This paper was presented as part of the Mini-Conference at IEEE INFOCOM 2011

Exploiting Graphics Processors for High-performance IP Lookup in Software Routers

Jin Zhao, Xinya Zhang, Xin Wang
School of Computer Science
Fudan University
Shanghai, China
Email:{jzhao,06300720198,xinw}@fudan.edu.cn

Yangdong Deng
Department of Electronic Engineering
Tsinghua University
Beijing, China
Email: dengyd@tsinghua.edu.cn

Xiaoming Fu
Institute of Computer Science
University of Goettingen
Goettingen, Germany
Email:fu@cs.uni-goettingen.de

Abstract—**As the physical link speeds grow and the size of routing table continues to increase, IP address lookup has been a challenging problem at routers. There have been growing demands in achieving high-performance IP lookup cost-effectively. Existing approaches typically resort to specialized hardwares, such as TCAM. While these approaches can take advantage of hardware parallelism to achieve high-performance IP lookup, they also have the disadvantage of high cost. This paper investigates a new way to build a cost-effective IP lookup scheme using graphics processor units (GPU). Our contribution here is to design a practical architecture for high-performance IP lookup engine with GPU, and to develop efficient algorithms for routing prefix update operations such as deletion, insertion, and modification. Leveraging GPU's many-core parallelism, the proposed schemes addressed the challenges in designing IP lookup at GPU-based software routers. Our experimental results on real-world route traces show promising gains in IP lookup and update operations.**

I. Introduction

One of the core functionalities of a router is to determine the next-hop port by comparing the incoming IP address against a set of stored prefixes in routing table. Currently, Classless Inter-Domain Routing (CIDR) is widely deployed to allocate the IPv4 address space efficiently [1]. With CIDR, the prefix lengths may vary from 1 to 32. The use of CIDR complicates the lookup process, requiring a lookup engine to search variable-length IP prefixes in order to find the longest prefix matching (LPM) for the destination address.

Due to the continuous growth in network link speeds and routing table size, IP lookup engines face enormous performance challenges. First, the lookup engines have to be able to answer an ever-increasing number of lookup queries over a few hundred thousand routing prefixes. As an evidence, it was reported that the number of IP prefixes in today's Internet routing tables exceeds 300K and is still growing [2]. At the same time, IP lookup engines also have to accommodate demands for other routing table operations such as addition/deletion of prefixes, and the modification of next-hop for existing prefixes. To keep pace with the Internet traffic and routing table growth, researchers have been continuously exploring the approaches to high-performance IP lookup engine design.

Generally, existing solutions can be classified into hardware-based and algorithmic-based approaches. Today's high-end routers typically use specialized hardwares, such as ternary content addressable memories (TCAM). However, TCAM suffers from high cost due to high circuit density. It also offers little flexibility to adapt to new addressing and routing protocols. Gupta et al. [3] presented a hardware-based lookup scheme which require a maximum of two memory accesses for each lookup. Currently, there have been many efforts in scaling software routers using off-the-shelf, general-purpose PC [4], [5]. Algorithmic-based approaches usually employ special data structures to perform longest prefix matching (LPM). The most commonly used data structure is trie [6] or its extensions, e.g. Lulea [7], Tree Bitmap [8]. Upon a lookup request, the trie is traversed from root to leaves. At each level one or more bits of the IP prefix are used to make branching decisions. The main performance bottleneck for trie-based IP lookups is the memory access penalty since LPM will involve additional memory accesses when traversing down the trie hierarchy.

In this paper, we investigate a new approach to building cost-effective and high-performance IP lookup engines and the corresponding routing table update schemes using Graphics Processing Units (GPUs). Building an efficient IP lookup engine on GPUs, however, is a non-trivial task due to the challenging data-parallel programming model provided by the GPU, and also due to the significant demand in performance. We present *GALE*, a GPU-Accelerated Lookup Engine, to enable high-performance IP lookup in software routers. We leverage the Compute Unified Device Architecture (CUDA) [9] programming model to enable parallel IP lookup on the many-core GPU. Besides the introduced parallelism in IP lookup, we also adopt an IP lookup scheme that could achieve O(1) time complexity for each IP lookup. The key innovations include: (1) IP addresses are directly translated into memory addresses using a large *direct table* on GPU memory that stores all possible routing prefixes. Only one direct memory access is required to locate the next-hop information. This scheme is thus scalable to very high line speeds with small computing overhead and memory access latency. (2) Though efficient for lookup, the large routing table makes routing updates, which happen fairly frequently in core routers, much more time-consuming and complex. Multiple prefixes need to be modified when only a single original prefix is inserted or deleted. To address this, we also map the route-update operations to CUDA in order to exploit GPU's vast processing

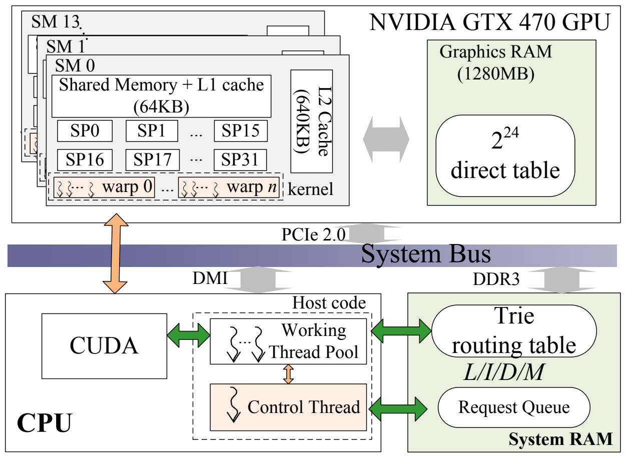

Fig. 1. GPU-based IP lookup engine

power.

With the experiments on NVIDIA GeForce GTX 470, featuring 448 computing cores, we have made the following new observations. First, *GALE*, can achieve IP lookup throughput at about 250 $million/s$, which outperforms trie-based lookup on CPU by a factor of 5 to 6. Second, the update throughput of *GALE* reaches up to 20,000 updates per second, which is far faster than the Internet's regular route-update frequency (say, several hundreds of packets per second).

It is noteworthy that *GALE* only focus on IP lookups and routing update operations, instead of the entire functionality set of software routers. Recently, there have been works in accelerating software router using GPUs [10], [11]. With their focuses on providing an entire framework for software routers, these previous works, however, assume that the routing table is static, and fail in addressing the routing table update overhead. *GALE* could be incorporated into existing software routers as an alternative plug-in with readily available off-the-shelf GPU and its drivers. Further, for software routers, GPUs, with fully programmability, is far superior to servers with multi-core GPU in term of the price/performance ratio.

The rest of this paper is structured as follows. In section II, we then outline our design architecture and then provide algorithms and implementations for major operations involved in IP lookup engines.In section III, we study the performance of the proposed schemes, focusing on issues for high-performance IP lookup. Finally, section IV provides concluding remarks.

II. GPU-ACCELERATED IP LOOKUP ARCHITECTURE

A. GPU-Accelerated Lookup Engine

We now describe the proposed GPU-accelerated IP lookup engine, *GALE*, as shown in Fig. 1. The main motivation of *GALE* is to provide high-performance IP lookup. *GALE* also considers routing update operations: modification (M), insertion (I) and deletion(D). The rationale behind *GALE*'s design lies in the observations that the routing prefix lookup request is stateless. Therefore, a batch of independent lookup requests can be processed in parallel. GPU can be leveraged to speed up routing lookup by using already available CUDA platform.

Besides the traditional trie-based routing table in system memory, *GALE* also stores the next-hop information for all the possible IP prefixes in a fixed-size large memory block on GPU, which is referred to as *direct table*. The reason for maintaining two routing tables is that there is an inherent tradeoff between efficiency and overhead when considering a direct table versus trie for particular functionality. The direct table is used for fast IP lookup while the trie is kept for route-update. Upon route-update requests, the corresponding prefix entries in both the direct table and the trie will updated. The request queue is used to aggregate the lookup and update requests received from different network interfaces. We assume that the request are already classified according to their operation type. GALE employs a control thread and a pool of working threads to implement the lookup and update requests. The processing of requests is divided into the following steps:

1) Control thread reads a group of requests of the same type from the request queue classified by L, I, M, D operations.
2) Control thread activates one idle working thread in the thread pool to process the group of requests.
3) Working threads invoke corresponding GPU and CPU code to perform lookup and update operations.

Note that the operations on trie is done by CPU code while the the lookup operations and update operations on direct table, are mapped to GPU using CUDA.

With GALE, the parallelism in request processing is exploited in two aspects. First, as the CPU we use also has multi-cores, the working threads can be potentially executed simultaneously. Meanwhile, each working thread will instruct the GPU to launch one kernel. As GTX 470 supports multiple kernels on different SMs, the different request groups thus can be scheduled in parallel. Second, inside one kernel, the group of requests are of the same type. Parallelism is thus achieved by mapping these requests to the CUDA threads, which in turn are scheduled in parallel to different SPs on GPU.

B. Lookup

Observing that IP-lookup operations are far frequent than route-update operations, we seek to trade some performance in update for better lookup throughput. We explore an IP lookup solution with $O(1)$ memory access and computational complexity.

GALE seeks to store the next-hop information for all the possible IP prefixes in one direct table. In this way, each IP address has exactly one-to-one mapping relation with entries in the direct table. The entries in direct table is derived from the trie structure. How to update entries in the direct table will be discussed in the next subsection. Direct table in turn enables simple translation from IP address to the memory address of matched prefix entry. The longest-prefix matching is thus transformed into an exact match. The IPv4 addresses have a length of 32 bits. Using CIDR, the direct table will scale up to $2^{32} = 4G$ possible IP prefix entries. Such an storage overhead is beyond most GPUs' memory capacities.

Fortunately, real-world traces [2] reveal that the IP prefix length distribution tends to aggregate. About 40% of the IP prefixes have the length of 24, and over 99% of the IP prefixes are less than or equal to 24. IP prefixes longer than 24 are rare. We leverage this fact and propose a solution that storing all the possible prefixes with length less than 24 in one direct table. In consequence, only $2^{24} = 16M$ prefix entries are required. We may use a separate long table to store the prefixes that are longer than 24 bits. However, since the majority of IP prefixes are not longer than 24 bits, we just omitted the further discussion on the long table.

With this direct table that stores all possible IP prefixes, IP lookup is fairly simple in terms of computational complexity since only one address translation is enough to find the corresponding routing entry. Meanwhile, only one memory access is required to fetch the next-hop destination.

The direct table $dtable$ is stored in GPU memory starting from memory address dt_base. Upon a lookup for the incoming IP address a.b.c.d, the leftmost significant 24 bits a.b.c are used as index to its corresponding next-hop information in the direct table. The lookup algorithm can be implemented very efficiently as shown below. Note that the dotted IP address notation a.b.c.d is regarded as an integer ($a \times 2^{24} + b \times 2^{16} + c \times 2^{8} + d$) in the algorithm.

Algorithm 1: IP lookup with direct table

```
Input: $ip\_addr$
Output: $nexthop$
$index$ = the leftmost 24 bits of $ip\_addr$;
$nexthop = dtable[index]$;
return $nexthop$;
```

With NVIDIA's CUDA, *GALE* can simultaneously execute a group of IP lookup requests on GPU's many-core architecture. By matching the incoming IP lookup requests in parallel on GPU, the overall throughput of IP lookup can be significantly improved.

C. Routing table update

When a route-update is applied, it generally causes the insertion, modification or removal of existing prefix entries in the routing table. In GALE, route-update will involve operations to both the direct table and the trie structure. Insertion a prefix entry in trie will involve traversing down the trie from root and finding the right place to add a new entry node. Similarly, modification and removal will also involve trie-traversal to modify or delete an existing entry. Such update operations on the trie-based routing table can be implemented effectively using existing available approaches.

As direct table has a fixed size, we actually need not allocate or release entry space upon route-update operations. Insertion, modification, and removal operations can be essentially generalized to modification to one or more prefix entries on the direct table.

Though efficient for lookup, the direct table added complexity to route-updates. The complexity comes from the correlations between the routing entries when using longest prefix matching.

In order to address the problem, we seek to use GPU's massively parallel computing cores to implement route-updates. To alleviate the trie-traversal operations in GPU, we introduce a *length table* to denote the prefix length. Length table, $ltable$, is stored in GPU memory starting from location lt_based and has the same size as direct table. The values in length table indicate the prefix lengths of the prefix entries at the corresponding offset in direct table.

We now summarize the key methods we selected to apply route-updates.

1) Insertion and Modification: Insertion and modification are essentially the same operation for direct table. Upon a new route prefix ($ip\_prefix/len, nexthop$),*GALE* will write the new next-hop information to the corresponding IP prefixes. The update range depends on the length of this new prefix. However, whether to update the nexthop information depends on if the new prefix length is larger than the entry's existing prefix length.

We describe the prefix modification algorithm as below.

Algorithm 2: Prefix modification in direct table

```
Input: $ip\_prefix, len, nexthop$
$start\_range$ = the leftmost 24 bits of $ip\_prefix$;
$end\_range = 2^{24-len} + start\_range - 1$;
for
$index = start\_range; index \leq end\_range; index++$
do
    if $len \geq ltable[index]$ then
        $dtable[index] = nexthop$;
        $ltable[index] = len$ ;
    end
end
```

Clearly, the for loop can be executed in parallel using different threads.

2) Deletion: The deletion operation is similar to modification except that during updating the range of entries, the next-hop information is modified to the parent node's nexthop information in the trie. Therefore, deleting an entry is as follows: replacing the $nexthop$ and prefix length of the updated entry with the parent's $nexthop$ and prefix length. The parent node is obtained from the trie-traversal during deleting the entry node in trie by CPU.

Since the updates on different prefix entries are independent(for loop in the algorithm), the update operations on direct table now can be mapped to different parallel threads.

Algorithm 3: Prefix deletion in direct table

```
Input: $ip\_prefix, len, parent$
$start\_range$ = the leftmost 24 bits of $ip\_prefix$;
$end\_range = 2^{24-len} + start\_range - 1$;
for
$index = start\_range; index \leq end\_range; index++$
do
    if $len == ltable[index]$ then
        $dtable[index] = dtable[parent]$;
        $ltable[index] = ltable[parent]$;
    end
end
```

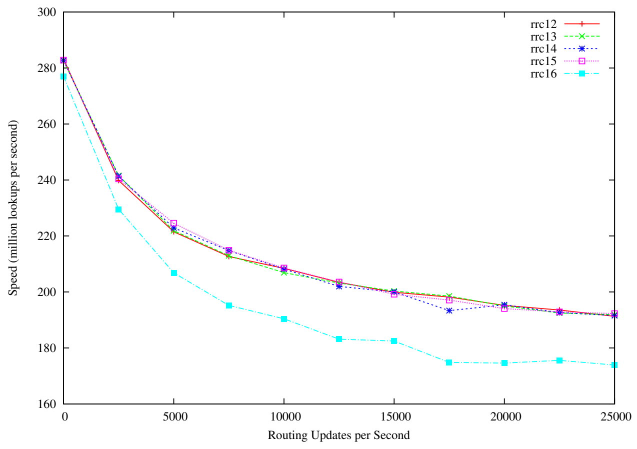

Fig. 2. IP lookup performance on 5 routing tables with high route-update frequency.

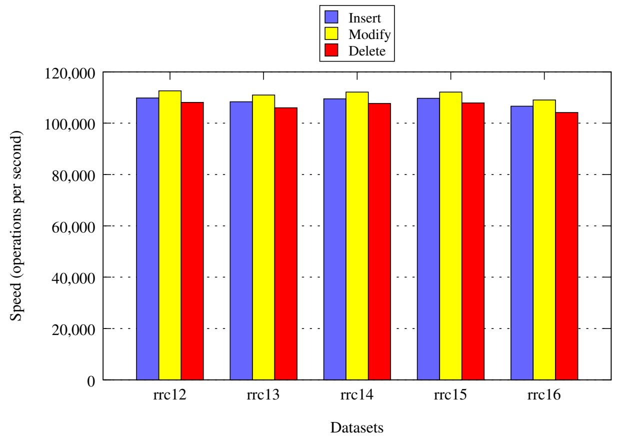

Fig. 3. Insert, Modify, and Delete speed on rrc12, rrc13, rrc14, rrc15 and rrc16.

III. Experiments

A. Experiment setup

The routing dataset we used in the experiments are from FUNET [12] and RIS [13]. Our evaluation of *GALE* is based on its implementation in desktop PC, which replays lookup and update operations on the routing tables captured by real-world route traces. This gives a more realistic routing table as compared to a randomly generated table.

All experiments are run on a desktop PC equipped with a 2.66 GHz Intel Core i5 750 CPU (with 4 cores), 4GB of DDR3 SDRAM, Gigabyte GA-P55-USB3L motherboard, and an NVIDIA GeForce GTX 470 graphics card with 1280 MB of global memory and 448 stream processors. As we are concerned with the cost-effectiveness of *GALE*, we also report here that the whole hardwares on the general-purpose PC used in our experiments cost US$ 1,122, in which the NVIDIA GTX 470 GPU contributed $428. So far as software is concerned, we used a 64-bit x86 Fedora 13 Linux distribution with unmodified kernel 2.6.33. The CUDA SDK version is 3.1 downloaded from NIVIDA's official website.

Unless otherwise specified, throughout the experiments we use the following default configuration:

- With 8 working threads.
- Every CUDA thread (running on GPU) takes just one lookup request, or updates one entry while performing update tasks.
- Every CUDA grid contains 4×4 thread blocks, and every block contains 256×1 threads.

B. Performance

1) IP lookup performance: To understand the baseline performance, we first examine the IP lookup performance of *GALE*. The routing tables are from http://data.ris.ripe.net/. We choose five sets: rrc12, rrc13, rrc14, rrc15 and rrc16 for experiments. The routing tables traced by BGP update message dump on June 1 2010 from 8:00 a.m. to 8:00 p.m. are used as routing-update data. Due to the unavailability of public IP request traces associated with their corresponding routing tables, IP request traces are generated randomly from the routing table.

The results shown in Fig. 2 suggest that the IP lookup performance will be hampered by the route-update frequency. Since the update operations will compete GPU's computing resources, the lookup throughput will decrease if route updates frequently. However, the absolute lookup throughput is still very high even with extreme route-update conditions. As can been seen from the figures, the proposed scheme leads to significantly faster lookup. This confirmed our design objecives in designing high-performance IP lookup.

2) Insert, Modify, and Delete speed on large routing tables: The insertion throughput is measured as a total of 2^{24} entries over the time needed to add all the entries to the direct table. Similarly, the modification throughput the deletion throughput are measured as the direct table size over the consumed time to modify or delete the lines one by one in the direct table. The routing tables, namely rrc12, rrc13, rrc14, rrc15 and rrc16, all have a size of over routing entries. Note that routing table size here refers to the number of prefix entries in traditional routing table, or the number of valid nodes in trie-based table.

With respect to route-updates, as observed in Fig. 3, the performances in inserting, modifying, and deleting the routing

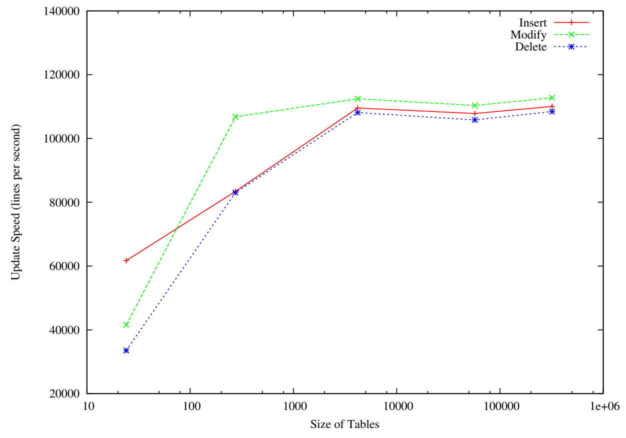

Fig. 4. Insert, Modify, and Delete speed v.s. routing table size

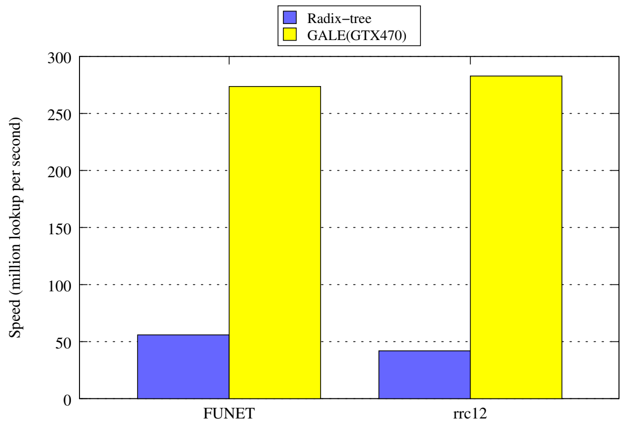

Fig. 5. Comparison between GALE and Radix-tree implementation

entries are generally lower than that of lookup because *GALE* has traded the complexity in update for efficient lookup.

3) Insert, Modify, and Delete speed on routing tables with different sizes: We now migrate the route-update experiments to routing tables with different sizes. The data source is from rrc12 dump in Test 2. We change the routing tables size to 24, 274, 4190, 57080, and 318851 respectively.

We can see in Fig. 3 that with the increase in routing table size (i.e. the number of nodes in trie), the update performance will also increase. This is because when there are more routing prefixes, the dependencies in updating the direct table are less.

The results in Fig. 3 and 4 suggest that the update throughput is also significant accelerated by GPU. Considering that the real-world routing update operations are not so frequent, say, about a few thousand BGP updates per second, the proposed schemes also suffice.

4) Lookup performance comparison between GALE and trie: Furthermore, we also compare the lookup performance between *GALE* and a radix-tree scheme. The radix-tree scheme, as one kind of trie-based implementation, runs the routing algorithm from NetBSD. Clearly, as shown in Fig. 5, GALE is superior than the trie-based scheme by a factor of 5-6 as it takes O(1) complexity to get the lookup result.

IV. Conclusions

It is a challenge to design efficient IP lookup algorithms in software routers that can meet the potentially competing objectives in cost-effectiveness, high-performance lookup and low cost route-update. To address this challenge, the following key innovations has been made in this paper: (1) We present the design and evaluation of a GPU-accelerated IP lookup engine, *GALE*, that exploits the massive parallelism to speedup parallel routing table lookups. (2) Meanwhile, we also propose to use a direct table for efficient IP lookup with small computing overhead and memory access latency. More specifically, it can achieve O(1) computational complexity and only one memory access for each lookup request. (3) Emphasizing on practicality, we finally designed efficient algorithms that can map route-update operations, deletion, insertion, and modification to GPU's parallel architecture. Our performance evaluation of *GALE* is highlighted with the replay of routing update from real-world traces. Experiment results show that, with proper design, there is the potential for significant improvement in lookup throughput. With a much better performance/price ratio, and highly flexible programmability, GPUs are ready to shed light on high-performance routing processing.

Acknowledgment

This work was supported in part by STCSM under grant 08dz150010E, by NSFC under grant 60803119, by 863 program of China under grant 2009AA01A348, and by Shanghai Educational Development Foundation under grant 2007CG07. Xin Wang is the corresponding author.

References

[1] Y. Rekhter and T. Li, "An architecture for ip address allocation with cidr," RFC 1518.

[2] "Bgp routing table analysis report," 2010, http://bgp.potaroo.net/ .

[3] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM '98*, Apr 1998, pp. 1240–1247.

[4] M. Dobrescu, N. Egi, K. Argyraki, B. gon Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009, pp. 15–28.

[5] K. Argyraki, S. A. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedveschi, and S. Ratnasamy, "Can software routers scale?" in *PRESTO 2008*, Aug. 2008.

[6] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of ip address lookup algorithms," *IEEE Network*, vol. 15, pp. 8–23, 2001.

[7] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM '97*, Sep. 1997, pp. 3–14.

[8] W. Eatherton, Z. Dittia, Z. Dittia, and G. Varghese, "Tree bitmap : Hardware/software ip lookups with incremental updates," *ACM SIGCOMM Computer Communication Review*, vol. 34, pp. 97–122, 2004.

[9] NVIDIA, "Compute unified device architecture," http://www.nvidia.com/ .

[10] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, "Ip routing processing with graphic processors," in *Design, Automation, and Test in Europe conference (DATE) '10*, Mar 2010.

[11] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *ACM SIGCOMM 2010*, Aug. 2010.

[12] FUNET, http://www.nada.kth.se/ snilsson/ .

[13] RIS, "Routing information service," http://data.ris.ripe.net/ .