Humboldt-Universität zu Berlin

Bachelor Thesis

Offloading network packet classification to GPUs using CUDA and dpdk

David Schramm

1. Reviewer Prof. Dr. Florian Tschorsch
Lehrstuhl für Technische Informatik
Humboldt-Universität zu Berlin

2. Reviewer Prof. Dr.-Ing. Eckhard Grass
Drahtlose Breitbandkommunikation
Humboldt-Universität zu Berlin

March 9, 2022

David Schramm
Offloading network packet classification to GPUs using CUDA and dpdk
Bachelor Thesis, March 9, 2022
Reviewers: Prof. Dr. Florian Tschorsch and Prof. Dr.-Ing. Eckhard Grass

Humboldt-Universität zu Berlin

Institut für Informatik
Unter den Linden 6
10099 and Berlin

Abstract

This thesis is about the use of CUDA and dpdk to offload network packet classification from CPUs to GPUs and using the GPU's architecture to accelerate certain classification algorithms. It evaluates different implementation variants and compares it to standard classification approaches of dpdk running on a CPU; using real hardware. It comes to the conclusion that GPU-based classifiers scale well on large rulesets and offer themselves to use them for these.

Abstrakt

Diese Bachelorarbeit befasst sich mit der Benutzung von CUDA und dpdk um Netzwerk-Paketklassifikation auf diese auszulagern und die Architektur von GPUs auszunutzen um bestimmte Klassifikationsalgorithmen zu beschleunigen. Es werden verschiedene Implementationen ausgewertet und diese mit den Standard-Klassifikationsalgorithmen von dpdk, welche auf CPUs laufen, auf echter Hardware verglichen. Sie kommt zu dem Schluss, dass GPU-basierte Klassifizierer sehr gut mit großen Regelsätzen skalieren und sich anbieten, sie dafür einzusetzen.

Acknowledgement

I would like to thank Dr. Sven Hager for bringing me to the idea of writing this thesis and providing helpful feedback. I would also like to thank my family and friends for their support which allowed me to focus on this work the whole time.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Thesis Structure . . . 3

2 Background **5**
- 2.1 Network Packet Classification . . . 5
  - 2.1.1 Bitvector classification algorithm . . . 6
- 2.2 Data Plane Development Kit . . . 10
- 2.3 GPUs . . . 15
  - 2.3.1 SIMT . . . 15
  - 2.3.2 CUDA . . . 15

3 Implementation **21**
- 3.1 Functionality . . . 21
  - 3.1.1 Kernel . . . 22
  - 3.1.2 Architecture . . . 28
- 3.2 Kernel Launching Types . . . 29
  - 3.2.1 On-Demand Kernel Launching . . . 29
  - 3.2.2 Asynchronous Kernel Launching . . . 30
  - 3.2.3 Persistent Kernel . . . 31
- 3.3 Computation Versions . . . 33
  - 3.3.1 k-ary search . . . 33
  - 3.3.2 SIMD . . . 35
  - 3.3.3 Bigger Burst Size . . . 35
- 3.4 Impact of memory type . . . 36

4 Evaluation **39**
- 4.1 DPDK ACL . . . 39
- 4.2 Classification Kernel . . . 39
- 4.3 Rule Generation . . . 40
- 4.4 Packet Generation . . . 40

4.5 Benchmark 41

5 Conclusions **51**
5.1 Offloaded Classification on GPUs 51
5.2 Future Work 52

Bibliography **55**

A Appendix **61**
A.1 Source Code 61
A.2 Hash of PDF 61

1 Introduction

1.1 Motivation

Today, in times of cloud computing and virtualization, the need for high-performance networking and the resulting fast network packet processing is greater than ever before. The rising speed of network connections results in more workload of tasks like routing, encryption, classification and filtering of network packets on servers and especially firewalls.

One of the most common tasks is network packet classification, where packets are compared against a set of rules consisting of ranges in multiple fields. They are then processed further based on the matched rules; mostly the one with the highest priority [LS98].

The classical approach for such task is to run the corresponding algorithm on the CPU. But the higher the packet rates become, the less computational capacity of the CPU is left for the actual main applications which are also executed on it. Thus, it is popular to offload network related workload to dedicated devices like co-processors, NPUs (Network Processing Unit) or the NICs itself and therefore relieve the CPU. So called SmartNICs are using mostly ASICs and FPGAs and, as of late, SoCs with ARM architecture [LS98] to take common work, like QoS, checksum calculations, segmentation/reassembly, encryption, but also flow classification and filtering off from CPUs. They deliver great performance on high packet rates but come with a high cost and are mostly less flexible. Another approach offloads packet processing to GPUs. Their SIMT (see 2.3.1) (Single Instruction Multiple Threads) architecture allows the process of a packet burst in parallel. This is particularly useful for classification algorithms which profit from highly parallelization and where the corresponding implementation using FPGAs or ASICs would be too expensive or utterly unviable.

1.2 Problem Statement

The most part of traditional packet processing on CPUs is done in kernel space and is interrupt-driven; which is less efficient (see 2.2). Thus frameworks exist for offloading it to user-space applications permitting faster and more efficient receiving of packets in bursts by polling them directly from the buffers of the NICs. They provide a variety of algorithms for different tasks and therefore allow creating of well adapted processing stacks. One popular framework is the Data Plane Development Kit (see 2.2). Its provided algorithms do run on the CPU and fully exploit their capabilities like SIMD (Single Instruction Multiple Data) or encryption intrinsics [@22s]. Especially classification algorithms benefit from the use of SIMD, since it enables comparisons of multiple rules or fields against one packet in parallel, which results in faster lookups and higher packet rates.

The difference between SIMD and SIMT is that SIMD requires data locality for an issued instruction and is therefore limited by the processors register size [Wik22]. Using SIMT (see 2.3.1), one instruction is executed on multiple threads where data locality is no must, providing more flexibility. Also, much more threads can be executed on GPUs in parallel which allows much higher data throughput than SIMD.

Most research in this area focuses on using GPUs for other tasks (routing, encryption or string pattern matching) than packet classification based on arbitrary fields. And the greatest portion of works [Qu+15; ZSP14; Han+10], which address packet classification, do not benchmark their developed applications in real environments. This makes it hard to compare against other established solutions, like SmartNICs.

The APUnet [You+17] paper also uses dpdk and compares integrated with external GPUs in terms of efficiency. It considers the PCIe bus as an major factor which decreases the throughput of external GPUs. It implements many algorithms but none for classification. It uses OpenCL instead of CUDA and is therefore limited by the few provided optimization possibilities. The goal of this thesis is to use dpdk and CUDA for developing a network packet classifier running on GPUs which can be used for stateless packet filtering and compare its performance against CPU-only approaches in terms of packet rate and computational efficiency.

1.3 Thesis Structure

Chapter 2

The background chapter provides the necessary definitions and knowledge to generally understand, what packet classification (see 2.1) is, how and why dpdk (see 2.2) is used to accelerate packet processing and how the architecture (see 2.3) of a GPU works.

Chapter 3

The implementation chapter explains how a GPU-based classifier, which uses the bitvector search algorithm, is implemented (see 3.1). It explains different launching types (see 3.2), which are used in a stateless layer 2 firewall, and advanced optimizations (see 3.3.1 and 3.3.2).

Chapter 4

The evaluation chapter benchmarks different versions of a stateless layer 2 firewall (see 4.5) and compares its performance with a classifier provided by dpdk (see 4.1).

Chapter 5

The conclusions chapter summarizes the results of the evaluation chapter and tries to answer whether offloaded packet classification to GPUs is worthwhile or not.

2 Background

2.1 Network Packet Classification

Network packet classification is a well known problem for which many solutions exist. The problem is simple: finding the matching rule with the highest priority for a given packet out of a set of rules. However, there is no perfect solution which stands out amongst all other in all aspects. Often, the algorithms, which provide the highest lookup rates [GM99], have poor rule update rates or need a great amount of memory. But memory costs are high in many environments, especially when the classification is hardware-based and depends on particular memory types, like TCAM or CAM [Zhe+06]. Many algorithms do not scale well while the number of rules increases. Either the lookup time grows to an unjustifiable time or the search structure gets too big [GM99; SSV99; Sin+03]. The bitvector classification algorithm[LS98] and its variants provide a good ratio in terms of lookup/update time and big rulesets.

Network Packet A typical network packet consists of two types of data: control and user. Commonly, the control data is well known formatted in fields of fixed size and lays at the start, it is also referred as the *header*. The user data, or mostly named as the *payload*, has no strict structure and is appended to the header. The rules do only consist of fields in the header, thus the payload is generally not important for classification.

Header fields are discrete numerical values of a fixed size and position inside the header.
Let $h_i \in D_i$ be the i-th header field with a finite set $D_i = [0, u_i],\ u_i \in \mathbb{N}_0$.
We call $\mathbf{h} = (h_1, ..., h_n)$ a header with $dim(h) = n$ fields.

Rules and Rulesets Rules usually consider subsets of all available fields for an header $\mathbf{h}$ and consist of ranges.
Let $\mathbf{h}$ be a header where *all* $dim(\mathbf{h}) = n$ fields are referred by the rules.
We call $\mathbf{r} = (r_1, ..., r_n)$ with $r_i = [a_i, b_i] \subseteq D_i$ and $a_i, b_i \in \mathbb{N}_0$ a **rule** for the header $\mathbf{h}$.

A rule $\mathbf{r} \subseteq D_1 \times \cdots \times D_n$ *matches* on a header $\mathbf{h} \in D_1 \times \cdots \times D_n$ iff. $\mathbf{h} \in \mathbf{r} \Leftrightarrow \forall i \in [1, n] : h_i \in r_i$.
We call $(\mathbf{r}, x)$ a rule $\mathbf{r}$ with rule number $x \in \mathbb{N}$.
We call $R = \{(\mathbf{r}_1, x_1), \ldots, (\mathbf{r}_n, x_n)\}$ with $x_1 \neq x_2 \neq ... \neq x_n$, $\mathbf{r}_1, \ldots, \mathbf{r}_n \subseteq D_1 \times \cdots \times D_n$ a **ruleset**.

Finding all matching rules for a given header $\mathbf{h}$ out of a ruleset R can be described as: $M = \{(\mathbf{r}, x) \in R \mid \mathbf{h} \in \mathbf{r}\}$. The matching rule with the highest priority is $(\mathbf{r}, x) \in M$ with $\forall(\mathbf{r}', x') \in M : x \leq x'$, in other words: the one with the lowest rule number.

2.1.1 Bitvector classification algorithm

The bitvector classification algorithm [LS98] consists of two steps. The first step is a binary search for all rule dimensions to find the corresponding bitvectors for a packet. In the second step, these found bitvectors are bitwise-anded to a resulting one. This bitvector is then linear searched in ascending order for non-zero bits.

Search Structure For creating the search structure, it decomposes all rules $\mathbf{r}$ of the ruleset R in its n dimensions (or fields), which will be searched through.
For each i-th dimension it creates an sorted array $A_i = ((a_1, b_1), ..., (a_m, b_m))$ of disjunct ranges where each range a_j has a bitvector b_j. The k-th bit in the bitvector b_j is set iff. $\exists(\mathbf{r}, x) \in R : x = k \wedge a_j \subseteq r_i$ or simpler: if the j-th range of A_i lays inside the range of the k-th rule for this particular i-th dimension (see figure 2.1).

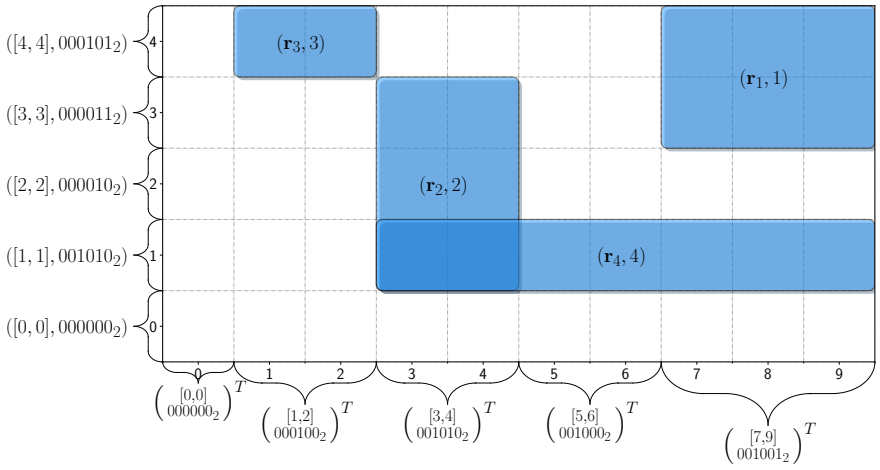


Fig. 2.1.: Generation of the bitvector arrays $A_{1,2}$ for a two-dimensional ruleset
Under the x-axis are the corresponding tuples of A_1, left of the y-axis, the one of A_2.

Searching To find the matching highest-priority rule $(\mathbf{r}, x) \in R$ for a header $\mathbf{h} = (h_1, ..., h_n)$, one does a binary search over the sorted array A_i for each i-th dimension to find the corresponding bitvector b_i. If b_i cannot be found because h_i lays in no range of A ($\forall (a, b) \in A_i : h_i \notin a$), then no rule $\mathbf{r} \in R$ matches.
If all bitvectors $B = (b_1, ..., b_n)$ are found, one unites them to a bitwise-anded vector $\mathbf{b}$. If the k-th bit in $\mathbf{b}$ is set, then the rule with number k matches on $\mathbf{h}$, since the k-th bit of all $b_1, ..., b_n$ must be set. Thus, for finding all matching rules for $\mathbf{h}$, one iterates over $\mathbf{b}$ to find all non-zero bits and their positions. The positions of these bits are also the numbers of the matching rules. To find the matching rule with the highest priority, one iterates over the bitvector and stops if the *first* non-zero bit occurred.
The binary search over all n dimensions in the first step has a complexity of $O(n \cdot log(m))$ with $m = max\,(\{|A_i| \mid i \in [1, n]\})$. And the linear search in the resulting bitvector has a complexity of $O(|R|)$. Thus, the lookup time for matching rules of a packet $\mathbf{p}$ is in $O(n \cdot log(m) + |R|)$.

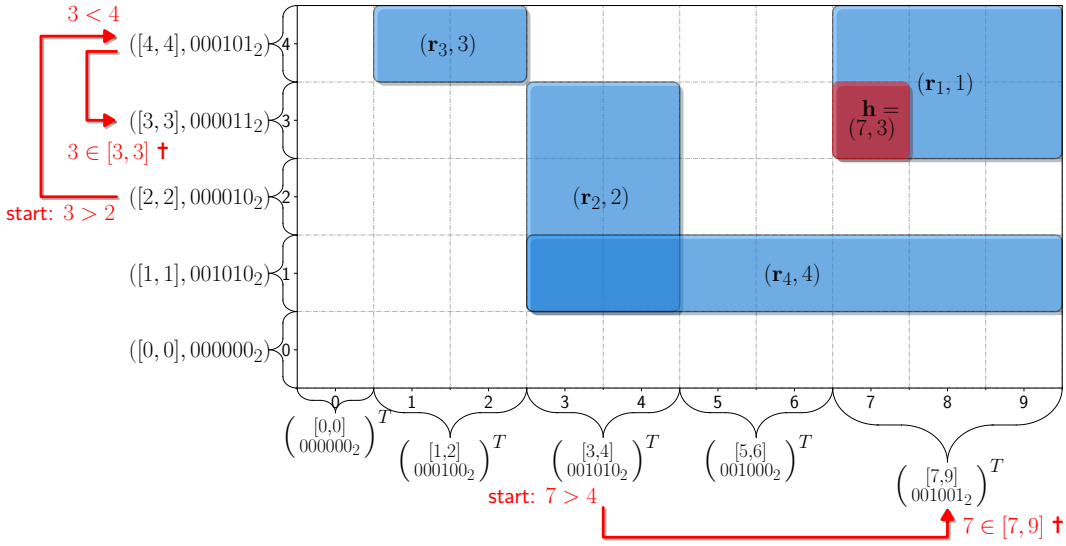


Fig. 2.2.: Binary search for the header $h = (7, 3)$ in each of the two arrays $A_{1,2}$.

Figure 2.2 shows the binary search in for the header $h = (7, 3)$ in each of the 2 dimensions. The tuples $([7, 9], 001001_2) \in A_1$ and $([3, 3], 000011_2) \in A_2$ have been found. The bitwise anded-bitvector is $001001_2 \;\&\; 000011_2 = 000001_2$. And the position of the least-significant non-zero bit represents the highest priority rule. Since $\text{ffs}(000001_2) = 1$, $(r_1, 1)$ is the matching rule for $\mathbf{h}$ with the highest priority.

Aggregated Bitvectors Bitvectors b are usually implemented as arrays a_w of w-bit unsigned integers (mostly $w = 32$ or $w = 64$). When searching for the highest-

priority rule, one iterates over $\mathbf{a}_w$ of the bitwise-anded bitvector $\mathbf{b}$ until the first non-zero w-bit integer occurs.
The check for non-zeroness of an integer is done in one processor cycle. And getting the least/most significant non-zero bit of an non-zero integer also, if intrinsics like *__ffs* ("find first set") [@22d] or *__clz* ("count leading zeros") [@22c] are used. Thus, the linear search using this implementation results in a runtime in $O\left(\frac{|R|}{w}\right)$ for a ruleset R.
Since it is very common that packets do only match on very few rules out of $|R|$ rules [BV01; GM99; GM00], the occurrence of non-zero integers in $\mathbf{a}_w$ is very low. Its therefore probable that one iterates over many zero integers until the first non-zero one comes. To speed it up, one stores a additional aggregated bitvector [BV01] b_{agg} per bitvector b. In b_{agg}, a i-th bit is set iff. the i-th integer of a_w is non-zero. b_{agg} is also implemented as an array a_{agg_w} of w-bit unsigned integers. If all a_{agg_w} are found, one bitwise-ands them to $\mathbf{a}_{agg_w}$. Thus, each i-th non-zero bit in $\mathbf{a}_{agg_w}$ means that the bitwise-and of the i-th integers of all a_w is non-zero.
Finding the highest-priority rule from $|R|$ rules with additional aggregated bitvectors consists of iterating through $\mathbf{a}_{agg_w}$ of the bitwise-anded aggregated bitvector $\mathbf{b}$ to find the first non-zero integer in $\mathbf{a}_w$ at position i. One then gets the first non-zero bit at position j from the i integer from $\mathbf{a}_w$, which can be done in one processor cycle. The highest-priority rule is the $(w \cdot i + j)$-th.
Since each bit in $\mathbf{a}_{agg_w}$ stands for one w-bit integer in $\mathbf{a}_w$, $\mathbf{a}_{agg_w}$ stores $\frac{|R|}{w}$ bits. Thus, iterating through $\mathbf{a}_{agg_w}$ is in $\Omega\left(\frac{\frac{|R|}{w}}{w}\right) = \Omega\left(\frac{|R|}{w^2}\right)$, but still in $O\left(\frac{|R|}{w}\right)$, since it may happen, that the bitwise-and of the i-th integer across all found a_w is zero. Then, one has to continue iterating through $\mathbf{a}_{agg_w}$. However, the linear search using aggregated bitvectors results is still an order of magnitude faster [BV01] than using plain bitvectors. One can also speed the search up by using the maximum integer size for which bitwise intrinsics are supported; usually 64-bit.
Using non-aggregated bitvectors, the bitvector classification algorithm has a lookup time in $O\left(d \cdot log(m) + \frac{|R|}{w}\right)$.
Using aggregated, the lookup time is in $\Omega\left(d \cdot log(m) + \frac{|R|}{w^2}\right)$ but in worst-case still in $O\left(d \cdot log(m) + \frac{|R|}{w}\right)$.

k-ary Search The binary search per i- th dimension in the array $A_i = ((a_1, b_1), ..., (a_n, b_n))$ takes $O\left(log\left(|A_i|\right)\right)$. One can speed this up by using a k-ary search with $k > 2$. During a k-ary search, one splits the current to-be-searched subset A_i' into k equally sized sets $s_1, ..., s_k$. For each of these sets s_j, one compares h_i in parallel with the first element a_{j_1} of s_j. If $h_i \in a_{j_1}$, then the matching range is found.

If $h_i < a_{j_1} \Leftrightarrow h_i < [l,u]_{a_{j_1}} \Leftrightarrow h_i < l$, then all ranges on the *right* of a_{j_1} are also greater than h_i and do not match. If $h_i > a_{j_1} \Leftrightarrow h_i > [l,u]_{a_{j_1}} \Leftrightarrow h_i > u$, then all ranges on the *left* of a_{j_1} are also smaller than h_i and do not match.
If a s_j and s_{j+1} exist, where $h_i > a_{j_1}$ and $h_i < a_{(j+1)_1}$, then there may be a range inside s_j on which h_i matches and one sets $A_i'' = s_j \setminus \{a_{j_1}\}$ and recursively does the k-ary search on A_i''. Else, there is no range in A_i on which h_i matches.
Since the comparison of the k sets tooks place in parallel for each iteration, the complexity of one k-ary search is $O\left(log_k(|A_j|)\right)$.
Using the binary search and aggregated bitvectors, the classification has a lookup time in $O\left(n \cdot log(m) + \frac{|R|}{w}\right)$ with $m = max\left(\{|A_i| \mid i \in [1,n]\}\right)$, using a k-ary search, it is in $O\left(n \cdot log_k(m) + \frac{|R|}{w}\right)$; much faster than the usual one.

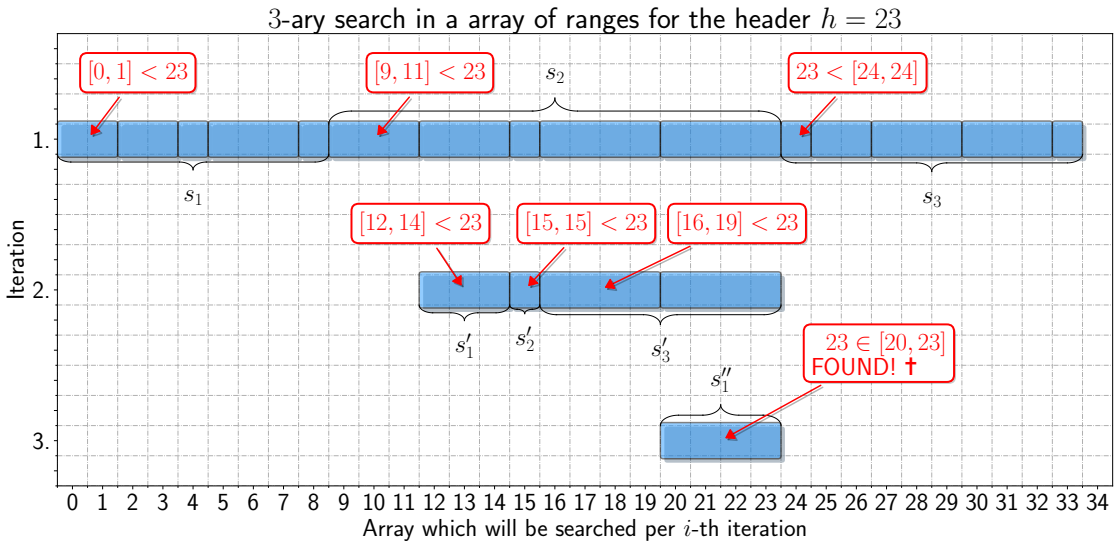


Fig. 2.3.: 3-ary search for the matching range on $h = 23$ in an array A.

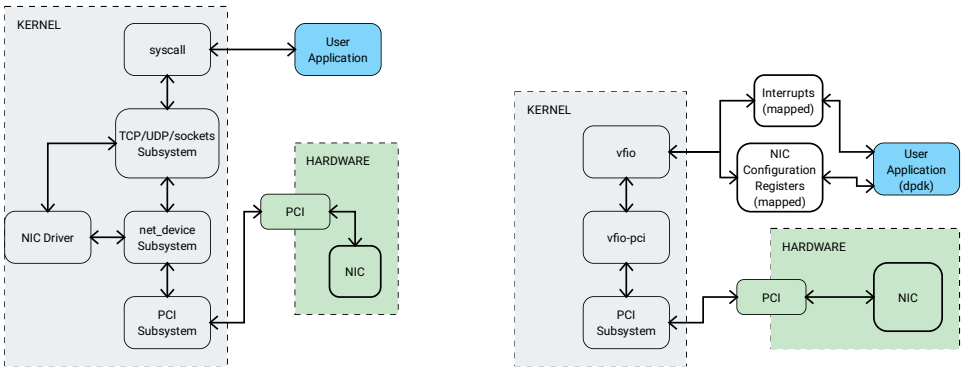
Figure 2.3 illustrates a 3-ary search for the matching range on $h = 23$ in an array A of 15 ranges. For the second iteration, $s_2 \setminus \{[9,11]\}$ is chosen as A'', because the first range of s_2 ($a_{2_1} = [9,11]$) is smaller than h and the first range $a_{3_1} = [24,24]$ of the neighboured subset s_3 on the right is bigger than h. Therefore the matching range must lay inside of $s_2 \setminus \{[9,11]\}$. In the second iteration, the first range a_{3_1} is smaller than h, thus the matching range must lay on the right of it. $A'' = s_3 \setminus \{[16,19]\}$ is set for the third iteration, in which the matching range $[20,23]$ is found.

2.2 Data Plane Development Kit

The **D**ata **P**lane **D**evelopment **K**it (dpdk) is a framework for packet processing in user-space, allowing lower latency and higher packet rates by bypassing the interrupt driven Linux kernel networking stack (see 2.2). It provides libraries for common tasks like routing, classification, metering, QoS and encryption and offers a platform-independent API for different devices (particulary NICs) using environment abstraction layers [@22h]. Applications can follow a pipelined, event-driven or run-to-completion packet processing model [@22h]. However, each of them depend on polling packets from and storing them in NIC or ring buffers.

Kernel Networking Stack Usually user-space applications use a socket API [@21c] from their standard library for communication with other hosts using for example TCP/IP or UDP/IP. See Figure 2.4a. For doing this, an application usually requests a socket descriptor and uses functions like *connect* for connecting to hosts or *bind* for accepting incoming connections. When a application sends data to the socket, the kernel allocates a packet buffer in kernel-space memory (*sk_buf*) which mainly consists of control data, headers for each network layer and payload [@19a; @16]. This buffer is then passed layer by layer to the corresponding kernel implementations until layer 2. Then, the resulting *sk_buf* is passed to the appropriate NIC which implements the *net_device* API. This driver then copies the whole packet data from the *sk_buf* buffer to a memory region which is accessible by the NIC and tells it that there is a packet ready to be transmitted. As one can see, the packet data traverses through several layers and subsystems until it reaches the NIC port. The memory copies from user-space to kernel-space and context switching due to syscalls when using the socket API slow down the overall packet rate.

Host to NIC Communication The NIC needs to access the host memory for storing received or reading to-be-transferred packet data [@19c; @19a]. Therefore the NIC driver uses MMIO (Memory Mapped Input/Output) [@21a] to map the configuration registers to kernel-space memory (purple line in figure 2.5) [@19a]. It then allocates a continuous memory region which is used to store a queue of receive descriptors (see figure 2.5). Each RX descriptor points to a buffer in memory which shall be [**f**]illed with the packet data. The driver then tells the NIC where this queue is located using the configuration registers. If a packet is recevied by the NIC, the DMA (Direct Memory Access) controller accesses the host memory through PCI and fills a free packet buffer [@19a]. It then updates the status of the corresponding RX



(a) User application using the socket API. [@19a]

(b) User application using dpdk. [@19a]

Fig. 2.4.: Kernel network stack and dpdk using VFIO.

descriptor and notifies the host using an interrupt that a packet has been received (figure 2.5).

Bypassing the Kernel Networking Stack DPDK bypasses the kernel network stack and thus saves the overhead by context switching due to syscalls and copying between layers in the kernel socket subsystem [@19a]. For doing that, it needs to directly communicate with the NIC using the interface provided by the PCI subsystem. It uses the VFIO system which allows user-space applications to communicate directly with a PCI device. It works by loading a generic PCI driver for the NIC. VFIO can then provide a file descriptor for interrupts and a mapping to the NIC configuration registers (see figure 2.4b). The queues and packet buffers, which dpdk tells the NIC to read and store packets into, need to be accessible for the NICs DMA controller. Hence, dpdk pins the memory, preventing it from swapping and changing its location. And uses IOVA (I/O Virtual Addresses) [@19c], which translates the virtual memory addresses to physical or I/O virtual addresses, understandable by the controller [@19b].

Memory Management The usage of dynamically allocated memory and memory manipulation functions provided by standard libraries (*libc*) lead to cache misses and address translation overhead which should be avoided in latency-bound applications like packet processing [@22w]. Therefore, dpdk provides a memory pool library [@22x] as one of its core components, which is used by all other libraries which work on memory. It allows the creation of memory pools of fixed sized objects. The slot

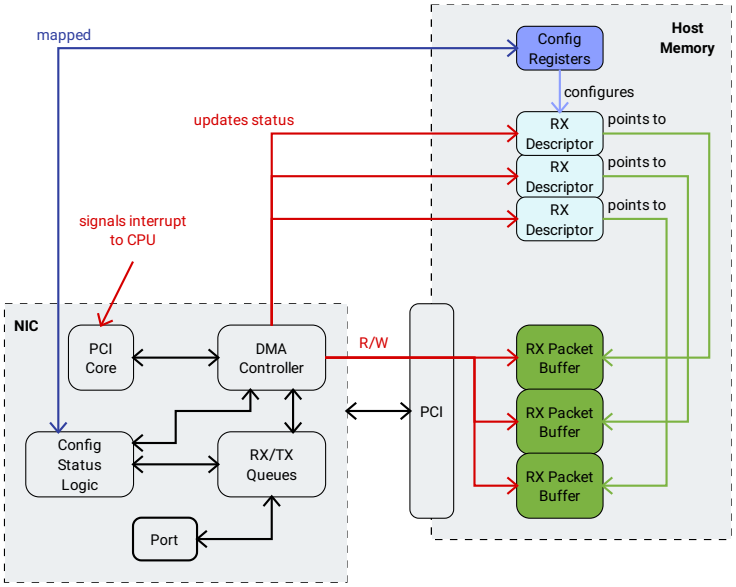


Fig. 2.5.: Kernel packet queue with packet data in skbufs [@19a].

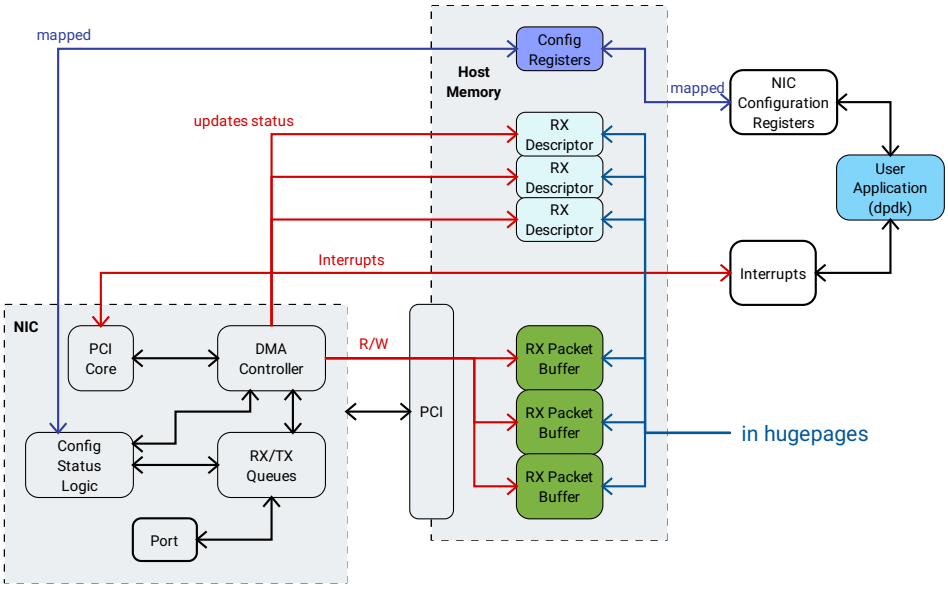


Fig. 2.6.: dpdk packet queue using memory in hugepages [@19a].

management inside a pool is done with ring buffers. The library takes also care of an equal distribution of objects across RAM channels and platform native alignment of them [@19c]. Allocated pools are mostly large, especially when packets are stored inside of them. The translation from the virtual-memory to physical-memory address of objects would fill up the CPU's TLB (translation lookaside buffer) causing costly computation of physical addresses, when normal 4K-memory pages are used [@22i; @19c]. Thus, the pools are allocated in huge-page-memory which reduces the amount of translations and entries in the TLB.
A closely related library is the one for message buffers [@22ag], its memory is also provided by an memory pool. It is used to store packet buffers and provide metadata about it [@22ah]. For example common header fields like source and destination ethernet MAC addresses, timestamps or checksum results. A experimental feature of this library is the ability to specify an own external memory buffer which is used to store the packet data. This allows the use of host pinned memory which can be accessed by devices like GPUs (see 2.3.2).

Port Initialization In dpdk, one configures *per port* and not *per NIC* using an uniform platform-independent ethernet device API [@22f; @22aj]. It allows the specification of the ports TX/RX queue size and NIC-dependent offload functionality, like checksum calculations or DMA mapping. One can also configure the port to use more than one RX/TX queue. Applications can profit from the equal distribution of packets across the queues by using multiple pairs of RX/TX threads with its own queues; resulting in higher packet rates.

Packet Polling and Transmitting An application receives network packets from a port by continuously polling the RX queue for new packets [@22h; @22e]. This provides much lower latency than waiting for an event [@22h]. But it comes with the downside of high cpu utilization, since the processing is done in a loop (see listing 2.1)and if there are no incoming packets, the applications repeatedly calls the receive function, wasting CPU cycles.

The poll-mode-drivers [@22e] offer polling for and transmitting of packet bursts. This allows more efficient packet processing since, mostly, packets come in bursts into a RX queue. Polling packet per packet would then result in much more overhead caused by the calls to the receive function. But one cannot arbitrarily increase the burst size, since it would lead to packet loss due to full rx queues and high latency, when much less packets are send in a given time, causing the application to wait until enough packets are in the queue. The choice of size for the RX/TX

queue and burst is therefore highly application-dependent. If an application can use proper parallelization, it would be beneficial to adapt the burst size to the maximum number of packets which can be processed in parallel.

```c
#include <stdint.h>
#include <unistd.h>
#include <rte_mbuf.h>
#include <rte_ethdev.h>

#define BURST_SIZE 64

extern void do_processing(uint16_t, struct rte_mbuf **);

void main_loop(uint16_t port_id, uint16_t queue_id) {
    struct rte_mbuf *bufs[BURST_SIZE];

    while(1) {
        const uint16_t nb_rx=rte_eth_rx_burst(port_id, queue_id,
                                              bufs, BURST_SIZE);
        if(!nb_rx) // no packet received, poll again
            continue;

        // process received packets
        do_processing(nb_rx, bufs);

        const uint16_t nb_tx=rte_eth_tx_burst(port_id, queue_id,
                                              bufs, nb_rx);
        // free not sent packets
        rte_pktmbuf_free_bulk(bufs+nb_tx, nb_rx-nb_tx);
    }
}
```

Listing 2.1: Typical main loop of a burst-oriented application.

Multi-threading The environment abstraction layer provides a multi-threading API [@22n] which wraps around pthread. The main difference is, that it also makes sure that the threads run on different logical CPU cores (lcores) by setting the core affinity. This prevents the overhead of task switching. Additionally, one should exclude the lcores which are used by the dpdk-application from the process scheduler of the kernel. This leads to much fewer interruptions of the application threads.

2.3 GPUs

The architecture of a GPU differs greatly from a CPU. The CPU is built to do serialized workload fast, providing just a few threads, which execute independently. It fulfils these goals by using big caches and having a high clockrate per core. [22z] The GPU allows processing of data highly parallelized. Therefore, it consists of much more processing units than caching and control, resulting to much more threads which can be concurrently executed. This lead to high memory and instruction throughput when the workload is uniform and not conditional.

2.3.1 SIMT

Single **I**nstruction **M**ultiple **T**hreads [@22m] refers to the main principle of the GPUs execution model. On a GPU, one scheduler is responsible for multiple threads. He puts n threads in groups G_i and issues a single instruction per cycle to each G_i [@22m]. One can compare this with SIMD, where one instruction is executed on multiple grouped data units in parallel. But unlike SIMD, threads in G_i do not require data alignment or grouping, which leads to much more flexible processing [@22m]. But one benefits by having strided memory access across data, since this lead to lower cache misses. [@22p]
In SIMT, the execution flow of threads in G_i can also diverge, meaning that threads inside G_i take different paths in the application due to data-dependent branching [@22m]. Then, the scheduler has to serialize the different pending instructions per branch. For each of the n sets s_j of threads in G_i, which took the same path, he pauses the rest in G_i and issues the corresponding instruction of s_j. This divides the instruction throughput by n [@22q]. Thus, it is highly recommended to write the application in a way which results in low or no branching inside of G_i. However, having different execution paths across different G_i and G_k is no problem. It is also possible for threads to communicate with each other through different types of memory or intrinsics. This allows writing algorithms which use thread-level coordination.

2.3.2 CUDA

Hardware Implementation The core components of NVIDIA GPUs are the *streaming multiprocessors* [@22l], responsible for creating, executing and scheduling groups of threads. One multiprocessor can execute multiple groups in parallel. These groups

have a fixed size of 32 threads and are also called *warps* [@22m].

Each thread of a warp has its own local memory, counters and registers, which is assigned by the multiprocessor. The amount of local memory and registers, a multiprocessor can distribute to warps, is limited. Thus the number of parallel warps which can be executed per multiprocessor is also. The number of concurrent warps depends directly on how much registers and memory a thread uses. In CUDA, it is also expressed as *occupancy* [@22b], the theoretical number of concurrent warps a multiprocessor can execute by given register and memory usage.
Since a multiprocessor maintains all thread-local information, like the pending instructions and filled registers, he can switch between the execution of different warps. This allows the multiprocessors to hide latencies caused by stalled warps which wait for memory operations or divergent threads [@22m].

Compute Capability The *compute capabiltiy* [@22g] describes the constraints a NVIDIA GPU has. For example: how many concurrent warps can be excuted per multiprocessor, how big the local memory is per multiprocessor is or how many registers a multiprocessor has.

Blocks And Grids An application launches threads in the form of grids. A grid consists of n blocks and a block of m threads [@22l]. The application can describe the grid and block size as a one-, two- or three-dimensional tuple. For differentiation, each thread in a thread block has an per-block unique id. The blocks are then equally distributed across all multiprocessors. Each multiprocessor splits the blocks into $\lceil \frac{m}{32} \rceil$ warps by their ascending thread id and executes them. The number of threads m per block is limited by the compute capability. As the number of concurrent warps per multiprocessor is also, an application should launch at least as many blocks as multiprocessors are available and adapt the block size to it. This will keep all multiprocessors busy and result in much less paused warps caused by the concurrent warps per multiprocessor limit. Therefore the overall performance is increased.

Synchronization Since warps of a block are executed independently, no assumptions, about when threads of other warps are executed and if they reached a point in the program, can be made. If an application depends on that all threads of a block are consistent at some point, it must do an explicit synchronization by using intrinsics like *__syncthreads()* [@22y]. This forces all threads in a block to wait until *all* threads of the block reached this point in the code and have seen all of their memory operations. For example if shared memory is used to exchange data across

warps of a block. If concurrent writes to same memory locations may happen, one must use atomics, since they are not ordered or serialized. Per-warp synchronization [@22y] is also possible, which forces threads of a warp to reconverge if they took different execution paths before. In addition, data can be exchanged between threads of the same warp without the use of shared memory. Using so called *warp intrinsics* it possible to shuffle data accross the threads by their warp-local id. One can also vote about a predicate across a warp without additional memory. Providing very efficient communication which can be used for parallelization of algorithms.

Memory In CUDA, one generally distinguishes between 3 types of memory types [@22ab]. **Global memory** can be accessed by all threads on the GPU. It is used to exchange data between the host and GPU and is the biggest but slowest of all memory types, since it lays in the device memory. The accesses of all threads per warp are coalesced to as few transactions as possible. The transactions are done in words up to 16 byte. [@22aa] Thus if threads in a warp access memory at n addresses very distant from each other, it leads to n serialized transactions dividing the throughput by n. Therefore, reads and writes should happen on consecutive addresses, preventing cache misses and resulting in fewer transactions.
Local memory [@22ad] is private to each thread. It is used to store non-constant data structures, which are to big for registers. Also, if a thread uses more registers than available, they are swapped to local memory. This is known as *register spilling*. Local memory is as slow as global, since it also lays in device memory. However, in contrast to global, the memory access is done in 32-bit words and is coalesced as long as the threads of a warp access the same address relative to their local memory. [@22aa]
Shared memory [@22ae] is block-local and can be accessed by each thread of the same block, providing much higher bandwidth than global memory, sice it sits near each multiprocessor. The size per processor depends on the compute capability but, with 64KB to 164KB, it is quite small. It can be used as a manual cache, storing frequently used data from global memory into it. Commonly it is used to exchange partial results of computations across threads in a block. It is valid as long as threads of the associated block are being executed. The performance of shared memory is achieved by having 32 banks of which each 32 sequential 32-bit words fall into the banks successively. [@22aa] Meaning that if the threads in a warp access 32-bit words at 32 sequential addresses in shared memory, each of the word is loaded from an distinct bank. Since memory can be loaded from a bank independently, these accesses are done in parallel. The throughput is 32 times greater than the one, where the addresses would fall into the same bank.

General Workflow The listing 2.2 shows a simple program, which uses CUDA to calculate a histogram of an artificial generated monochrome image. For doing that, one must allocate memory on the GPU to store the workload into it (see 49 and 55). It is also possible to make host memory accessible for the GPU by pinning (page-locking the memory, preventing swapping) and mapping it into the address space of the GPU (see 61). This type of memory does not require copying but has a lower bandwidth than device memory. It is often to communicate between host- and GPU-side code. Code which runs on the GPU needs to be declared as a function with GPU access qualifiers, these functions are also named *kernels* (see 9). The kernel is then invoked with its arguments and the grid/block size (see 65). Each thread warp accesses the image buffer at coalesced addresses so that the number of transactions from global memory is minimal. The threads of a block do also write their results in a partial histogram (see 15), laying in shared memory, resulting on faster atomic operations (see 28). After each thread finished iterating, the partial histogram is then accumulated to the one in global memory (see 34).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define X_DIM 1920
#define Y_DIM 1080
#define MAX_NUM 255

// the kernel, executed on the GPU
__global__  void cal_histogram(const size_t x_dim,
                               const size_t y_dim,
                               int *img,
                               unsigned long long *hist) {
    // shared memory, used to calculate
    // the partial histogram of each block
    __shared__ unsigned long long l_hist[MAX_NUM+1];
    const size_t size=gridDim.x*blockDim.x,
                 start=blockIdx.x*blockDim.x+threadIdx.x;

    if(threadIdx.x<=MAX_NUM)
        l_hist[threadIdx.x]=0;

  // prevent threads to write into uninitialized memory
    __syncthreads();

    for(size_t i=start; i<x_dim*y_dim; i+=size)
        // atomic opertaion on shared memory needed,
        // since threads access it concurrently
        atomicAdd_block(&l_hist[img[i]], 1);
```

```
30
31    // wait for all for writes to l_hist becoming visible
32      __syncthreads();
33
34      // writing partial histogram results to global memory
35      if(threadIdx.x<=MAX_NUM)
36          atomicAdd(&hist[threadIdx.x], l_hist[threadIdx.x]);
37  }
38
39
40  int main(void) {
41      srand(time(NULL));
42
43      const size_t hist_size=sizeof(int)*X_DIM*Y_DIM;
44
45      int *hostMem=NULL, *gpuMem=NULL;
46      unsigned long long *gpuHistogram=NULL;
47
48      hostMem=(int *) malloc(sizeof(int)*X_DIM*Y_DIM);
49      // allocating device memory
50      cudaMalloc(&gpuMem, sizeof(int)*X_DIM*Y_DIM);
51
52      for(size_t i=0; i<X_DIM*Y_DIM; ++i)
53          hostMem[i]=rand()&MAX_NUM;
54
55      // copy image data to GPU device memory; global memory
56      cudaMemcpy(gpuMem, hostMem, sizeof(int)*X_DIM*Y_DIM,
57                 cudaMemcpyHostToDevice);
58      free(hostMem);
59
60      // creating host mapped memory,
61      // accessible from GPU; global memory
62      cudaHostAlloc(&gpuHistogram, hist_size, cudaHostAllocMapped);
63      memset(gpuHistogram, 0, hist_size);
64
65      // 28 blocks of 1024 threads
66      cal_histogram<<<28, 1024>>>(X_DIM, Y_DIM, gpuMem, gpuHistogram);
67      cudaStreamSynchronize(0);
68
69      cudaFree(gpuMem);
70
71      for(size_t i=0; i<=MAX_NUM; ++i)
72          printf("%lu: %llu times\n", i, gpuHistogram[i]);
73
74      cudaFreeHost(gpuHistogram);
75      return 0;
76  }
```

Listing 2.2: Simple CUDA kernel for calculating a histogram.

3 Implementation

The classifier proposed in this thesis uses a highly parallelized version of the bitvector classification algorithm with aggregated bitvectors (see 2.1.1) and implements dpdk's API for lookup tables [@22v; @22ak]. This API is provided by all classification algorithms in dpdk. It is heavily used in the pipelined execution model of dpdk, where this abstract interface is used to interact with multiple chained algorithms for different stages, like routing or firewalling.

3.1 Functionality

Table API The table API is designed to make a lookup for a burst of up to 64 packets [@22am]. One adds and removes pairs of a rule and payload [@22al]. For classification, one calls a method which then finds the matching highest-priority rule of the ruleset for each packet. Then, a pointer to the corresponding deposited rule payload is stored in a user-provided buffer [@22am]. Additionally a bitvector is set, where each i-th bit represents, if a rule for the i-th packet has been found or not [@22am].

Ruleset Specification For creating an classifier instance, one needs to call a particular function of the table API with the ruleset properties. A ruleset consists of an arbitrary number of fields which are interpreted as unsigned integers. Each field needs to be specified by its size (1, 2 or 4-bit) and its offset from the beginning of the packet data.

Packet Data Since the classification takes place on the GPU, packet data needs to be accessible for a kernel. Copying data to the device memory at each invocation of the lookup function would result into higher latency. Also, the rules do mostly consist of only a few fields which lay in the packet header. However, even the copy of only the header results in needlessly transferred bytes. The overhead can be stint by using host-pinned memory (see 2.3.2). This requires dpdk to store the packets

inside the allocated pinned memory zone. Therefore, one needs to specify a memory pool for the packets with an external memory buffer, which is the host-pinned one.

3.1.1 Kernel

There are a two main possibilities for implementing the bitvector classification algorithm as a kernel. The first one splits the search and unification stage into two kernels and calls them in order. But this requires that the first kernel stores pointers of the found bitvectors for each dimension into *global* device memory, so that the second kernel can unify them and search for the first non-zero bit. Another approach uses one single kernel for the whole classification. This kernel firstly searches in each dimension for the matching range and its corresponding bitvector. He then stores a pointer of each bitvector in *shared* memory. Since the search is done with much more threads than the unification stage needs, an arbitrary amount of warps can now exit. The left ones are then used for the unification.
This technique is also called kernel fusion [Fil+15]. In contrast to the first approach, it saves the time needed by the additional launch and provides much faster access to the results of the first stage using shared memory. Thus, this classifier is also implemented as one single kernel.

Search Structure The search structure is generated on the host after each ruleset update and copied to device memory. Since rulesets are infrequently updated, the higher bandwidth of device memory predominates the transfer overhead to it. For each dimension, two sorted arrays of 32-bit unsigned integers are created, where one stores the lower and one the upper bounds for each range. The corresponding bitvectors consist of 64-bit unsigned integers and lay coalesced in one large array.

Parallelization Many aspects of the bitvector classification algorithm can be parallelized. The most obvious one is the classification of all 64 packets in parallel. Since one should avoid data-dependent branching inside the same warp, each packet shall be processed using a dedicated warp. This would result into 64 warps with only one active thread per warp out of 32 launched ones; a very ineffective use.
But the searches for a matching range per dimension are also independent on each other and thus can be done in parallel. If one would use one warp per packet to let each of its thread search for the matching range in an other dimension, the overall runtime is reduced by a factor of n, with $n =$ number of dimensions. But commonly, rules do not consist of 32 or more field, and if they do, one has to split the threads

up to more warps. Despite the spare threads, one has still branch divergence inside the warps which causes serialization and leads to an effective runtime reduce by a factor smaller than n. Thus, it is wiser to let all threads in a warp search in the same dimension, reducing the divergence.

Therefore, one can launch n warps per packet. But CUDA only allows 1024 threads per thread block [@22ao] and shared memory is needed for saving the pointers of the found bitvectors for each dimension. This limits the number of warps per block to $\frac{1024}{32} = 32$. If $n \leq 32$, the search can be done completely in parallel inside one block. But if $n > 32$, one has to either launch more than one block per packet and use global memory for synchronization or let the warps of a block sequentially search in more than one dimension (see figure 3.1).

This implementation uses the latter option and launches not more than 32 warps per packet. For having a high occupancy (see 2.3.2), as many packets are processed in one block as multiple of n warps fit inside of it, depending on the compute capability[@22ao]. So if the rules consist of 5 dimensions, one has a group of 5 warps per packet and a block consists of at most $\left\lfloor \frac{32}{5} \right\rfloor = 6$ groups. For a burst of 64 packets, $\left\lceil \frac{64}{6} \right\rceil = 11$ blocks will be launched.

After each range per dimension is found and the pointers to the bitvectors are stored in shared memory, the kernel now needs to find the highest-priority rule in the unification stage. Since all warps of a packets are located in the same block, one could use all of them to iterate coalesced over the bitwise-anded bitvectors. In each iteration, all warps must exchange the position of their found first non-zero bit in shard memory. If any warps have found a non-zero bit, the minimum position p out of all found positions is chosen and the warps can exit. Since the positions are equal to rule numbers, p is also the highest-priority rule.

The threads of the warps access the bitvectors consecutively (see figure 3.2). Since aggregated bitvectors and 64-bit integers are used, each thread checks $64 \cdot 64 = 4096$ rules per iteration. As a warp consists of 32 threads, one warp checks $32 \cdot 64 \cdot 64 = 131072$ rules per iteration. Thus, if the ruleset is smaller than the number of rules a single warp can check in one iteration, the use of multiple warps for one packet is unnecessary and causes synchronization overhead. One may even not benefit from the use of multiple warps for larger rulesets, because they must be costly synchronized after each iteration using shared memory. In a warp, one can synchronize and find the least significant non-zero bit using warp vote functions. Hence, this kernel uses only one warp per packet for the unification.

Iteration \ Warp	1	2	3	...	31	32	#Processed Fields
1.	1	2	3	...	31	32	32
2.	33	34	35	...	63	64	64
3.	65	66	†	...	†	†	66

Fig. 3.1.: Processed fields by warp per iteration; rules consist of 66 dimensions.

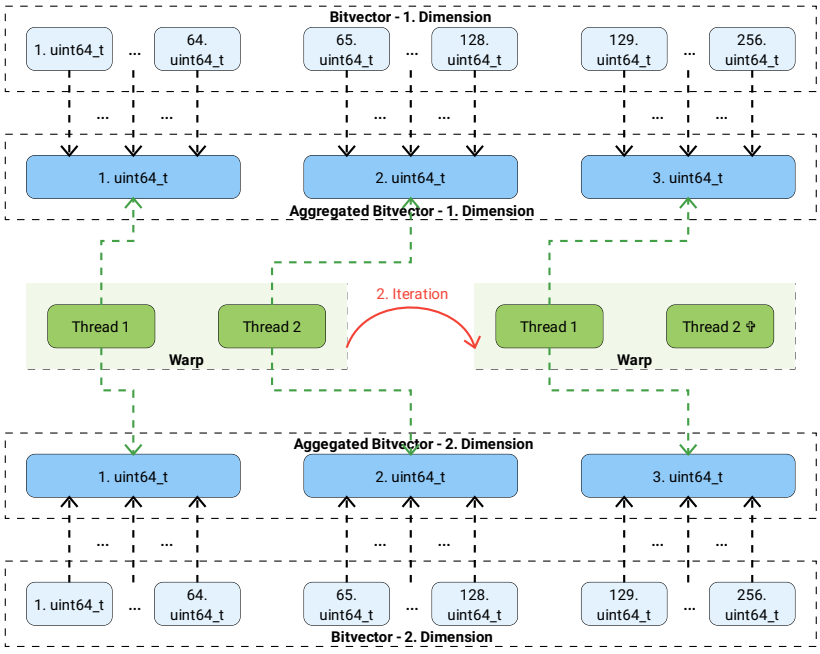


Fig. 3.2.: Access pattern on two bitvectors by a warp consisting of 2 threads.

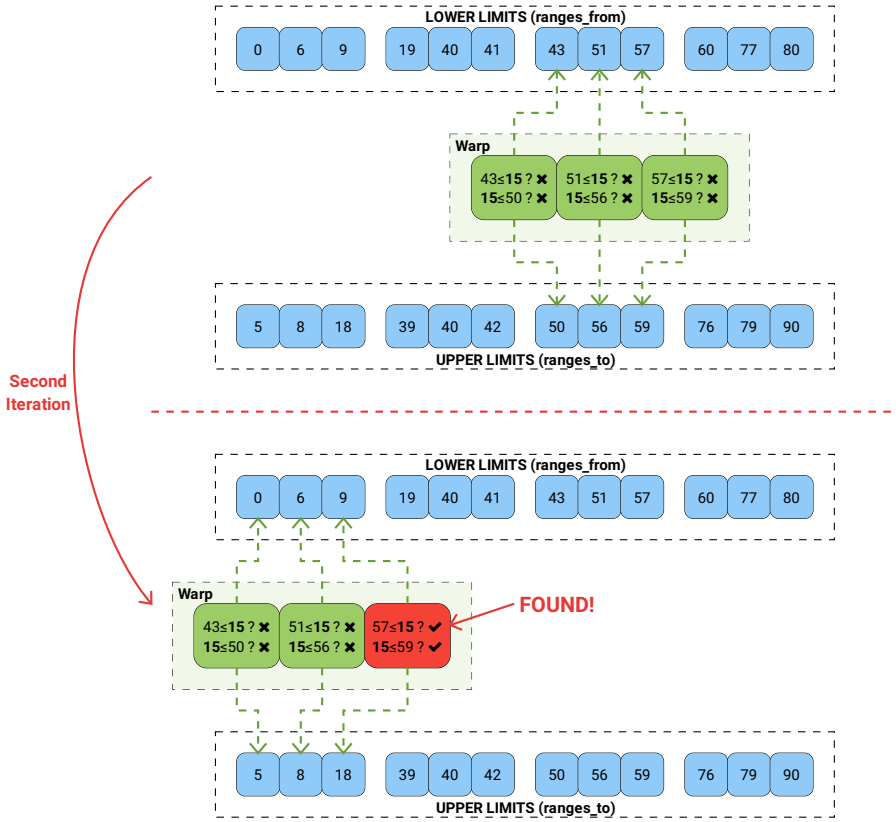


Fig. 3.3.: Blockwise binary search for the matching range on 15 in one dimension by a warp consisting of 3 threads.

Implementation The kernel frequently uses warp-level intrinsics for synchronization and information exchange. Since one warp for each dimension is used, 32 ranges of the to-be-searched array are compared against the header value of this dimension in parallel. Hence, the array is partitioned in blocks of 32 ranges and the binary search is done blockwise (see figure 3.3).

In each iteration, the threads in a warp compare the header field against their current range in the block. Their range is calculated by their *threadId*; so a thread with *threadIdx.x*$=n$ always compares the n-th range in block with the header field. Listing 3.1 show the kernel code which does the blockwise binary search. All threads of a warp enter synchronized the loop and calculate their current to-be-compared range based on the middle of the block borders and their local ID inside the warp (see line 9 ff.). They then compare the lower and upper limits of their range with the header field h (see line 11 ff.). Using the warp-level intrinsic *__ballot_sync(mask, predicate)*, the results of these comparisons are propagated as a bitmask across the warp. A set bit at the i-th position means that the predicate of the i-th threads is

non-zero. Which effectively means that set bits in the bitmask l represent all threads for which the lower bound of the range is less or equal to h and the set bits in r represent all threads for which the upper bound is greater or equal to h. Thus, when the result of the bitwise-and of the masks is non-zero, the matching range is found. Because of the comparisons of all ranges in one block happen in parallel, the runtime of this blockwise binary search is in $O\left(\log_2\left(\frac{N}{32}\right)\right) = O(\log_2(n) - 5)$, with n=number of ranges and a warp size of 32. This is a reduce by a constant time of 5 in contrast to a single-threaded binary search.

```
__syncwarp();
// get the value for the current field from thread 0 in our warp
v=__shfl_sync(UINT32_MAX, v, 0);

long start_end[2]= {0, (long) num_ranges[threadIdx.z]>>5};
long mid;
uint32_t l, r;
while(start_end[0]<=start_end[1]) {
  // calculate middle, 0 <= threadIdx.x <= 31
  mid=(((start_end[0]+start_end[1])>>1)<<5)|threadIdx.x;
  // compare and propagate results across the warp
  l=__ballot_sync(UINT32_MAX, ranges_from[threadIdx.z][mid]<=v);
  r=__ballot_sync(UINT32_MAX, v<=ranges_to[threadIdx.z][mid]);
  if(l&r) { // found
    if(__ffs(l&r)-1==threadIdx.x) {
      bv[threadIdx.y][threadIdx.z]=bvs[threadIdx.z]+mid*BV_BS;
      non_zero_bv[threadIdx.y][threadIdx.z]=
        non_zero_bvs[threadIdx.z]+mid*NON_ZERO_BV_BS;
    }
    __syncwarp();
    break;
  }

  start_end[!l]=((start_end[0]+start_end[1])>>1)+(-1*!l)+(l!=0);
}

__syncthreads();
// only use one warp per packet for unification
if(threadIdx.z)
  return;
```

Listing 3.1: Part of the kernel code which does the blockwise binary search in each dimension in parallel.

The code in listing 3.2 does the search for the first non-zero bit in the bitwise-anded bitvector. All threads of a warp iterate coalesced over the aggregated bitvectors (see figure 3.2) in ascending order (see line 7 ff.) and are bitwise-anding them. Each thread iterates over each i-th non-zero bit (line 14 ff.) of its current aggregated bitvector integer from the least to the most significant one. Since each bit in a 64-bit integer of a aggregated bitvector stands for one bitwise-anded 64-bit integer of the bitvector, the thread bitwise-ands the corresponding bitvector-integer of the i-th bit (line 19 ff.). If this integer is non-zero, it found the first matching rules of the current aggregated-bitvector-integer (line 25 ff.).
Each thread in a warp accesses the bitvectors by an offset of their ID inside the warp. Thus, rules encoded in a bitvector-integer by a thread with a lower ID have a higher priority than another. Using *__ballot_sync*, the information about which thread has found a non-zero bitvector-integer is propagated as a bitmask (line 32 ff.). Hence, the least significant non-zero bit i of the mask *tm* stands for the thread which found the bitvector-integer with the highest-priority rule (line 38). This thread calculates the position of the highest-priority rule using the index of the bitvector-integer and its first non-zero bit.
Since the threads of the warp compare their corresponding aggregated-bitvector-integer in each iteration in parallel, the runtime of the unification and search stage is in $O\left(d \cdot \frac{|R|}{64 \cdot 32}\right)$ with a ruleset R and d dimensions. In contrast to a single threaded unification with a runtime of $O\left(d \cdot \frac{|R|}{64}\right)$, the runtime is reduced by a factor of 32 (the warp size).

```
1   // all bitvectors found, now getting highest-priority rule
2   int nz_bv_b=threadIdx.x;
3   uint32_t in_loop=__ballot_sync(UINT32_MAX,
4                                  nz_bv_b<NON_ZERO_BV_BS);
5
6   while(nz_bv_b<NON_ZERO_BV_BS) {
7     // bitwise-and the abv-integers of each dimension
8     uint64_t x=non_zero_bv[threadIdx.y][0][nz_bv_b];
9     for(int field_id=1; field_id<num_fields; ++field_id)
10      x&=non_zero_bv[threadIdx.y][field_id][nz_bv_b];
11
12    int pos;
13    uint64_t y=0;
14    // iterate over the set bits of the bitvise-anded abv-integer
15    while((pos=__ffsll(x))) {
16      const int p=(nz_bv_b<<6)|(pos-1);
17
18      // bitwise-and the corresponding bv-integer of the set
19      // abv-integer bit
20      y=bv[threadIdx.y][0][p];
```

```
21        for(int field_id=1; field_id<num_fields; ++field_id)
22          y&=bv[threadIdx.y][field_id][p];
23
24        // found the first non-zero bv-integer
25        // of the current abv-integer
26        if(y)
27          break;
28
29        x=(x>>pos)<<pos;
30      }
31
32      // has any thread found a non-zero bv-integer?
33      const uint32_t tm=__ballot_sync(in_loop, __ffsll(y));
34      if(tm) {
35        // get the bv-integer with the highest priority
36        if((__ffs(tm)-1)==threadIdx.x) {
37          // calculate the highest-priority-rule out of the first
38          // set bit in bv-integer with the highest priority
39          const size_t rule=(nz_bv_b<<12)+((pos-1)<<6)+__ffsll(y)-1LU;
40          matched_entries[pkt_id]=&entries[entry_size*rule];
41          lookup_hit_vec[pkt_id]=1;
42        }
43        return;
44      }
45
46      // no non-zero bit found, continue with iteration
47      nz_bv_b+=blockDim.x;
48      in_loop=__ballot_sync(in_loop, nz_bv_b<NON_ZERO_BV_BS);
49    }
50
51    if(!threadIdx.x)
52      lookup_hit_vec[pkt_id]=0;
53  }
```

Listing 3.2: Part of the kernel code which does the unification of the found bitvectors and linear search for the first non-zero bit.

3.1.2 Architecture

With the implemented table API of dpdk, the classifier can be integrated in all packet processing models. For testing the performance of it, the run-to-completion-model is chosen. Since it can be easily implemented.
Figure 3.4 shows the general architecture of the implementation. It is a layer 2 firewall which filters incoming packets from a NIC based on a given ruleset. For receiving the filtered packet stream, a tap interface[@22r] *fw0* using dpdk is created.

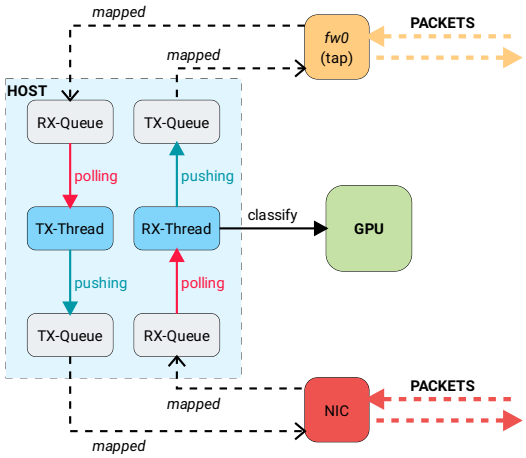


Fig. 3.4.: General architecture with one RX/TX queue per NIC and 2 threads.

It serves as the endpoint behind the firewall. Packets transmitted through *fw0* are passed directly to the NIC. The incoming packets of the NIC are classified using the kernel running on a GPU. If the action of the found highest-priority rule for a packet is not DROP, a incoming packet will be transmitted to *fw0*. Multiple RX/TX queues can be used for the NIC and *fw0*. For each queue, a thread is created using dpdk's multi-threading library, which ensures, that each runs on a different logical CPU core. Since each thread is continuously polling a RX-queue for incoming packets, the CPU cores are always busy. On the other side, the latency until a packet is processed is very low and the model allows receiving packets in large bursts (> 512 packets) from which the kernel generally benefits, since this leads to fewer transfers.

3.2 Kernel Launching Types

3.2.1 On-Demand Kernel Launching

The simplest way to classify packets using the GPU is launching a kernel on-demand when a new burst of packets come in (see figure 3.5). The RX-thread continuously polls from the NIC until a burst of packets is received. He then launches the kernel with an array of pointers pointing to the corresponding packet buffers in pinned host memory, accessible by the GPU. The kernel then reads the header fields from these buffers and tries to find the matching highest-priority rule for each packet using the ranges and bitvectors stored in the GPU device memory. He then writes the found entries into pinned host memory. Based on these found entries, the RX-thread

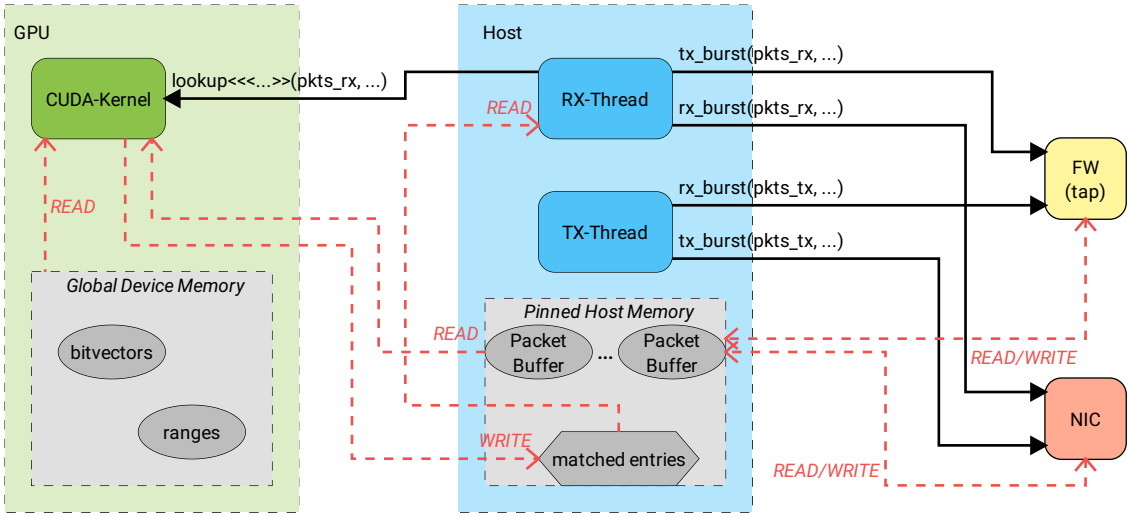


Fig. 3.5.: On-demand kernel launching using one queue per interface and two threads.

transmits each packet or silently drops it.
A great advantage of this launch method is, that the GPU is only utilized when packets come in. So that more processing power is left for other applications. However, it also leads to higher latency due the launch overhead for each burst and explicit synchronization. When a burst is received, the thread waits for the kernel to complete its classification and cannot receive additional bursts in the meantime.

3.2.2 Asynchronous Kernel Launching

For hiding the kernel launch overhead and access latency on pinned host memory, one can launch the kernel asynchronously for each incoming burst. This works by using CUDA streams[@22k]. A stream is a FIFO channel in which commands, like a launch of a kernel or a memory copy, can be issued in. CUDA allows creating many of these channels and the multiprocessor can dynamically schedule the execution of warps from different kernels in different streams. Thus, he is able to hide latencies of kernels by switching between the streams and executing the warps, which are not stalled[@22o].
One can exploit this by creating a ring buffer, which stores the streams where a kernel is currently classifying a packet burst (see figure 3.6). Instead of having one thread per RX-queue of the NIC, we have at least one *dequeue*-thread and one *enqueue*-thread. The *enqueue*-thread is continuously polling the NIC's RX-queues, for each received burst, he launches a kernel in a different stream identified by the

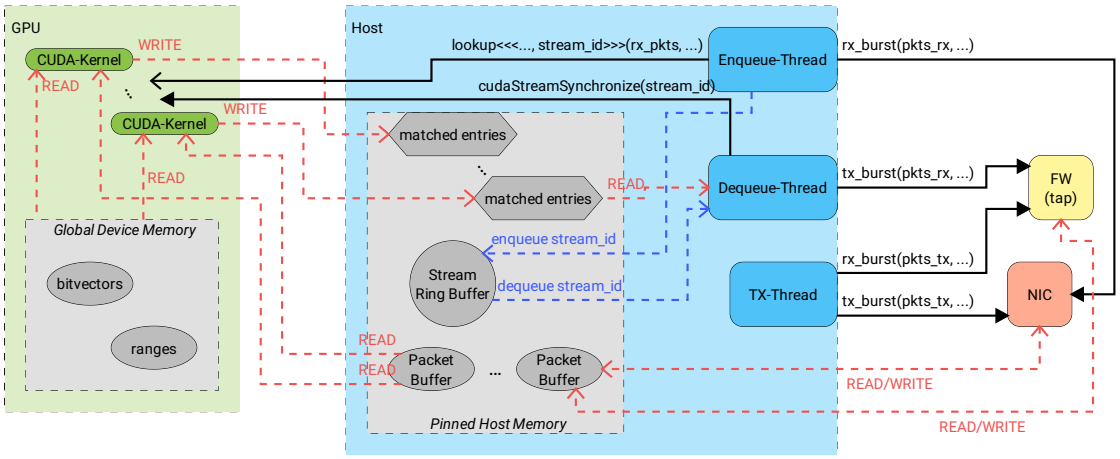


Fig. 3.6.: Asynchronous kernel launching using one queue per interface, a *dequeue*- and *enqueue*-thread and a TX-thread.

stream_id without waiting for it to finish. He then enqueues the *stream_id* in the ring buffer and continues polling. On the other side, the *dequeue*-thread continuously dequeues the ring buffer. For each *stream_id* he calls *cudaStreamSynchronize* which blocks until all commands in the stream have been completed, and so the launched kernel in this stream also. Based on the entries of the highest-priority rule for each packet, he then transmits a packet to the tap or not.
Like in the on-demand launching model, the GPU is not utilized when no packets are received. Additionally, memory access latencies are hidden by the multiprocessor and the incoming packet rate is higher since the RX-thread is not waiting for the classification. On the downside, more RX/TX descriptors and packet buffers are needed, since kernels are executed in parallel and thus more packets need to be stored at all times. Also, the launch latency cannot be hidden and so the processing latency is still quite high.

3.2.3 Persistent Kernel

For eliminating the launch time completely, one can create a so called *persistent kernel*, which is launched at the application start and then runs on the GPU until the program exits. This kernel communicates with the RX-thread using mapped pinned host memory (see 2.3.2) (see figure 3.7). If a RX-thread received a burst of packets, it writes the number of new packets and their buffer pointers into a for the kernel well-known location. The kernel continuously polls for this number until it is non-zero. He then does the classification of the packets and writes the found entries

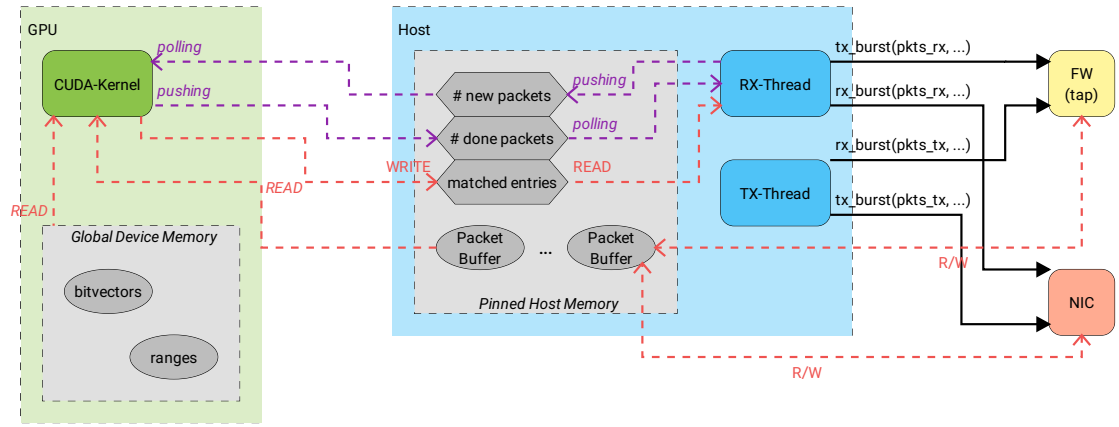


Fig. 3.7.: Persistent kernel using one queue per interface and two threads.

back into pinned memory. On the other side, the RX-thread continuously polls the number of classified packets until it is equal to the number of the enqueued ones. It then transmits the packets to the tap or not, depending on the matched entries.

One can also create multiple queues with its own persistent kernel. But the possible number of parallelly executed kernels is limited by the amount of multiprocessors[@22j] and how many warps can persistently reside on them; depending on the compute capability[@22ao]. Since the warps never exit, a multiprocessor cannot process more warps than he can execute in parallel[@22l]. This also shortens the number of parallelly processed packets. If the burst size is bigger, the kernel must serially process a part of the packets.

The reached very low latency using this method comes with the cost of high GPU utilization due polling.

3.3 Computation Versions

3.3.1 k-ary search

Using a k-ary search[2.1.1] instead a blockwise binary search noticeably accelerates the classification. The corresponding kernel uses one warp per dimension and thus is able to do a 32-ary search. Listing 3.3 shows the code which implements it. All threads of a warp split the current to-be-searched subset of the array up to 32 subsets until it is smaller than 32 elements. The subset, a thread compares to the header field *v*, is calculated based on its local warp ID (see line 8). It then compares the first range of its subset with *v* (see line 10 ff.). The result of each thread is propagated across the warp using *__ballot_sync* as two bitmasks *l* and *e*. Where each i-th set bit in l means that the first range's lower limit of the i-th thread is smaller than or equal to *v*. And each i-th set bit in *r* means that the first range's upper limit of the i-th thread is greater or equal to *v*. So if *l*&*r* is non-zero, a thread found the matching range (see line 13 ff.).
If it is zero, the new subset has to be calculated. The higher the thread ID is, the greater the lower limit of the first range of the thread's subset is, since the ranges are sorted in ascending order. Hence, the matching range must be in the subset of the thread with the highest ID where the lower limit of the first range is still smaller than *v*. This is also the thread behind the least-significant bit in *l* (see line 26 ff.). The start of this subset is propagated using the warp level intrinsic *__shfl_sync(mask, val, thread_id)*, which propagates *val* of the thread with the id *thread_id* across all threads in *mask*; providing much faster exchange than using shared memory. The subset is then again split up into 32 sets and the iteration continues.
When the last subset is smaller than 32 ranges, the threads access (line 36) and compare it coalesced for the last time (line 38 ff.). If no range of the subset matches on *v*, then no rule matches on the packet.
All threads in a warp compare their subset in parallel and warp-level intrinsics are used for finding the next to-be-searched subset (runtime in $O(1)$). Thus, the runtime of a search in one dimension lays in $O(log_{32}(n))$ on an array of n ranges; much faster than the blockwise binary search of $O(\log_2(n) - 5)$.

```
1   __syncwarp();
2   // get the value for the current field from thread 0 in our warp
3   v=__shfl_sync(UINT32_MAX, v, 0);
4   long size=num_ranges[threadIdx.z]>>5;
5   long start=0, offset;
6   uint32_t l,r; //left, right
7   while(size) {
```

```
8       // calculate current offset of the subset
9       offset=start+((long) threadIdx.x)*size;
10      // compare first element of subset with header field
11      l=__ballot_sync(UINT32_MAX,ranges_from[threadIdx.z][offset]<=v);
12      r=__ballot_sync(UINT32_MAX,v<=ranges_to[threadIdx.z][offset]);
13      if(l&r) { // found
14        if((__ffs(l&r)-1)==threadIdx.x) {
15          bv[threadIdx.y][threadIdx.z]=bvs[threadIdx.z]+offset*BV_BS;
16          non_zero_bv[threadIdx.y][threadIdx.z]=
17            non_zero_bvs[threadIdx.z]+offset*NON_ZERO_BV_BS;
18        }
19        __syncwarp();
20        goto found_bv;
21      }
22      if(!l) // nothing found, all first ranges greater than field
23        goto found_bv;
24
25      // calculate thread with the rightmost subset,
26      // where the lower limit of the first range <= v
27      r=__popc(l)-1;
28      // use old offset of this thread as the new start
29      start=__shfl_sync(UINT32_MAX, offset+1, r);
30      // new size is old one, partitioned for
31      // 32 threads (a warp)
32      size=r==31?(num_ranges[threadIdx.z]-start)>>5:(size-1LU)>>5;
33
34      __syncwarp();
35    }
36    // access last subset coalesced
37    offset=start+threadIdx.x;
38    // do only compare if own offset not above boundary
39    if(offset<num_ranges[threadIdx.z]) {
40      if( (ranges_from[threadIdx.z][offset]<=v)&
41          (v<=ranges_to[threadIdx.z][offset])) {
42        bv[threadIdx.y][threadIdx.z]=bvs[threadIdx.z]+offset*BV_BS;
43        non_zero_bv[threadIdx.y][threadIdx.z]=
44          non_zero_bvs[threadIdx.z]+offset*NON_ZERO_BV_BS;
45      }
46    }
47    __syncwarp();
48
49  found_bv:
50
51    __syncthreads();
```

Listing 3.3: Part of the kernel code which does the 32-ary search in each dimension in parallel.

3.3.2 SIMD

Currently, the search structure is implemented as two 32-bit integer arrays per dimension. One array consists of the lower limits of each range and the other of the upper. Each integer stores exactly one limit. But commonly, the header fields have a size of $w = 16$ or $w = 8$ bit and thus less than 32 bit are needed for storing the corresponding range limits. Hence, the upper $32 - w$ bit of each integer for a range are unused *but* transferred from global memory of the GPU to a thread, when it compares the range. This divides the information throughput by $\frac{32}{w}$.
But CUDA supports SIMD intrinsics[@22a]. Depending on the field size $w = 32, 16, 8$ one can pack $\frac{32}{w}$ limits into one 32-bit unsigned integer and use the per word, halfword or byte intrinsic for comparison with the header field v. This raises the throughput by $\frac{32}{w}$ times and also speeds up the comparison. Since one thread compares up to 4 ranges in parallel when the header field is 8-bit long.

3.3.3 Bigger Burst Size

The classifier implements dpdk's table API[@22v; @22ak] where the lookup function uses a 64-bit bitmask to let the table know which packet of the burst is valid. Since each bit in the mask stands for one packet, the maximum burst size is 64.
But the GPU can process much more packets in parallel. For example: the rules have 5 dimensions and for each dimension, one warp in a block is used for searching. Since a block can only have at maximum 1024 threads[@22ao] and thus $\frac{1024}{32} = 32$ warps, one block can classify $\left\lfloor \frac{32}{5} \right\rfloor = 6$ packets. On compute capability 8.6, the number of resident blocks per multiprocessor is 16 [@22ao]. Thus, one multiprocessor can classify $16 \cdot 5 = 80$ packets nearly in parallel. Hence, a GPU with 28 multiprocessors, can classify $28 \cdot 80 = 2240$ packets without heavy serialization; so $\frac{2240}{64} = 35$ times more packets than the table API allows.
Also, the bigger the burst size is, the less kernel launches are needed for classification. Therefore, using a bigger burst size results in less launch overhead and higher packet throughput, since the GPU is able to process more packets at all time. It also utilizes, beforehand unused, resources of the GPU and allows the multiprocessors to hide latencies better, because more warps for switching to are available.
Hence, the classifier provides a lookup-function for arbitrarily large packet bursts, which is not available from dpdk's table API.

3.4 Impact of memory type

In CUDA, the memory types differ in size, cache strategy and location on-chip (see 2.3.2). Thus, the memory type used for different data has an high impact on the overall performance of the classification.

Search Structure The first data which is in the device memory is the search structure of the bitvector classification algorithm (see 3.1.1). The ranges and bitvectors for each dimension are quite large. They do not fit into shared memory and thus reside in global device memory. CUDA provides different caching and access strategies for *normal* writeable or *constant* global memory [@22ac]. Constant memory is global memory with its own cache. Requests to the same address by multiple threads in a warp will be merged into one request to the cache and, if the cache access results in a miss, to global device memory. If accesses on different addresses are requested, they will be served *serialized* [@22ac].
Since the classification kernel does not write to the structure, it is possible to place it into constant memory. But each thread in a warp of the kernel accesses a range or bitvector based on its ID and thus always access memory on a different address. Hence, putting the search structure in constant memory has no performance gain and may even degrade it, because the requests are serialized.

Packet Data The packet data lays in mapped pinned host memory (see 2.3.2). When the kernel runs persistently on the GPU (see listing 3.4), the packet data is updated on the host side at any time. Thus, the data cannot be cached and is therefore marked as *volatile* (see line 13). Each warp of a packet classifies exactly one particular field. Since all threads of a warp would access the same location in a packet buffer for getting the field value, this would lead to multiple transfers of the same memory location. Thus, for each incoming burst, only one thread per warp is chosen, which fetches the packet field and propagates it across the warp using a warp-level intrinsic.

```
__global__ void bv_search(uint32_t **ranges_from, // lower bounds
                          uint32_t **ranges_to, // uppe rbounds
                          uint64_t *num_ranges,
                          // offsets of packet field from start
                          // and its size
                          uint32_t *offsets, uint8_t *sizes,
                          // bitvectors per dim., global memory
                          uint64_t **bvs, uint64_t **non_zero_bvs,
                          const uint32_t num_fields,
                          // entries, global memory
                          const uint32_t entry_size,
                          uint8_t *entries,
                          // packet data, shall not be cached
                          volatile uint8_t **pkts,
                          void **matched_entries,
                          uint8_t *lookup_hit_vec,
                          // number of incoming and done packets,
                          // used to communicate with the host,
                          // shall not be cached
                          volatile uint *num_pkts,
                          volatile uint *num_done_pkts,
                          // used to check if kernel should exit
                          volatile uint8_t *running) {
```

Listing 3.4: Signature of the kernel which runs persistently on the GPU.

4 Evaluation

For evaluating the classifier's performance, different versions of the firewall will be tested against rulesets which are generated randomly or by ClassBench[TT07a]. For reference the performance of the dpdk ACL library *rte_table_acl*[@22s] is consulted.

4.1 DPDK ACL

The ACL library of dpdk[@22s] provides a classifier for a variable number of fields. It uses multiple multibit tries with a stride size of $8bit = 1byte$ [@22s; @22an] for finding the highest priority rule out of a given ruleset. The rules can consist of an arbitrary number of fields with sizes from one to eight bytes [@22t]. The library processes four consecutive bytes [@22t] and thus enforce that the fields need to be grouped into four byte blocks, except the first field [@22t]. It uses up to eight multibit tries by default depending on the ruleset to provide good performance. DPDK also implemented the table API for it [@22ai]. Additionally, the library chooses the SIMD instructions with the biggest register sizes, the processor supports, for achieving maximum parallelization [@22u]. For the benchmark, the table API *rte_table_acl* [@22ai] of the ACL library is used and the trie limit is raised to 32.

4.2 Classification Kernel

The classification kernel of this thesis does a 32-ary search (see section 3.3.1) and uses SIMD instructions (see section 3.3.2) for comparing the packet header with the search structure (sec. 3.1.1).

4.3 Rule Generation

For testing the performance of the firewall using the implemented classifier and others, rulesets need to be generated. The rules consists of 5 dimensions: source & destination IPv4 address, source & destination port and protocol number.

Randomly Generated Ruleset The simplest generation of such ruleset is to generate each range for a rule pseudo-randomly with uniform distribution. The used number generator is *rand()*[@21b] from the GNU C library.

ClassBench ClassBench [TT07a] is used in many researches in the area of packet classification. It generates rulesets based on data gained by real filter sets from internet service providers, network equipment vendors or other researchers [TT07b]. Thus the generated rulesets are similar to real-world ones. Testing the firewall against these rulesets provides good insight on how it performs in real-world environment.
To generate rulesets, ClassBench needs a parameter file which includes all information on how the rules shall be. It ships with three types of files: ACL, Firewall and IP-Chain. For the this benchmark, the firewall parameter files are choosen.

4.4 Packet Generation

The packet generation is done using *pktgen-dpdk* which is capable of creating 10Gbit/s line rate traffic with packets of minimum size [@22ap]. The minimum ethernet frame size is $64byte$ [18], the inter-packet gap $12bytes$ and the preamble& start frame delimiter field is $7 + 1 = 8bytes$ large. Thus, $12 + 64 + 8 = 84bytes$ are needed for sending one packet. Hence, *pktgen-dpdk* is able to create a packet rate of $\frac{10 \cdot 10^9 bit/s}{8bit \cdot 84bytes} \approx 14,880,952pps \approx 14.88Mpps$.

Generation Scheme The packets are generated by incrementing the corresponding values for each header field. For generating a new packet, each header's current value is incremented by the particular increment in table 4.1.

	Src IPv4	Dst Ipv4	Src Port (16-bit)	Dst Port (16-bit)	Protocol Number
start	0.0.0.0	0.0.0.0	0	0	0
end	255.255.255.255	255.255.255.255	65535	65535	255
increment	0.1.2.5	0.1.2.5	1	1	1

Fig. 4.1.: Generation scheme used to generate the packets.

4.5 Benchmark

For benchmarking the implemented classifier, the firewall with all of its three kernel launching types (see 3.2) is tested against randomly and by ClassBench generated rulesets (see 4.3) of different sizes. Additionally, different burst sizes (see 3.3.3) and number of queues are considered.

Packet Rate Measurement For getting the packet rates depending on the number of rules, the measurement is done in the following way. For each number of rules n, m rulesets with n rules are created. For each firewall application, the packet rate on each of the m corresponding rulesets is measured 10 times with a gap of two seconds between each measurement. Out of the $10 \cdot m$ packet rates, the average and standard deviation is calculated.
m is set to 4 for randomly generated rulesets and to 5 for rulesets generated by ClassBench, since it is shipped with 5 parameter files for firewall rules. The ruleset sizes are created exponentially using the schema 4^x with $x \in \mathbb{N}$ and $x > 2$. The only exception is the biggest ruleset with 50.000 rules, since it is the current set limit for the GPU-based classifier so that the search structure of the bitvector classification algorithm still fits into the GPU global device memory (see 3.1.1).

Test Environment The test environment consists of two hosts connected with Mellanox ConnectX-3 NICs which are capable of transmitting at a line rate of $40Gbit/s$. See Figure 4.2. On the left, a server generates traffic with *pktgen-dpdk* as described in section 4.4 and transmits it to the host which runs the firewall.
Both the host and the server run on Debian 11 and the Linux kernel with version 5.15, the logical CPU cores which are used by the firewall application are isolated from the kernel scheduler and SMP load balancer [@17]. The host has an Intel Core i5-11600K with 16GB RAM and a NVIDIA RTX 3060 installed. In the server, a Intel Xeon E3-1230v5 with 8GB of RAM is installed. The processors are capable of AVX-512 vector instructions, hence the ACL library of dpdk will use them.

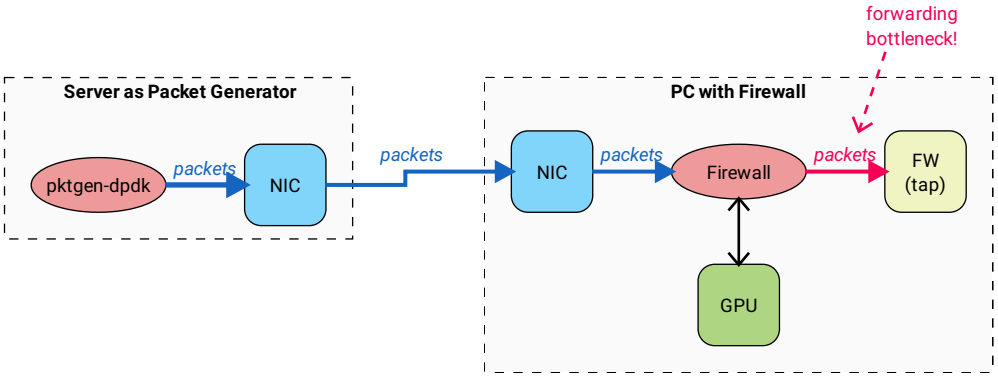


Fig. 4.2.: Test environment, the forward of classified packets is the bottleneck.

Benchmark with Forward to tap The graphs in figure 4.3 show the average packet rates and its standard deviations (as error bars) of different firewall configurations on randomly and by ClassBench generated rulesets.
Independently from the ruleset type, each rule does accept the packet and thus, causes the firewall to forward it to the tap interface. Hence, after a packet is classified, it will be forwarded to the tap. This approach utilizes the tap's queues fully and makes possible bottlenecks visible. When the rule action would also be randomly generated, it is possible that on some rulesets, the firewall performs better than on other. Because, if the most of the generated packets would match on rules which drop them, a RX-thread (see figure 3.5) would not send them to the tap after classification and therefore does not need to wait for the successful transmit. This leads to less process time for a packet burst, resulting in an overall higher packet rate. Hence, using random actions and tap forwarding would result in falsified statistics.
All packet rates of the firewall using the own classifier decrease very slowly with the rising number of rules on both ruleset types. Whereas the packet rate using dpdk's ACL library is highly dependent on the type. On completely random generated rulesets, the classifier delivers the highest packet rates but does not scale as well as the GPU-based one with the number of rules. On rulesets generated by ClassBench, it fails to build its multibit tries on numbers of rules above 1024.
However, doubling the number of queues for the on-demand launched kernel and ACL library of dpdk leads to an improvement of $\approx 40\%$ of the packet rate. A possible cause may be that a single RX-thread is busy transmitting its classified packets to the tap. For each pair of NIC RX-queue and tap TX-queue, one RX-thread is launched. Hence, having multiple queues results in multiple RX-threads which classify and transmit packets. Latencies due to waiting RX-threads are more hidden, the more

queues are created. Therefore, the packet rate increases. The great impact of the queue number indicates a bottleneck at the packet forward to the tap (see figure 4.2). Thus, for better statistics with less influences, a benchmark without packet forward to the tap interface is done.

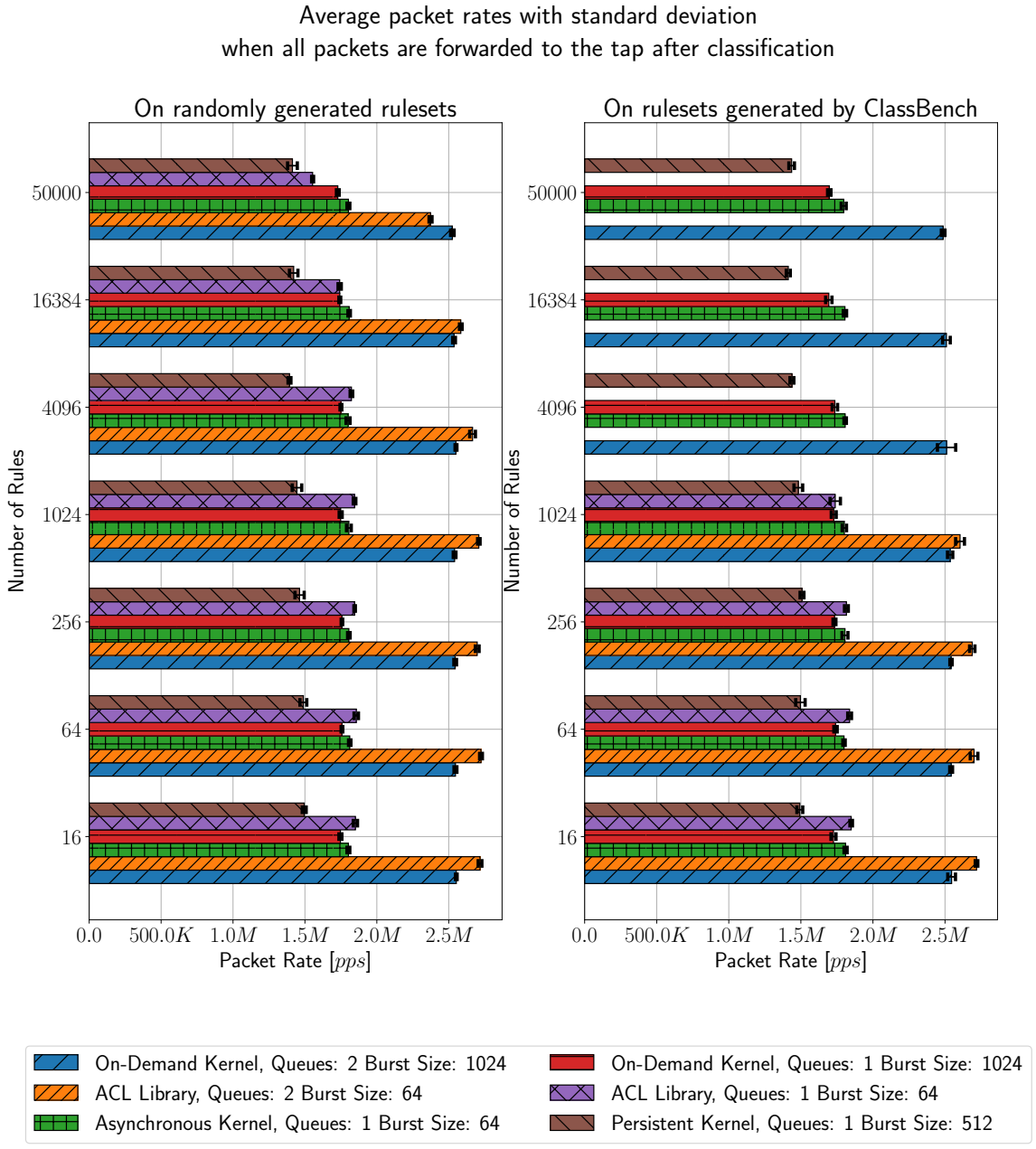


Fig. 4.3.: Packet rates on randomly and by ClassBench generated rulesets when all packets are forwarded to the tap after classification.

Benchmark without Forward to tap The graphs in figure 4.4 show the average packet rates and standard deviation (as error bars) on randomly and by ClassBench generated rules when all packets are not forwarded to the tap interface. Since the RX-threads (see figures 3.5 and 3.7) do not need to wait that the packet are sent, the overall rates are much higher.
The rates of the firewall using the classifier of this thesis are decreasing slowly. Considering the on-demand launched kernel, the burst size has an high impact on the packet rate. Taking a look on only on-demand launched kernels with one queue, one can see, that the higher the burst size is, the higher the packet rate becomes. Using an burst of 1024 instead of 64 packets increases the performance by an factor of ≈ 2.1.
The high disparity between the packet rates on randomly and by ClassBench generated rulesets come from different ways how they are generated. Whereas rules in the randomly generated rulesets differ highly, rules from ClassBench have more similarities. See Listing 4.1. This causes that on a single header field, mostly more than one rule from by ClassBench generated rulesets would match. Hence during the unification stage of the bitvector classification algorithm (see 2.1.1), the kernel must iterate over more false positives in the bitwise-anded aggregated bitvector for finding the highest-priority rule. Thus, the classification time of a packet is more dependent on its header values. Which causes the higher fluctuation observable by the bigger standard deviation error bars of the packet rates from the on-demand launched kernels on by ClassBench generated rulesets. As one can see on the on-demand launched kernels with a burst size of 64 and 1024 packets, the use of two queues lessens the fluctuation. Since the varying time a single RX-thread needs to wait for the kernel to finish is better hidden.
The ACL library of dpdk [@22s] reaches the highest packet rate. However, it does not scale well with a rising number of rules. On completely randomly generated rulesets, its rate becomes less than the GPU-based classifier on rulesets with more than 4096 rules. On rulesets generated by ClassBench, it fails to build its tries for rulesets with 4096 rules and above. Of all kernel launching types for the classifier proposed by this thesis, the highest packet rate is achieved by using an asynchronous launched kernel (see 3.2.2) The persistent kernel (see 3.2.3) has the worst performance, even with a burst size of 512 packets, since the number of parallelly executed kernels and the grid size cannot be as large as in other launching types (see 3.2.3). This leads to more serialized processing of packet bursts inside the kernel which causes a longer process time of one burst.

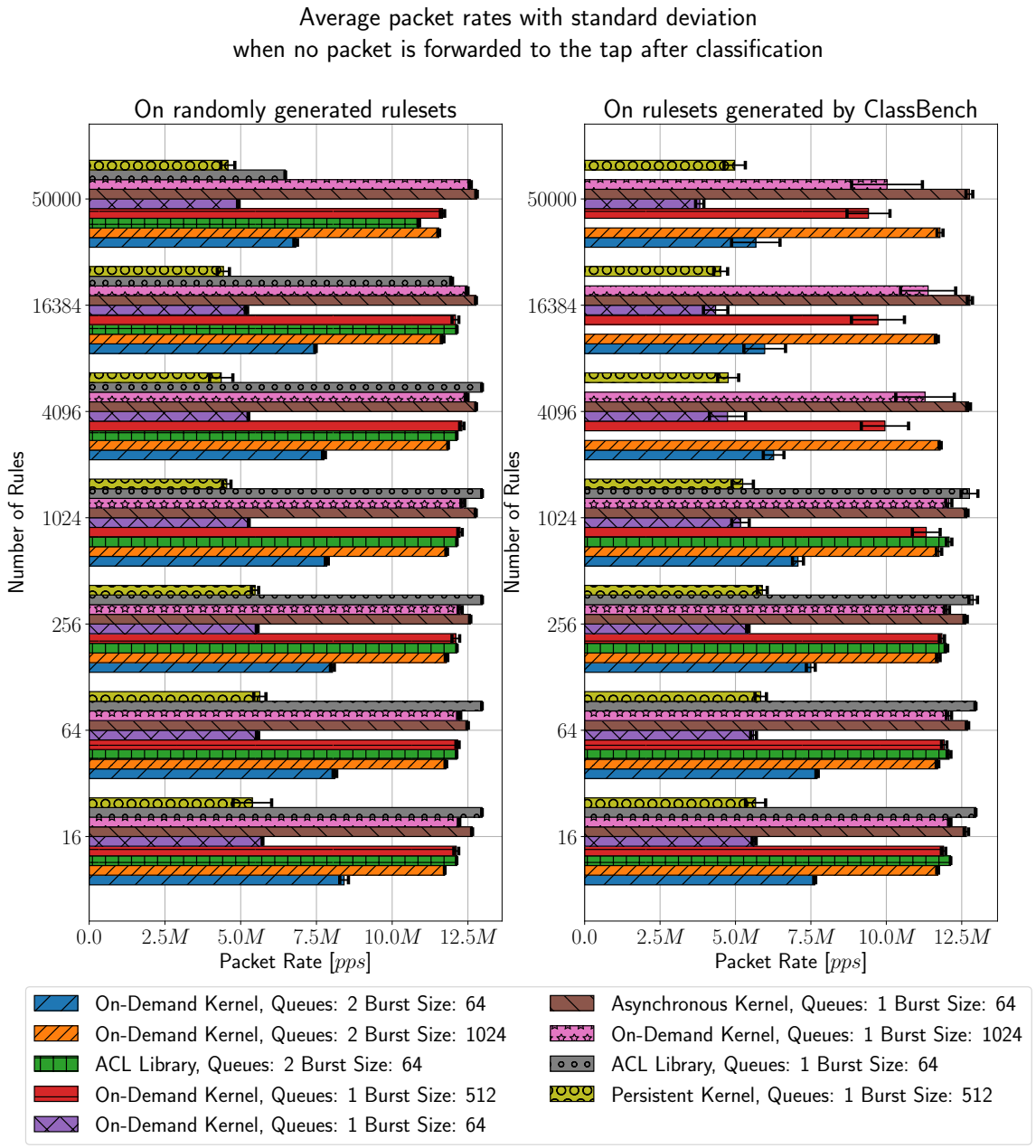


Fig. 4.4.: Packet rates on randomly and by ClassBench generated rulesets when no packet is forwarded to the tap after classification.

```
# randomly generated rules
# SRC IP  DST IP  SRC PORT  DST PORT  PROTO ACTION
188.182.235.0/24 184.116.18.0/21 23110-40408 26800-52967 79-144 DROP
40.237.6.0/19 253.70.230.28/31 23893-47378 8804-57092 28-171 ACCEPT
245.0.0.0/8 206.0.0.0/9 39370-50033 29349-59589 136-194 ACCEPT
23.0.0.0/8 168.94.42.0/22 4660-18767 31372-56559 181-189 DROP
135.195.127.0/23 164.80.248.1/25 31123-61433 26-5479 103-223 ACCEPT

# rules generated by classbench
# SRC IP  DST IP  SRC PORT  DST PORT  PROTO ACTION
177.73.69.57/32 15.66.202.19/32 119-119 0 DROP
177.237.31.1/32 15.66.202.19/32 123-123 0 ACCEPT
182.127.16.193/32 15.66.202.19/32 53-53 0 ACCEPT
180.227.86.43/32 6.226.51.166/32 123-123 0 ACCEPT
191.230.145.64/32 6.226.50.3/32 53-53 0 ACCEPT
```

Listing 4.1: excerpt from a randomly generated ruleset and a rulset from ClassBench

Latency Figure 4.5 shows the average RTT (round trip time) of a packet generated by a ping flood, where as many as possible ICMP echo requests[81] are sent to the firewall, depending on the rule size.

The measurement is done by using the *ping* tool from the iputils package[@22af]. The server in figure 4.2 has set an IP address on the corresponding NIC and the host has set an IP address in the same subnet on the tap interface. For getting the RTT depending on the number of rules, the measurement is done in the following way. For each number of rules n, m rulesets with n rules are created. For each firewall application, the average RTT on each of the m corresponding rulesets is measured for 10 seconds by using the *ping* tool on the server to send the ICMP echo requests to the host and receive the ICMP reply per packet. Out of the m RTTs, the average and standard deviation is calculated.

m is set to 4 for randomly generated rulesets and to 5 for rulesets generated by ClassBench. The RTT is measured for different kernel launching types, burst and queue sizes. Independently on the ruleset type, each rule accepts the packet and causes the firewall to forward it to the tap.

All configurations of the on-demand launched kernel have nearly the same RTT of $\sim 31\mu s$ independently from the number of rules. Hence, the burst or queue sizes have an very low effect on the RTT.

The RTT of the asynchronous launched kernel is on the same level as the on-demand launched one. Only the persistent kernel scales worse with the number of rules and has an at least ~ 3 times higher RTT than all other launching types. However, the ACL library of dpdk has the best RTT of $18\mu s$ which is almost half as large as the ones of the GPU-based classifiers.

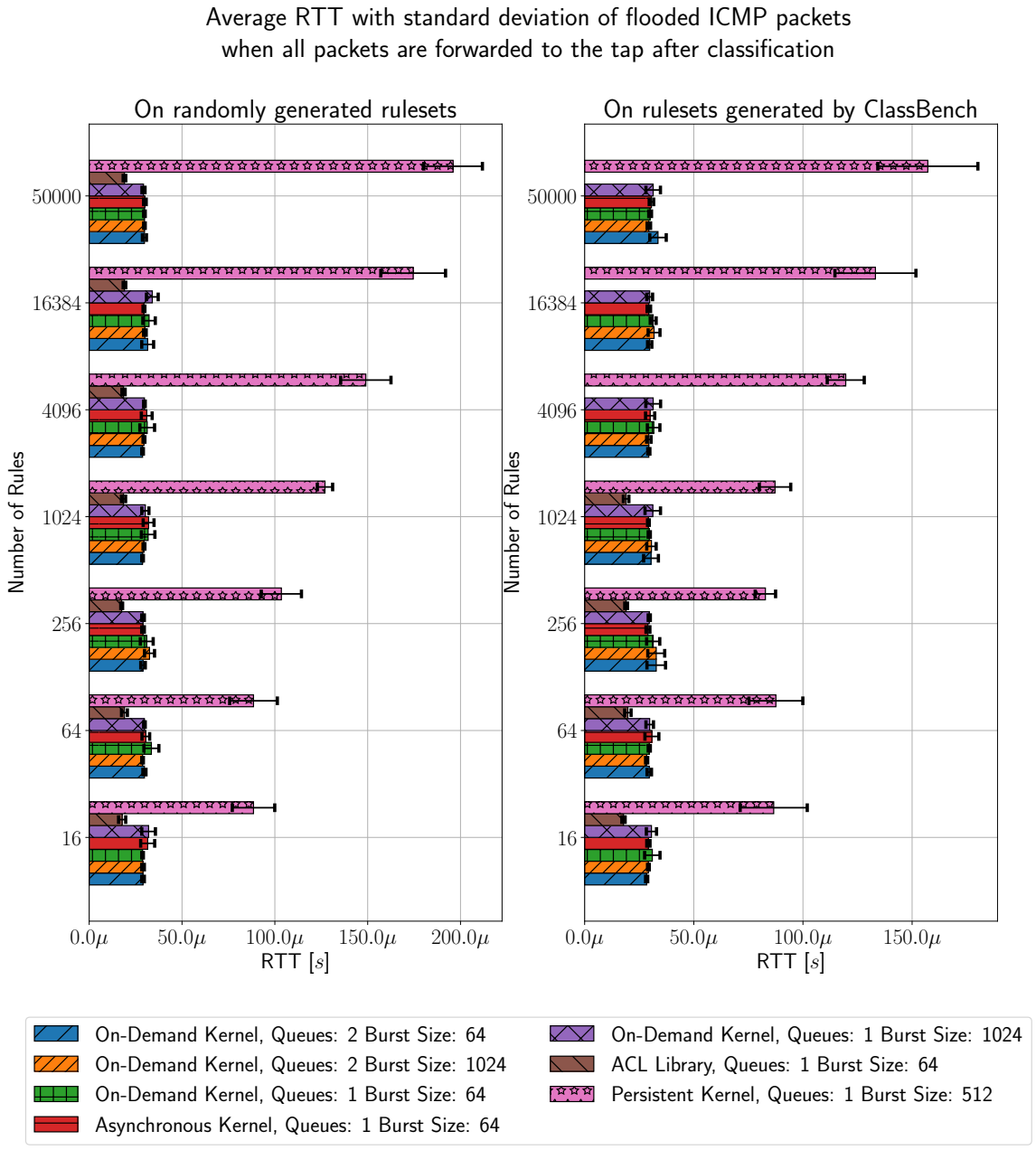


Fig. 4.5.: Average RTT with standard deviation of flooded ICMP packets.

CPU Utilization For getting an insight of the CPU utilization of the firewall proposed by this thesis, the number of used CPU cycles of the firewall configurations with the highest packet rates is measured using the Linux performance analyzing tool *perf* [@20]. For each number of rules n, m rulesets with n rules are created. For each firewall application, the number of CPU cycles is measured 3 times for 10 seconds for each of the m rulesets. Out of the $3 \cdot m$ measurements, the average and standard deviation is calculated. m is set to 4 for randomly generated rulesets and to 5 for rulesets generated by ClassBench.
Figure 4.6 shows the measured CPU cycles on randomly and by ClassBench generated rulesets. As one can see, the on-demand launched kernel needs fewer CPU cycles than the asynchronously called one, independently on the ruleset type. The higher number of CPU cycles of the latter one may be caused by its architecture (see 3.2.2). Instead of a single RX-thread per queue, one *enqueue*- and *dequeue*-thread is used. Since both of them poll on a queue or ring buffer, the overhead is much higher than the use of a single RX-thread like in the on-demand launched architecture (see 3.2.1). The on-demand launched kernel is also on the same level as the ACL library on randomly generated rulesets but scales much better with the rising number of rules. On rulesets generated by ClassBench, the classifier by dpdk fails to build its multibit tries for more than 1024 rules. However, one has to note that the measurement also includes the search structure generation phase of each classifier. Since the GPU-based classifier proposed by this thesis and the classifier by dpdk use different classification algorithms and thus generate their search structure in a completely different way, the gathered measurements need to be handled with care.

Summary In relation to the ACL library of dpdk, the classifier scales much better on rulesets generated by ClassBench in terms of packet rate. On randomly generated rulesets, it is also on a par with the ACL library and even slightly better. However, the performance highly depends on the launch type and packet burst size. The highest packet rate is achieved when the kernel is launched asynchronously. The on-demand launched classification kernel can keep up with the dpdk ACL library, if a burst size of 1024 packets is used. For smaller bursts, the performance is generally worse.
In terms of latency, the lowest ones of the GPU-based classifiers are the ones with on-demand or asynchronous launched kernels. But theirs are still two times higher than the classifier of the ACL library. Considering the CPU utilization, the classifier scales not worse than the one by dpdk and better on large rulesets. The persistent kernel has no advantage over all other launching methods and worse performance than all other.

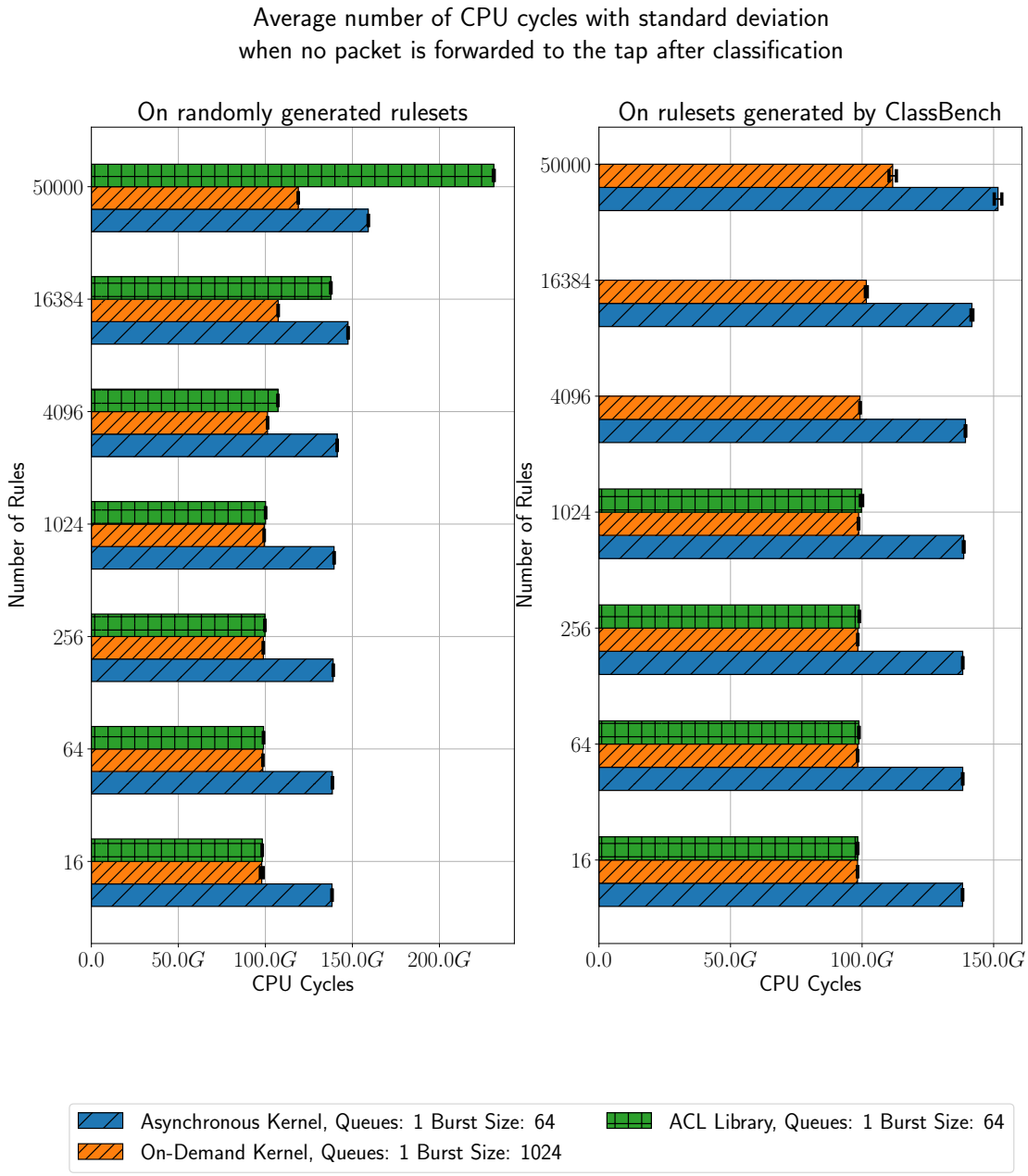


Fig. 4.6.: Average number of CPU cycles with standard deviation when no packet is forwarded to the tap after classification.

5 Conclusions

5.1 Offloaded Classification on GPUs

Table API Since the lookup method of the table API of dpdk [@22ak] is designed to do a synchronous lookup of a packet burst with at maximum 64 packets, the optimal configuration of the classifier cannot be achieved when the table API is used as it is currently designed. Much better performance of the GPU-based classifier is achieved by using bigger packet bursts. Hence, instead of providing a 64-bit packet mask to the classifier, the number of packets could be provided directly. Since the lookup function is designed for synchronized calls, the classifier can not use asynchronously launched kernels, but this launch type is the one with the best performance. Therefore, one may not use the table API at all for classification on the GPU.

Launching Types and Configuration The best performance is gained when large packet bursts (≥ 512) are classified by a kernel at once, since the GPU's multiprocessors then potentially hide latencies better. An asynchronous launching model has the most potential for classification on GPUs. It scales well with larger rulesets and, unlike the persistent model, only utilizes a GPU when packets are received. If the burst sizes are great enough, one may even synchronously launch the kernels.

Scaling on Rulesets The results show that the GPU-based classifier delivers the same or slightly better packet rate on randomly generated rulesets as the solution by dpdk. And on realistic rulesets, it scales much better (see figure 4.4) with the number of rules. Hence, a GPU-based classifier is ideal for environments where large rulesets are used.

Difficulties The main problem of external GPU-offloaded packet classification is the communication between host and GPU. The GPU must access packet data in host memory, which results into PCIe transfers between host and GPU. This leads to an higher processing latency than a CPU-only classifier has. For making the

overhead worthwhile, the burst size must be chosen properly. Additionally, the use of GPUs may require a adjustment of the processing architecture, since synchronous communication with the GPU and degrades the classifiers performance.

Offloading Offloading packet classification to GPUs can provide a great performance increase in terms of packet rate (see 4.5) and CPU utilization on large rulesets, when the right architecture and proper burst size is chosen. CPU-side threads are still needed but they are only used to receive and transmit packets, which allows the use of more CPU processing power for other tasks.
But the offloading comes with the price in form of an higher processing latency (see 4.5) which may be a problem depending on the use-case.
However, the results show that processing stacks can benefit from the combination of dpdk and GPUs for classification and create inducement for implementing other classifications as algorithms, like decision trees, as well.

5.2 Future Work

Hybrid Approach Since classifiers running on the CPU have a lower processing latency (see the ACL library of dpdk 4.5) but do not scale as well as GPU-based ones on larger rulesets, both of them could be used in combination. Then, depending on the current ruleset, one could use the CPU-based one for small sets and thus provide a high packet rate and low latency. After the rulesets reached a particular size, and the CPU-based classifier's performance decreases, one switches to the classifier running on a GPU. This hybrid approach may offer very good results in both latency and packet rate. The optimal border for switching can be found by analyses of benchmarks.

Other Classification Algorithms GPU implementations of other classification algorithms with different characteristics which can be used with dpdk would provide further insights of the potential use of GPUs for packet classification. Tree based algorithms like HyperSplit [Qi+09] also perform well on large rulesets and can be easily parallelized. HyperSplit creates a binary tree for which each traversal splits one dimension of the ruleset in half and thus narrows the ruleset down to the matching rules. But instead of splitting a dimension only in half, one could possibly split a dimension in $2 < k$ parts and let threads on a GPU compare the corresponding

header field of a packet with each part in parallel. Which would convert the tree's binary search into a k-ary search.

Packet Processing Model Instead of using a run-to-completion model, one could also use the pipelined [@22h] one, since the GPU-based classifier can be implemented with the table API of dpdk [@22v]. Additionally, the event-driven model [@22h] can be used with the asynchronous launching type of the kernel (see 3.2.2). Then the *dequeue*-thread can enqueue the classified packets in another ring which is used polled by the TX-threads.

Bibliography

[BV01] F. Baboescu and G. Varghese. “Scalable Packet Classification”. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 199–210 (cit. on p. 8).

[22z] *CUDA C++ Programming Guide*. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. 2022 (cit. on p. 15).

[Fil+15] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska. “Optimizing CUDA Code by Kernel Fusion: Application on BLAS”. In: *J. Supercomput.* 71.10 (Oct. 2015), pp. 3934–3957 (cit. on p. 22).

[GM00] P. Gupta and N. McKeown. “Classifying Packets with Hierarchical Intelligent Cuttings”. In: *IEEE Micro* 20.1 (Jan. 2000), pp. 34–41 (cit. on p. 8).

[GM99] P. Gupta and N. McKeown. “Packet Classification on Multiple Fields”. In: *SIGCOMM Comput. Commun. Rev.* 29.4 (Aug. 1999), pp. 147–160 (cit. on pp. 5, 8).

[Han+10] S. Han, K. Jang, K. Park, and S. Moon. “PacketShader: A GPU-Accelerated Software Router”. In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 195–206 (cit. on p. 2).

[81] *Internet Control Message Protocol, Echo or Echo Reply Message, p. 14*. RFC 792. Sept. 1981 (cit. on p. 46).

[LS98] T. Lakshman and D. Stiliadis. “High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching”. In: *SIGCOMM Comput. Commun. Rev.* 28.4 (Oct. 1998), pp. 203–214 (cit. on pp. 1, 5, 6).

[Qi+09] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. “Packet Classification Algorithms: From Theory to Practice”. In: *IEEE INFOCOM 2009*. 2009, pp. 648–656 (cit. on p. 52).

[Qu+15] Y. Qu, H. Zhang, S. Zhou, and V. Prasanna. “Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU”. In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2015, pp. 87–98 (cit. on p. 2).

[Sin+03] S. Singh, F. Baboescu, G. Varghese, and J. Wang. “Packet Classification Using Multidimensional Cutting”. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM 03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pp. 213–224 (cit. on p. 5).

[SSV99] V. Srinivasan, S. Suri, and G. Varghese. “Packet Classification Using Tuple Space Search”. In: *SIGCOMM Comput. Commun. Rev.* 29.4 (Aug. 1999), pp. 135–146 (cit. on p. 5).

[18] “Table 4–2—MAC parameters, interPacketGap and minFrameSize”. In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), p. 161 (cit. on p. 40).

[TT07a] D. Taylor and J. Turner. “ClassBench: a packet classification benchmark”. In: *IEEE/ACM Transactions on Networking* 15.03 (May 2007), pp. 499–511 (cit. on pp. 39, 40).

[TT07b] D. Taylor and J. Turner. “ClassBench: a packet classification benchmark, Chapter 3, Analysis of Real Filter Sets”. In: *IEEE/ACM Transactions on Networking* 15.03 (May 2007), pp. 499–511 (cit. on p. 40).

[Wik22] Wikipedia contributors. *Single instruction, multiple data — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-March-2022]. 2022 (cit. on p. 2).

[You+17] G. Younghwan, M. Jamshed, Y. Moon, C. Hwang, and K. Park. “APUNet: Revitalizing GPU as Packet Processing Accelerator”. In: (2017) (cit. on p. 2).

[Zhe+06] K. Zheng, H. Che, Z. Wang, B. Liu, and X. Zhang. “DPPC-RE: TCAM-based distributed parallel packet classification with range encoding”. In: *IEEE Transactions on Computers* 55.8 (2006), pp. 947–961 (cit. on p. 5).

[ZSP14] S. Zhou, Shreyas G. Singapura, and V. Prasanna. “High-performance packet classification on GPU”. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 2014, pp. 1–6 (cit. on p. 2).

Webpages

[@22a] *1.11. SIMD Intrinsics*. 2022. URL: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__SIMD.html#group__CUDA__MATH__INTRINSIC__SIMD (visited on Feb. 18, 2022) (cit. on p. 35).

[@22b] *1.4.1.3. Occupancy*. 2022. URL: https://docs.nvidia.com/cuda/turing-tuning-guide/index.html#sm-occupancy (visited on Feb. 18, 2022) (cit. on p. 16).

[@22c] *1.9. Integer Intrinsics, __clz*. 2022. URL: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html#group__CUDA__MATH__INTRINSIC__INT_1gd45548f57952ed410fa5d824984f16d0 (visited on Jan. 19, 2022) (cit. on p. 8).

[@22d] *1.9. Integer Intrinsics, __ffs*. 2022. URL: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html#group__CUDA__MATH__INTRINSIC__INT_1gaf1eb22243e29e0b7222adee8ae7d4e4 (visited on Jan. 19, 2022) (cit. on p. 8).

[@22e] *11. Poll Mode Driver*. 2022. URL: http://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html# (visited on Mar. 2, 2022) (cit. on p. 13).

[@22f] *11.4.3. Device Configuration*. 2022. URL: http://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html# (visited on Mar. 2, 2022) (cit. on p. 13).

[@22g] *14.1. CUDA Compute Capability*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#cuda-compute-capability (visited on Mar. 2, 2022) (cit. on p. 16).

[@22h] *2. Overview*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/overview.html (visited on Jan. 19, 2022) (cit. on pp. 10, 13, 53).

[@22i] *2.4.2. Use of Hugepages in the Linux Environment*. 2022. URL: https://doc.dpdk.org/guides/linux_gsg/sys_reqs.html (visited on Jan. 19, 2022) (cit. on p. 13).

[@22j] *3.2.6.2. Concurrent Kernel Execution*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#concurrent-kernel-execution (visited on Feb. 18, 2022) (cit. on p. 32).

[@22k] *3.2.6.5. Streams*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams (visited on Jan. 19, 2022) (cit. on p. 30).

[@22l] *4. Hardware Implementation*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation (visited on Feb. 18, 2022) (cit. on pp. 15, 16, 32).

[@22m] *4.1. SIMT Architecture*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture (visited on Feb. 18, 2022) (cit. on pp. 15, 16).

[@22n] *42. Multi-process Support*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html (visited on Jan. 19, 2022) (cit. on p. 14).

[@22o] *5.2.3. Multiprocessor Level*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#multiprocessor-level (visited on Feb. 18, 2022) (cit. on p. 30).

[@22p] *5.3.2. Device Memory Accesses*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses (visited on Mar. 2, 2022) (cit. on p. 15).

[@22q] *5.4.2. Control Flow Instructions*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#control-flow-instructions (visited on Mar. 2, 2022) (cit. on p. 15).

[@22r] *50. Tun|Tap Poll Mode Driver*. 2022. URL: https://doc.dpdk.org/guides/nics/tap.html (visited on Feb. 18, 2022) (cit. on p. 28).

[@22s] *52. Packet Classification and Access Control*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html (visited on Feb. 3, 2022) (cit. on pp. 2, 39, 44).

[@22t] *52. Packet Classification and Access Control, 52.1.1. Rule definition*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html (visited on Feb. 3, 2022) (cit. on p. 39).

[@22u] *52. Packet Classification and Access Control, 52.1.3. Classification methods*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html (visited on Feb. 3, 2022) (cit. on p. 39).

[@22v] *53.4. Table Library Design*. 2022. URL: http://doc.dpdk.org/guides/prog_guide/packet_framework.html#table-library-design (visited on Feb. 3, 2022) (cit. on pp. 21, 35, 53).

[@22w] *65. Writing Efficient Code*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/writing_efficient_code.html (visited on Jan. 19, 2022) (cit. on p. 11).

[@22x] *9. Mempool Library*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/mempool_lib.html#mempool-library (visited on Jan. 19, 2022) (cit. on p. 11).

[@22y] *B.6. Synchronization Functions*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#synchronization-functions (visited on Mar. 2, 2022) (cit. on pp. 16, 17).

[@22aa] *CUDA Device Memory Accesses*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses (visited on Jan. 19, 2022) (cit. on p. 17).

[@22ab] *CUDA Memory Hierachy*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy (visited on Jan. 19, 2022) (cit. on p. 17).

[@22ac] *CUDA Memory Hierachy, Constant Memory*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#:~:text=8.x%2C%20respectively.-,Constant%20Memory,-The%20constant%20memory (visited on Jan. 19, 2022) (cit. on p. 36).

[@22ad] *CUDA Memory Hierachy, Local Memory*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#:~:text=to%20these%20constraints.-,Local%20Memory,-Local%20memory%20accesses (visited on Jan. 19, 2022) (cit. on p. 17).

[@22ae] *CUDA Memory Hierachy, Shared Memory*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#:~:text=Capability%206.x).-,Shared%20Memory,-Because%20it%20is (visited on Jan. 19, 2022) (cit. on p. 17).

[@19a] *How can DPDK access devices from user space?* 2019. URL: https://codilime.com/blog/how-can-dpdk-access-devices-from-user-space/ (visited on Feb. 10, 2022) (cit. on pp. 10–12).

[@22af] *iputils package*. 2022. URL: https://github.com/iputils/iputils/ (visited on Mar. 3, 2022) (cit. on p. 46).

[@17] *isolcpus Linux kernel parameter*. 2017. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cpu-partitioning/isolcpus (visited on Mar. 3, 2022) (cit. on p. 41).

[@22ag] *Mbuf Library*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html (visited on Mar. 2, 2022) (cit. on p. 13).

[@22ah] *Mbuf Library, 10.6. Meta Information*. 2022. URL: https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html (visited on Mar. 2, 2022) (cit. on p. 13).

[@19b] *Memory in DPDK Part 2: Deep Dive into IOVA*. 2019. URL: https://www.dpdk.org/blog/2019/09/14/memory-in-dpdk-part-2-deep-dive-into-iova/ (visited on Feb. 10, 2022) (cit. on p. 11).

[@19c] *Memory in DPDK, Part 1: General Concepts*. 2019. URL: https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/ (visited on Feb. 10, 2022) (cit. on pp. 10, 11, 13).

[@21a] *Memory-mapped I/O — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-February-2022]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Memory-mapped_I/O&oldid=1060192503 (visited on Feb. 10, 2022) (cit. on p. 10).

[@20] *perf: Linux profiling with performance counters*. 2020. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on Mar. 6, 2022) (cit. on p. 48).

[@21b] *rand(3) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man3/srand.3.html (visited on Feb. 22, 2022) (cit. on p. 40).

[@22ai] *RTE Table ACL*. 2022. URL: http://doc.dpdk.org/api/rte__table__acl_8h.html (visited on Feb. 26, 2022) (cit. on p. 39).

[@22aj] *rte_ethdev.h, rte_eth_dev_configure()*. 2022. URL: https://doc.dpdk.org/api/rte__ethdev_8h.html#a1a7d3a20b102fee222541fda50fd87bd (visited on Mar. 2, 2022) (cit. on p. 13).

[@22ak] *rte_table.h*. 2022. URL: https://doc.dpdk.org/api/rte__table_8h.html (visited on Feb. 3, 2022) (cit. on pp. 21, 35, 51).

[@22al] *rte_table.h, rte_table_op_entry_add*. 2022. URL: https://doc.dpdk.org/api/rte__table_8h.html#aaab350845daf6098fe049d1868d078ea (visited on Mar. 3, 2022) (cit. on p. 21).

[@22am] *rte_table.h, rte_table_op_lookup*. 2022. URL: https://doc.dpdk.org/api/rte__table_8h.html#a9e058e310bef6b604ef665e070c88b41 (visited on Mar. 3, 2022) (cit. on p. 21).

[@22an] *Search Structure of ACL Library*. 2022. URL: https://github.com/DPDK/dpdk/blob/35cb5bd236301277d7fa7c571cfa57c07f5322af/lib/acl/acl.h#L41-L77 (visited on Feb. 26, 2022) (cit. on p. 39).

[@16] *sk_buff*. 2016. URL: https://wiki.linuxfoundation.org/networking/sk%5C_buff (visited on Feb. 10, 2022) (cit. on p. 10).

[@21c] *socket(2) — Linux manual page*. 2021. URL: https://www.man7.org/linux/man-pages/man2/socket.2.html (visited on Feb. 22, 2022) (cit. on p. 10).

[@22ao] *Table 15. Technical Specifications per Compute Capability*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications (visited on Jan. 19, 2022) (cit. on pp. 23, 32, 35).

[@22ap] *The Pktgen Application*. 2022. URL: https://pktgen-dpdk.readthedocs.io/en/latest/ (visited on Feb. 24, 2022) (cit. on p. 40).

Appendix

A

A.1 Source Code

The source code of the GPU-based classifier proposed by this thesis with all of its variants can be found under `https://github.com/daschr/cuda_firewall`.

A.2 Hash of PDF

The SHA256 sum of the corresponding PDF is

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.