An Efficient Update Mechanism for GPU-Based IP Lookup Engine Using Threaded Segment Tree

Yanbiao Li[1], Dafang Zhang[1], Gaogang Xie[2], Jintao Zheng[1], and Wei Zhao[1]

[1] College of Information Science and Engineering, Hunan University, Changsha, 410082, China
[2] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China

Abstract. Recently, the Graphics Processing Unit (GPU) has been used to deploy high-speed software routers. On this platform, designing an efficient IP lookup engine is still a challenging task, especially when taking into account the comprehensive performance under frequent updates. Existing solutions either fail in dealing with update overhead, or can not provide stable throughput. In this paper, we first propose the Threaded Segment Tree, a novel tree-like structure. On this basis, we then present a fast IP lookup engine with an efficient parallel update mechanism. According to our experiment results on real-world data, the proposed mechanism reduces the memory accesses on the GPU and the overall update overhead by at least 82.5% and 89.6% respectively. Moreover, it also ensures the lookup engine provides stable throughput under highly frequent updates, which only decreases by less than 1% even though update frequency increases to $100,000$ $updates/s$.

Keywords: GPU, IP lookup, parallel update, segment tree.

1 Introduction

IP address lookup, as a key function of modern routers for packets forwarding and classifying, aims to determine a proper next hop for each incoming packet, through comparing its destination address against all prefixes stored in the Forwarding Information Base (FIB). It is always modeled as a Longest Prefix Matching (LPM) problem.

1.1 Summarize of Prior Arts

Classic solutions to LPM fall into two major categories. Hardware-based solutions always provide very fast lookup [1, 2], but their low flexibilities and high consumptions on power and cost make them unadaptable to large tables, or to growing requirements on scalability. By contrast, software-based solutions are proved more flexible due to some tree-like data structures [3–5]. But processing LPM on them requires multiple memory accesses for one lookup. Though optimized by many techniques [6, 7], their performance are still difficult to meet today's link speed.

J. Su et al. (Eds.): ICoC 2013, CCIS 401, pp. 134–144, 2013.

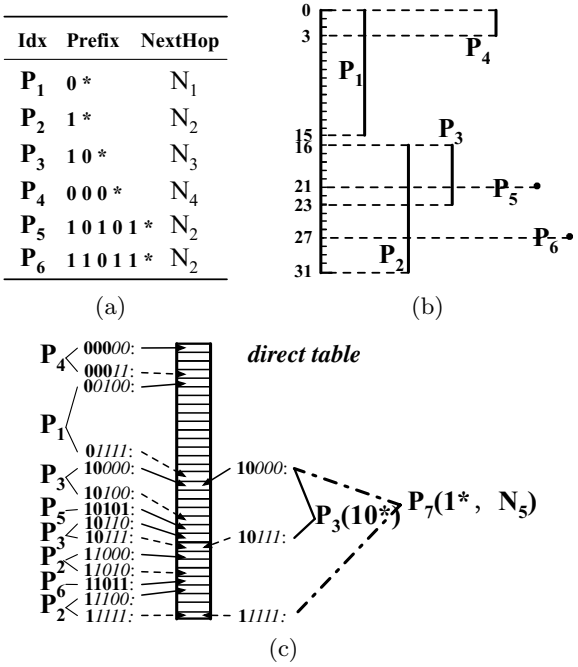


Fig. 1. (a) FIB. (b) A sample of GALE's direct table. Deleting $P_3(10*)$ from it and then inserting $P_7(1*, N_5)$ into it must be processed in order. (c) Transforming prefixes into segments. To simplify examples, we suppose the focused maximum prefix length is 5 (which is 24 in practice).

Fortunately, some GPU-based software routers [8–10] have been proposed to provide both high throughput and high scalability. However, with major focus on the entire framework design, they all treat the routing table as static and thus fail in dealing with update overhead. In view of this, J. Zhao et al. [11] presented a GPU-Accelerated Lookup Engine (GALE), which provides both fast lookup and the solution to route updates. Due to its update mechanism, a update request, after being used to update a trie, is mapped into a range of unit modifications toward a direct table stored on the GPU. However, different updates may be mapped to the same unit. So, even with a careful length checking, breaking their order may also lead to incorrect updates (as shown in Fig. 1(c)). Accordingly, in GALE, not all updates can be processed in parallel, which obstructs it to benefit enough from GPU's parallelism.

1.2 Our Approach and Key Contributions

In this paper, we first implement the TBL24 of DIR-24-8 scheme [12] on the GPU, just like GALE, to enable O(1) lookup. Then, instead of using a trie, we present a novel tree-like structure, Threaded Segment Tree (TST), to process

off-line updates on the CPU, achieving more efficient on-line updates toward the TBL24 on the GPU. We call the proposed IP lookup engine TSTT, for its two most important components are TST and TBL24 respectively.

We make three key contributions in this paper. Firstly, after transforming the FIB into a compact segment tree, we design a special leaf-pushing technique, to divide all prefixes into several non-intersecting segments. Besides, we present a series of algorithms to thread necessary segments during off-line updates, ensuring threaded segments cover all update information without any intersecting. As a result, on-line updates can be processed completely in parallel.

The rest of this paper is organized as follows. Section 2 presents the system architecture of our proposed scheme. And the details of TST are proposed in Sect. 3. Section 4 discusses the performance evaluation experiments. At last, a short conclusion are given in Sect. 5.

2 System Architecture

Since there are many novel designs toward the entire framework of GPU-based software routers [8, 9], we only focus on the IP lookup engine, which can be deployed into such routers as an additional plug-in [11]. With the help of an optimized packet I/O engine [8], which manages packet queues and extracts pending packets' destination addresses for lookup, we pay our major attention to the GPU-based accelerator for table lookup and update.

As shown in Fig. 2, our system architecture is based on Compute Unified Device Architecture (CUDA), in which, all program codes are divided into two cooperative parts, the *Host* and the *Device*, executed respectively on the CPU and the GPU. In the *Device*, the TBL24, which stores all prefixes no longer than 24 (the rest are stored in a small TCAM, which is not shown in Fig.2), provides O(1) lookup. In the *Host*, as managed by a control thread, a group of working threads process lookup and update requests, by utilizing the computing resources of both the CPU and the GPU.

Processing route updates on TBL24 always needs help from additional structures [11]. In TSTT, TST plays such a role. Actually, in our case, all prefixes no longer than 24 are transformed into segments, each of which corresponds to a range of units of the TBL24. Then, a TST is constructed on bidis of these segments, which is stored in the host memory (on the CPU) for off-line updates. When a route update arrives, it is first used to update this TST, producing several unit modifications toward the TBL24. During such off-line updates on the CPU, all produced unit modifications are collected, and are then processed on the GPU to update the TBL24 for on-line updates.

As the GPU works in Single Instructions with Multiple Threads (SIMT), batch processing is required for improving performance. As long as all unit modifications produced during off-line updates are kept independently with each other, all of them can be processed in parallel on the GPU. In this case, we can

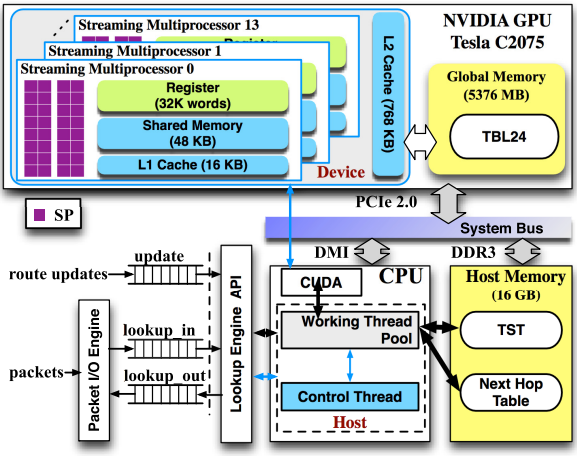


Fig. 2. TSTT's architecture based on CUDA

collect all produced unit modifications in batchs, and send them to the GPU batch by batch, by utilizing multiple streams[1].

Besides, to avoid storing complicated next hop information (such as multi-next-hop [13]) on the GPU, we store their index in the TST and the TBL24 instead, with entire next hop information stored in a separated Next Hop Table on the CPU.

3 Threaded Segment Tree (TST)

3.1 Segment Tree and Prefix Transforming

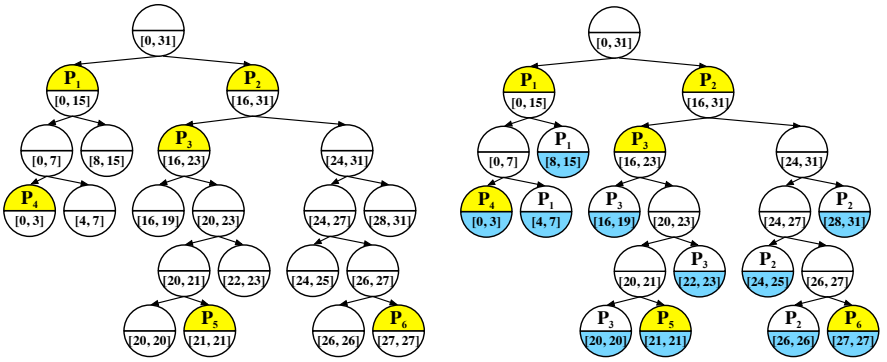
Segment Tree is a special binary search tree that supports dynamic lookup and update. Some of its extensions have been already used in IP lookup [14] and packet classifying [15]. They all transform prefixes into segments in a straightforward way. As shown in Fig. 1(b), the maximum prefix length is supposed to be max, if the length and value[2] for a prefix are denoted as len and pre respectively, then, its corresponding segment can be calculated as $[pre \times 2^{max-len}, pre \times 2^{max-len} + 2^{max-len} - 1]$.

3.2 Building Leaf-pushed Segment Tree

After transforming all prefixes into segments (as shown in Fig. 1(b)), a segment tree can be easily constructed. But in our case, the segment tree is only used to produce segments that represents unit modifications toward TBL24 during

[1] In CUDA, a steam is a sequence of operations executed in order.

[2] For a prefix formatted as $a.b.c.d/len$, its prefix length is len and its prefix value is $(a \times 2^{24} + b \times 2^{16} + c \times 2^8 + d) >> (32 - len)$.



(a) Before leaf pushing. (b) After leaf pushing.

Fig. 3. Segment tree built on the FIB shown in Fig. 1(a)

Algorithm 1. Leaf Pushing

Input: $curNode, nextHop$
1 **if** $curNode = NULL$ ***or*** $curNode.isPrefixSeg$ **then**
2 | **return**;
3 **end**
4 LeafPushing ($curNode.leftChild, nextHop$);
5 **if** $curNode.isLeaf$ **then**
6 | ModifyNode ($curNode, nextHop$);
7 **end**
8 LeafPushing ($curNode.rightChild, nextHop$);

off-line updates. In order to generate as less segments as possible, we build a compact segment tree (as shown in Fig. 3), in which a segment should not be broken unless necessary. Originally, a prefix corresponds to only one segment (let's call it *prefix segment*), whose value is set as the entry index of this prefix. So, each update request needs only modify just one prefix segment, enabling update process be simple and efficient.

However, since two prefix segments may intersect with each other, on-line updates produced by them may toward the same unit, making it unavailable to process them in parallel. To address this issue, we introduce a special leaf pushing technique[3] to push all prefix segments' values into leaf segments (shown in Fig. 3(b)). Unlike the traditional leaf-pushing algorithm [6], there are two important special rules for ours: **1)** All prefix segments still reserve their own values after leaf pushing. **2)** Each value is pushed from some prefix segment down to all possible leaf segments, until reaching another prefix segment. This algorithm is described in Algorithm 1.

[3] Leaf Pushing is a widely used technique for Trie-based IP lookup.

After leaf pushing, each leaf segment contains an index number that corresponds to a route entry, so as all prefix segments. On the other hand, a segment can be represented by several leaf segments. In another word, instead of producing unit modifications by all prefix segments, we can use a group of leaf segments to do the same thing. Since any two leaf segments will not intersect with each other, all produced unit modifications in this way can be processed in parallel now.

3.3 Threading Leaf Segments during Off-line Updates

Based on the leaf-pushed segment tree, several leaf segments can be used to produce unit modifications without intersecting. The next problem is to determine which segments are required. In this section, we present a series of algorithms to solve this problem, by threading all necessary leaf segments during off-line updates. Since modifying an existing prefix can be treated as inserting a new prefix, we only discuss how to delete/insert a prefix.

Prefix Deletion. To delete a prefix, we begin with looking up its corresponding segment. If no one matches, nothing needs to do. Otherwise, the value of the matched segment should be modified to the value of the nearest segment along the path from it to the tree root (if no one found, use the default). Then, its new value should be pushed down.

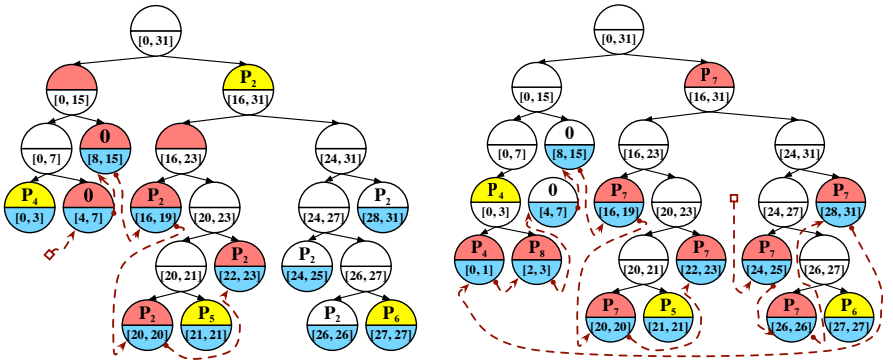
For example, as shown in Fig .4(a), to delete $P_3(10*)$, its corresponding prefix segment $[16, 23 : P_3]$ should be modified to P_2 (the nearest segment along the path from it to the root is $16, 31 : P_2$), which is then pushed down to three leaf segments: $[16, 19]$, $[20, 20]$ and $[22, 23]$. While for deleting $P_1(0*)$, the value of its corresponding segment $0, 15 : P_1$ is modified to the default (e.g. 0). And the default is then pushed down to two leaf segments: $[4, 17]$ and $[8, 15]$. During the deleting process, we use a doubly linked list to thread all leaf segments modified. This algorithm is described in Algorithm 2.

Prefix Insertion. Inserting a prefix is namely inserting a segment. If this segment is already exist, its value should be set as the inserting prefix's entry index. Then, this new value should be pushed down. During this process, all leaf segments modified should be threaded in the double linked list. If the inserting prefix segment is not exist, a series of leaf segments should be broken into new segments until the inserting segment has been produced. During this process, all generated leaf segments should be threaded. On the other hand, if a threaded segment has been broken, it must be removed from the list.

For example, as shown in Fig. 4(b), to insert $P_7(1*, N_5)$, since its prefix segment $[16, 31]$ is already exist, the value of this segment is set as P_7, which is then pushed down. While for inserting $P_8(0001*)$, $[0, 3 : P_4]$ is broken into $[0, 1 : P_4]$ and $[2, 3 : P_8]$, which should be threaded in turn. This algorithm is described in Algorithm 3.

Algorithm 2. Delete Segment

Input: $curNode, delSeg, nextHop$
1 **if** $curNode.seg = delSeg$ ***and*** $nextHop \neq DEFAULT$ **then**
2 $curNode.seg.value = DEFAULT$; $curNode.isPrefixSeg = FALSE$;
3 LeafPushing $(curNode, nextHop)$;
4 **end**
5 **else**
6 **if** $curNode.isLeaf$ **then**
7 **return**;
8 **end**
9 **if** $curNode.isPrefixSeg$ **then**
10 $nextHop = curNode.seg.value$;
11 **end**
12 $mid = (curNode.seg.low + curNode.seg.high)/2$;
13 **if** $mid < delSeg.low$ **then**
14 DeleteSegment $(curNode.rightChild, delSeg, nextHop)$;
15 **end**
16 **if** $mid \geq delSeg.high$ **then**
17 DeleteSegment $(curNode.leftChild, delSeg, nextHop)$;
18 **end**
19 **end**



(a) Delete $P_3(10*)$ and $P_1(0*)$. (b) Insert $P_7(1*, N_5)$ and $P_8(0001*, N_2)$.

Fig. 4. Threading leaf segments during off-line updates. Only one direction's connection of the double linked list is shown.

4 Experimental Evaluation

In this section, based on four real-world routing data sets collected from the RIPE RIS Project [16] (shown in Table.1), we conduct a group of experiments to evaluate TSTT's performance, and demonstrate its superiorities in comparison with GALE. The experimental system is set up on a server with an Intel CPU (Xeon E5-2630, 2.30GHz, 6Cores) and an NVIDIA GPU (Tesla C2075, 1.15 GHz,

Algorithm 3. Insert Segment

```
   Input: curNode, insSeg
 1 if curNode.seg = insSeg then
 2 |   curNode.seg.value = DEFAULT; curNode.isPrefixSeg = FALSE;
 3 |   LeafPushing (curNode, nextHop);
 4 end
 5 else
 6 |   if curNode.isLeaf then
 7 |   |   BreakSegment (curNode);
 8 |   end
 9 |   mid = (curNode.seg.low + curNode.seg.high)/2;
10 |   if mid < insSeg.low then
11 |   |   InsertSegment (curNode.rightChild, insSeg);
12 |   end
13 |   if mid ≥ insSeg.high then
14 |   |   InsertSegment (curNode.leftChild, insSeg);
15 |   end
16 end
```

448 Cores) on the basis of CUDA 5.0. We measure all concerned metrics through the *NVIDIA Visual Profiler* [17].

Table 1. Routing Data Sets (Collected at Jan. 1, 2013)

Data Set	Location	Total Prefixes	Total Updates
rrc11	New York (NY), USA	442,176	1,177,425
rrc12	Frankfurt, Germany	450,752	4,049,260
rrc13	Moscow, Russia	456,580	2,025,239
rrc14	Palo Alto, USA	446,160	1,388,217

4.1 Memory Accesses Required for On-line Updates

To evaluate the performance of route update, we replay a week's update traces of $rrc12$ and a whole day's update traces for all tables. After processing off-line updates in TST, we get a list of threaded segments, each of which represents several memory writes toward a range of units of the TBL24 on the GPU. While for GALE, due to the length map checking, it requires both memory reads and writes on the GPU to finish on-line updates. We evaluate the performance of on-line updates by measuring the average produced global memory accesses per update on the GPU.

As depicted in Fig. 5(a), the average produced memory accesses in TSTT are far less than that in GALE in all cases. In fact, TSTT achieves a reduction by 92.5% $\sim$ 98.1%, which is still 82.5% $\sim$ 97.3% even ignoring memory reads in

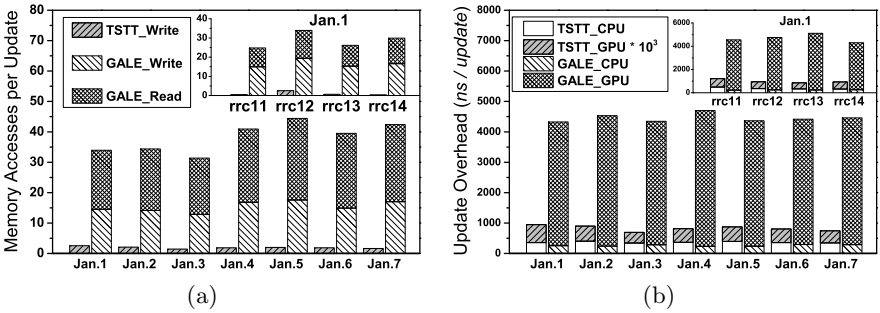


Fig. 5. (a) average memory accesses produced per update on the GPU. (b) overall update overhead on both the CPU and the GPU per update.

GALE. Such a significant improvement benefits from our mechanism essentially. Actually, in TSTT, all produced unit modifications cover all update information without any redundancy. So, excluding any of them must lose some update information. In another word, the number of memory modifications for on-line updates are minimized by TSTT.

4.2 Overall Update Overhead

Then, we take into account the overall update overhead (in time cost) for both off-line updates (on the CPU) and on-line updates (on the GPU). Since TSTT's on-line update overhead are too small, we **times it by** 10^3.

As shown in Fig. 5(b), the average update overhead per update of TSTT's on-line updates is only 0.72 ns in the worst case, achieving a speedup than GALE by a factor over 5000. Such a fast speed benefits from that the number of produced unit modifications in TSTT are minimized, and all of them can be processed in parallel on the GPU.

However, processing off-line updates in TSTT costs more time than that in GALE by a factor of $2.3 \sim 3.4$. That's because some additional operations are required in TSTT, such as leaf pushing and list management. Even though, in comparison with GALE, TSTT's overall update overhead is still reduced by $89.6\% \sim 93.5\%$, which demonstrates clearly that TSTT's update mechanism is more efficient.

4.3 Comprehensive Performance

In order to evaluate the comprehensive performance, we process a $16M$ generated lookup requests in TSTT and GALE respectively, with update frequency increasing to $100,000$ $updates/s$. Then, we measure the throughputs with Million Lookups Per Second (MLPS) in all cases.

As presented in Fig. 6, without any updates, TSTT provides the same throughput (539 $MLPS$) as GALE. That's because their lookup approaches are

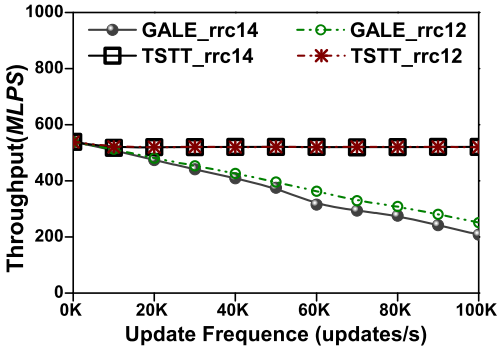


Fig. 6. Comprehensive performance on rrc12 and rrc14

all based on the DIR-24-8 scheme. However, as update frequency is increasing, TSTT's superiority becomes more and more significant. Actually, in GALE, the throughput decreases by 53.4% and 61.4% on $rrc12$ and $rrc14$ respectively. But such descents in TSTT are even below 0.4%. In another word, with the help of an efficient update mechanism, TSTT enables more stable throughput under frequent updates than GALE.

5 Conclusion

In this paper, we have proposed an efficient update mechanism for a GPU-accelerated IP lookup engine. By deploying the TBL24 of DIR-24-8 onto GPU's global memory, our proposed engine, TSTT, enables O(1) lookup. Moreover, we presented an novel tree-like structure, Threaded Segment Tree (TST), to help update the TBL24 on the GPU. Actually, by threading necessary leaf segments during off-line updates, the number of unit modifications for on-line updates are minimized, and all of them can be processed completely in parallel. According to the experiment results, using our mechanism, the average required memory accesses for on-line updates and the overall update overhead on both the CPU and the GPU in average are reduced by at least 82.5% and 89.6% respectively. What's more, due to the proposed update mechanism, the throughput in TSTT has been proved more stable. Actually, it only decreased by at most 0.9% even if update frequency increases to $100,000/s$.

Acknowledgment. This work is supported by the National Basic Research Program of China (973) under Grant 2012CB315805, and the National Science Foundation of China under Grant 61173167.

References

1. Jiang, W., Wang, Q., Prasanna, V.K.: Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit ip lookup. In: IEEE INFOCOM 2008 The 27th Conference on Computer Communications, pp. 1786–1794. IEEE (2008)

2. Le, H., Jiang, W., Prasanna, V.K.: Memory-efficient IPv4/v6 lookup on FPGAs using distance-bounded path compression. In: 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 242–249. IEEE (2011)
3. Huang, K., Xie, G., Li, Y., Liu, A.X.: Offset addressing approach to memory-efficient IP address lookup. In: 2011 Proceedings IEEE INFOCOM, pp. 306–310. IEEE (2011)
4. Eatherton, W., Varghese, G., Dittia, Z.: Tree bitmap: hardware/software IP lookups with incremental updates. ACM SIGCOMM Computer Communication Review 34(2), 97–122 (2004)
5. Song, H., Kodialam, M., Hao, F., Lakshman, T.: Scalable IP lookups using shape graphs. In: 17th IEEE International Conference on Network Protocols, ICNP 2009, pp. 73–82. IEEE (2009)
6. Srinivasan, V., Varghese, G.: Fast IP address lookups using controlled prefix expansion. ACM Transactions on Computer Systems (TOCS) 17(1), 1–40 (1999)
7. Bando, M., Chao, H.J.: Flashtrie: hash-based prefix-compressed trie for ip route lookup beyond 100gbps. In: 2010 Proceedings IEEE INFOCOM, pp. 1–9. IEEE (2010)
8. Han, S., Jang, K., Park, K., Moon, S.: Packetshader: A GPU-accelerated software router. ACM SIGCOMM Computer Communication Review 40(4), 195–206 (2010)
9. Zhu, Y., Deng, Y., Chen, Y.: Hermes: an integrated CPU/GPU microarchitecture for IP routing. In: Proceedings of the 48th Design Automation Conference, pp. 1044–1049. ACM (2011)
10. Mu, S., Zhang, X., Zhang, N., Lu, J., Deng, Y.S., Zhang, S.: IP routing processing with graphic processors. In: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, pp. 93–98 (2010)
11. Zhao, J., Zhang, X., Wang, X., Deng, Y., Fu, X.: Exploiting graphics processors for high-performance IP lookup in software routers. In: 2011 Proceedings IEEE INFOCOM, pp. 301–305. IEEE (2011)
12. Gupta, P., Lin, S., McKeown, N.: Routing lookups in hardware at memory access speeds. In: Proceedings of the INFOCOM 1998, Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 3, pp. 1240–1247. IEEE (1998)
13. Wenping, C., Xingming, Z., Jianhui, Z., Bin, W.: Research on multi next hop rip. In: International Forum on Information Technology and Applications, IFITA 2009, vol. 1, pp. 16–19. IEEE (2009)
14. Chang, Y.K., Lin, Y.C., Su, C.C.: Dynamic multiway segment tree for ip lookups and the fast pipelined search engine. IEEE Transactions on Computers 59(4), 492–506 (2010)
15. Su, C.F.: High-speed packet classification using segment tree. In: IEEE Global Telecommunications Conference, GLOBECOM 2000, pp. 582–586. IEEE (2000)
16. RIPE network coordination centre, `http://www.ripe.net`
17. NVIDIA Corporation: NVIDIA CUDA Profiler User Guide, Version 5.0 (October 2012)