Journal of Parallel and Distributed Computing 167 (2022) 136–147

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Efficient hierarchical hash tree for OpenFlow packet classification with fast updates on GPUs

Yu-Hsiang Lin, Wen-Chi Shih, Yeim-Kuan Chang [*]

Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan

A R T I C L E I N F O

Article history:
Received 17 June 2021
Received in revised form 8 April 2022
Accepted 21 April 2022
Available online 4 May 2022

Keywords:
Packet classification
OpenFlow
Hash table
GPU

A B S T R A C T

Packet classification is an important functionality of modern routers/switches, needed in packet forwarding, Quality of Service (QoS), firewall etc. In order to better utilize routers on the Internet, Software Defined Network (SDN) decouples control plane from data plane to fulfill centralized management. Based on OpenFlow standards, packet classification in SDN is designed for multi-field rules which are more complex than traditional 5-tuple rules. In the paper, we propose a novel packet classification algorithm, called hierarchical hash tree (H-HashTree), based on the two IP address fields and the 7 exact-match fields to partition rules into groups. An extended Bloom filter is also proposed to accelerate search process by skipping groups in the hash tree. To further improve the performance, H-HashTree is implemented on GPU. We tested on 100K rules including synthesized rules containing characteristics of ACL, FW, and IPC with different wildcard ratios in exact-match fields, and real OpenFlow rules from Open vSwitch. Compared with the existing state-of-the-art algorithms, CutTSS and TabTree [19][18], H-HashTree achieves the best performance on both search and update speeds. H-HashTree achieves 1.17-13.9 and 2.48-12.7 times faster in search speed and 2.03-6.0 and 1.87-4.53 times faster in rule updates from synthesized rulesets than CutTSS and TabTree, respectively. On the GPU platform, H-HashTree can achieve up to 114 MPPS in search speed and less than 0.04 usec/rule in rule updates.

1. Introduction

The research of packet classification algorithms must always be improved to meet the requirement of modern network. Packet classification [13] is a process to find out which flow each packet belongs to in the router. Each flow is specified by certain rules. The router is responsible for distinguishing packets into different flows. The router processes all packets belonging to the same flow followed by pre-defined rule and its action. A traditional rule contains five fields including Source and Destination IP addresses, Source and Destination ports and protocol number. Each rule has its own priority value The rule with highest priority is selected to be the final matched rule and is returned to the controller when multiple rules match against the incoming packet. Geometrically, high priority rules tend to overlay lower priority rules. The purpose of packet classification is to match incoming packets among thousands of rules (we refer to as rule table) as fast as possible. In legacy network system, the rule table is considered to be stable. Some packet classification algorithms are not capable of doing updates and must rebuild the whole data structure if necessary.

Nowadays, with the emerging of Software Defined Networks (SDN) [22], network system becomes more flexible, and much easier for administrators to control and manage. OpenFlow is one of well-known standards that support SDN and it provides more match fields than traditional 5-fields. Compared with legacy network system, SDN has a better ability to handle modern development of network applications and continuously changing of network requirements. These powerful properties rely on dynamically updating rules of the classifier and fast packet classification speed. As a result, focusing on fast updates are definitely essential for classifier today and in the future. The rise of general-purpose GPU (GPGPU) has made the implementation of GPU-assisted network applications easier than ever. Many GPU-accelerated network applications and frameworks [9][35][15][10] have adopted discrete GPU or APU to enhance their performance. The powerful arithmetic processing ability and fast context switch of GPU make the computation of hash procedure and large data (such as many-field OpenFlow packets, the number of flow match fields can up to 45 in OpenFlow version 1.5.1 [23] compared with traditional 5-field packets) more effective than modern CPU.

[*] Corresponding author.
E-mail addresses: c1884725@gmail.com (Y.-H. Lin), yotrew@ksvcs.kh.edu.tw (W.-C. Shih), ykchang@zmail.csie.ncku.edu.tw (Y.-K. Chang).

https://doi.org/10.1016/j.jpdc.2022.04.018
0743-7315/© 2022 Elsevier Inc. All rights reserved.

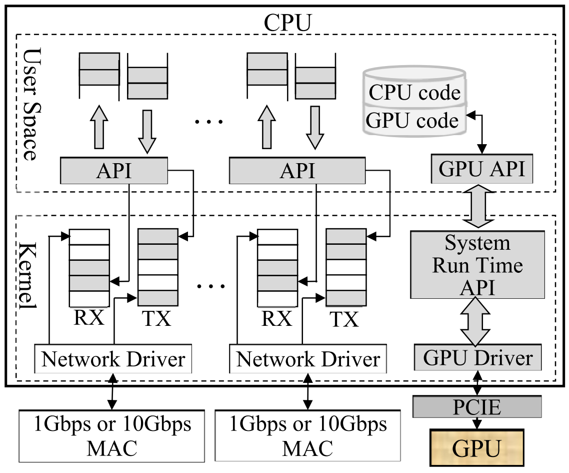

Fig. 1. Software architecture of the switch.

Fig. 1 shows the software architecture [32] of the switch that we consider in this paper. The switch is connected to the network by one or more 10Gbps MAC links. Also, the switch is connected with a GPU card via PCIe bus. CPU receives the packets from network and extracts the needed packet headers for packet classification. In CPU scenario, the classification data structure is constructed and stored in CPU and so the classification process is performed inside the CPU. By assuming minimum-sized 64-byte packets as in [11], one 10Gbps link accounts for 19.5M packets per second. This means that CPU has to be fast enough to classify 19.5M packets per second. Because packet's data is surrounded by an ethernet preamble, frame check sequence, ethernet epilogue, the effective throughout [1] that is only about 71.4% of the original throughput is 14.88 MPPS for packets of size 64 bytes. Our performance results shown in this paper indicate the highest throughput of the best existing schemes that CPU can support for 10K rules is 12 MPPS. As a result, the computation capability of CPU is not sufficient for this large amount of packets and so we may need to resort to GPU. In GPU scenario, the classification data structure is constructed in CPU and then copied to GPU and the packet headers are sent to GPU in a batch fashion. However, the bandwidth of PCIe bus between CPU and GPU must be large enough to transfer the headers from CPU to GPU. Consider the 257-bit 12-field headers that we study in this paper. PCIe 2.0 $\times$ 16 (64 Gbps) can transfer the headers of 249M packets. As a result, the bandwidth of PCIe 2.0 $\times$ 16 is capable of transferring the needed headers to GPU for packet classification. After finishing the packet classification process, GPU sends the results back to CPU.

In this paper, we introduce a packet classification algorithm suitable for classifying not only traditional 5-field packets but also OpenFlow packets. We build a hash-based decision tree, which we refer to as hierarchical hash tree (H-HashTree). Compared with [6], H-HashTree achieves high classification speed, fast rule updates and much more memory efficiency. H-HashTree outperforms the existing state-of-the-art algorithms CutTSS and TabTree [19][18] in both searching and updates.

The rest of this paper is organized as follows. In section 2, we review the recently proposed packet classification algorithms and in section 3, OpenFlow analysis and GPU are described. In Section 4, we introduce the proposed hierarchical hash tree (H-HashTree) data structure and enhancement by the extended Bloom filters. The experimental results are presented in section 5. This paper is concluded in the final section.

2. Related work

In this section, we first review some recently proposed packet classification schemes based on tuple space search for supporting fast update operations. Then, we analyze the flow tables of the OpenFlow. Finally, we introduce the GPU architecture that is our target.

The packet classification algorithms can be categorized into four types, namely, the exhaustive search, decomposition-based scheme, decision tree, and tuple space search. Exhaustive search schemes that check all rules include linear search and the TCAM schemes. Decomposition schemes first construct a search data structure for each field independently. Then all the data structures are searched by using the corresponding packet headers of the incoming packets. Finally, the search results from all search data structures are ANDed to find the final matched rules. The typical example is RFC [12]. Decision tree schemes first construct a decision tree based on the field values of all rules, and then perform the search operations by traversing the decision tree. HiCuts [13] and HyperCuts [29] are two most famous decision trees. Recently proposed decision trees include recursive endpoint-cutting (REC) [5], NeuroCuts [21], and CutSplit [20]. Decision trees have fast search performance, but their updates are slow and do not meet the requirement of SDN. Tuple space search (TSS) [30] partitions rules into many tuples according to the number of specific bits or other field properties. We have to search all the tuples against header values of the packets to find the final matched rule.

The advantages of TSS are that no rule is duplicated and the update operation is fast. In basic TSS, a hash table is constructed for each tuple to support fast rule update. The bits associated with each tuple are extracted form the packet headers to form a hash key to search the tuple. When the ruleset becomes large, it is possible to generate too many tuples and so the search performance will be degraded.

Since TSS can support fast updates, many recently proposed schemes that targeted on flow table lookups in SDN are based on TSS. One noticeable example is the priority sorting tuple space search (PSTSS) [26] in Open vSwitch (OVS). OVS is a software OpenFlow switch that is open-sourced and publicly available. OVS is designed for network management and protocol supports and used in NetFlow and sFlow. PSTSS records the highest rule priority of each tuple and all tuples are searched in the decreasing order of their priorities. When a matched rule is found and its priority is higher than the highest priority of the next tuple, the lookup process can be terminated.

Another TSS-based scheme called TupleMerge [7][8] reduces the number of tuples by relaxing the restrictions of using the same set of bits in one tuple in order to merge many tuples into one. TupleMerge controls the number of tuples to be merged by the following rule. If the number of collisions in hash tables is greater than a predefined threshold, TupleMerge will use more tuples to place these rules.

TabTree [18] a TSS-assisted bit-selecting tree that takes advantage of both decision tree and fast updates of TSS approaches. The construction of TabTree can be divided into three steps. Given an N-dimensional rule R $= (F_1, ..., F_i, ..., F_N)$ and a threshold value vector T $= (T_1, ..., T_i, ..., T_N)$, F_i is a *big field* if the range length of field $F_i >$ threshold value T_i; Otherwise, F_i is a *small field*. For a W-bit field F_i with threshold 2^K, F_i is a small field if and only if there are no wildcard (*) at its most significant W − K bits. These W − K bits can be called *selectable bits*. Each rule partition is built in a decision tree by selecting the most distinguishing selectable bits in each tree node. So, rules can be mapped into its children nodes in the most balanced fashion. There are two bit-selecting strategies, brute force and greedy described as follows.

In brute force strategy, we look for minimum standard deviation. Assuming there are M rules and B unused *selectable bits*. Find at most b bits at one-time from *selectable bits*, which partitions rules into $n = 2^b$ subsets in the most balanced fashion. We find the smallest *costFunc*(b bits) from all C_b^B combinations, where x_i is the number of rules in the i-th subset:

$$costFunc\,(bbits) = \sqrt{\frac{\sum_{i=1}^{n}(x_i - x)^2}{n}}, \quad \text{where } x = \frac{M}{n}.$$

In greedy strategy, we choose at most b bits by selecting one bit at a time, where each selected single bit is with the smallest *imbalance* value among current selectable and unused bits. The branching progress stops when the tree depth reaches a predefined value, or the number of rules in tree node is less than binth, or tree node cannot be further separated, or bit selecting will lead to rule duplication due to wildcard.

The third step checks the number of rules in each tree leaf. If # of rules $\leq$ binth, they use linear search. Otherwise, these rules are further constructed based on PSTSS. This scheme can avoid rule replication problem as well as create a relatively balanced tree. But the drawback is that the lookup process need to search all the subsets and thus increase the search time.

CutTSS [19] also uses concept of small fields to partition rules in a ruleset into subsets, similar to TabTree. There are several differences between the two. The first is to assign priority to each subset according to the maximum priority of all rules in the subset, and sort these subsets. Therefore, if the priority of the matched rule is greater than the priority of the next subset, the search process can be terminated. The second is to use the Fixed Cutting (FiCuts) algorithm to build the tree in the first stage. The difference between HiCuts and FiCuts is that the FiCuts is cut based on the small field of each subset, because it will not cut a large scale field, which can avoid the problem of rule replication. The third is to compute the search performance between linear search and PSTSS and determine which one is better. However, this scheme is likely to have poor performance in large scale classifier. One reason is that it does not limit the depth of the tree, which will cause more time to traverse the tree, and a more unbalanced tree. Another reason is that the rules stored in some leaf nodes are too many and so it becomes the bottleneck during searching.

In [33], a hierarchical 3-Layer Hash Tree based on the 7 exact-match fields is proposed. The 3-Layer Hash Tree is a Decision-tree extended by adding hash tables to process the fields with exact values. Because these exact-match fields contain only exact value or wildcard, we can simply group the rules with same type into a group, and each group has its own hash table. Because the exact-match fields contain only exact value or wildcard, we need 2^n hash tables if the layer contains n fields. A Bloom filter is used in each layer to detect whether some hash tables can be skipped to speed up the search performance. One disadvantage is the memory explosion on system memory. As shown in the experimental results of [33], the 3-Layer Hash Tree built for 12K rules takes up to 1190 MB memory space. The memory problem can be solved by hash table compression to reduce the memory to 70 MB. However, with the hash table compression, real time updates are no longer supported and so the needs of frequent rule updates in SDN environment can not be fulfilled. Another issue that the real time update can't be satisfied is the rule expansion in the third layer. In layer 3, they perfectly hash the rule with *VLAN priority* and *IP TOS* into 512-bit hash table. Also, the rule expansion in layer 3 results in 512 update operations.

In the first two layers, the Bloom filter and possibility bitmaps are used to skip unnecessary search in subgroups. The over-reliance of certain type of rules in Bloom filter restrains the Bloom filter's performance. Moreover, the worst case of updating Bloom filter needs up to 64 insertions in two hash tables which is intolerant in modern network environment. Cooperating with ignore flag, the Bloom filter can only work on small rulesets with limited variety of rule types.

The 3-Layer Hash Tree only focuses on using the 7 exact-match fields. We know that there are still some traditional 5-field rules in the ruleset, which resulting in performance bottleneck. That all 7 exact-match fields are wildcard means we can only search these rules linearly. The synthesized OpenFlow rulesets used in [33] exclude all traditional 5-field rules. Therefore, their experimental results cannot reveal the true performance if the algorithm is implemented in practical environment.

Varvello et al. [31] used GPU to accelerate linear search, tuple search, and bloom filters search. The bloom filters search algorithm achieves up to 115 MPPS classification speed for small rules-sets (up to 5k rules). However, the performance dramatically decreases to 20 MPPS for 100k rules due to increasing number of classes. The one-level entropy-based hashing scheme (named IG) was proposed recently in [11] to process packet classification in GPU. IG uses a hash function of the 18-bit hash key to split the original ruleset into small subsets. It requires only a 2MB database to store all rules. The classification phase uses the same hash function to calculate the hash value that points to the address of the 64-bit pointer array. The 64-bit pointer point to a sub-table which contains a sub-list of possible matched rules. Then, it uses a simple linear search to find the rule matching the incoming packet. In order to speed up searching, the authors proposed to use GPU to perform parallel linear search.

Table 1
Characteristics of OpenFlow rules (ClassBench-ng).

% of rules having wildcard in all 7 additional fields		
ruleset	OF1	OF2
percentage	2.14 %	1.50 %
% of rules having non-wildcard in Src or Dst MAC fields		
ruleset	OF1	OF2
percentage	97.33 %	98.31 %

3. Openflow analysis and GPU

3.1. Openflow table analysis

OpenFlow switch is composed of several flow tables, each of which is made up of many flow entries or called rules. In OpenFlow 1.0, rules have 12 fields that include 5 traditional fields and 7 additional fields. These 5 traditional fields are source and destination IP addresses (32 bits) in prefix format, source and destination Ports (16 bits) in range format and protocol number (8 bits) in exact-match format. The other 7 additional fields consist of one metadata field named ingress port and six exact-match fields, namely source MAC address (48 bits), destination MAC address (48 bits), Ethernet type (16 bits), Vlan ID (12 bits), VLAN priority (3 bits), and IP TOS (6 bits). The ingress port is implementation dependent but we use 16 bits as in Open vSwitch. The exact-match format means the values can be either a number or wildcard. This means these 7 fields are suitable for subgrouping rules with hash method, providing better performance in packet classification, especially in updates. Source and destination IP fields are prefixes of lengths 0 to 32, source and destination port fields are ranges, protocol field and the other 7 fields are exact-match fields. Ingress port does not define a specific size in the specification of OpenFlow 1.0 [24], it is a Metadata field that relates to the treatment of a packet, which is not extracted from the packet data itself and depends on what platform we implement OpenFlow Protocol. Open vSwitch is one of the most popular open source software virtual

Table 2
Rule type analysis of OF1.

1	2	3	4	5	6	7	8	9	
src ip	dst ip	src port	dst port	proto	ingress port	src mac	dst mac	eth type	Percent (%)
	✓		✓	✓			✓	✓	39.87
✓	✓			✓			✓	✓	33.02
✓			✓	✓		✓		✓	9.29
✓	✓						✓		7.63
							✓		3.19
	✓	✓	✓						1.87
	✓	✓	✓				✓		1.49
✓						✓			0.77
	✓						✓		0.64
✓				✓		✓		✓	0.63
					✓				0.51
						✓			0.30
	✓								0.26
	✓			✓			✓	✓	0.17
✓	✓	✓	✓			✓	✓		0.16
✓	✓	✓	✓				✓		0.11
✓									∼ 0.0

*17 types and Vlan ID, Vlan priority, IP ToS are all wildcards

Table 3
Rule type analysis of OF2.

1	2	3	4	5	6	7	8	9	
src ip	dst ip	src port	dst port	proto	ingress port	src mac	dst mac	eth type	percent (%)
✓	✓	✓	✓				✓		81.05
✓	✓			✓			✓	✓	7.88
	✓		✓	✓			✓	✓	5.65
✓	✓						✓		1.55
							✓		1.47
	✓	✓	✓						1.28
	✓								0.20
	✓	✓	✓				✓		0.18
					✓				0.18
✓						✓			0.16
	✓						✓		0.12
✓				✓		✓		✓	0.10
						✓			0.07
	✓			✓			✓	✓	0.03
✓									∼ 0.0

*15 types and Vlan ID, Vlan priority, IP ToS are all wildcards.

multilayer network switch that supports OpenFlow protocol. The proposed scheme in this paper follows the fields specified in the Open vSwitch Manual [25], so that the size of the Ingress port defined in the algorithm is 16 bits. Therefore, the total size of each rule in OpenFlow 1.0 is 253 bits which is two times longer than traditional 5-field rules of 104 bits. The differences between traditional rules and OpenFlow rules are such that we need a different way to deal with packet classification in order to sustain fast search as well as fast update in SDN environment.

From the observation of OpenFlow rules, most fields consist of exact values or wildcards, in particular, the Source and Destination MAC addresses. The MAC address consists of a 24-bit manufacturer prefix - Organizationally Unique Identifier (OUI) that uniquely identifies a vendor, manufacturer, or other organization and a 24-bit device identifier, meaning there are 2^{24} different combinations of device IDs within a single OUI.

In recent network environment, technology changes on both protocols (the uptake of IPv6) and network architectures (the adoption of Software Defined Networking, SDN). Especially, IPv6 and OpenFlow increase the complexity of rule matching problems so that researchers encounter new challenges on ASIC design. ClassBench-ng [16] provides IPv4, IPv6, and OpenFlow 1.0 classification rule sets taken from operational environments, where IPv4 and IPv6 prefixes have been taken from core routers. Classification rule sets come from Access Control Lists (ACLs) applied at a university network's perimeter, while the analysis of OpenFlow data is based on a set of Open vSwitches running in a cloud datacenter environment.

There are two OpenFlow seed files provided by ClassBench-ng, namely OF1 and OF2. We generate OpenFlow rulesets based these seeds and show the performance of our proposed method in later sections. Table reveals that some rules are equivalent to traditional 5-tuple rules whose 7 additional fields are all wildcard. Interesting results shown in Table conclude that a lot of rules have non-wildcard field values in source or destination IP address fields and prefix length of those non-wildcard source or destination IP address field concentrates on 30 to 32. According to the MAC address analysis for OF1 in Table 2 and OF2 in Table 3, we found that over 97% of rules have at least one non-wildcard MAC field. There are less than 3% of rules whose source and destination MAC addresses are wildcards at the same time. Most importantly, each MAC address is relatively different from each other. As a matter of fact, MAC address is expected to be unique. The uniqueness of MAC addresses in the rules results in low collision in hashing method. This is a critical attribute that we can use to divide rules into subgroups for packet classification.

3.2. GPU architecture

Discrete GPU plays the role of a PCIe peripheral device to communicate with host machine via PCIe bus. GPU consists of hundreds of thousands of cores (e.g. 2304 cores in AMD RX580) and

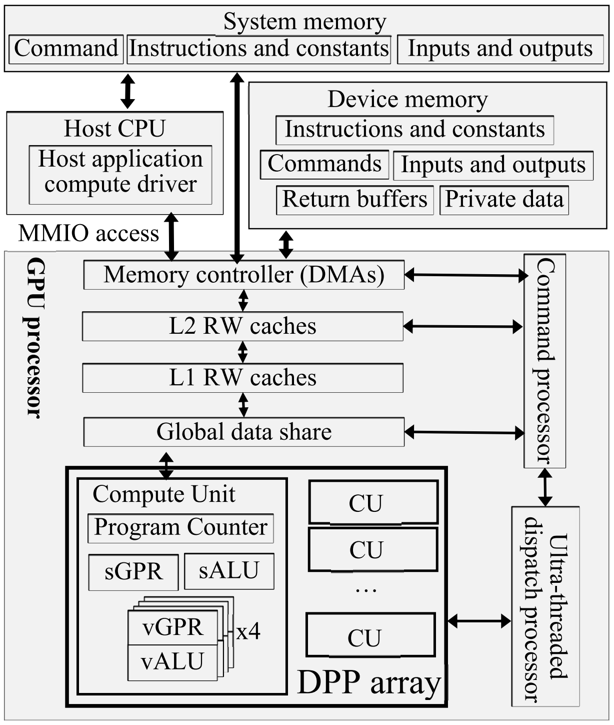

Fig. 2. "Radeon" GPU Architecture.

GDDR memory with high memory bandwidth (e.g. 8 GB GDDR5 with 256 GB/s in AMD RX580). The program processed by GPU is called kernel that is a function with special prefixes (e.g. __global__ or __device__). Kernel codes contain single instruction and multiple data (SIMD) instructions executed by a group of threads with different data under the same instruction stream in a lock-step manner (called a warp in NVIDIA or a wavefront in AMD hardware). The benefits of GPU over CPU include powerful computation capability on arithmetic operations supported by runtime API and fast hardware thread switching that can hide memory access latency.

In this paper, the target GPU is AMD rx580 that utilizes Graphics Core Next (GCN) 4 (generation 4) architecture whose instruction set architecture is the same as GCN 3. From the reference guide of AMD GCN 3 architecture [2], we show an abbreviated hardware architecture of GPU processor in Fig. 2. The main components of GPU processor include a data-parallel processor (DPP) array, a command processor, a memory controller, and an ultra-threaded dispatch processor. The GCN command processor reads the commands which the host has written to the register in the system memory address space. After the operations of the command complete, the command processor sends hardware-generated interrupts to the host. The GCN memory controller can directly access GCN device memory and host-specified areas of system memory without the participation of the processor, which eliminates large overhead from interrupt calls. The ultra-threaded dispatch processor (scheduler) is responsible for the fast context switching between threads. Threads are put to sleep when they need resources with high response latency like memory access.

The host applications cannot write data directly to the device memory of GPU and they can only instruct GPU to copy data between system memory and device memory. This can be achieved in the following two ways.

1) Request DMA engine on GPU to write data from the address of the source data in CPU memory to the location at the offset in GPU memory. For example, in AMD ROCm HIP, it is done by calling hipMemset or hipMemcpy.
2) Upload a kernel function to run on GPU and access the memory through PCIe bus, then process and store it in GPU memory. This concept is similar to unified virtual memory (UVM) where GPU can access any page of the entire system memory and simultaneously copy the data on-demand to its own device memory for high bandwidth access.

There are 4 separate SIMD units in each CU for vector processing, each of which consists of 16 vALU associated with vGPR. The vALU-assisted instructions perform an arithmetic or logical operation on data for each thread. As a result, each CU can operate on 64 threads at the same time. A group of 64 work items (data) is called wavefront. Each wavefront has a single program counter and is the minimum granularity for work on CU. When a wavefront is created, the program counter is initialized to the first instruction of the program. In AMD GPU with GCN architecture [3][4], each CU contains 4 SIMD units for vector processing. Each SIMD unit simultaneously executes a single operation on a wavefront of 16 work items (data), but each can be working on a separate wavefront. Also, each 16-wide SIMD processes one 64-wide wavefront over 4 clock cycles. In each clock cycle, the scheduler issues one vector instruction of the wavefront that is active on CU to one of the four SIMD units. So over 4 cycles each SIMD has an instruction issued to it and the scheduler is back to the start.

Streams allow the overlap of computation time of CPU receiving packets of the next batch and communication time of GPU classifying the packets of the current batch. In other words, a kernel from one stream is running in GPU, while a data copy from another stream is also running in parallel. As a result, the overall GPU utilization is improved with concurrent memory copy and kernel execution.

The Radeon Open Compute (ROCm) project [27] is an open source development platform built for High Performance Computing (HPC). The ROCm ecosystem is comprised of technology including math library, a set of tools (e.g. debugger, performance analysis etc.), programming models (e.g. OpenMP, HIP, OpenCL etc.) or frameworks (e.g. TensorFlow, PyTorch etc.). Heterogeneous-Compute Interface for Portability (HIP) [28] is a clang-based C++ runtime API and GPU language to create portable GPU programs for AMD and NVidia devices in a single source code. The compiler of HIP programing model is called HIPCC and it can be used to replace nvcc in existing CUDA code. HIPCC will call nvcc or hcc depending on what platform the code is running and include appropriate platform-specific headers and libraries.

4. Proposed scheme

In this paper, we propose a scheme called hierarchical hash tree (*H-HashTree*) that carefully selects some fields to divide the rules into groups. A hash table is constructed from the rules of each group to further divide the rules into buckets. The rules are indexed into buckets of hash tables with simple hash functions which perform XOR operations on the hash keys. If the number of rules in a bucket exceeds a predefined threshold, we further partition the bucket into smaller subsets using other fields. If the number of the rules in a bucket is smaller than the threshold, we simply save these rules in the linked list for linear search. While classifying packets, the needed header values are extracted from each packet header to form the search key for the same hash function used in the hash table construction process to locate the bucket for further search. Each packet will eventually reach some buckets using these hash functions. We fully match these candidate rules against the packet headers to determine which rules really match the packet. If there are multiple matched rules, we choose the rule with the highest priority and output its rule action for this packet. However, if there isn't any rule matched, we shall not receive this packet and just drop it.

As described earlier, OpenFlow rules are longer than traditional 5-D rules and the fields other than IP address and port fields are in the exact-match format that are either exact values or wildcards. Table 1 shows that the prefixes of two IP address fields of OpenFlow rules are very specific. The recently proposed TSS based algorithms [19][18][7][20][8][34] focused on partition rules into tuples based on these two IP address fields. For OpenFlow rules with long prefixes in IP address fields, it is hard to gain much benefits from these algorithms since the goal of tuple space search is to scatter the rules into a reasonable number of groups based on prefix lengths. Because of the characteristics of OpenFlow rulesets and the degeneracy of TSS-based algorithms built for these rulesets, we must find another method to support both fast search speed and real time updates. Compared with the existing schemes, we will show that the proposed scheme performs better in both search and update.

The proposed H-HashTree consists of three parts: main hierarchical hash tree, extended Bloom filter and extra hierarchical hash tree. The main hierarchical hash tree has 4 layers as shown in Fig. 3. In each layer, we use certain fields to partition rules into groups and then the rules of each group is stored in the corresponding hash table. If the number of rules in any entry (i.e., bucket) of the hash table exceeds a predefined threshold, we put these rules in next layer by further partitioning them into groups based on some unused fields. In layer 1, we partition rules into four (2^2) groups by checking if source and destination MAC address fields are wildcard or not. In layer 2, we partition rules into eight (2^3) groups by checking if ingress port, Ethernet type and Vlan ID fields are wildcard or not. In layer 3, we partition rules into four (2^2) groups by checking if the prefix length of source and destination IP address fields are longer than a predefined threshold or not (we use length of 16 in the experiment). In the final layer, we partition rules into four (2^2) groups by checking if Vlan priority and IP ToS fields are wildcard or not.

The extended Bloom filter stores bitmasks for the purpose of detecting if there are any hash tables in layers 1 and 2 that do not need to be searched for an incoming packet. Any rule that cannot be processed by extended Bloom filter is inserted into extra hierarchical hash tree, e.g. rules with wildcards in all four fields, Ethernet Type, VLAN ID, Source MAC, and Destination MAC. The extra hierarchical hash tree is constructed by 2 layers in which the first layer is identical to the third layer of main hierarchical hash tree (partitioned by source and destination IP address fields). In the second layer, we partition rules into eight (2^3) groups by checking if ingress port, Vlan priority and IP ToS fields are wildcard or not. Fig. 4 shows the search algorithm of the proposed algorithm, where the part of layer-3, layer-4, extra-layer-1, and extra-layer-2 are omitted because they are similar to the first two layers.

4.1. Main hierarchical hash tree

In the first layer, we choose source and destination MAC fields that can uniformly distribute the rules in a ruleset into a number of subsets. Four groups are formed according to whether source and destination MAC field are wildcard or not. Each group is implemented as a hash table in which each entry stores either a pointer to layer-2 node or a pointer to a bucket. Based on our analysis shown in previous subsection, almost every OpenFlow rule contains a non-wildcard source or destination MAC address field. Source and destination MAC addresses are the longest fields among all fields that can provide outstanding diversity of field values for efficient partitioning, as shown in Table 2 and Table 3, for OF1 and OF2 rulesets of 30K rules, respectively. The collisions of field value in source and destination MAC addresses nearly do not exist, which means the field value of most rules containing non-wildcard MAC address is easy to be used to distinguish from each other. The pseudo code of the hash function for MAC addresses is shown in Fig. 5.

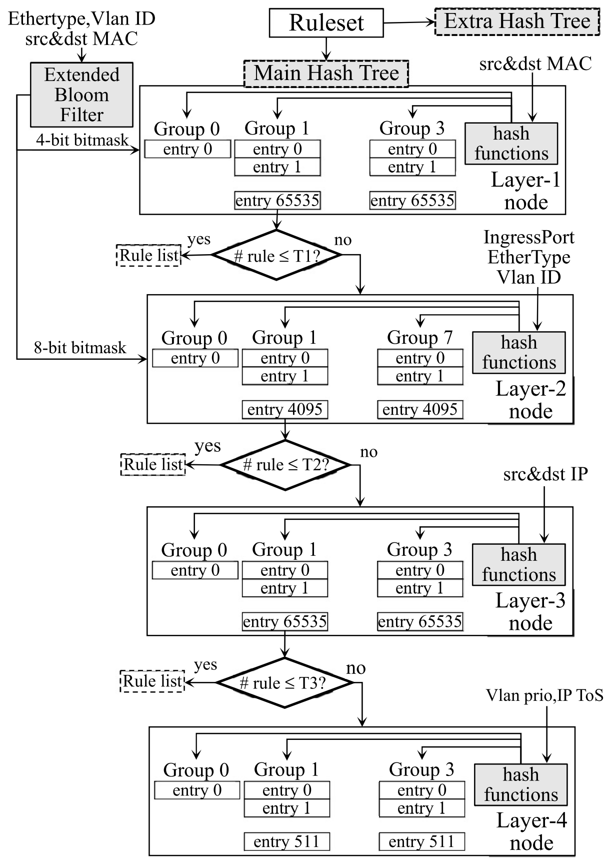

Fig. 3. Overview of Main Hierarchical Hash Tree.

```
//final  and mask are global variables;
L1_search(layer1, pkt)
{
  mask = BloomFilter(pkt.EtherType, pkt.VLANID,
                     pkt.srcMAC, pkt.dstMAC);
  for (g=3; g>=0; --g) { // from Group 3 to Group 0
     if (mask[g] == 0) continue;
     value = L1_hash(pkt.srcMAC, pkt.dstMAC, g)
     entry = layer1[g][value];
     if (entry.ptr != NULL) {
          if (entry.type == bucket) match = Linear_search(entry.ptr);
          else match = L2_search(entry.ptr, pkt);
          if (match.priority > final.priority) final = match;
     }
  }
}
L2_search(layer2, pkt)
{
  for (g=7; g>=0; --g)  {// from Group 7 to Group 0
     if (mask[g+4] == 0) continue;
     value = L2_hash(pkt.VLANID, pkt.EtherType, pkt.ingressPort, g)
     entry = layer2[g][value];
     if (entry.ptr != NULL) {
          if (entry.type == bucket) match = Linear_search(entry.ptr);
          else match = L3_search(entry.ptr, pkt);
          if (match.priority > final.priority) final = match;
     }
  }
}
```

Fig. 4. Layer-1 search algorithm.

```
value = 0;
if(srcMAC ≠ *){
    value ^= (srcMAC & 0xFFFF)
    value ^= (srcMAC & 0xFFFF0000) >> 16
    value ^= (srcMAC & 0xFFFF00000000) >> 32
}
if(dstMAC ≠ *){
    value ^= (dstMAC & 0xFFFF)
    value ^= (dstMAC & 0xFFFF0000) >> 16
    value ^= (dstMAC & 0xFFFF00000000) >> 32
}
```

Fig. 5. The 16-bit XOR folded hash function for MAC addresses of layer-1.

In layer 2, we follow the grouping strategy similar to layer 1. But this time, rules are divided into 8 groups by using Ingress port, Ethernet type and Vlan ID fields. In general, all these 8 groups have to be searched for each packet. A layer-2 node is comprised of 8 hash tables of size 4096 entries. To compute the group ID to which a rule belongs, ingress port, Ethernet type and Vlan ID fields correspond to b_2, b_1, and b_0 of group ID $b_2b_1b_0$, respectively. If a rule has a non-wildcard value in the field corresponding to b_i for i = 0 to 2, $b_i = 1$ for group ID of this rule. Otherwise, $b_i = 0$ for group ID of this rule. For example, if only Ethernet type field value of a rule is non-wildcard, this rule will be put in group 2. Similar to layer-1 node, each entry in the hash table saves either a pointer to layer-3 node or a pointer to a bucket.

In layer 3, we use two IP address fields to divide rules into 4 groups by checking whether the source or destination IP prefix length is longer than 16. All these 4 groups are implemented with hash tables of sizes 65526. Since there is no Bloom filter support, all four groups have to be searched for each packet. The reason why we separate rules with prefix length 16 is that most rules are already classified in the linear search buckets in the previous layers, only few rules left over from the buckets with # of collision more than predefined threshold. We don't have to separate rules into fine-grained tuples like tuple space search does. Most tuples won't be used, and the number of rule types is not too many. Therefore, we can simply classify rules into four groups. Also, OpenFlow and ACL rules tend to contain long prefix length, IPC and FW rules contain considerable amount of short prefix length. We use prefix length 16 as a compromised criterion for grouping rules.

In the last layer of main hierarchical hash tree, we use the remaining two fields to classify rules, which are Vlan priority and IP ToS. The two fields are the shortest among exact match fields, providing less ability to classify rules into subgroups by hash method. As a result, we apply the two fields in the final layer. Although Vlan priority and IP ToS are all wildcard in OpenFlow rules [16], we still implement the hash table without losing generality because Vlan ID is of size 16 bits which theoretically provide great ability to classify rules if it is non-wildcard. The same reason can be applied to Vlan priority and IP ToS, but they are too short and cannot provide good hash performance for rule grouping. We retain these two fields in layer-4 hash table since we expect that after distributing rules onto the first three layers, each resulting group in layer 4 is small enough to make Vlan priority and IP ToS work without too many collisions. We separate rules into 4 groups by checking whether Vlan priority and IP ToS are wildcard or not. All these four groups are of size 65536 hash tables and have to be searched for each packet. All rules in the hash table entries are saved in linked lists.

4.2. Extended bloom filter phase

We propose an extended Bloom filter to be used in the first two layers of the H-HashTree in order to accelerate the search process.

	Eth	VlanID	SrcMac	DstMac	Ingress
R1	65535	*	FF:00:00:FF:FF:00	*	1
R2	65535	3	*	FF:01:00:FF:FF:00	*

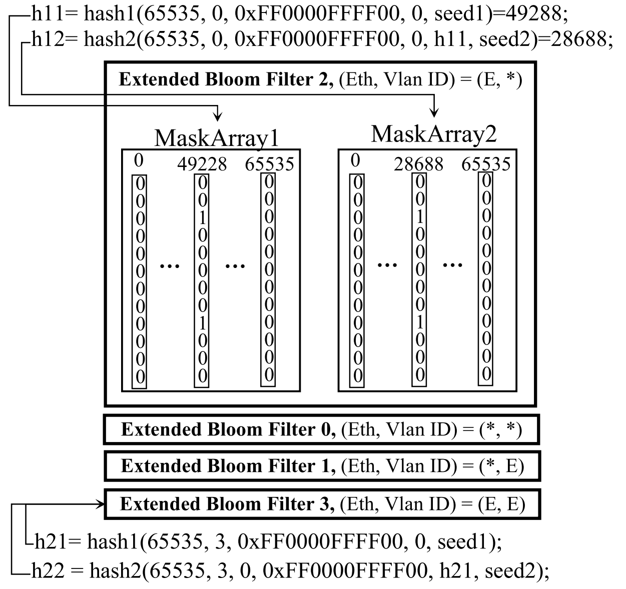
h11= hash1(65535, 0, 0xFF0000FFFF00, 0, seed1)=49288;
h12= hash2(65535, 0, 0xFF0000FFFF00, 0, h11, seed2)=28688;
Extended Bloom Filter 2, (Eth, Vlan ID) = (E, *)
MaskArray1
0 49228 65535
MaskArray2
0 28688 65535
Extended Bloom Filter 0, (Eth, Vlan ID) = (*, *)
Extended Bloom Filter 1, (Eth, Vlan ID) = (*, E)
Extended Bloom Filter 3, (Eth, Vlan ID) = (E, E)
h21= hash1(65535, 3, 0xFF0000FFFF00, 0, seed1);
h22 = hash2(65535, 3, 0, 0xFF0000FFFF00, h21, seed2);

Fig. 6. Diagram of extended Bloom filter insertions.

There are four extended Bloom filter for the rules whose Ethernet Type and Vlan ID field values are in the form of (*, *), (*, E), (E, *), and (E, E). We associate two bits, b_1 and b_0, of a 2-bit ID with Ethernet Type and Vlan ID field, respectively. Then, b_1 is set to 1 if Ethernet Type field value of a rule is non-wildcard; otherwise, b_1 is set to 0. The bit b_0 is set similarly for Vlan ID field. As a result, if a rule is associated with ID $= i$ for $i = 0$ to 3, it will be inserted into extended Bloom filter i.

Each extended Bloom filter consists of two bitmask arrays, namely MaskArray1 and MaskArray2, hashed by two hash functions based on 4 fields, Ethernet Type, Vlan ID, source MAC, and destination MAC fields. The hash functions for these two arrays are as follows.

$$h1 = \text{hash1}(\text{Eth}, \text{VlanID}, \text{SrcMac}, \text{DstMac}, \text{seed1});$$

$$h2 = \text{hash2}(\text{Eth}, \text{VlanID}, \text{SrcMac}, \text{DstMac}, h1, \text{seed2});$$

The entries of these two arrays are in the form of 12-bit bitmasks that will be explained later. When computing the hash functions, the wildcard value is treated as zero in these four fields. The bitmask saved in extended Bloom filter consists of 4 bits for layer-1 node and 8 bits for layer-2 node. A set bit in the bitmask means we have to search the corresponding group. An unset bit in the bitmask means any matched rule does not exist in the group definitely.

Fig. 6 shows an example that rule R1 should be put into group 2 in layer 1 and into the group 4 in layer 3 if the bucket of R1 in layer-1 node exceeds the threshold. As a result, the 12-bit mask is computed to be (001000001000). Since R1 has the field values of (E, *) in Ethernet Type and Vlan ID fields, it is inserted in to extended Bloom filter 2. As shown in the figure, hashed indices h11 and h12 are assumed to be 49228 and 28688, respectively. Therefore, the computed mask of (001000001000) will be put into the 49228th entry by ORing it with the old mask in MaskArray1. Similar operations are performed in MaskArray2. Fig. 6 also shows that another rule R2 should be inserted into extended Bloom filter 3 with the computed mask of (001000100000).

If rules have wildcard values in all Ethernet Type, VLAN ID, source MAC and destination MAC fields, we do not add them in the extended Bloom filters because they cannot be scattered by the hash functions. Instead, we build an extra hierarchical hash tree for them. The benefit of the extra hash tree is that we turn to search a hash tree with less layers and less rules which potentially speed up searching process.

When a packet comes in with headers of (Ether Type, Vlan ID, srcMac, dstMac) = (65535, 4095, FF:00:00:FF:FF:00, 00:00:00:00:00:FF), we have to search all four extended Bloom filters. We have to form four input headers to search extended Bloom filter 2. These four input headers are (65535, 4095, 0, 0), (65535, 4095, 0, 00:00:00:00:00:FF), (65535, 4095, FF:00:00:FF:FF:00, 0), and (65535, 4095, FF:00:00:FF:FF:00, 00:00:00:00:00:FF). When processing one of the four input headers, we use hash1() function to search MaskArray1 and hash2() function to search MaskArray2. Two 12-bit masks are obtained and they are ANDed together. Then, from the four input headers, we obtain 4 bitmasks that again are ORed together to have the intermediate mask for one extended Bloom filter. All four intermediate masks that are computed for extended Bloom filters 0 to 3 are ORed together to obtain the final 12-bit mask.

4.3. Extra hash tree

The extra hash tree consists of two layers. Its layer1 node is exactly the same as layer-3 node of the main hierarchical hash tree. The layer-2 node uses Ingress Port, VLAN priority, and IP ToS to build the hash tables of size 512 because Ethernet Type and VLAN ID are both wildcards. All these 4 groups have to be searched for each packet. The processes of insertions, deletions and searching are the same as the main hierarchical hash tree.

4.4. Stream reduction for update

In GPU processing, each thread is responsible for a rule update (either insert or delete). Assuming the data we need already resides in GPU, and device memory is pre-allocated on host side. Each thread deals with the necessary steps for updating rules, and finally reaches the bucket where the rule is going to be inserted or deleted. It is known that a basic data parallel model on GPU is called a wavefront. A wavefront groups every 64 data (we will refer to data as rules later) for parallel execution. If we have to update multiple rules in a single bucket using wavefront, modification operations by multiple threads at the same time may cause race condition because rules are sorted by priority in the bucket.

If we are in multi-core CPU scenario, we may think of applying critical section such as spin lock to each bucket. But in GPU, applying critical section to the bucket may cause dead lock due to lock-step execution. Thus, it is necessary to implement a spinlock-free update for GPU.

We separate update process into three steps: finding bucket ID, streaming reduction and bucket condensing. Each step is blocked by a global fence, which means there will be three GPU kernel functions for each step and executed sequentially.

1) Finding bucket ID:

We map each rule to a specific thread on GPU and locate buckets that need to be modified. If the update operation is insertion, we append rules to the end of the buckets. If the update operation is deletion, we disable the rule and leave it blank in the bucket. We do not sort rules with priority from high to low in this step because of race condition between threads.

2) Streaming reduction:

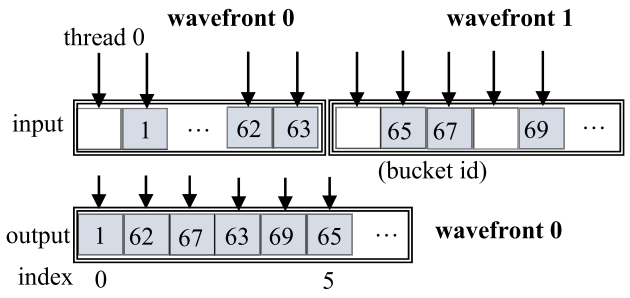

Fig. 7. Basic idea of streaming reduction.

To make sure that all rules in the buckets sorted from high to low priority, we map each bucket to a specific thread in order to avoid race condition. In this case, some buckets will be updated, but others will not, which leaves some of the threads idle in SIMD computing process. To avoid computing resource wasted from sparse data sets, we re-arrange the buckets so that the buckets which need to be updated are piled together as shown in Fig. 7. The goal we are trying to reach is similar to *Prefix Sum* [14] but is simpler and causes less overhead because we do not need data permutation to be static. In the figure, we mark the bucket IDs which the threads need to modify. The GPU threads just modify the buckets corresponding to the aligned bucket IDs.

3) Bucket condensing:

As long as the buckets are re-arranged, we fully utilize the computing resource of GPU and continue to focus on sorting bucket contents left from step 1. If the update operation is insertion, we traverse from the tail of the buckets and put the appended rules back to the correct position of the buckets. If the update operation is deletion, we cover the blanked positions by shifting the rule and adjust the array to the correct size.

4.5. Extended counting bloom filter for update

Counting Bloom filter [17] promotes Bloom filter by using a counter instead of a single bit for each entry in the hash table. When an element is inserted into counting Bloom filter, the corresponding counters are incremented, deletions can be done by decrementing the counters. Counters with non-zero value in counting Bloom filters share the same meaning to bits which sets to 1 in Bloom filters. Once the counter is zero after a serial of deletions, the entry can safely set to zero in Bloom filter. We apply the idea of counting Bloom filter to replace original extended Bloom filter in the proposed scheme, including 4-bit masks in layer-1 node and 8-bit masks in layer-2 node of main hierarchical hash tree.

5. Experimental results

We implemented the proposed algorithms on both CPU (i.e., host) and GPU (i.e., device). The CPU is an 6-core AMD Ryzen-5 2600x @ 3.6 GHz with 16 GB DDR4 RAM, and the GPU is an AMD Radeon RX580 with 8 GB GDDR5 RAM. The GPU contains 36 Compute Units (CUs) @ 1257 MHz, each of which contains 64 stream processors. Therefore, the GPU has 2304 stream processors so that we can run 2304 threads concurrently. The operating system of host is Ubuntu 18.04.3 LTS (bionic). The compiler used for GPU is HIPCC from AMD ROCm, introduced for compiling Heterogeneous Compute Interface for Portability (HIP) programs. HIPCC is a clang-based C++ compiler appended with AMD GPU API. For CPU programs, compiler is the GNU Compiler Collection (gcc, g++).

We use the rulesets generated from ClassBench-ng [16], which provides four different characteristics of filter seeds: ACL (ACL1∼5),

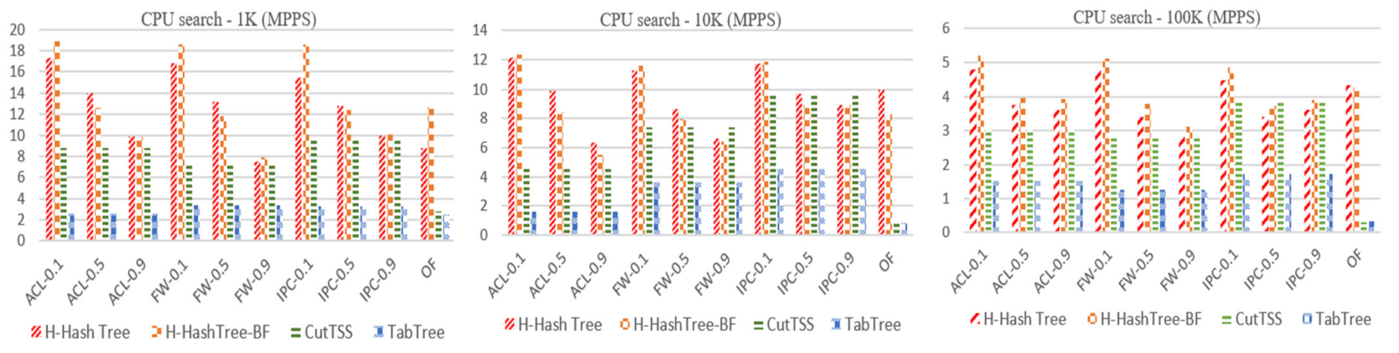

Fig. 8. CPU search performance of 1K, 10K, and 100K rules.

Table 4
Prefix length distribution of OpenFlow rules.

ruleset	Percentage	Src IP	Dst IP
OF1	non-wildcard values	51.61%	85.22%
	prefix lengths 30-32	51.39%	85.21%
OF2	non-wildcard values	90.74%	97.94%
	prefix lengths 30-32	90.62%	97.91%

FW (FW1∼5), IPC (IPC1∼2) and OF1/OF2. OF1 and OF2 rulesets are real OpenFlow 1.0 rulesets collected from Open vSwitch, but ACL, FW and IPC rulesets are traditional 5-tuple rulesets. In order to generate OpenFlow rulesets containing characteristics of ACL, FW and IPC, we randomly generate the other fields with wildcard ratio 0.1 for the best case, 0.5 for the average case and 0.9 for the worst case. The field values of these fields are randomly selected, and the ruleset size we use in experiments are 1K, 10K and 100K. We compute the rule size by byte-alignment, where each rule is of size 53 bytes.

The trace datasets generated by ClassBench-ng consist of 1000K packets and they include two types, one contains the packet headers that have 100% matches and another contains the packet headers that incur 10% misses. We consider whether the header fields of the packet match the rules in the rulesets. So, the searching process only takes the header of trace data to match rules rather than injects packet traffic through network interface to test your rules. In other words, we focus on the computation instead of network I/O plus computation.

Throughput with trace of 100% match

We extend source code of CutTSS and TabTree from 5 fields to 12 fields and evaluate classification performance with 1K, 10K, and 100K rulesets. Both CutTSS and TabTree focus on dividing rules into groups based on prefix fields and so the performance results remain the same in rulesets with various wildcard ratios. In CPU scenario, the results for 100K rulesets in Fig. 8 show that H-HashTree extended by Bloom Filter (H-HashTree-BF) has better performance than H-HashTree for all rulesets. H-HashTree performs better than CutTSS and TabTree for ACL and FW rulesets with all wildcard ratios and for IPC rulesets of wildcard ratio 0.1. H-HashTree for rulesets IPC-0.5 and IPC-0.9 and H-HashTree-BF for rulesets IPC-0.5 performs a little worse than CutTSS, but still outperforms TabTree. Our proposed algorithm is better than CutTSS and TabTree in most cases for 1K and 10K rulesets. The search performance results in CPU scenario for rulesets of various sizes are similar. The performance in the result is a little of a downgrade for bigger rules set (i.e., 100K) but not too much. Therefore, our proposed algorithm should be suitable for packet classification in most network environments.

For real OpenFlow rulesets (OF) in Fig. 8, there is an obvious performance advantage for the proposed schemes over CutTSS and TabTree. As shown in Table 4, most OpenFlow rules have longer prefix length in source and destination IP fields, resulting in the worst-case performance for CutTSS and TabTree. The performance of the H-HashTree-BF is slightly worse than H-HashTree because the benefit gain from extended Bloom filter during searching hash tree does not pay off the overhead of checking extended Bloom filter. But this situation does not happen on GPU which we will describe subsequently. Fig. 9 shows the search performance on CPU in terms of number of memory accesses.

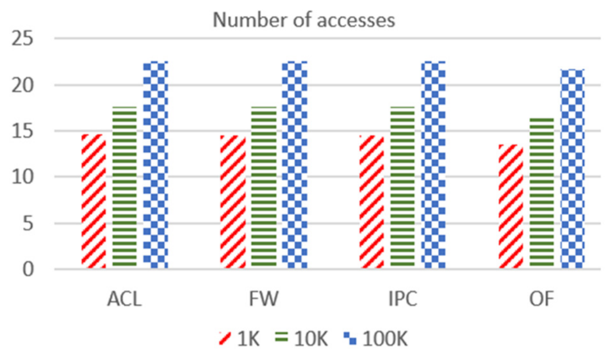

Fig. 9. Average number of memory accesses on CPU for H-HashTree-BF.

By considering the throughput (13.7 MPPS) required to support one 10Gpbs link, it is obvious that CPU is not powerful to support the needed computation capability of packet classification. As a result, we can resort to GPU for the required computation capability.

In GPU scenario, H-HashTree-BF has better performance than H-HashTree in all rulesets as shown in Fig. 10. The impact of checking extended Bloom filter for bitmask is not a shortcoming anymore because of the strong computational capability and efficient context switch between threads for memory latency hiding on GPU. The configuration of GPU performance evaluation uses.

Fig. 10 shows the search performance in GPU scenario. The GPU configuration is set as BLOCK_PER_GRID=72 and THREAD_PER_BLOCK = 512. For 100K rulesets, the throughput of our proposed scheme can reach up to 60∼100 Mpps. In contrast to the CPU scenario, using GPU to classify packets in large rulesets is more efficient. The throughputs of 10K and 1K rulesets can reach up to 210 MPPS and 333 MPPS, respectively. Using GPU can efficiently accelerate the classification of the packets.

Throughput with traces of 10% miss ratio

In the practical network environment, all packets on the Internet will pass through the classifier. When a packet miss happens, ovs-vswitchd or the controller will specify a rule to classify the miss-classified packet. This is how rule updates are triggered. In the previous experiments, the performance is based on the assumption of packets that will not be misclassified in the algorithm. We need to figure out how long it takes to identify packet miss in

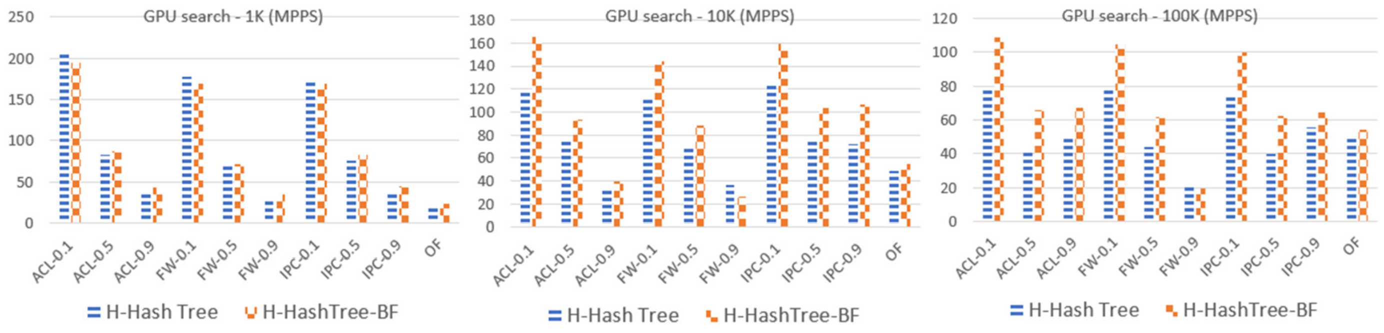

Fig. 10. GPU search performance of 1K, 10K, and 100K rules.

Table 5
CPU throughputs of 10% miss and all match among different wildcard ratios and OF rulesets.

Search (MPPS)	all match	10% miss
H-HashTree WD 0.1 ruleset	4.746	4.919
H-HashTree WD 0.5 ruleset	3.554	3.724
H-HashTree WD 0.9 ruleset	3.297	3.036
H-HashTree (OF) ruleset	4.341	4.237
H-HashTree-BF WD 0.1 ruleset	5.114	5.409
H-HashTree-BF WD 0.5 ruleset	3.844	4.033
H-HashTree-BF WD 0.9 ruleset	3.592	3.250
H-HashTree-BF (OF) ruleset	4.272	4.245
CutTSS ruleset	3.059	2.582
CutTSS OpenFlow (OF) ruleset	0.307	0.177
TabTree ruleset	1.443	1.322
TabTree (OF) ruleset	0.334	0.181

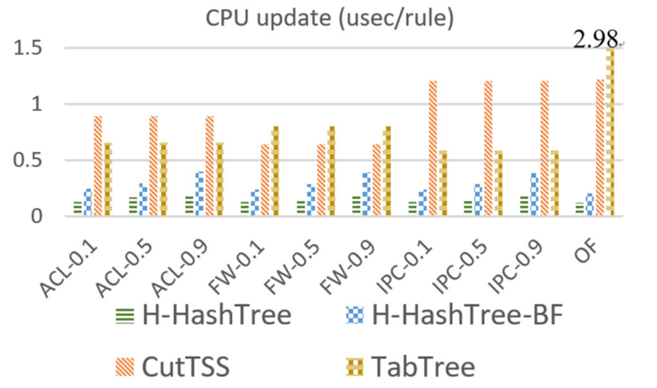

Fig. 11. CPU update times of 100K ACL, FW, and IPC with different wildcard ratio and OF rulesets.

Table 6
GPU throughput comparison with different wildcard ratios and OF rulesets.

Search (MPPS)	all match	10% miss
H-HashTree WD 0.1 ruleset	77.5	81.6
H-HashTree WD 0.5 ruleset	42.2	43.8
H-HashTree WD 0.9 ruleset	41.6	39. 9
H-HashTree (OF) ruleset	49.2	49.7
H-HashTree-BF WD 0.1 ruleset	104.5	118.7
H-HashTree-BF WD 0.5 ruleset	63.3	67.9
H-HashTree-BF WD 0.9 ruleset	50.4	51.5
H-HashTree-BF (OF) ruleset	4.3	4.2

order to take the needed action for rule updates. To simulate performance with packet misses, we randomly exclude 10% rules from the ruleset, and build hash tree with the remaining 90% of rules (so do CutTSS and TabTree). In Table 5 and 6, we average throughputs of all 100K rulesets with same wildcard ratios for traces of 100% match and 10% miss. In CutTSS and TabTree, they define the highest priority of rule saved in a tuple as the priority of the tuple and arrange tuples in the order of priority from high to low. They search packets from the tuple with highest priority and return when priority of current match rule is higher than priority of the next tuple. Because even though we can match rules in the following tuples, the priority won't be higher than the current one. The reason why performance downgrades in CutTSS and TabTree with the trace of 10% miss is because if we cannot find a matched rule, we must exhaustively search all tuples.

For H-HashTree and H-HashTree-BF with wildcard ratios 0.1 and 0.5, the performance of 10% miss is better than 100% match, but downgrades in wildcard ratio 0.9 and nearly equal in real OpenFlow ruleset (OF). Although we give priority to each hash table like CutTSS and TabTree, we do not search packet from hash table of the highest priority to the lowest, which means the performance of our approach will not necessarily downgrade like CutTSS and TabTree. Generally speaking, our approach tends to have better reaction to packet misses when wildcard ratio of a ruleset is not high.

Evaluation on Incremental Update

To perform the experiments on updates, we first build data structure with the whole ruleset and randomly select 5% rules for deletion and insertion, then average deletion and insertion times are obtained (shown as μsec per rule). Rule updates can be performed in CPU or GPU. If update operations are performed in CPU and search operations are executed in GPU, the modified data structures after inserting or deleting rules have to be copied to GPU.

In CPU scenario, as we expect that H-HashTree has better performance than H-HashTree-BF because we have to maintain extended counting Bloom filter and extra hierarchical hash tree. The performance drops about 50% in Proposed-scheme-BF as shown in Fig. 11. Compared with CutTSS and TabTree, H-HashTree-BF is still better because we only have to search at most 6 hash tables (4-layer main hash tree plus 2-layer extra hash tree) to reach the buckets that need to be updated. For each bucket needed to be modified, we insert rules according to their priorities, and we delete by disabling rules from the bucket and adjust the bucket size based on the number of remaining rules in this bucket. In GPU scenario, we pre-allocate extra device (GPU) memory that is really needed in order to handle rule updates on GPU. The reason is that technically we cannot do dynamical device memory allocation while kernel function is running on GPU.

Updating 5% rules randomly chosen from 100K rulesets (means launch 5K rules to GPU for update) in fact does not fully utilize the computational resource of GPU. But we still get an acceptable speedup compared with CPU except in FW-0.9 ruleset as shown in Fig. 11. We observe that sizes of many buckets (also known as number of rules saved in the buckets) in FW-0.9 rulesets scale much more than other buckets, resulting in unbalanced concur-

Table 7
Update impact on search performance.

Search (MPPS)		
ruleset	H-HashTree-BF	H-HashTree-BF with 5% updates mixed
WD ratio 0.1	105.687	105.65
WD ratio 0.5	72.319	63.469
WD ratio 0.9	46.915	46.869
OF	54.379	54.366

rent execution of threads on GPU. For example, if there are two threads among the same wavefront, one thread needs to update a bucket with 5 rules but another thread needs to update a bucket with 900 rules, the former thread must wait for later thread to complete even though it has finished its work earlier. The same situation also happens during searching FW-0.9 on GPU. The reason why we explain it until now because the imbalance of concurrent execution impacts the most in updates. Comparing H-HashTree-BF with H-HashTree on GPU, we can see that the performance of H-HashTree-BF drops only about 25%. To measure how update operation affects search performance on GPU, we mix 5% rule updates while searching packets. The measurement can be concluded as below.

$$mixed\_perf = \frac{1\,(sec) - \text{time of update 5\% rules}}{\text{avg. search time of each packet}}$$

In Table 7, update operations show nearly no significant impact to search performance. In the case of wildcard ratio 0.5 rulesets, which is also the worst case among the four types of rulesets, search performance of H-HashTree-BF with 5% updates mixed is about 87.7% of that without mixed updates.

Memory Consumption

In this paper, we do not adjust hash table size according to the utilization rate. The hash table size of each layer is predefined as mentioned previously. As the wildcard ratio of the ruleset increases, the memory consumption grows dramatically as shown in Table 8 while CutTSS and TabTree remain low memory footprint. The different memory consumption between H-HashTree-BF and H-HashTree is from extended counting Bloom filters and extra hash tree, causing about 100 bytes or more per rule in average.

Comparison with previous work

The multi-layer Bloom filter algorithm implemented on NVIDIA GTX-580 GPU in [31] achieves the throughput of 39-58 MPPS for 2K ACL rules and the throughput of 43.9 MPPS for 1K FW rules. Our proposed scheme obviously outperforms the multi-layer Bloom filter algorithm. Since the extended Bloom filters are not used in [31], the updates are not supported.

For the entropy-based hash algorithm proposed in [11], the throughput is 4 MPPS using IG scheme and 2 MPPS using MSB for 100K ACL in CPU scenario. However, our throughput is between 4 and 5 MPPS. Especially, the performance of our proposed in smaller rules is better than the IG scheme in [11]. The throughput of 1k-fw in our experiment is at least 10Mpps but their result is about 4.1Mpps.

In [11], the throughput is 4 MPPS using entropy and 2MPPS using MSB for 100K ACL in CPU scenario. The throughput of our proposed schemes is between 4 MPPS and 5 MPPS. Especially, the performance of our proposed in smaller rules is better than the method in [11]. The throughput of 1K-FW ruleset in our experiment is at least 10 MPPS but their result is about 4.1 MPPS. In addition, our performance evaluation is conducted for 12-field rules while the results in [11] are for 5-field rules. Since the rules may contain wild-cards in some fields, they must be duplicated among the 256K sub-tables indexed by 18-bit keys. As a result, the total number of rules is significantly expended to up to about 15 million rules for a 100k rule-set. Based on the proposed information entropy-based criterion, the maximal Information Gain (IG) is computed for finding a uniform hash key 18 bits from source and destination addresses are determined. Since computing IG is a pre-computation process, dynamic updates cannot be supported.

Suppose that the packet length is 64 bytes. The performance of 10K-ACL with 10% wildcard ratio of our proposed scheme is up to 109Gbps. To compare with [11], their performance is up to 125Gbps for 10K-ACL. The difference comes from the operating frequency of GPU where they use 1733Mhz NVIDIA GTX-1080 GPU and we use 1257Mhz AMD RX580. Our proposed algorithm is more suitable for the future networks than [11] because we consider 12-field rules and support rule updates.

Table 8
Memory consumption of 100K ACL, FW, IPC with different wildcard ratios and OF rulesets.

Memory (bytes/rule)				
ruleset	H-HashTree	H-HashTree-BF	CutTSS	TabTree
ACL-0.1	62.246	144.788		
ACL-0.5	106.812	205.692	71.948	67.478
ACL-0.9	1,749.996	1,850.644		
FW-0.1	62.512	147.446	70.761	69.111
FW-0.5	100.672	202.408		
FW-0.9	1,716.9	1,820.406		
IPC-0.1	62.02	142.52		
IPC-0.5	105.335	201.78	71.833	65.062
IPC-0.9	1,798.03	1,896.245		
OF	69.905	146.555	86.348	70.858

6. Conclusions

In this paper, we proposed a packet classification algorithm using hierarchical hash tree which supports both high searching speed and fast updates. We implement extended Bloom filters to check whether we can skip unnecessary search in main hash tree. The advantage of extended Bloom filters is that its performance impact is not affected by types of rulesets. We isolate rules which cannot be hashed by extended Bloom filter and put them in an extra hash tree. We search the extra hierarchical hash tree only when the miss match of bitmasks probably happens, resulting in potential throughput improvement by searching smaller hash tree with fewer rules. In both main and extra hierarchical hash trees, we define a threshold to make sure the subset rules are small enough for linear search and stop further creating nodes to the next layer. We showed that our method is also feasible on parallel computing platform like GPU. Due to powerful computation capability and special hardware architecture designed for GPU, we can easily overcome the defects of our algorithm implemented by CPU such as the relatively high overhead needed in extended Bloom filter.

The proposed H-HashTree performs better with state-of-the-art algorithms in both searches and updates. The proposed schemes have better reaction to packet miss (downgrades only in the worst case), which makes our approach suitable for frequent updates in SDN environment.

CRediT authorship contribution statement

Yu-Hsiang Lin: Conceptualization, Methodology, Software. **Wen-Chi Shih:** Data curation, Formal analysis, Writing – original draft. **Yeim-Kuan Chang:** Conceptualization, Methodology, Project administration, Writing – review & editing.

Declaration of competing interest

To the best of our knowledge, the named authors have no conflict of interests or otherwise.

References

[1] fmadio, What is 10GBIT Line Rate? in: the Packet Sniffer Blog, Fmad Engineering, June 30, 2021, https://www.fmad.io/blog-what-is-10g-line-rate.html.
[2] AMD, Inc., AMD GCN3 Instruction Set Architecture rev1.1, August 2016.
[3] AMD, Inc., AMD GCN1 Instruction Set Architecture rev1.1, December 2012.
[4] AMD, Inc., White Paper | AMD Graphics Cores Next (GCN) Architecture, June 2012.
[5] Y.-K. Chang, H.-C. Chen, Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA, Comput. J. 62 (2) (Feb. 2019) 198–214.
[6] Yeim-Kuan Chang, Tung-Yin Chi, Hash-based OpenFlow packet classification on heterogeneous system architecture, in: Eleventh International Conference on Ubiquitous and Future Networks, 2019.
[7] J. Daly, E. Torng, TupleMerge: building online packet classifiers by omitting bits, in: IEEE ICCCN, 2017.
[8] J. Daly, V. Bruschi, L. Linguaglossa, et al., TupleMerge: fast software packet processing for online packet classification, IEEE/ACM Trans. Netw. 27 (4) (2019) 1417–1431.
[9] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, Sotiris Ioannidis, GASPP: a GPU-accelerated stateful packet processing framework, in: 2014 USENIX Annual Technical Conference.
[10] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, APUNet: revitalizing GPU as packet processing accelerator, in: USENIX Symposium on Networked Systems Design and Implementation, 2017.
[11] S. Greenberg, T. Sheps, D.A. Leon, Y. Ben-Shimol, Packet classification using GPU and one-level entropy-based hashing, IEEE Access 8 (2020) 80610–80623, https://doi.org/10.1109/ACCESS.2020.2990331.
[12] P. Gupta, N. McKeown, Packet classification on multiple fields, in: ACM Sigcomm, August 1999.
[13] P. Gupta, N. McKeown, Classifying packets with hierarchical intelligent cuttings, IEEE MICRO 20 (1) (2000) 34–41.
[14] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, Aug. 2007, pp. 851–876.
[15] Cheng-Liang Hsieh, Ning Weng, Wei Wei, Scalable many-field packet classification for traffic steering in SDN switches, IEEE Trans. Netw. Serv. Manag. 16 (1) (March 2019).
[16] Jiří Matoušek, Gianni Antichi, Adam Lučanský, ClassBench-ng: recasting ClassBench after a decade of network evolution, in: 2017 ACM/IEEE Symposium on Architectures for Networking and Communications.
[17] Member Li Fan, Pei Cao, Jussara Almeida, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Netw. 8 (3) (June 2000).
[18] W. Li, Tong Yang, Y.-K. Chang, TabTree: a TSS-assisted bit-selecting tree scheme for packet classification with balanced rule mapping, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2019.
[19] W. Li, T. Yang, O. Rottenstreich, X. Li, G. Xie, H. Li, B. Vamanan, D. Li, H. Lin, Tuple space assisted packet classification with high performance on both search and update, IEEE J. Sel. Areas Commun. (April 2020).
[20] W. Li, X. Li, H. Li, G. Xie, Cutsplit: a decision-tree combining cutting and splitting for scalable packet classification, in: Proc. IEEE INFOCOM, Apr. 2018, pp. 2645–2653.
[21] E. Liang, H. Zhu, X. Jin, I. Stoica, Neural packet classification, in: ACM SIGCOMM, 2019.
[22] N. McKeown, T. Anderson, H. Balakrishnan, OpenFlow: enabling innovation in campus networks, Comput. Commun. Rev. 38 (2) (2008) 69–74.
[23] Open Networking Foundation, OpenFlow Switch Specification Version 1.5.1, 26 March 2015.
[24] Open Networking Foundation, OpenFlow Switch Specification Version 1.0.0, December 31, 2009.
[25] Open vSwitch manual Version TSS 2.11.90, ovs-fields – protocol header fields in OpenFlow and open vSwitch, http://docs.openvswitch.org/en/latest/ref/.
[26] B. Pfaff, et al., The design and implementation of open vSwitch, in: Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15), May 2015, pp. 117–130.
[27] Advanced Micro Devices, Inc., ROCm core technology, site: https://github.com/RadeonOpenCompute.
[28] Advanced Micro Devices, Inc., ROCm developer tools, site: https://github.com/ROCm-Developer-Tools.
[29] S. Singh, F. Baboescu, G. Varghese, J. Wang, Packet classification using multidimensional cutting, in: SIGCOMM '03, 2003.
[30] V. Srinivasan, S. Suri, G. Varghese, Packet classification using tuple space search, in: SIGCOMM 99, 1999, pp. 135–146.
[31] M. Varvello, R. Laufer, F. Zhang, T.V. Lakshman, Multilayer packet classification with graphics processing units, IEEE/ACM Trans. Netw. 24 (5) (Oct. 2016) 2728–2741.
[32] WOLF Advanced Technology, VPX3U-RTX5000E-SWITCH, https://wolfadvancedtechnology.com/images/datasheets/VPX3U-RTX5000E-SWITCH_Datasheet.pdf.
[33] Yeim-Kuan Chang, Tung-Yin Chi, Hash-based OpenFlow packet classification on heterogeneous system architecture, in: 2019 Eleventh International Conference on Ubiquitous and Future Networks.
[34] Sorrachai Yingchareonthawornchai, James Daly, Alex X. Liu, A sorted-partitioning approach to fast and scalable dynamic packet classification, IEEE/ACM Trans. Netw. 26 (4) (August 2018).
[35] Shijie Zhou, Viktor K. Prasanna, Scalable GPU-accelerated IPv6 lookup using hierarchical perfect hashing, in: IEEE Global Communications Conference, 2015.

Yeim-Kuan Chang received PhD degree in computer science from Texas A&M University, College Station, in 1995. He is currently a professor and the department chair in the Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan. His research interests include Internet router design, computer architecture, and multiprocessor systems.

Yu-Hsiang Lin received the M.S. degree in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, Republic of China, in 2019. His research interests include high-speed packet processing in hardware.

Wen-Chi Shih received the M.S. degree in Information Management from Cheng Shiu University, Taiwan, Republic of China, in 2011. He is currently working toward the Ph.D. degree in Computer Science and Information Engineering at National Cheng Kung University, Taiwan, Republic of China. His current research interests include high-speed networks, network security and high-speed packet processing.