshed, Y. Moon, and et al., “Apunet: Revitalizing gpu as packet processing accelerator,” in *NSDI*. USENIX Association, 2017.

[10] S. Mukherjee, Y. Sun, P. Blinzer, and A. K. Z. et al., “A comprehensive performance analysis of hsa and opencl 2.0,” in *ISPASS*. IEEE, 2016.

[11] D. Stancevic, “Zero copy i: User-mode perspective,” https://www.linuxjournal.com/article/6345, 2003, accessed 17-Sep-2021.

[12] J. M. Batalla, G. Mastorakis, C. X. Mavromoustakis, and et al., “Network services chaining in the 5g vision,” *IEEE Communications Magazine*, vol. 55, no. 11, pp. 112–113, 2017.

[13] “Opencl overview,” https://www.khronos.org/opencl/, [accessed: 21-Sep-2018].

[14] Khronos OpenCL Working Group, *The OpenCL Specification*, 2015, [accessed: 29-Sep-2018]. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf

[15] VMware, “Network functions virtualization (nfv),” https://www.vmware.com/topics/glossary/content/network-functions-virtualization-nfv, 2021, [accessed: 20-Sep-2021].

[16] V. G. Cerf and E. R. Cain, “The dod internet architecture model,” *Comput. Networks*, vol. 7, pp. 307–318, 1983.

[17] H. Scheidl, “Gpu image processing using opencl,” https://towardsdatascience.com/get-started-with-gpu-image-processing-15e34b787480, 2018, [accessed: 29-Sep-2021].

[18] Renderpool, “Cpu vs. gpu rendering: Which is best for your studio projects?” https://renderpool.net/blog/cpu-vs-gpu-rendering/, 2020, [accessed: 29-Sep-2021].

[19] J. Fung and S. Mann, “Using graphics devices in reverse: Gpu-based image processing and computer vision,” in *2008 IEEE International Conference on Multimedia and Expo*, 2008.

[20] D. Lustig and M. Martonosi, “Reducing gpu offload latency via fine-grained cpu-gpu synchronization,” in *HPCA ’13*. IEEE, 2013.

[21] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style gpu programming for gpgpu workloads,” in *InPar’12*. IEEE, 2012.

[22] N. Brunie, C. Collange, and G. Diamos, “Simultaneous branch and warp interweaving for sustained gpu performance,” in *ISCA ’12*. IEEE Computer Society, 2012.

[23] T. Barbette, C. Soldani, and L. Mathy, “Fast userspace packet processing,” in *ANCS’15*. IEEE, 2015.

[24] H. Sundell and P. Tsigas, “Lock-Free and Practical Deques using Single-Word Compare-And-Swap,” Chalmers University of Technology and Göteborg University, Department of Computing Science, Tech. Rep., 2004.

[25] The Open Group Base Specifications Issue 7, “mmap - map pages of memory,” https://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html, 2018 Edition.

[26] “ConnectX-4 Lx EN Cards,” https://www.mellanox.com/page/products_dyn?product_family=219&mtag=connectx_4_lx_en_card.

[27] P. Gupta, S. Lin, and N. McKeown, “Routing lookups in hardware at memory access speeds,” in *INFOCOM ’98*. IEEE, March 1998.

[28] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Commun. ACM*, vol. 18, pp. 333–340, Jun. 1975.

[29] S. Frankel, R. Glenn, and S. Kelly, “The AES-CBC Cipher Algorithm and Its Use with IPsec,” Internet Requests for Comments, Network Working Group, RFC 3602, 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3602.txt

[30] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” Internet Requests for Comments, Network Working Group, RFC 2104, 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2104.txt

[31] OpenSSL, “Tls/ssl and crypto library,” https://github.com/openssl/openssl, 2019, [accessed: 01-Mar-2019].

[32] S. Han, “System design for software packet processing,” EECS Department, UCB, Tech. Rep., 2019.

[33] S. Kent, “Ip encapsulating security payload (esp),” Internet Requests for Comments, RFC Editor, RFC 4303, 12 2005.

[34] H. Dhall, D. Dhall, S. Batra, and P. Rani, “Implementation of ipsec protocol,” in *ICACCT*, Jan 2012, pp. 176–181.

[35] R. Morris, E. Kohler, and J. Jannotti et al., “The click modular router,” *SIGOPS Oper. Syst. Rev.*, vol. 33, pp. 217–231, Dec. 1999.

[36] J. Hwang, K. K. Ramakrishnan, and T. Wood, “Netvm: High performance and flexible networking using virtualization on commodity platforms,” in *NSDI’14*. USENIX Association, 2014.

[37] W. Zhang, G. Liu, and W. Zhang et al., “Opennetvm: A platform for high performance network service chains,” in *HotMIddlebox*. ACM, 2016.

[38] J. Tseng, R. Wang, J. Tsai, and et al., “Accelerating open vswitch with integrated gpu,” in *KBNets ’17*. ACM, 2017.