Scalable GPU-accelerated IPv6 Lookup using Hierarchical Perfect Hashing

Shijie Zhou
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: shijiezh@usc.edu

Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: prasanna@usc.edu

Abstract—**IPv6 has been proposed to fulfill the increasing demand of IP addresses. As the data rate and volume of network traffic keep increasing and the Internet evolves, high-speed IPv6 lookup for large routing tables is essential. In this paper, we propose a novel IPv6 lookup approach based on hierarchical perfect hashing. The lookup complexity of the proposed algorithm is *O*(1) in the worst case. The performance is independent of the size of the routing table. Each lookup is performed by examining up to 3 perfect hash tables. Each hash table uses a range of bits of the input IP address as lookup key. We develop a simple scheme to choose appropriate key length for each hash table, which can efficiently reduce the total memory requirement. We implement our design on a state-of-the-art Compute Unified Device Architecture (CUDA) platform. Experimental results show that our GPU-accelerated lookup engine is scalable to sustain a high throughput of over 1.6 billion lookups per second (GLPS) for routing tables from 10K to 1M. This corresponds to 80% of the peak throughput of the target platform. Compared with a state-of-the-art GPU-based IPv6 lookup engine, our design demonstrates 2x improvement with respect to throughput for large tables.**

I. INTRODUCTION

Internet Protocol address (IP address) serves as an identification of each device in a computer network. Today, the pool of unallocated IPv4 addresses is exhausted. To fulfill the increasing demand of IP addresses, IP designers have proposed to migrate from IPv4 to IPv6, which uses a 128-bit number to represent an IP address [1].

IP lookup is a core function of Internet routers to forward IP packets. It requires to perform longest prefix match (LPM) [2] based on the destination IP address. Most of the existing approaches for IPv4 lookup are trie-based [3] or tree-based [4]. However, it is challenging to scale these approaches to support IPv6 lookup for large routing tables due to:

- Since IPv6 addresses are longer than IPv4 addresses, trie-based approaches require more trie-levels and thus more memory accesses.
- The lookup complexity for tree-based approaches is $O(\log N)$, where N is the size of the routing table. Performance deteriorates as N increases.

This work has been funded by U.S. National Science Foundation under grants CCF-1320211 and CCF-1116781.

Novel techniques are required to achieve high-throughput and scalable IPv6 lookup. Perfect hashing [5] is a hashing technique to map a set of elements to a set of integers without collision. Searching a perfect hash table requires $O(1)$-time. Our approach is based on perfect hashing to guarantee $O(1)$ memory accesses for IPv6 lookup, regardless of the size of the routing table.

Both software and hardware platforms have been explored to perform IP lookup. While hardware platforms such as Ternary Content Addressable Memory (TCAM)-based [6], [7] and Field-programmable Gate Array (FPGA)-based [8] architectures can deliver high throughput, the processing flexibility is limited. Software-based routers gain interest because of extensibility and customizability [4]. But the throughput of software-based routers is not as high as hardware-based solutions. While CUDA programming model has been used to gain dramatic speedup for floating point intensive applications [9], it is challenging to obtain speedup for integer-based throughput oriented applications. In recent years, there has been an increasing trend in exploring GPUs to accelerate networking applications [10-16]. In this paper, we implement our IPv6 lookup engine on a state-of-the-art CUDA platform.

The main contributions of this paper are:

- A novel approach for IPv6 lookup based on hierarchical perfect hashing, which needs $O(1)$ memory accesses per lookup. The throughput is independent of the routing table size.
- A simple and efficient mechanism to choose the length of hash keys for each hash table to reduce memory footprint.
- Optimized implementation on a state-of-the-art CUDA platform which achieves a throughput of 1.6 GLPS for large routing tables. This corresponds to over 2x improvement compared with a state-of-the-art GPU-based IPv6 lookup engine.

The rest of the paper is organized as follows. Section II provides the background. Section III introduces related work. Section IV discusses the algorithm of our approach. Section V reports experimental results. Finally, Section VI concludes the paper.

II. Background

A. *Trie-based and Tree-based Algorithms for IP lookup*

Entries in the routing table are specified using various lengths of prefixes. Each entry also contains the information of the corresponding next hop and output interface. For each IP packet, routers need to select an entry which has the longest prefix match with the destination IP address [2]. Assuming each IP address is a 6-bit number, TABLE I shows an example of a routing table. Based on TABLE I, for an IP packet with destination IP "110000", the router selects Entry 3 (E3) and forwards the packet to Interface 1.

TABLE I: Example of Routing Table

Entry	Prefix	Interface
E1	10****	2
E2	11****	4
E3	110***	1
E4	1110**	3

Trie-based and tree-based algorithms are widely used for IP lookup [17]. Trie represents each prefix by a node. The path from the root to a node is determined based on the prefix value. At each node, one or multiple bits are used to make a branching decision. If only one bit is used to make each branching decision, the trie is called a uni-bit trie [17]. If multiple bits (called a stride) are used to make each branching decision, the trie becomes a multi-bit trie [17]. Fig. 1 depicts the uni-bit trie and multi-bit trie (stride length s=2) built for the routing table in TABLE I. IP lookup is performed by traversing the trie until a leaf node is reached. Some variants of trie-based approaches, such as LC-tries [18] and Tree Bitmap [19], can realize a table compression at the cost of more memory accesses.

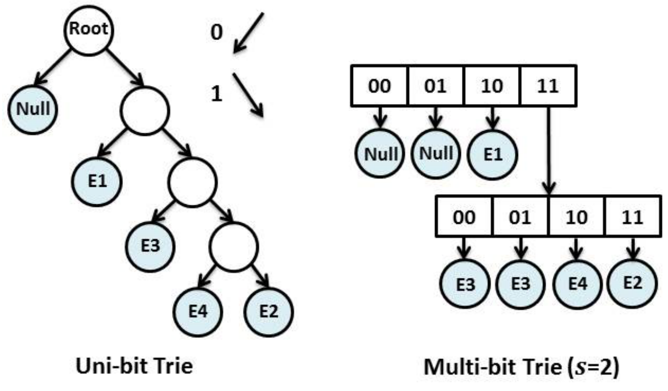

Fig. 1 Uni-bit trie and Multi-bit trie for Table I

The lookup complexity for trie-based approaches is $O(w)$, where w is the maximum prefix length. Traversing a uni-bit trie or multi-bit trie requires many memory accesses for IP lookup [20]. This is problematic when each memory access incurs long latency.

Range-trees are also widely adopted to perform IP lookup [17]. Each prefix can be represented by a range. For example, the prefix of E1 in TABLE I can be specified by a lower bound "100000" (32 in decimal) and a upper bound "101111" (47 in decimal). All such pairs of boundaries are used to produce a set of non-intersecting sub-ranges. An implicit range-tree can be built based on the sub-ranges. The lookup complexity for tree-based approaches is $O(\log N)$, where N is the size of the routing table. Thus, the number of memory accesses increases as N increases, resulting in poor scalability to support large routing tables.

B. *Perfect Hashing*

Hashing is a fundamental technique widely used in various applications. Compared with other hashing techniques, perfect hashing has the advantage that its mapping is injective without hash collisions. A two-level scheme with universal hashing at each level is widely used to build perfect hash table [5]. Universal hashing [5] refers to randomly selecting a hash function from a family of hash functions which has a certain mathematical property. It can guarantee a low number of collisions in expectation. Given P hash keys, the two-level universal hashing scheme works as follows:

- A first-level hash table A is built using a random universal hash function H. Since this may produce collisions, some hash slots of A may contain multiple elements.
- If a hash slot of A only contains one element, it stores the corresponding hash value.
- If a hash slot of A contains more than one element (say p elements, $1 < p \leq P$), it points to a second-level hash table B which has p^2 hash slots. A second-level universal hash function h is randomly chosen to hash the p elements in B without collision.

Such a perfect hash table requires $O(P)$-space [5].

III. Related work

A. *Pefect Hashing for IPv4 Lookup*

[20] proposes an approach for IPv4 lookup based on perfect hashing and direct addressing. For a given routing table, the approach defines a parameter n_{opt} ($n_{opt} < 32$), which is the length of hash keys of the perfect hash table. The prefixes shorter than n_{opt} -bit are expanded into n_{opt} -bit hash keys; the other prefixes use the first n_{opt} bits as hash keys. A perfect hash table for the hash keys is built by iteratively increasing the hash table size until there is no collision. Each hash key corresponds to a direct addressing vector of size $2^{(32-\ n_{opt})}$. To perform a lookup, the algorithm uses the first n_{opt} bits of the input IP as lookup key and performs hash function to locate a direct addressing vector, then uses the remaining $(32 - n_{opt})$ bits of the input IP as an index to access the corresponding element of the vector. However, the approach of [20] for IPv6 lookup raises following issues:

- Since IPv6 uses 64 bits for forwarding [21], each direct addressing vector will contain $2^{(64-n_{opt})}$ elements. The memory requirement will be prohibitive.
- During the construction of the perfect hash table, if a hash collision occurs, [20] enlarges the hash table and restarts the construction. The perfect hash function is not guaranteed to be found.

B. *GPU-based IPv4 lookup*

A GPU-accelerated software router called PacketShader is presented in [10]. The algorithm needs one memory access if the input IP matches a prefix which is less than 24 bits, otherwise it needs one more memory access. PacketShader achieves a throughput of 78 MLPS for forwarding IPv4 packets based on a 282K-entry routing table. In [11], a large direct table is built based on the fact that most of IPv4 prefixes are no longer than 24 bits. The direct table is stored on GPU for lookup. Each input IP has exactly one matching entry in the direct table to enable $O(1)$ memory access. [11] achieves a throughput of 250 MLPS for 440K-entry IPv4 routing tables. [14] implements the IPv4 lookup engine using leaf-pushed segment tree. Such tree-like data structure supports dynamic update. The work achieves a throughput of 539 MLPS for 440K-entry IPv4 routing tables.

C. *GPU-based IPv6 lookup*

A multi-bit trie based IPv6 lookup engine is presented in [15]. The multi-bit trie is encoded into a state jump table to enable efficient global memory accesses on GPU. Meanwhile, CPU maintains a copy of the multi-bit trie and performs offline updates. The best performance is achieved when the trie contains 20 levels. For a real-life IPv6 routing table with 11K entries, the achieved throughput is 658 MLPS. Another multi-bit trie based IPv6 lookup engine using GPU is presented in [16]. The best performance is achieved when the stride length is 8. The kernel sustains a throughput of 3.6 GLPS for 5K entry IPv6 routing tables. However, the data transfer overhead between the CPU and GPU is not considered.

IV. ALGORITHMS

In this section, we first discuss our approach to build a single perfect hash table, then we introduce the entire hashing structure which is a 3-level hierarchical perfect hash table. Throughout this paper, we use T_1 to denote a first-level perfect hash table and S_1 to denote a hash slot of T_1; similarly, T_2, T_3, S_2 and S_3 can be defined.

A. *Single Perfect Hash Table*

We adopt a two-level universal hashing scheme [5] to construct each perfect hash table. Algorithm 1 shows how to build up a perfect hash table given a set of key-value pairs. We leverage the multiply-shift universal hashing family [22] which can be represented as:

$$h_a(x) = ((a \bullet x) \bmod 2^w) / 2^{w-l} \quad (1)$$

The hash functions of this family map a w-bit key to a l-bit integer. In Equation (1), x is the hash key and a is a random w-bit positive odd integer. The number of hash slots in the hash table is 2^l. We choose this universal hashing family because:

- It needs a small number of parameters. When collision occurs, only the parameter a of the hash function needs to be re-chosen.
- The computation required by the hash functions are simple and fast multiply-shift operations.

Algorithm 1 Constructing a perfect hash table

```
Let $P$ denote the number of hash keys
Let $key_p$ denote the $p$th hash key ($0 < p \le P$)
Let $value_p$ denote the value of $key_p$
Let M denote the number of hash slots in the first-level universal hash table
Let $slot[m]$ denote the $m$th hash slot of the first-level universal hash table ($0 < m \le M$)
Let $n_m$ denote the number of hash keys hashed to $slot[m]$ (Initially $n_m$=0)
Let $list[m]$ denote the linked list which stores the hash keys hashed to $slot[m]$
Let $H_{family}$ denote the family of universal hash functions
Build_table ($key_1, \dots, key_P, value_1, \dots, value_P$)
 1:  $H$ = A random hash function of $H_{family}$
 2:  For $p$=1 to $P$ do
 3:    $idx = H(key_p)$
 4:    $n_{idx}=n_{idx}+1$
 5:    $list[idx]$.add($key_p$,$value_p$)
 6:    $slot[idx].key=key_p$
 7:    $slot[idx].value=value_p$
 8:  End for
 9:  For $p$=1 to $P$ parallel do
10:    If $n_p > 1$ then
11:      Create a hash table $T_2$
12:      $H_{T_2}$ = A random hash function of $H_{family}$
13:      Initialize each slot of $T_2$
14:      For $j$=1 to $n_p$ do
15:        $idx = h_{T_2}(list[p].key_j)$
16:        If $T_2.slot[idx]$ is empty then
17:          $T_2.slot[idx].key = list[p].key$
18:          $T_2.slot[idx].value = list[p].value$
19:        Else
20:          Goto Line 12
21:        End if
22:      End for
23:      $slot[p].pointer$ points to $T_2$
24:    End if
25:  End for
```

Algorithm 2 illustrates how to perform a lookup based on a single perfect hash table. In the worst case, a lookup needs two memory accesses.

B. *Hierarchical Perfect Hash Table*

It is desirable to build a single perfect hash table for all the prefixes. However, to realize this, all the prefixes need to be expanded into w-bit numbers, where w is the maximum prefix length. Such expansion results in excessive waste of memory [20]. To reduce the waste of memory, we develop a hierarchical perfect hashing scheme which contains 3-level perfect hash table. If the lookup result can not be determined by the current level perfect hash table, the input IP is directed to the next-level perfect hash table. Each perfect hash table extracts a specific range of bits from input IP as lookup key. We develop an efficient scheme to decide the *range* for each hash table, which further reduces the memory requirement.

```
Algorithm 2 Performing a Lookup
Let $A$ denote the first-level hash table
Let $H$ denote the hash function of $A$
Let $n_m$ denote the number of elements hashed to the $m$th slot of $A$, $(0 < m \le |A|)$
Let $B_m$ denote the second-level hash table
Let $h_m$ denote the hash function of $B_m$
Lookup ($x$)
1:  $idx = H(x)$
2:  If $n_{idx}=1$ do
3:    If $x = A[idx].key$ then
4:      Return $A[idx].value$
5:    Else
6:      Return $no\_match$
7:    End if
8:  Else if $n_{idx}>1$ do
9:    $B_{idx} = A[idx].pointer$
10:   $j = h_{idx}(x)$
11:   If $x = B_{idx}[j].key$ then
12:     Return $B_{idx}[j].value$
13:   Else
14:     Return $no\_match$
15:   End if
16: End if
```

1) First-level Perfect Hash Table: The first-level perfect hash table uses the first 32 bits of input IP as the lookup key (*range* is '[1,32]'). To construct the first-level perfect hash table, prefixes shorter than 32 bits are expanded to 32-bit hash keys; prefixes longer than 32 bits use the first 32 bits as hash keys. We use Algorithm 1 to construct T_1 for all the hash keys. We choose the *range* '[1, 32]' for T_1 due to:

- Our analysis of real-life routing tables [24] reveals that (1) prefixes shorter than 32 bits constitute less than 2.6% of the routing table (2) while prefixes with exact 32 bits constitute over 30%. Thus, selecting a broader *range* leads to many expansion computations.
- Selecting a narrower *range* increases the complexity of the construction for the second-level and third-level perfect hash tables.

2) Second-level and Third-level Perfect Hash Table: If an S_1 has prefix(es) longer than 32 bits hashed into it, T_2 is required to perform LPM. For example, assume an S_1 is created for the prefix "2001:256:4A2F:/48"; the input "2001:256:0:0" matches this S_1 but it does not match the corresponding prefix.

If many prefixes are hashed into the same S_1 and the length gap of these prefixes is large, we create a T_3 to reduce the memory footprint. For example, "2001:256:8000:/33" and "2001:256:8000:1234/64" are hashed into the same S_1; if we only create a T_2, the latter prefix needs to be expanded into 2^{64-33} numbers, resulting in too much waste of memory. In this case, we create a T_2 for "2001:256:8000:/33" and a T_3 for "2001:256:8000:1234/64", respectively.

For a specific S_1, the *range* selection for T_2 and T_3 depends on the prefixes hashed to the S_1. We propose an efficient scheme to choose the *range* for each T_2 and T_3. The default *range* for T_2 and T_3 is '[33,48]' and '[49,64]', respectively. The default setting guarantees that when converting prefixes into hash keys, any prefix will not be expanded into more than 2^{15} hash keys. Among all the prefixes hashed to an S_1, the S_1 records the following values:

- *min*: the minimum length of such prefixes which are longer than 32
- *mid*: the maximum length of such prefixes which are longer than 32 but shorter than 49
- *max*: the maximum length of these prefixes

We use Algorithm 3 to choose the *range* of T_2 and T_3.

```
Algorithm 3 Range selection for $T_2$ and $T_3$
1:  If $max \le 32$ then
2:    $T_2$ is not needed
3:  Else
4:    If $max\text{-}min \le 16$ then
5:      Range of $T_2$ is $[33, max]$
6:      $T_3$ is not needed
7:    Else
8:      Range of $T_2$ is $[33, mid]$
9:      $T_3$ is needed
10:     Range of $T_3$ is $[mid, max]$
11:   End if
12: End if
```

The proposed scheme guarantees that any expansion from prefix to hash keys is no more than 2^{15}. It also avoids creating any unnecessary T_2 and T_3. Algorithm 4 illustrates the steps to perform an IPv6 lookup based on the 3-level perfect hash table.

V. Performance Evaluation

A. Experimental Setup

Our implementation is based on CUDA 6.0. The platform is based on dual 8-core Intel E5-2665 processors running at 2.4 GHz. One NVIDIA K40 Kepler GPU running at 745 MHz is installed as accelerator. It has 15 streaming multiprocessors (SMX) with 2880 CUDA cores in total. The GPU and CPU are connected via a PCIe 3.0×16 bus.

```
Algorithm 4 IPv6 Lookup
Let $T_i$ denote the $i$th-level hash table ($i$ = 1,2,3)
Let $S_i$ denote the hash slot of $T_i$
IPv6_Lookup (ip)
 1:  $key_1$ = Get_key(ip, [1,32])
 2:  $S_1$ = $T_1$.lookup($key_1$)
 3:  If $key_1 \neq S_1$.key then
 4:      Return no_result
 5:  Else
 6:      temp_result = $S_1$.result
 7:      If $S_1$.max < 32 then
 8:          Return temp_result
 9:      Else
10:          $key_2$ = Get_key(ip, $S_1$.range)
11:          $T_2$ = $S_1$.$T_2$
12:          $S_2$ = $T_2$.lookup($key_2$)
13:          If $key_2 \neq S_2$.key then
14:              Return no_result
15:          Else
16:              temp_result = $S_2$.result
17:              If $S_1$.max-$S_2$.min < 16 then
18:                  Return temp_result
19:              Else
20:                  $key_3$ = Get_key(ip, $S_2$.range)
21:                  $T_3$ = $S_2$.$T_3$
22:                  $S_3$ = $T_3$.lookup($key_3$)
23:                  If $key_3 \neq S_3$.key then
24:                      Return temp_result
25:                  Else
26:                      Return $S_3$.result
27:                  End if
28:              End if
29:          End if
30:      End if
31:  End if
```

We collected real-life backbone IPv6 routing tables from the Routing Information Service (RIS) [23]. Each IPv6 routing table typically contains around 11K entries. We used the same approach in [4] to synthesize large IPv6 routing tables and IP traces. Overall throughput is the main metric when we evaluate the performance. We define overall throughput as the number of lookups performed per second.

B. *Multi-core Implementation*

We first implemented our design on the multi-core platform (dual 8-core Intel E5-2665 processors) without using the GPU accelerator. For comparison, we also implement the widely used implicit range-tree approach and multi-bit trie (s=4 and s=8) approach (discussed in Section II-A) on the same platform. We show the performance for various routing table sizes in Fig. 2. It can be observed that our approach achieves much higher throughout and the performance does not deteriorate as the table size increases.

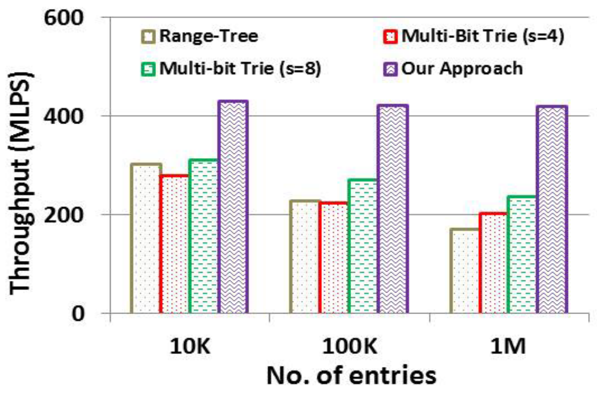

Fig. 2 Performance on Multi-core Platform

C. *CUDA Implementation Details*

In this section, we discuss the details of our GPU-based implementation. In order to increase throughput, our GPU-based design uses batch processing. Every time when a kernel is invoked, a batch of input IPs are transferred from host (CPU) memory to device (GPU) memory and 30K threads are launched to process the batch. After the processing is completed, the lookup results are transferred back from device memory to host memory. However, the data transfer between host memory and device memory suffers long latency due to the limited bandwidth of PCIe bus. To address this issue, we use the multi-stream technique to overlap kernel execution and data transfer [24].

D. *Kernel Performance*

In this section, we compare the kernel performance of our design with range-tree and multi-bit trie approaches. We calculate the kernel performance based on the assumption that IPs are initially stored in the device memory. Thus, the kernel performance does not consider the data transfer overhead between the CPU and GPU. We show the kernel performance of various approaches in Fig. 3. We observe that the throughput of our design is not negatively affected by the size of the routing table.

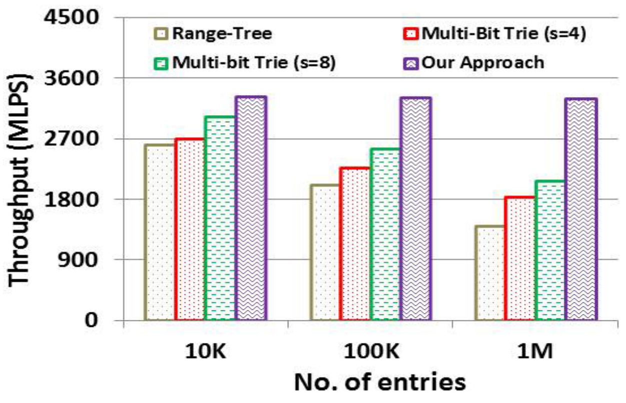

Fig. 3 Kernel Performance

E. *GPU-accelerated Implemenation*

This section presents the overall lookup performance including the data transfer overhead. The overall lookup performance for various sizes of table sizes is depicted in Fig. 4. Our approach is scalable to sustain a high throughput of over 1.6 GLPS for various table sizes from 10K to 1M. This corresponds to 3.6x speedup compared with our multi-core implementation. The PCIe bus of our target platform has a peak bandwidth of 16 GB/s in each direction. Since we transmit 8 bytes per IPv6 address for each lookup, this results in a peak throughput of 2 GLPS. Thus, our design achieves 80% of the peak performance of the target platform. Compared with a state-of-the-art GPU-based IPv6 design [15], our design achieves 2x improvement in throughput.

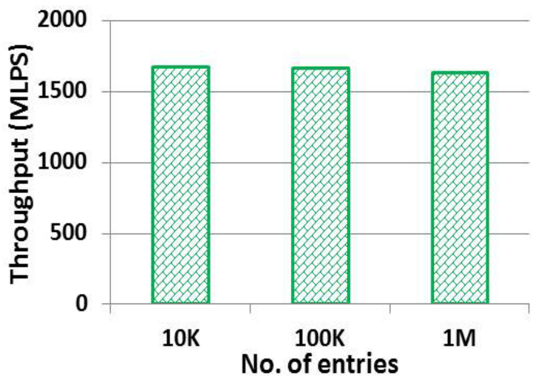

Fig. 4 Overall Lookup Performance

VI. Conclusion

IP lookup is a core network function. Achieving high-performance IPv6 lookup remains a challenging problem. In this paper, we proposed a novel approach based on hierarchical perfect hashing for IPv6 lookup. Using this approach, each IPv6 lookup was performed by examining at most 3 perfect hash tables, which required only 6 memory accesses in the worst case. We conducted comprehensive experiments on a state-of-the-art CUDA platform. Results show that our design achieves high throughput, while the throughput does not degrade as the size of the routing table increases. Including the data transfer overhead between the CPU and GPU, our design sustains a throughput of over 1.6 GLPS for very large routing tables. This corresponds to 2x improvement in throughput compared with a state-of-the-art GPU-based IPv6 lookup design.

References

[1] "Internet Protocol Version 6 (IPv6) Specification," http://tools.ietf.org/html/rfc1883

[2] S. K. Maurya, "Design and Implementation of Longest Prefix Matching Content Addressable Memory for IP Routing," ProQuest, UMI Dissertation Publishing , ISBN-10: 1243433043, 2011.

[3] S. Sahni and K. S. Kim, "Efficient Construction of Multibit Tries for IP Lookup," IEEE/ACM Transactions on Networking (TON), 11(4):650-662, 2003.

[4] T. Ganegedara and V. K Prasanna, "100+ Gbps IPv6 Packet Forwarding on Multi-Core Platforms, IEEE Global Communications Conference (GLOBECOM), pp. 2096-2101, 2013.

[5] "Universal and Perfect Hashing," http://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture notes/lect0215.pdf

[6] K. Zheng, C. Hu, H. Lu, and B. Liu, "A TCAM-based Distributed Parallel IP Lookup Scheme and Performance Analysis," IEEE/ACM Trans. Netw., vol. 14, no. 4, pp. 863-875, 2006.

[7] M. J. Akhbarizadeh, M. Nourani, R. Panigrahy, and S. Sharma, "A TCAM-based Parallel Architecture for High-speed Packet Forwarding, IEEE Transactions on Computers, vol. 56, no. 1, pp. 58-72, 2007.

[8] H. Le and V. K. Prasanna, "Scalable Tree-Based Architectures for IPv4/v6 Lookup Using Prefix Partitioning," IEEE Transactions on Computers, vol. 61, no. 7, pp. 1026-1039, 2012.

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

[10] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in Proc. of ACM SIGCOMM, vol. 40(4), pp.195-206, 2010.

[11] J. Zhao, X. Zhang, X. Wang, Y. Deng and X. Fu, "Exploiting Graphics Processors for High-Performance IP Lookup in Software Routers," In Proc. of INFOCOM, pp. 301-305, 2011.

[12] S. Zhou, P. R. Nittoor and V. K. Prasanna, "High-performance Traffic Classification on GPU," In Proc. of SBAC-PAD, pp. 97-104, 2014.

[13] S. Zhou, S. G. Singapura and V. K. Prasanna, "High-performance Packet Classification on GPU," In Proc. of HPEC, pp. 1-6, 2014.

[14] Y. Li, D. Zhang, G. Xie, J. Zheng and W. Zhao, "An Efficient Update Mechanism for GPU-Based IP Lookup Engine Using Threaded Segment Tree," second CCF Internet Conference of China, pp. 134-144, 2013.

[15] Y. Li, D. Zhang, A. X. Liu and J. Zheng, "GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers," ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 1-12, 2013.

[16] T. Li, H. Chu, A. and P. Wang, "IP Address Lookup using GPU," in Proc. of High Performance Switching and Routing, pp. 177-184, 2013.

[17] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," IEEE Network, vol. 15, no. 2, pp. 8-23, 2001.

[18] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," IEEE Journal on Selected Areas in Communications, pp. 1083-1092, 1999.

[19] W. Eatherton, Z. Ditta and G. Varghese, "Tree Bitmap: Hardware Software IP lookup with Incremental Update," ACM SIGCOMM Computer Communication Review, vol. 34, no. 2, pp. 97-122, 2004.

[20] S. Giordano, F. Oppedisano, G. Procissi and F. Russo, "A Novel High-Speed Micro-Flows Classification Algorithm based on Perfect Hashing and Direct Addressing," IEEE Global Telecommunications Conference (GLOBECOM), pp. 448-452, 2007.

[21] M. Wang, S. Deering, T. Hain, and L. Dunn, "Non-random Generator for IPv6 Tables," In Proc. of IEEE Symposium on High Performance Interconnects, Hot Interconnects, pp. 35-40, 2004.

[22] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, "A Reliable Randomized Algorithm for the Closest-pair Problem," Journal of Algorithms, 25:19-51, 1997.

[23] "RIPE. Ripe Routing Information Service (RIS)," http://www.ris.ripe.net/

[24] "CUDA C Best Practices Guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ #axzz3FJsE6o9I