High-Performance Traffic Classification on GPU

Shijie Zhou, Prashant Rao Nittoor and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: {shijiezh, nittoor, prasanna}@usc.edu

***Abstract*—Traffic classification is an essential task in network management. 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 obtaining speedup can be challenging. In this paper, we design a high-performance traffic classifier based on an alternate representation of the C4.5 decision-tree algorithm and implement it using Compute Unified Device Architecture (CUDA). To remedy the unbalanced nature of the decision-trees arising in traffic classification, we convert the C4.5 decision-tree into a set of completely balanced range-trees. Classification is performed by searching the range-trees and merging the search results. We optimize our design by storing the range-trees using compact arrays without explicit pointers in shared memory. By exploiting thread level parallelism, we develop throughput-optimized as well as latency-optimized designs. Experimental results show that for a typical decision-tree containing 128 leaf nodes and 6 features, our design achieves a throughput of over 1600 million classifications per second (MCPS). Compared with the state-of-the-art multi-core implementation, our design demonstrates 16x improvement with respect to throughput. We also demonstrate similar performance improvements on a variety of decision-trees with respect to number of leaf nodes, structure of the tree and number of features.**

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

I. Introduction

Many of the network management tasks such as flow prioritization, traffic shaping and diagnostic policing rely on accurate network traffic classification. During traffic classification [1], network traffic flows are categorized into a number of classes according to various parameters such as port number, average packet size, etc. Once a traffic class is decided, a predetermined policy can be applied to either guarantee a certain quality (eg. media streaming service) or to provide best-effort delivery. As the Internet grows rapidly, high-performance traffic classifiers are essential in order to handle the high data rates and volume of traffic in the network.

Traffic classification can be performed by various techniques. Traditional traffic classification is on the basis of transport layer port number. However, since many applications today tend to dynamically allocate port numbers, classifiers based on port number are not reliable. Deep packet inspection (DPI) [2] inspects the actual payload of the packet and can provide high accuracy. But DPI-based techniques can not handle traffic with encrypted payloads (eg. Skype). Heuristic-based approaches [3] classify traffic based on heuristic patterns, but the accuracy is relatively low. Statistics-based classification relies on statistical analysis of traffic features. Machine-learning (ML) algorithms are widely used in statistics-based classifiers. ML-based traffic classifiers can achieve high accuracy [4].

In recent years, there has been an increasing trend in exploring GPUs to realize networking functions [5], [6]. CUDA is a programming model which has been used to show dramatic speedup for floating point intensive applications [7]. However, novel techniques are needed to achieve high performance for online algorithms such as traffic classification.

C4.5 decision-tree based traffic classification is well studied on reconfigurable hardware platforms [8], [9]. However, due to the unbalanced nature of the C4.5 decision-trees used for traffic classification, it is challenging to realize high-performance C4.5 decision-tree based traffic classifiers on GPU. In this paper, we propose a traffic classifier on GPU based on an alternate representation of C4.5 decision-tree. We propose two designs: one for high-throughput and the other for low-latency. The main contributions of this work are:

- Our traffic classifier is based on a novel alternate representation of C4.5 decision-tree algorithm. The C4.5 decision-tree is converted into completely balanced range-trees to perform classification. Compared with a straightforward C4.5 decision-tree implementation, we observe 1.88x throughput improvement.
- 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. An alternate parallel implementation is also developed to reduce the per flow classification latency.
- Experimental results show that our high-throughput design can achieve a throughput of over 1600 MCPS for a typical C4.5 decision-tree. The classification accuracy is over 95%.
- Our design achieves 16x improvement with respect to throughput compared with a highly optimized implementation on the state-of-the-art multi-core platforms.

The rest of the paper is organized as follows: Section II introduces the background and related work; Section III presents the challenges in using GPU for C4.5 decision-tree based classification; Section IV discusses the details of our approach; Section V contains experimental results; Section VI concludes the paper.

This material is based upon the work supported by the National Science Foundation under Grant No. 1018801. The target platform from the University of Southern California's Center for High-Performance Computing is gratefully acknowledged.

II. Background and Related work

A. ML-based Traffic Classification

Traffic classification is performed by checking either packet payloads or packet headers. ML-based approaches study the statistical properties of traffic flows by examining the packet headers. The classification decision is made based upon a set of feature values. A feature can be a characteristic of a single packet (a packet-level feature), or statistics of a set of packets (a flow-level feature). A packet-level feature can be obtained directly from a packet header; a flow-level feature needs to be calculated based on a flow of packets. For instance, the maximum packet size of a specific traffic flow can be viewed as a flow-level feature. TABLE I lists candidate features used by various algorithms. Required features can be obtained through traffic feature extraction engine [11]. [9] has shown that using the features computed from the first 4 packets of a traffic flow can achieve an accuracy of 97.92%.

TABLE I: Candidate Features

Source port number	Destination port number
Average packet size	Variance of packet size
Maximum packet size	Minimum packet size
Maximum inter-arrival time	Minimum inter-arrival time
# of Bytes in forward direction	# of Bytes in back direction

In the literature, many ML-algorithms have been applied for traffic classification, including Naive Bayesian [12], Support Vector Machine (SVM) [16] and C4.5 decision-tree [9]. [4] studies and evaluates the effectiveness of various ML-based traffic classifiers for SSH traffic and Skype traffic. A total number of 22 features, including both the packet-level features and the flow-level features are used to build the ML-based classifiers. Their experimental results show that the accuracy using C4.5 decision-tree based approach is over 83% and 97% for classifying SSH traffic and Skype traffic, respectively. [4] also shows that C4.5 decision-tree based classifiers achieve the highest accuracy among all the considered algorithms.

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 consists of host function and kernel function. The kernel function is invoked by the CPU but executed in parallel by many threads on the GPU. The call to a kernel function involves specifying a configuration which defines the number of threads in a thread block and the number of such blocks. A thread block can be one-dimensional, two-dimensional or three-dimensional. Thread blocks constitute a grid, which can also be one-dimensional, two-dimensional or three-dimensional. Inside a thread block, each group of 32 threads shares the same program counter and executes the same instruction in every clock cycle (SIMT execution model). Such a group of 32 threads is called a warp, which is the basic execution unit on GPU.

A GPU device is composed of several streaming multiprocessors (SMX). In Kepler [10], each SMX has 192 processing units. Each processing unit is called a CUDA core. When a warp encounters long latency operations, the warp scheduler switches it with another warp which is ready to be executed. When threads in a warp take different branches of a program due to conditional statements, divergence occurs. Divergence in thread execution leads to under-utilization 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. Optimizing a CUDA program to avoid divergence is very difficult [14].

CUDA platforms provide various types of memory. Global memory is the largest memory which resides in the device DRAM and can be accessed by every thread. Access latency to the global memory is over hundreds of cycles. In Kepler [10], accessing global memory goes through a two-level cache hierarchy, L1 cache and L2 cache. L1 cache is private to each SMX and L2 cache is shared by all SMXes. Shared memory is on-chip memory and dedicated to each thread block. Threads within the same thread block can cooperate via shared memory. Accessing shared memory can be as fast as accessing registers if there is no shared memory bank conflict. Registers are the fastest on-chip memory. Registers allocated to a thread can not be shared with other threads.

C. Related Work

Many existing traffic classifiers are proposed on reconfigurable hardware platforms such as field-programmable gate array (FPGA). In [8], an FPGA-based architecture is presented based on the C4.5 decision-tree algorithm; optimizations are applied to reduce the number of memory accesses. [15] is based on k-Nearest-Neighbour algorithm and can achieve high accuracy with large training data sets. In [9], a tool is developed which can automatically generate Verilog codes for a binary-tree. The tool is used to map C4.5 decision-tree based traffic classifiers onto FPGA.

Multi-core platforms also draw much attention in the community for traffic classification. In [16], an SVM-based classifier is employed on a dual Xeon platform with 24 cores. The classifier handles both feature extraction and traffic classification, and achieves a throughput of 7 MCPS. In [17], a modification of C4.5 decision-tree algorithm is adopted to design the traffic classifier. The design is implemented on a dual AMD Opteron 6278 platform with 16 cores and can classify 98 million flows per second.

There are not many efforts in exploring GPU for traffic classification. [18] is a DPI based classifier using a modification of the Zobrist hashing algorithm to encode the traffic signatures. It stores the dictionary of signatures in the cached memory of the GPU and harnesses hashing for examination. The classifier is deployed on commodity hardware and achieves 1.8 MCPS throughput. Another DPI based classifier is presented in [19]. They use CPU for buffering the payloads of packets and constructing flows, and then transfer the flows to GPU for processing. The classifier can handle 10 Gbps traffic read from the network interface card and GPU processes flows with a throughput of 12 MCPS. To the best of our knowledge, our work is the first ML-based traffic classifier on GPU which can sustain a throughput of more than 1000 MCPS.

III. Challenges in using GPU for Decision-Tree based Classification

A. C4.5 Decision-tree

C4.5 decision-tree algorithm is a well-known machine learning algorithm proposed in [20]. The decision trees built by the C4.5 algorithm are based on a set of training data using the concept of information entropy. The training data is a set of classified samples with each sample having a number of features. A C4.5 decision-tree consists of internal decision nodes and terminal leaf nodes. At each decision node, the C4.5 algorithm chooses the feature of the data that most effectively splits the set of samples into two subsets. The C4.5 algorithm then is applied on the smaller subsets. If all the samples in a subset belong to the same class, the C4.5 algorithm creates a leaf node corresponding to that class. Fig. 1 shows an example of a C4.5 decision-tree involving 2 features: source port number (SP) and maximum packet size (MPS). If the traffic flow has SP with 80 and MPS with 220, the C4.5 decision-tree will classify it as HTTP class.

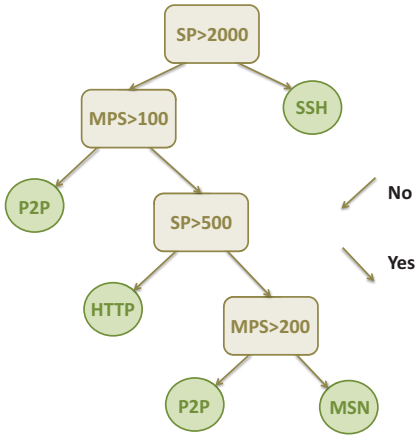

Fig. 1: Example of C4.5 decision-tree

B. Challenges

There are many challenges to be addressed for C4.5 decision-tree based traffic classifiers to achieve high performance on GPU.

Firstly, traversal on a tree data structure involves many conditional statements. A conditional statement introduces divergence within a warp. GPU handles divergence by executing the two divergent codes serially one after another. The overhead caused by the divergence is significant. Greater the depth of the tree, higher the branching resulting in underutilization of GPU resources.

Secondly, the C4.5 decision-trees used in traffic classification are highly unbalanced [17]. Traversing a highly unbalanced tree is inefficient on GPU. Since different threads take different amounts of time to reach the leaf node, many threads become idle when waiting for the slower threads. The threads on GPU can not start the next classification until the classification performed by the "slowest" thread in a warp completes.

Thirdly, the performance of the C4.5 decision-tree based traffic classifiers on GPU highly depends on the traffic pattern Based on Fig 1, classifying a mix of SSH traffic and MSN traffic will result in much worse performance than classifying pure SSH traffic.

IV. GPU specific Data Structure and Optimizations

In this section, we discuss the data structure and the optimizations in our design. To reduce the divergence overheads from the unbalanced nature of C4.5 decision-trees, we represent a C4.5 decision-tree as a rule set table (RST), and convert the RST into completely balanced range-trees to perform classification.

A. Rule Set Table Representation

We adopt the same approach of [17] to represent a C4.5 decision-tree as a rule set table (RST). For each leaf node of the decision-tree, there is a path from the root consisting of d decision conditions (d is the depth of the leaf node). Each decision condition is based on a specific feature. For each path from the root to a leaf node, we classify the decision conditions into separate groups based on the feature. Then inside each group, we extract the common range of all the conditions. The extracted range represents the condition of the specific feature to reach the leaf node. If a decision-tree involves M features, M ranges will be obtained for each leaf node. The M ranges constitute a rule. If the number of leaf nodes is N, the resulting RST will have N rules. Note that the size of the RST does not depend on the shape of the decision-tree. Table II shows the RST built for the decision-tree in Fig. 1.

TABLE II: Example of RST

Rule ID	SP	MPS	App. Class
1	(2000, 65535]	(0, 65535]	SSH
2	(0, 2000]	(0, 100]	P2P
3	(0, 500]	(100, 65535]	HTTP
4	(500, 2000]	(0, 200]	P2P
5	(500, 2000]	(200, 65535]	MSN

B. Multi-field Classification

Given an RST with multiple fields (or features) in each rule, the classification problem can be solved by approaches of packet classification [21]. We adopt a range-tree search and bit-vector (BV) based algorithm [22] to perform the classification. The algorithm involves three phases: (1) pre-processing the RST to build a range-tree for each feature, (2) performing range-tree search on each tree to produce an intermediate result (represented as a BV) for each feature and, (3) merging the intermediate results to obtain a final classification result.

1) Pre-processing RST: In this phase, we construct an implicit range-tree [23] for each feature. First, for each column of RST, values of all the boundaries are extracted and sorted. Duplicated values are removed so that the remaining values are unique. We refer to the range between any 2 adjacent unique values as *unique range*. The binary range-tree is built upon the unique values. Note that the depths of leaf nodes differ by at most 1 level. Each leaf node of the range-tree is associated with a BV. The size of BV is equal to the number of rules.

Each bit corresponds to a rule and indicates whether the range satisfies the rule. Fig. 2 illustrates an example of constructing the range-tree for the SP feature based on TABLE II.

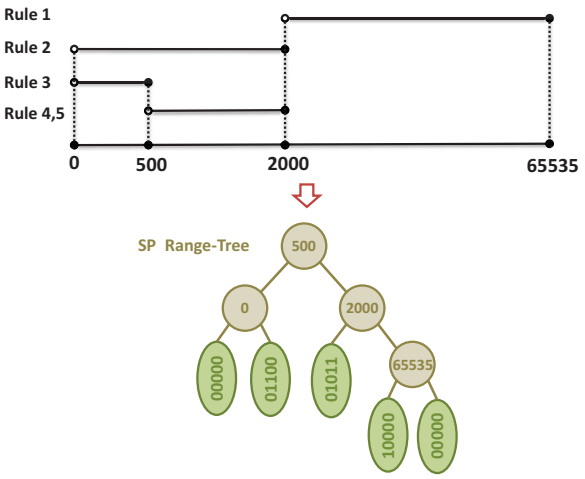

Fig. 2: Example of constructing a range-tree

2) Search: For such an implicit range-tree, the tree-search terminates upon reaching the leaf-node level. The BV associated with the reached leaf node represents the intermediate result of this feature. For example, if the SP of input traffic is 60, based on the range-tree in Fig. 2, "01100" will be obtained, indicating the SP feature matches Rule 2 and Rule 3. We implement two variants of the search:

- High-throughput design: every thread performs M (the No. of features) range-tree searches sequentially.
- Low-latency design: M range-tree searches are performed in parallel by multiple threads within the same warp.

At the end of the search phase, M BVs are obtained.

3) Merge: The merge phase simply performs bitwise AND operations on the M BVs to get a final BV. In the final BV, the non-zero bit indicates which rule satisfies all the features. For instance, for the traffic with SP of 60 and MPS of 1000, the intermediate BVs are "01100" and "10101" based on Table II, respectively. The final BV will be "00100", indicating Rule 3 is matched. Therefore, such traffic is classified as HTTP. Note that there will be at most 1 non-zero bit in the final BV. The reason is that for a traffic flow, there is only 1 leaf node of the original C4.5 decision-tree that can be reached; that leaf node corresponds to exactly 1 rule in the RST.

C. Optimizations

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 use arrays to store the range-trees. To hold the tree with depth of d, the size of the array is $2^d - 1$. The root of the tree is stored in array[0]. For the node stored in array[i] ($0 \leq i < 2^d - 1$), its left child node is stored in array[$2i+1$] while its right child node is stored in array[$2i+2$]. To move from a node to its children, we 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 [24]. In a perfect binary tree, all the leaf nodes have the same depth. To convert the range-tree into a perfect binary tree, for any non-leaf node which is not fully filled, we duplicate it to its missing child's location in the array. The BVs of leaf nodes are calculated based on the perfect binary tree. Fig. 3 depicts the perfect binary tree converted from the range-tree in Fig. 2. Note the conversion of the range-tree will not lead to observable negative impact on performance. This is due to: (1) the original range-tree is already highly balanced; (2) the depth of range-tree does not change, which determines the latency of the "slowest" thread.

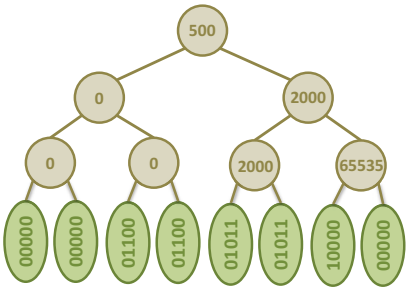

Fig. 3: Complete SP range-tree

Based on the above storage of range-trees, search and merge steps are illustrated in Algorithm 1[1]. The number of required memory accesses to traverse each range-tree is equal to the depth of the range-tree. The computation complexity of Algorithm 1 is O($M \times \log N$) for search and O($M \times N$) for merge. Note that each range-tree is built upon the unique ranges of a feature and the number of unique ranges of each feature is less than N.

Algorithm 1 Search and Merge

Let $Tree_m[\,]$ $(m = 1, ...M)$ denote the range-tree arrays
Let L_m denote the number of non-leaf nodes in $Tree_m$
Let f_m denote the features of input traffic
Let BV_m store the intermediate results
Let N denote the number of rules
Search and Merge(f_1,...,f_M)

```
for m = 1; m < M; m++ do
    ID = 0;
    while ID < L_m do
        ID = ID * 2 + 1 + (f_m ≥ Tree_m[ID]);
    end while
    BV_m = Tree_m[ID];
end for
Final_BV = &_{m=1}^{m=M} BV_m;
return Class_of_rule(N − log_2(Final_BV));
```

L1 cache might be polluted by unnecessary data and the access pattern of L1 cache is irregular. On the contrary, shared memory is user controllable on-chip memory. We store range-trees and BVs in shared memory for fast access. Suppose an

[1]For simplicity, Algorithm 1 assumes non-leaf nodes store boundary values and leaf nodes store BVs. In fact, BVs and range-trees are separately stored.

N-leaf-node[2] decision-tree with M features is converted into an RST; RST is further translated into M completely balanced range-trees. The maximum memory consumption[3] is $M \times N$ bytes for storing M range-trees, and $\frac{M \times N^2}{2}$ bits for storing the associated BVs. For $N = 128$ and $M = 6$, the maximum memory consumption is less than 7 KB. State-of-the-art GPUs offer 48 KB shared memory per SMX; thus the range-trees and BVs can be fit in on-chip shared memory. When N is too large for shared memory to store all the data, we store range-trees in shared memory and BVs in global memory. Note that real-life C4.5 decision-trees typically have $80-120$ leaf nodes and $40-80$ unique ranges per feature [17].

In Kepler [10], shared memory is divided into 32 equally sized memory banks. Each bank has a bandwidth of 64 bits per clock cycle. Multiple concurrent accesses to the same memory bank leads to bank conflict. This results in the accesses serialized. Since threads in a warp examine the same level of a range-tree in each clock cycle, bank conflicts are very likely. Fig. 4 depicts a 16-way bank conflict scenario, which will take 16 clock cycles to resolve.

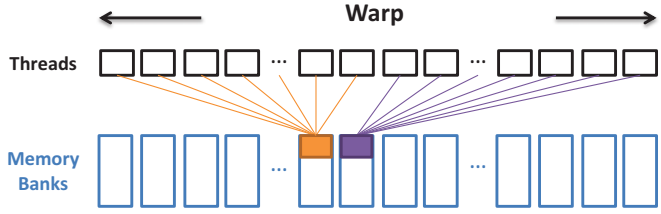

Fig. 4: 16-way bank conflict

To reduce shared memory bank conflicts, we can make multiple copies of range-trees. We store the copies of the same node in distinct shared memory banks. Threads choose which copy to access according to their thread indexes. For example, in Fig. 5 two copies of each node are made and the 16-way bank conflict of Fig. 4 can be reduced to an 8-way bank conflict.

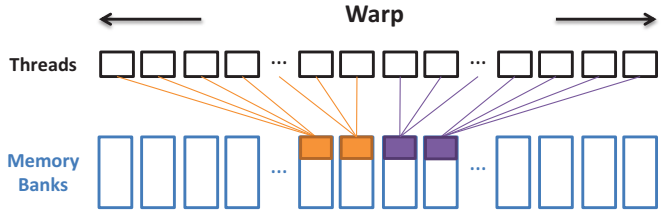

Fig. 5: 16-way bank conflict reduced to 8-way Bank conflict

V. Performance Evaluation

A. Experimental Setup

Our implementations are based on CUDA 5.5. The platform is 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. It has 13 SMXes with 2496 CUDA cores in total and is equipped with 5GB GDDR5. Hardware events are captured by NVIDIA visual profiler.

[2]Assume N is a power of 2.

[3]This occurs when every range of each feature is unique.

The data sets we use are provided by Tstat [25], a publicly available traffic trace. We target 6 features including protocol, source port number, destination port number, average packet size, maximum packet size and minimum packet size. Classifiers based on the 6 features have demonstrated high classification accuracy [9]. We consider 8 network traffic classes, which are HTTP, MSN, P2PTV, QQ_IM, Skype, Skype_IM, Thunder, and Yahoo_IM. Converting the C4.5 decision-tree to RST and constructing range-trees are done by the CPU.

We define classification accuracy as the average percentage of correctly classified flows. The classification accuracy is measured using WEKA [26], a widely used ML software. Comprehensive experiments were conducted to generate decision-trees with classification accuracy over 95%. Overall throughput is the main metric when we evaluate the performance. We define overall throughput as the number of classifications performed per second. We assume the features of traffic have been extracted and stored in the global memory of GPU. Throughout Section V, we use "thread block" and "block" interchangeably.

B. C4.5 Decision-tree vs. Proposed Approach

In this section, we compare the performance of the C4.5 decision-tree (without any modification) against our proposed algorithm which converts the C4.5 decision-tree to balanced range-trees. All the data required by threads is stored in the global memory and accessed through L1 and L2 caches. In these experiments, we employed 512 threads per block and 52 blocks per grid.

Statistics show that most of the decision-trees have 80 to 120 leaf nodes. Thus, the corresponding RSTs will contain $80-120$ rules. We define $unbalance\ factor$ (denoted as U) to reflect the unbalance degree of C4.5 decision-trees. Let d_{max} denote the depth of the tree, d_{min} denote the depth of the most shallow leaf node, and d_{avg} denote the average depth of leaf nodes. The unbalance factor of a decision-tree is given by:

$$U = \frac{d_{avg} - d_{min}}{d_{max}}$$

For example, the decision-tree in Fig. 6 has an unbalance factor of 0.33. Note that the more balanced a decision-tree is, the smaller its unbalance factor will be ($0 \leq U < 1$). A completely balanced decision-tree has a unbalance factor of 0.

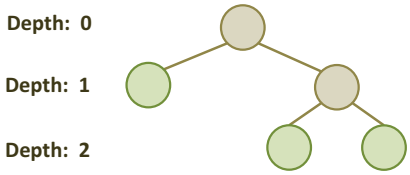

Fig. 6: A decision-tree with $U = 0.33$

We noticed that most unbalance factors of the generated decision-trees were in the range 0.5 to 0.6. Fig. 7 shows the performance of decision-tree approach and proposed approach under various unbalance factors. Our proposed approach achieves 1.88x higher throughput. As the unbalance factor increases, the throughput of the direct implementation of the decision-tree approach shows a slowly dropping trend, while

the performance of our approach is not affected. We summarize the advantages of our proposed solution compared with the direct implementation of the C4.5 decision-tree algorithm as follows:

- Our approach has much less divergence overhead. Every thread examines the same feature at a given time and the range-trees are completed balanced.
- In decision-tree based approach, threads which terminate classification earlier become idle. This makes the hardware resources under-utilized.
- Since we convert the decision-tree to completely balanced range-trees and perform searches on the balanced range-trees, the performance does not depend on the shape of the decision-tree or the traffic pattern.

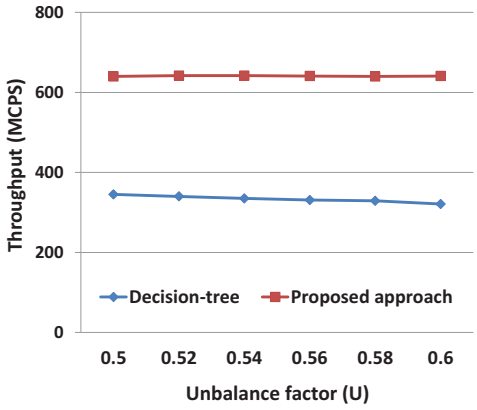

Fig. 7: Performance under various unbalance factors

C. Occupancy

Occupancy is defined as the ratio of the number of active warps per SMX to the maximum number of possible active warps [13]. It reflects how efficiently hardware is kept busy. Low occupancy implies hardware is under-utilized. However, increasing occupancy does not always improve performance. Moreover, occupancy often depends on the availability of shared memory and registers. TABLE III shows the compute capabilities of our target platform.

TABLE III: Compute Capability

Max. # of warps per SMX	64
Max. # of blocks per SMX	16
Max. # of threads per block	1024
Max. # of threads per SMX	2048

We vary the size of the block and the number of blocks to explore the impact of occupancy. TABLE IV shows the occupancy and throughput under various configurations. When the number of threads per thread block is 512 or 1024, upto 64 warps[4] can reside in each SMX. Since the target platform contains 13 SMXes, the last 2 configurations in TABLE IV hit the upper bound of possible concurrent launching threads. We have the following observations:

[4] 2×1024-thread block or 4×512-thread block

- As the occupancy increases, the throughput increases.
- When the number of threads reaches 26 K, occupancy is 99.8% and the throughput saturates; increasing the number of threads beyond 26 K does not improve the throughput.

TABLE IV: Occupancy and Throughput

# of threads/block	512	1024	512	1024	512	1024	512
# of blocks	1	1	13	13	26	26	52
Occupancy (%)	24.6	49.2	24.9	49.9	49.9	99.8	99.8
Throughput (MCPS)	41	78	520	613	613	640	640

D. Optimization using Shared Memory

In this section, we explore the effect of using shared memory in GPUs. Instead of repeatedly fetching the range-trees from the global memory, we store them in the shared memory for faster access. In the target platform, L1 cache and shared memory share a 64 KB memory segment per SMX. According to user's preference, ratio of L1 cache and shared memory can be 1:3, 1:1 or 3:1. We choose the first ratio which offers 48 KB shared memory per SMX. Since shared memory is partitioned among blocks, the amount of shared memory that each block can acquire depends on the number of blocks per SMX. We vary the size of the block to vary the number of blocks in each SMX. In Fig. 8, we show the performance under various configurations. We use the notation of "$A \times B$" along x-axis as follows: A refers to the number of threads per block and B refers to the number of blocks per SMX.

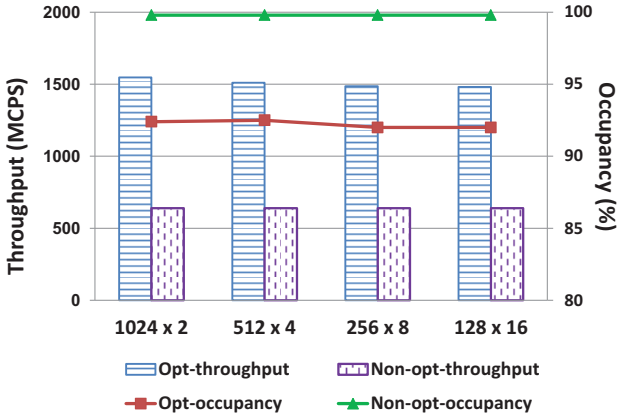

Fig. 8: Performance under various configurations

Compared with the non-optimized implementation, we observe that:

- Throughput is improved by 2.3x.
- Occupancy drops due to the use of shared memory.

We make multiple copies of range-trees to reduce the number of shared memory bank conflicts. We vary the number of copies under the configuration of "1024×2" to allow each block to acquire 24 KB of shared memory. Fig. 9 shows the results. We observe:

- Throughput increases by over 50 MCPS when the number of copies doubles from 2 to 4.

- Occupancy drops if we make more copies in the shared memory.
- The highest throughput is achieved when we make 4 copies; making more copies beyond this does not further improve performance.

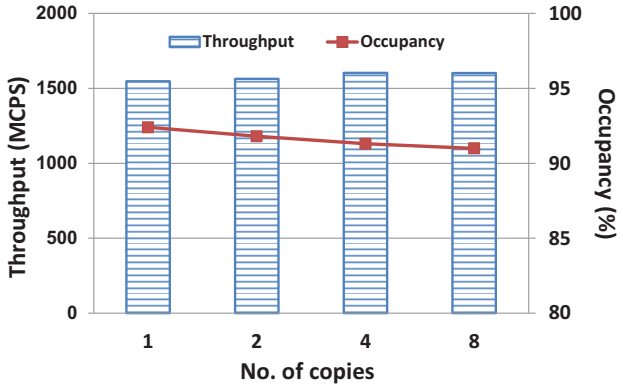

Fig. 9: Performance under various number of copies

E. Peak vs. Sustained Throughput

In order to determine the efficiency of our classifier, we calculate the theoretical peak throughput that can be achieved under ideal conditions. These ideal conditions include ignoring overheads such as shared memory bank conflicts, stalls caused by long latency operations and data dependencies. The peak throughput only depends on the number of executed instructions for classification. We counted the total number of instructions executed for a classification by our approach to be 363. On the target platform, each SMX features four warp schedulers and eight instruction dispatch units. The quad warp scheduler selects four warps, and two independent instructions per warp can be issued in each cycle. The clock rate of the GPU used in our experiments is 705.5 MHz. This results in a peak throughput of 6468 MCPS[5]. Thus, our design achieves 25% of the peak throughput. The main overheads leading to the gap between the peak throughput and the sustained throughput are:

- Reading feature values and writing back results require long-latency memory access to global memory.
- The executed instructions per cycle (IPC) is 1.4, which is less than 2. Execution dependency is the main cause for low IPC.
- Arithmetic operations take from 18 to 22 cycles [27]. Each thread switch introduces non-zero overhead.

F. Latency vs. Throughput

For latency-sensitive networks, we exploit more parallelism in our design to reduce the classification latency. As opposed to performing 6 range-tree searches sequentially for each classification, we harness 6 threads to perform 6 range-tree searches in parallel. By this approach, the average latency per classification can be reduced from 16.5μs to 7.7μs while still sustaining a throughput of 1281 MCPS. The throughput deteriorates due to the needed synchronization among threads before merging the BVs.

[5]Peak throughput = $2 \times 4 \times 32 \times 13 \times 705.5/363$ MCPS

G. Scalability

To study the scalability of our approach, we conduct experiments using synthesized RSTs. The following 3 parameters were varied: number of features (M), ratio of number of unique ranges per feature to the total number of rules (R) and total number of rules (N).

We fix M at 6, and vary R from 0.2 to 0.7 and N from 128 to 256. As shown in Fig. 10, the throughput degrades for larger N and R. The reasons for the deterioration are:

- As R or N increases, the size of range-trees becomes larger. Hence, the number of memory accesses increases correspondingly.
- When more shared-memory is required to store the range-trees, occupancy will be negatively affected.
- When N doubles, the number of executed instructions for merging the BVs doubles as well.

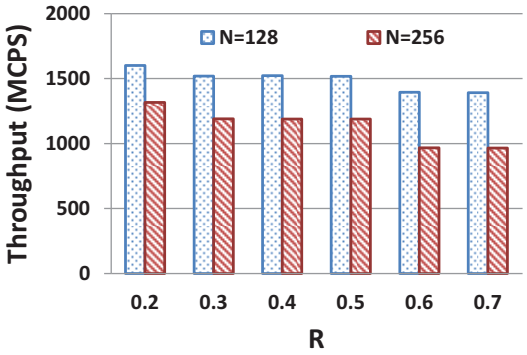

Fig. 10: Varying R

We fix R at 0.25, and vary M from 6 to 9 and N from 128 to 256. The results are shown in Fig. 11. Since the computation complexity of our approach is linear in M, throughput degrades as M increases.

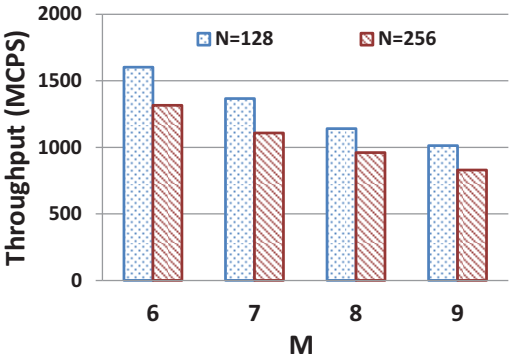

Fig. 11: Varying M

In summary, for any C4.5 decision-tree with upto 128 leaf nodes and upto 9 features, our approach achieves a throughput of at least 1000 MCPS. Note that a typical C4.5 decision-tree for traffic classification contains 80−120 leaf nodes [17].

H. Comparison with Multi-core Implementation

In this section, we compare the performance of our GPU based design with the state-of-the-art multi-core implementation [17]. The platform used in [17] is a dual-socket AMD Opteron 6278 with 16 physical cores. The clock rate of each core is 2.4 GHz. Using 6 features, we compare the throughput with various pairs of N and R. The comparison results in Fig. 12 show that our work achieves 16x higher throughput.

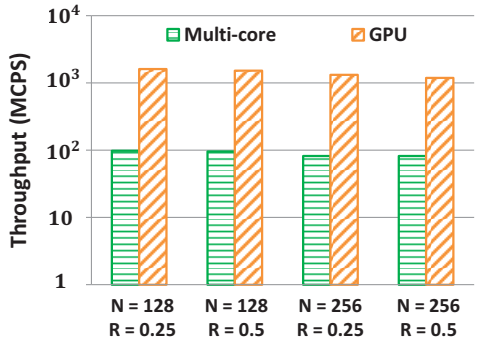

Fig. 12: Multi-core vs. GPU (using log scale)

VI. Conclusion

In this paper, we presented a high-performance traffic classifier on a GPU platform. The classifier was based on an alternate representation of C4.5 decision-tree algorithm. The C4.5 decision-tree was first converted into a rule set table, then a range-tree and BV based approach was adopted to perform classification. We use compact arrays to store range-trees without explicit pointers for efficient tree-search on GPUs. Our approach significantly reduces the divergence overheads. On-chip shared memory was exploited to achieve high performance. We also demonstrated alternate parallelism to reduce the per flow classification latency. For accelerating a typical decision-tree with upto 128 leaf nodes and 6 features, our design achieved a throughput of over 1600 MCPS. Compared with a state-of-the-art multi-core implementation, our work demonstrated 16x improvement with respect to throughput.

In the future, we plan to develop high-speed feature extraction algorithm with throughput at par with the traffic classifier in this paper, so that the entire traffic classification process chain could be pipelined. We observe that when the number of features increases, the throughput decreases in both high-throughput and low-latency implementations. In the high-throughput implementation, the deterioration is because of more computation per thread; whereas in the low-latency implementation, the deterioration is due to significant amount of time spent for synchronization among threads. In the future, we plan to develop an algorithm that makes the best of both designs to get better performance for more number of features.

References

[1] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, "Traffic Classification on the Fly," SIGCOMM Comput. Commun. Rev., vol. 36, no. 2, pp. 23-26, 2006.

[2] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," Micro, IEEE, vol. 24, no. 1, pp. 52-61, 2004.

[3] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel Traffic Classification in the Dark," SIGCOMM Comput. Commun. Rev., vol. 35, no. 4, pp. 229-240, 2005.

[4] R. Alshammari and A. N. Zincir-Heywood, "Machine Learning Based Encrypted Traffic Classification: Identifying SSH and Skype," in Proc. of CISDA, pp. 289-296, 2009.

[5] Y. Li, D. Zhang, A. X. Liu and J. Zheng, "GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers," in Proc. of ANCS, pp. 1-12, 2013.

[6] W. Sun and R. Ricci, "Fast and Flexible: Parallel Packet Processing with GPUs and Click," in Proc. of ANCS, pp. 25-36, 2013.

[7] 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.

[8] Y. Luo, K. Xiang, and S. Li, "Acceleration of Decision Tree Searching for IP Traffic Classification," in Proc. of ANCS, pp. 40-49, 2008.

[9] D. Tong, L. Sun, K. Matam, and V. Prasanna, "High Throughput and Programmable Online Trafficclassifier on FPGA," in Proc. of FPGA, pp. 255-264, 2013.

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

[11] S. Li and Y. Luo, "High performance flow feature extraction with multi-core processors," in Proc. of NAS, pp. 193-201, 2010.

[12] A. W. Moore and D. Zuev, "Internet Traffic Classification Using Bayesian Analysis Techniques," SIGMETRICS Perform. Eval. Rev., vol. 33, no. 1, pp. 50-60, 2005.

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

[14] B. Coutinho, D. Sampaio, F. M. Q. Pereira and W. M. Jr., "Profiling divergences in GPU applications," Concurrency and Computation: Practice and Experience, vol 25, no. 6, pp. 775-789, 2013.

[15] W. Jiang and M. Gokhale, "Real-Time Classification of Multimedia Traffic Using FPGA," in Proc. of FPL, pp. 56-63, 2010.

[16] F. Gringoli, L. Nava, A. Este, and L. Salgarelli, "MTCLASS: Enabling Statistical Traffic Classification of Multi-gigabit Aggregates on Inexpensive Hardware," in Proc. of IWCMC, pp. 450-455, 2012.

[17] D. Tong, Y. Qu and V. Prasanna, "High-throughput Traffic Classification on Multi-core Processors," in Proc. of HPSR, 2014.

[18] G. Szabo, I. Godor, A. Veres, S. Malomsoky and S. Molnar, "Traffic Classification over Gbit Speed with Commodity Hardware," IEEE Journal of Communications Software and Systems, Vol. 5, Num. 3., 2010.

[19] R. Leira, P. Gomez, I. Gonzalez and J.E. Lopez de Vergara, "Multimedia flow classification at 10 Gbps using acceleration techniques on commodity hardware," in Proc. of SaCoNeT, pp. 1-5, 2013.

[20] J. R. Quinlan, "C4.5: programs for machine learning," Morgan Kaufmann Publishers Inc., 1993.

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

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

[23] T. Ganegedara and V. Prasanna, "100+ Gbps IPv6 Packet Forwarding on Multi-Core Platforms," IEEE Global Communications Conference (GLOBECOM), 2013.

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

[25] "TCP Statistic and Analysis Tool," http://tstat.tlc.polito.it/traces.shtml

[26] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," SIGKDD Explor. Newsl., vol. 11, no. 1, pp. 11-18, 2009.

[27] "Fundamental CUDA Optimization," http://www.arc.vt.edu/userinfo/training /2013Sum_NVIDIA/VT05_Fundamental_CUDA_Optimization.pdf