# High-Performance Packet Classification on GPU

Shijie Zhou, Shreyas G. Singapura and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: {shijiezh, singapur, prasanna}@usc.edu

***Abstract*—Multi-field packet classification is a network kernel function where packets are classified and routed based on a pre-defined rule set. Recently, there has been a new trend in exploring Graphics Processing Unit (GPU) for network applications. These applications typically do not perform floating point operations and it is challenging to obtain speedup. This paper presents a high-performance packet classifier on GPU. We investigate GPU's characteristics in parallelism and memory accessing, and implement our packet classifier using Compute Unified Device Architecture (CUDA). The basic operations of our design are binary range-tree search and bitwise AND operation. We optimize our design by storing the range-trees using compact arrays without explicit pointers in shared memory. We evaluate the performance with respect to throughput and processing latency. Experimental results show that our approach scales well across a range of rule set sizes from 512 to 4096. When the size of rule set is 512, our design can achieve the throughput of 85 million packets per second and the average processing latency of 4.9 $\mu s$ per packet. Compared with the implementation on the state-of-the-art multi-core platform, our design demonstrates 1.9x improvement with respect to throughput.**

***Index Terms* — GPU, CUDA, High-Performance, Packet Classification**

## I. Introduction

Packet classification is a network kernel function performed by Internet routers. It enables various network services such as network security, Quality of Service (QoS) routing and resource reservation. During packet classification, incoming packets are classified against a set of predefined rules (rule set) and routed to a specific port according to the classification result. In traditional network applications, each predefined rule considers 5 header fields [1]: source/destination IP addresses, source/destination port numbers and transport layer protocol. In Software Defined Networking (SDN) [2], which is proposed as the next generation of network, up to 40 header fields [3] are considered in each rule. As the Internet traffic grows rapidly, high-performance packet classifiers are essential in order to handle the high data rates and volume of traffic in the network.

Both hardware and software platforms are harnessed in the literature to implement packet classifier. Ternary Content Addressable Memories (TCAMs) [4] are widely used because they can handle ternary match and provide high throughput. However, TCAMs are quite expensive and require a lot of power due to the massively parallel exhaustive search [5]. Field Programmable Gate Array (FPGA) technology is also an attractive option for implementing packet classifier [6] due to its reconfigurability and massive parallelism. But the FPGA-based packet classifiers suffer the constraint of the limited on-chip resources, and will undergo long processing latency when using external memory. Software-based routers have also been investigated in the community because of the extensibility and customizability [7], [8]. However, the throughput of the packet classifiers on the multi-core platforms is not promising.

A new trend in high-performance computing is to perform general purpose computing with GPUs. CUDA is a programming model which harnesses the power of GPU. It has been used to show dramatic speedup for floating point intensive applications [9]. Several recent works have also explored the CUDA platform to implement networking functions such as IP lookup [10], [11] to achieve improvement in throughput. State-of-the-art GPUs have 2880 CUDA cores while supporting 30720 threads [12]. Moreover, there are several types of memory with various features available on GPUs. However, how to efficiently use the powerful parallelism and the various types of memory for online algorithms such as packet classification still poses great challenges.

In this paper, we exploit the desirable features of GPU for designing a high-performance packet classifier. The main contributions of the work are summarized below:

- We exploit the massive parallel processing capabilities of GPU and propose a high-performance GPU-based packet classifier.
- We use compact arrays without explicit pointers to store range-trees which results in efficient tree-search on GPU.
- We fully exploit the limited on-chip shared memory to achieve high performance. We also apply optimizations to minimize the shared memory bank conflicts.
- When the size of rule set is 512, our design can achieve the throughput of 85 millions packets per second (MPPS).
- For a thorough evaluation of our design, we conduct experiments using various sizes of rule sets. The performance of the best case and the worst case is studied as well.

The rest of the paper is organized as follows: Section II introduces the background and related work. Section III presents the details of the algorithms. Section IV contains the experimental results. Section V concludes the paper.

This work has been funded by U.S. National Science Foundation under grant CCF-1320211.

978-1-4799-6233-4/14/$31.00 

TABLE I: Example of a rule set

| Rule ID | SA | DA | SP | DP | Protocol | Priority | Action |
|---|---|---|---|---|---|---|---|
| 1 | 175.77.88.155/32 | 192.0.0.0/8 | 0-65536 | 0-1000 | 0x06 | 1 | Action 1 |
| 2 | 11.77.88.2/32 | 180.0.96.12/32 | 16-16 | 0-65536 | 0x00 | 2 | Action 2 |
| 3 | 192.168.0.26/32 | 125.199.2.72/32 | 10-65536 | 200-300 | 0x11 | 3 | Action 3 |
| 4 | 10.0.11.0/20 | 137.135.88.159/32 | 0-1000 | 20-22 | 0x06 | 4 | Action 4 |
| 5 | 100.19.0.0/16 | 17.35.0.0/16 | 1000-1000 | 1020-1020 | 0x3f | 5 | Action 5 |

## II. Background and Related Work

### A. 5-field Packet Classification

In packet classification, an IP packet is classified based on 5 fields in the packet header: 32-bit source/destination IP addresses (denoted as SA/DA), 16-bit source/destination port numbers (denoted as SP/DP) and 8-bit transport layer protocol. Routers perform packet classification based on a predefined rule set. [1] studies the real-life firewall rule sets from several Internet Service Providers (ISPs) and finds that 0.7% of the rule sets contain more than 1000 rules while nearly 99% of the rule sets contain less than 500 rules.

Each rule has its own matching information of the five fields, a priority and an action to be taken if matched. Different fields require different types of matches: SA and DA require prefix match; SP and DP require range match; protocol field requires exact match. A packet matches a rule only when the five header fields are all matched. If a packet matches multiple rules, the action associated with the highest prioritized one will be taken. We show an example of a rule set in TABLE I.

### B. CUDA Programming Model

In this section, we briefly introduce the key features of CUDA programming model and GPU. Additional details can be found in [13].

A CUDA program contains host function and kernel function. The kernel function is called by the CPU but executed by a number of threads on GPU. A certain number of threads (upto 1024 threads) are grouped into a thread block and several thread blocks form a grid. All the threads in the grid will execute the invoked kernel function. Inside a thread block, each group of 32 threads shares the same program counter and executes the same instruction on different data in every clock cycle (SIMT execution fashion). The group of 32 threads forms a warp, which is the basic execution unit on GPU.

A GPU device is composed of several streaming multiprocessors (SMX). Threads inside the same thread block will reside in the same SMX. In Kepler [12], each SMX has 192 processing units. Each processing unit is called a CUDA core. Every SMX features four warp schedulers to allow four warps to be executed concurrently. When threads in a warp encounter long latency operations, warp scheduler will switch to another warp which is ready to be executed. On GPU, switching between threads introduces little overhead. Thus, maximizing the number of warps in a SMX helps to hide the latency of long operations and increase throughput. If threads in a warp take different branches of a program (eg. if, else), divergence among thread execution will lead to a waste of hardware resources. The reason is that all possible program execution paths have to be traversed by each thread; threads not satisfying the current execution condition will become idle.

CUDA platform provides various types of memory. Global memory is the largest off-chip memory on GPU and can be accessed by every thread. Access latency to global memory is over hundreds of cycles. Constant memory and texture memory are two small pieces of memory and can be cached by on-chip read-only cache of each SMX. In Kepler, accesses to global memory go through a two-level cache hierarchy, L1 cache and L2 cache. L2 cache is shared by all SMXs and L1 cache is private to each SMX. L1 cache is on-chip memory and can be configured as 16 KB (by default) or 48 KB per SMX. Shared memory is also on-chip memory and private to each thread block. L1 cache and shared memory share a 64 KB memory segment per SMX. Access to shared memory may take from 1 to 32[1] clock cycles. Registers are the fastest memory. There are 65536 32-bit registers per SMX [12]. Registers allocated to a thread can not be shared with other threads.

### C. Related Work

Most of the packet classification algorithms on general purpose processors fall into three categories: decision-tree-based, hash-based and decomposition-based algorithms. Decision-tree-based algorithms such as [14] employ several heuristics to cut the rule set space into smaller sub-regions in a multi-dimensional space. Each node in the decision-tree represents a sub-region. The idea of hash-based algorithms [15] is grouping the rules into a set of tuple spaces and each tuple is maintained as a hash table. The classification of an incoming packet can be performed in parallel over all tuples by searching each hash table. Decomposition-based approaches first search each field individually and then merge the intermediate results of each field. For example, [16] uses bit vectors (BV) to represent the intermediate results. Each bit corresponds to a rule and will be set to "1" if the input matches the corresponding rule.

There are not many efforts in developing packet classifier on GPU. [17] proposes an approach which exhaustively compares the packet against all matching conditions to minimize the divergence overhead. However, [17] is based on the assumption that the set of unique criteria is small for large rule set. [18] implements a decomposition-based approach and a decision-tree-based approach on GPU. The design is optimized by exploring a variety of memory architectures of CUDA. It achieves 10x improvement against the CPU-based implementation but does not discuss the throughput or latency in details. A hash-based algorithm is adopted in [19]. The implementation delivers the throughput of 10.7 MPPS and 4.8 MPPS with rule numbers of 500 and 2000, respectively.

[1]The variation of latency is due to shared memory bank conflict

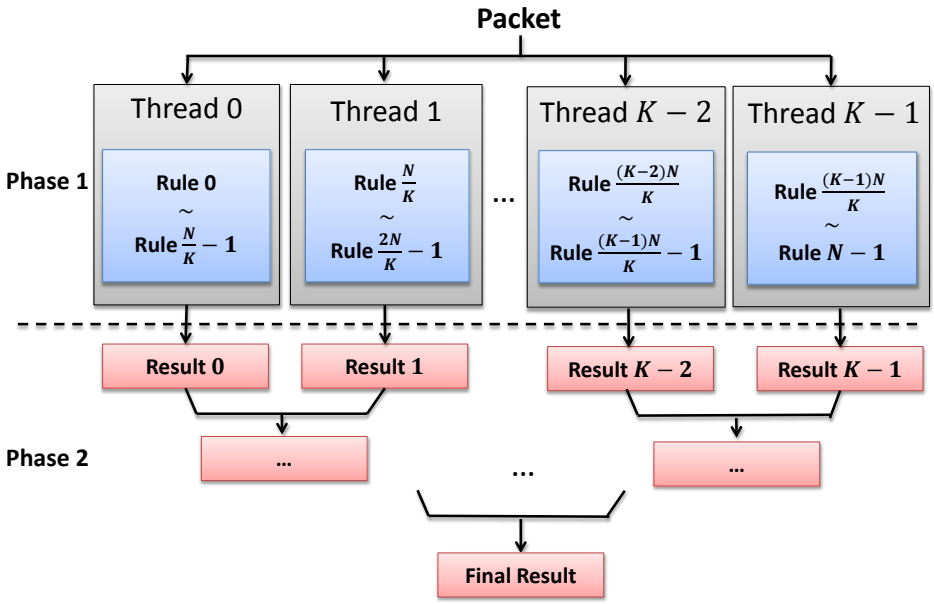


Fig. 1: Algorithm Overview

## III. Algorithm

### A. Algorithm Overview

We use $K$ threads to classify each incoming packet. Assume there are $N$ rules in the rule set and they are ordered by priority. Each thread is responsible for examining the packet against $\frac{N}{K}$ rules. After examination, each thread produces a local immediate classification result, which can be an exact rule index or no match. In the next step, a final classification result can be reported by identifying the highest prioritized matching rule among the $K$ immediate results in $\log K$ steps. In summary, the kernel function to classify each packet involves 2 phases:

- Phase 1: each thread examines $\frac{N}{K}$ rules and produces a local classification result.
- Phase 2: the rule with the highest priority among the $K$ local results is identified in $\log K$ steps.

The overview of this algorithm is depicted in Fig. 1.

### B. Tree Search & Bit Vector Based Algorithm

To identify the local result of each thread in Phase 1, we adopt a range-tree search and bit vector (BV) based algorithm proposed in [8]. The algorithm is a decomposition-based approach and the major steps are summarized below:

- Pre-process rules to construct a binary range-tree for each individual field. Every leaf node is assigned with BVs, which can infer which rules are matched when reaching the leaf node.
- Each thread performs binary range-tree search sequentially field by field. After 5 tree searches, 5 BVs are produced.
- Merge the 5 BVs by bitwise AND operation to obtain a final BV. Identify the first non-zero bit of the final BV and the corresponding rule is the local classification result. If the final BV is 0, it indicates there is no matching.

The pre-processing step is completed by host function. In this step, $N$ rules are first partitioned into $\frac{N}{K}$ groups. For each group, five trees are constructed for the five fields. For SA/DA field, prefixes are first translated into a set of ranges and further "flattened" to produce a set of non-intersecting sub-ranges. The range-tree is constructed by the sub-range boundaries and each leaf node corresponds to two sub-ranges. For SP/DP field, the construction of the range-trees is similar to SA/DA field, but prefix-to-range conversion is not required. For protocol field, all the unique values are extracted and mapped to a binary search tree.

The tree search step and merging BV step happen in Phase 1 for each thread. Fig. 2 illustrates the steps. Since every thread takes the same execution path, the latency of Phase 1 is determined by the "slowest" thread during each tree search. The "slowest" thread occurs when it has the largest tree to traverse. The computation complexity is O($\log \frac{N}{K}$) for range-tree/tree search, O($\frac{N}{K}$) for merging BVs, and O($\frac{N}{K}$) for identifying the first non-zero bit of BV.

### C. Tree Search on GPU

Classic tree implementation uses pointers to connect nodes with their children. Traversing such trees is not efficient on GPU because it leads to additional memory accesses and divergence overheads. Also, pointers require additional space. This is problematic given the limited on-chip memory. For time and space efficient tree-search, we implement a range-tree in the form of an array. To hold a $p$-level tree, the size of the array is set to $2^p - 1$. The root of the tree is located at array[0]. For the node at array[$i$] ($0 \leq i < 2^p - 1$), its left child node is located at array[$2i + 1$] while its right child node is located at array[$2i + 2$]. This particular arrangement makes it

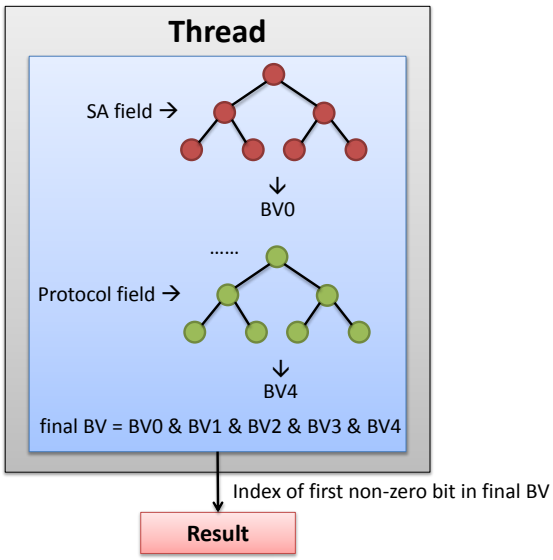


Fig. 2: Steps for Phase 1

easy to move from a node to its children: multiply the node's index by 2 and add 1 to go left or add 2 to go right.

However, performing a tree search on such a tree structure requires it to be a perfect binary tree [20]. In a perfect binary tree, all the leaf nodes have the same depth. The tree constructed in [8] will have leaf nodes differed by 1 level when it is not a perfect tree. To convert the range-tree into a perfect binary tree, for any non-leaf node which is not fully filled, we duplicate it to the missing child's location in the array. The BVs of leaf nodes are calculated based on the perfect binary tree. Note that this conversion will not impair the performance. This is due to that the "slowest" thread determines the processing latency and the length of the longest execution path does not change. Fig. 3 shows an example[2] in which the two ranges of five rules are first translated into an normal range-tree, then converted into a perfect range-tree and finally stored in an array.

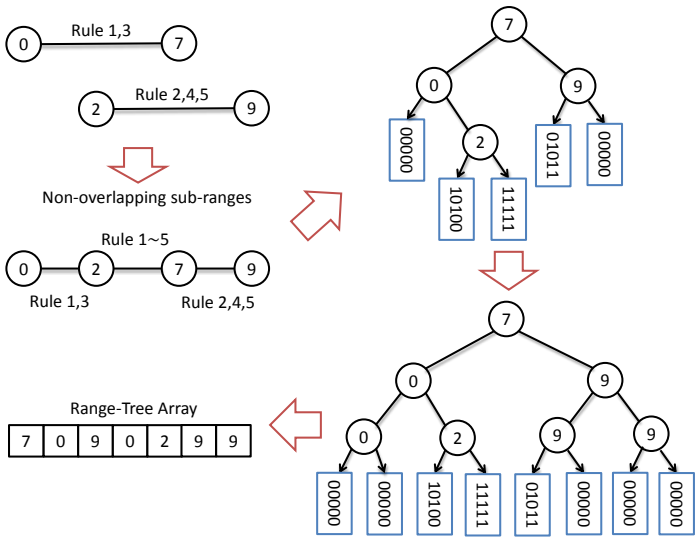


Fig. 3: Translate ranges into a perfect range-tree

[2]The bit vector in each blue box is the BV associated with the leaf node

Algorithm 1 illustrates how to perform a tree search using such a range-tree. The $ID$ returned by this algorithm can be used to determine the result BV. Note that searching values which are beyond the tree will also return an $ID$, but the corresponding BV only contains 0.

**Algorithm 1** Binary Tree Search

Let $Tree[\,]$ denote the array storing the tree
Let $ArraySize$ denote the size of $Tree[\,]$
**Range_Tree_Search**($input$)

```
1: ID = 0;
2: while ID < ArraySize do
3:    if input < Tree[ID] then
4:       ID = ID * 2 + 1
5:    else
6:       ID = ID * 2 + 2
7:    end if
8: end while
9: return ID;
```

### D. Identify Global Result

In Phase 2, a global result is identified among $K$ local results in $\log K$ steps. The operations in Algorithm[3] 2 are executed by each thread during Phase 2. The global result will be finally found in the first thread of the warp. Fig. 4 shows the process when $K$ = 4;

**Algorithm 2** Identify Global Result

Let $Local[\,]$ denote the array of local results
Let $Tid$ denote the thread index in the thread block
**Identify_Global_Result**()

```
1: for i = 0; i < logK; i++ do
2:    if Tid%2^(i+1) = 0 then
3:       Local[Tid] = min(Local[Tid], Local[Tid + 2^i])
4:    end if
5: end for
```

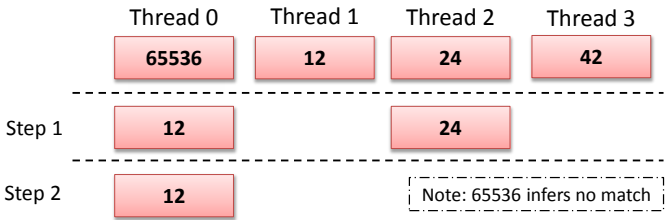


Fig. 4: Identify global result when $K$=4

### E. Mapping to Hardware

Since each local result produced in Phase 1 should be accessible by other threads and be accessed for multiple times in Phase 2, the local results are stored in an array using shared memory. All the perfect range-trees and their associated BVs are pre-constructed by CPU and transferred to GPU via PCIe. When the rule set is small, the range-trees and BVs can be fit in the shared memory for fast access; when the shared memory is

[3]In Phase 1, if a thread does not find any match, its local result will be set to a large number beyond any possible rule index.

not sufficient to hold all the range-trees and BVs, only certain upper levels of the range-trees are stored in shared memory and the rest of data will be stored in the global memory. The shared memory is configured as 48 KB per SMX by default. To keep more data of the range-trees and BVs in the shared memory of each thread block, there is only 1 thread block per SMX. Each thread block contains 1024 threads.

## IV. Performance Evaluation

### A. Experimental Setup

Our implementation is based on CUDA 5.0. The target platform contains a dual 8-core Intel E5-2665 processor running at 2.4 GHz. One NVIDIA K20 Kepler GPU running at 705.5 MHz is installed as the accelerator. The target platform has 13 streaming multi-processors (SMX) with 2496 CUDA cores in total and is equipped with 5GB GDDR5.

We generate synthetic rule sets and packet traces using the same methodology as [8]. Overall throughput and the processing latency per packet are the main metrics when we evaluate the performance. We define the overall throughput as the number of packets that can be classified by the classifier per second (MPPS). The processing latency per packet is defined as the average latency for classifying a packet. We assume the packets to be classified are initially stored in the global memory.

### B. Data Layout

We use a warp ($K$=32) to classify each incoming packet and there are 32 warps in each SMX. If $K$ is smaller, the memory consumption for BVs and the latency for the bitwise AND operations will increase. If $K$ is larger than 32, modifying the shared variables in Phase 2 across different warps requires synchronization. The synchronization among warps will lead to significant overhead.

On the target platform, shared memory is divided into 32 equally-sized memory banks [13]. Each bank is 4-byte wide and can only serve one operation per clock cycle. If two addresses of the memory request fall in the same memory bank, a bank conflict occurs and the access has to be serialized. In a warp, each thread classifies packets based on the trees constructed by $\frac{N}{32}$ rules. We optimize the shared memory data layout by arranging the trees required by the same thread in the same memory bank, and the trees needed by different threads in distinct memory banks. By this data layout, the chance of shared memory bank conflicts is minimized and we observe a 3.7x faster speed for shared memory accesses.

### C. Throughput and Latency

For evaluation, we vary the number of rules ($N$) from 512 to 4096. Fig. 5 and Fig. 6 show the throughput and processing latency under various sizes of rule sets, respectively. When $N$ is 512 or 1024, all the trees and BVs can be fit in shared memory. When $N$ is 2048, trees are stored in shared memory while BVs are stored in global memory. When N is 4096, parts (upper levels) of trees are stored in shared memory while the rest and all BVs are stored in global memory.

We observe that the throughput shows a dropping trend as the rule set becomes larger. The deterioration is due to:

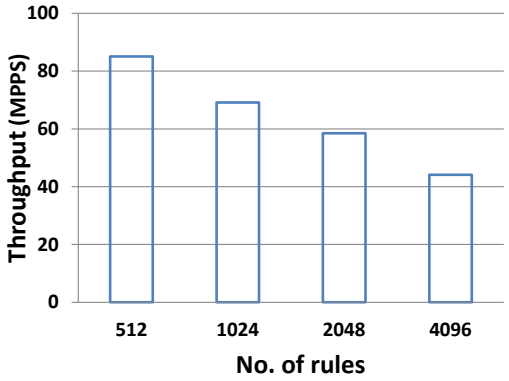


Fig. 5: Throughput

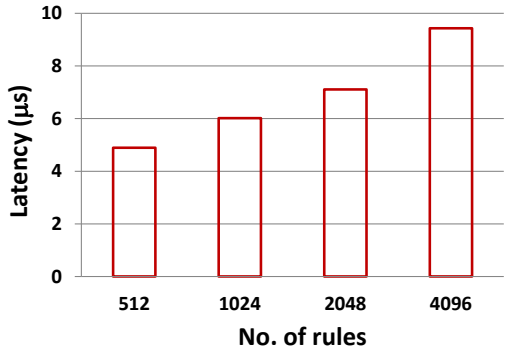


Fig. 6: Processing Latency

(1) each thread needs to go through larger trees; (2) each BV in the bitwise AND operation becomes longer; (3) when the shared memory can not hold all the trees and BVs, each thread needs to access the global memory in Phase 1. We also observe that the processing latency for Phase 1 increases as the rule set becomes larger, but the processing latency of Phase 2 is constantly 0.3 $\mu s$ per packet.

### D. Performance of Best and Worst Case

For a more thorough analysis of our design, we study the performance of the best case and the worst case. The systematized rule set has a lot of duplicate values in each field. The methodology to construct the range-trees is based on the unique values of each field. Thus the memory requirement and the size of each tree are dependent on the number of the unique values in each field. We define the best case as that all the five trees of each thread have only 1 level. In this scenario, all the range-tree/tree searches can be completed in one step. We define the worst case as that all the five trees of each thread are as large as possible. In this scenario, each value is a unique value of the field. If there are $\frac{N}{32}$ rules, the number of tree levels will be $\log(\frac{N}{32})$+1. The performance of the best case and the worst case is shown in Fig. 7 and Fig. 8, respectively.

When using the systematized rule set, the throughput can attain at least 68% of the performance of the best case. Even in the worst scenario, our design can still achieve the throughput of over 30 MPPS for the rule set with 4096 rules.

### E. Comparison with implementation on Multi-core

In this section, we compare the performance of our GPU-based design with the multi-core-based design in [8]. Both de-

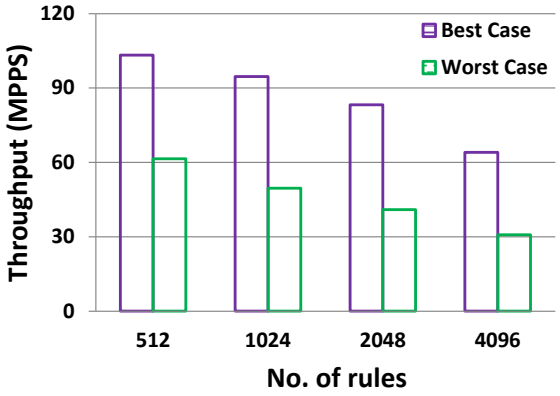


Fig. 7: Throughput


Fig. 8: Processing Latency

signs harness the decomposition-based approach using range-tree search and BV. The synthesized rule sets used for evaluation are also the same. The comparison results are summarized in TABLE II.

TABLE II: Comparison Summary

| No. of rules | | 1024 | 2048 | 4096 |
|---|---|---|---|---|
| Throughput (MPPS) | This paper | 69.1 | 58.5 | 44.1 |
| | [8] | 36.5 | 30.3 | 26.5 |
| Latency ($\mu s$) | This paper | 6.0 | 7.1 | 9.4 |
| | [8] | 0.13 | 0.15 | 0.18 |

It can be observed that our design achieves 1.9x improvement in throughput compared with [8]. However, the average processing latency for each packet is much longer than [8]. The longer latency on GPU is mainly due to:(1) the clock rate of GPU is slower than that of CPU; (2) in [8], five fields are searched in parallel while our approach performs sequential tree searches; (3) since there are 4 warp schedulers per SMX, only four warps (128 threads) per SMX are allowed to be executed concurrently.

## V. Conclusion

In this work, we proposed a BV and range-tree based approach for packet classification on the modern GPU platform. We fully exploited the limited on-chip shared memory to achieve high-performance. Optimizations were applied to minimize the shared memory bank conflicts. We conducted comprehensive experiments by varying the number of rules from 512 to 4096. Experimental results showed that when the rule set size was 512, our design achieved the throughput of 85 MPPS and the processing latency of 4.9 $\mu s$ per packet. Compared with the state-of-the-art multi-core implementation, our design demonstrated at least 1.9x improvement in throughput.

In the future, we plan to develop the hash-based packet classification algorithms on GPU. We will also explore other networking applications such as OpenFlow packet classification [3] on GPU.

## References

[1] P. Gupta and N. McKeown, "Packet classification on multiple fields," in Proc. of SIGCOMM, pp. 147-160, 1999.

[2] N. McKeown, T. Anderson,H.Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," SIGCOMM Comput. Commun. Rev., vol. 38, no.2, pp. 69-74, 2008.

[3] "Openflow switch specification 1.3.0" https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

[4] F. Yu, R. H. Katz, and and T. V. Lakshman, "Efficient Multimatch Packet Classification and Lookup with TCAM," IEEE Micro, vol. 25, no. 1, 50-59, 2005.

[5] D. E. Taylor, "Survey and taxonomy of packet classification techniques," ACM Comput. Surv., vol. 37, no. 3, pp. 238-275, 2005.

[6] W. Jiang and V. K. Prasanna, "Scalable Packet Classification on FPGA," in Very Large Scale Integration (VLSI) Systems, vol. 20, pp. 1668-1680, 2011.

[7] D. Liu, B. Hua, X. Hu and X. Tang. "High-performance Packet Classification Algorithm for Any-core and Multi-threaded Network Processor, in Proc. CASES, pp. 334-344, 2006.

[8] S. Zhou, Y. Qu and V. K. Prasanna, "Multi-core implementation of decomposition-based packet classification algorithms," in Parallel Computing Techniques (PaCT), pp. 105-119, 2013.

[9] C. Lee, W. W. Ro and J. Gaudiot, "Boosting CUDA Applications with CPUGPU Hybrid Computing," International Journal of Parallel Programming, vol. 42, no. 2, pp. 384-404, 2014.

[10] Y. Li, D. Zhang, A. X. Liu and J. Zheng, "GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers," in Architectures for Networking and Communications Systems (ANCS), pp. 1-12, 2013.

[11] T. Li, H. Chu and P. Wang, "IP Address Lookup Using GPU," in Proc. of HPSR, pp. 177-184, 2013.

[12] "Kepler GK110 whitepaper" http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[13] "CUDA C Best Practices Guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz34wp6JHEd

[14] S. Singh, F. Baboescu, G. Varghese and J. Wang, "Packet Classification using Multidimensional Cutting," ACM SIGCOMM, pp. 213-224, 2003.

[15] V. Srinivasan, S. Suri, G. Varghese and J. Wang, "Packet classification using tuple space search," ACM SIGCOMM, pp. 135-146, 1999.

[16] T. V. Lakshman, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," ACM SIGCOMM, pp. 203-214, 1998.

[17] A. Nottingham and B. Irwin, "Parallel packet classification using GPU co-processors," in SAICSIT Conf.ACM., pp. 231-241, 2010.

[18] C. Hung, Y. Lin, K. Li, H. Wang and S. Guo, "Efficient GPGPU-based parallel packet classification," in Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 1367-1374, 2011.

[19] K. Kang and Y. S. Deng, "Scalable packet classification via GPU metaprograming," in Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1-4, 2011.

[20] Y. Zou and P. E. Black, perfect binary tree, in Dictionary of Algorithms and Data Structures, National Institute of Standards and Technology.