Received 27 November 2014; revised 18 May 2015; accepted 11 July 2015.
Date of Publication 22 July 2015; date of current version 8 June 2016.

Digital Object Identifier 10.1109/TETC.2015.2460453

IP Address Lookup by Using GPU

HUNG-MAO CHU, TSUNG-HSIEN LI, AND PI-CHUNG WANG

Department of Computer Science and Engineering, National Chung Hsing University, Taichung 402, Taiwan

CORRESPONDING AUTHOR: P.-C. WANG (pcwang@nchu.edu.tw)

This work was supported by the Ministry of Science and Technology, Taiwan, under Grant NSC100-2628-E-005-007-MY3 and Grant NSC104-2221-E-005-010.

ABSTRACT We present a novel parallel IP address lookup architecture based on graphics processing unit (GPU) via compute unified device architecture (CUDA). Our architecture consists of two functions: 1) host function and 2) device function. The host function is executed by a CPU to construct and update the data structure of IP address lookup executed by the device function in a GPU. Both host and device functions are executed simultaneously to fully utilize computational resources. To shorten the lookup time, a trie-based data structure optimized for CUDA is developed. The trie-based data structure uses multi-bit stride to shorten the trie depth and also improves the efficiency of texture cache in GPUs. The experimental results show that a low-end G92 GPU can achieve a throughput of more than 1.3 billion packets per second for IPv4 routing tables with more than 350K prefixes while a high-end GT200 GPU can further double the performance. By employing dual data structures, the implementation can support several hundred thousand updates per second.

INDEX TERMS IP address lookup, CUDA, longest prefix matching.

I. INTRODUCTION

Explosive Internet traffic has led to a high bandwidth demand in backbone networks. To accommodate the great amount of traffic, optical fibers have been widely deployed to interconnect routers in core networks. These optical fibers usually have high bit rates, e.g. 160 gigabit per second (Gbps) in OC-3072. Achieving wire speed of an OC-3072 link requires to output 2.5 billion packets per second (GPPS) with the smallest 64-byte Ethernet frames. Since the development of CIDR in 1993 [1], each packet must be processed by an IP lookup engine to determine the longest matching routing prefix and extract the next-hop. Due to the increasing size of routing tables, IP lookup has become the performance bottleneck of an Internet router. The packet throughput cannot be fulfilled by legacy software-based IP lookup approaches. The current solutions usually rely on hardware parallelism to achieve high-performance IP lookup.

Ternary content addressable memory (TCAM) is a specialized memory which can perform parallelized comparisons. It can achieve high-speed packet forwarding by accomplishing one address lookup within one single TCAM access. For a TCAM with 4-ns access latency, the forwarding rate can be up to merely 250 million packets per second (MPPS) [2]. However, the forwarding rate could be degraded due to the slow update procedure for maintaining the order of TCAM entries [3]. TCAM also suffers from high power consumption and cost [4]–[6]. These obstacles also keep TCAM from shortening access latency. Network processor is another specialized hardware optimized for processing data packets. It provides a higher degree of hardware parallelism than a general purpose processor for executing multiple threads to hide memory access latency [7]. Field-programmable gate array (FPGA) is another popular hardware for IP packet forwarding [8]. It provides hardware programmability with the flexibility of employing and configuring different components by using hardware description language (HDL). To our knowledge, no existing research based on network processors and FPGAs can achieve a throughput of more than one GPPS.

GPU is a special-purpose electronic circuit to accelerate the creation of images for output to a display. In 2007, compute unified device architecture (CUDA), a programmable GPU platform developed by NVIDIA, is designed for general-purpose parallel computing. CUDA defines parallel thread execution (PTX) and instruction set architecture (ISA) and all processing units of CUDA concurrently execute the same fragments of a program. The number of processors in CUDA can be easily extended for a better performance. The programmable GPU is known as general-purpose

computing on graphics processing units (GPGPU) [9]. The CUDA parallel computing platform is composed of software and hardware. Software provides useful programming libraries and toolkits from version 1.0 to 5.5 [10]. In order to improve the GPU computing performance, the hardware version from 1.0 to 3.5 (Kepler) is enhanced by mounting more processors and refining the GPU architecture. A CUDA program consists of host and device functions for CPU and GPU, respectively. Initially, the host function moves the device function and data to a GPU device through a PCI-express (PCIe) bus. After that, the device function is performed with data for a specified purpose. When the device function is completed, the host receives a copy of the result from the GPU device. Both functions can be operated simultaneously, and a host function can also invoke many device functions to be executed on one or more GPUs. CUDA has successfully accelerated computations in many fields, such as medical imaging in magnetic resonance imaging, finding an astronomical body, high-quality video encoding, and so forth [11]. The increase in speed can reach more than one hundred times depending on the degree of parallelism.

In this paper, we propose a novel parallel IP address lookup architecture on GPU via CUDA. In the proposed architecture, the GPU acts as a co-processor to perform IP address lookups. The headers of the incoming packets from network interface cards (NICs) are directly transmitted to the GPU memory to avoid the bandwidth limitation of PCIe bus. We present a trie-based data structure optimized for CUDA. The data structure uses multi-bit stride to shorten the trie depth. Because the data elements nearby a trie root must be accessed in each lookup, the data structure improves the performance of memory accesses by raising spatial locality of texture cache in GPUs. It also supports IPv6 and incremental updates. The host function maintains the data structure and the device function uses the proposed data structure to perform IP address lookups. They can be executed simultaneously to fully utilize computational resources. The IPv4 routing tables used in our experiments have more than 350K IPv4 prefixes. Our experiments based on a low-end GPU show that a throughput of more than 1.3 GPPS can be achieved. The throughput could be further doubled by using a high-end GPU. The results indicate that a CUDA-based IP forwarding engine with the proposed data structure can provide the wire speed of OC3072. Our scheme also has superior performance for IPv6 routing tables. Besides, our proposed architecture can support several hundred thousand updates per second. The previous version of this paper has been presented at the 14th IEEE Conference on High Performance Switching and Routing (HPSR) [12]. The major improvedments of this paper include a feasible architecture of a GPU-based packet forwarding engine and the implementaion considerations. A comprehensive study of the existing research is also provided in this paper.

The remainder of this paper is organized as follows. Section 2 describes the related works. We present the characteristics of CUDA in Section 3. Section 4 introduces the proposed architecture of a CUDA-based packet forwarding engine. In Section 5, the proposed multi-bit trie algorithm for IP address lookup is discussed. Section 6 provides the performance evaluation. Finally, concluding remarks are given in Section 7.

II. RELATED WORKS

The major task of IP lookup schemes performs longest prefix matching (LPM) to determine packet routes. LPM is one of the most important issues in router design since an efficient IP lookup scheme can reduce the packet processing time and speed up the packet forwarding rate. Several common metrics for IP lookup schemes include searching speed, storage requirement, updating time, processing time, scalability, and flexibility. IP lookup schemes can be classified into software- and hardware-based according to their implementations.

Software-based schemes usually do not rely on a particular hardware to provide the flexibility of supporting heterogeneous platforms. Most software-based approaches on IP address lookup focus on reducing memory requirements and lowering the number of memory accesses. The data structures used by an IP address lookup algorithm include multi-bit trie [13], Bloom filters [14], bit-vectors [15], tree bitmap [16], etc. Trie is a common data structure with updatability for IP address lookup. Conventional trie-based data structures may have a large number of empty nodes to waste memory space and increase lookup latency [17]. Lim et al. replace redundant nodes by priority nodes to improve both memory size and lookup speed. When the IP address lookup procedure matches a priority node, the next-hop is determined immediately to stop the lookup procedure [18]. In [19], the approach uses a trie as an auxiliary data structure to create prefix vectors for all leaf nodes with their sub-prefix information. Its IP lookup procedure simply performs binary search on these prefix vectors to achieve good scalability and implementation flexibility.

Hardware-based schemes utilize specialized hardware, including TCAMs, network processors, FPGAs and GPUs, to achieve better performances. A summary of various hardware-based solutions can be found in [20]. Among these specialized hardware, GPUs have been used for packet processing with superior programmability. There are diverse approaches to achieving highspeed LPM using GPUs. Zhao et al. proposed an O(1) IP lookup engine, namely GALE [21], [22]. GALE maintains a trie-based data structure in the host function for updating and expanding all routing prefixes into a 2^{32}-entry array stored in GPU memory. Although recent graphics cards have large memory to accommodate 2^{32} entries, reaching O(1) performance is difficult to implement in practice because of the hierarchical memory architecture of GPU. Memory access time and cache hit rate correlate with each other. Moreover, updating process is another problem since one prefix update may rewrite massive and discontinuous entries. Yao et al. developed a hash-based approach, GPMHIA [23], where 25 hashing systems and a global table are employed for performing LPM. Each hashing system corresponds to one prefix length within 8 to 32.

When a hash collision occurs in one of the 25 hashing systems, the prefix is inserted into the global table to achieve O(1) lookup performance. The lookup procedure of GPMHIA exploits excessive multi-threads to search all hashing systems and the global table in parallel. Ideally, most matching prefixes appear in one of the 25 hash systems. However, a perfect hash function is difficult to produce in an on-fly system and the longest matching prefix may be found in the global table by linear search.

Lee et al. proposed a unique architecture, which considers the performance of memory accesses for different types of memory in GPUs [24]. They place the forwarding table in constant memory, one of the fastest memory in a CUDA GPU adapter, for speeding up the lookup procedure. Since the constant memory is too small to store a large forwarding table, this scheme is suitable for edge networks with a small number of prefixes. Li et al. used a trie-based data structure for fast IP lookup, namely GAMT [25]. The data structure consists of a multi-bit trie and a next-hop table, where the multi-bit trie is stored in the GPU memory. The whole data structure is also stored in the system memory. The LPM result produced by the GPU is a parameter returned to the CPU to determine the next-hop in a zoom table. The transmissions of packet headers, results, and control operations share the bandwidth of PCIe bus to limit the performance of IP lookup procedure.

In [26], an approach based on Bloom filters is presented with two 16-lane PCIe buses employed, one for communicating both host and device and another for connecting NICs and the CUDA adapter (or graphics adapter). Both PCIe buses independently control internal and external communications. Data transferring may exhaust the bandwidth of PCIe buses because of the high throughput of GPU. Moreover, Bloom filters may cause false positives to incur wrong LPM results. Han et al. presented a specialized system, PacketShader, to accommodate two sets of computer architecture for alleviating the bottleneck of PCIe bandwidth [27], where each set uses a CPU to control a GPU and two NICs via an independent I/O hub chipset. PacketShader uses a previous data structure, DIR-24-8-BASIC [28], for IP address lookup. The data structure can support atomic update operations to avoid race conditions between different processing units. The result shows that PacketShader equipped with two GPUs can achieve a throughput of 40 Gbps by forwarding 64-byte IPv4 packets.

III. CHARACTERISTICS OF CUDA-SUPPORTED GPUs

The CUDA architecture has its own device processor and memory but differs from general computer systems. Understanding the properties of CUDA and designing an appropriate data structure are thus important to fully exploit the capability of CUDA. In this section, we briefly introduce the characteristics of CUDA-supported GPU and look into the necessary considerations in designing an IP-lookup data structure.

A. STREAMING MULTIPROCESSOR

In the CUDA hardware, a streaming multiprocessor (SM) is the elementary hardware unit for thread execution. According to the hardware version, an SM is composed of 8, 32, or 96 streaming processors (SPs) and supports up to 8 or 16 blocks of concurrently executing threads. Threads in a block are further divided into multiple warps, where each warp contains 32 threads. A warp is the basic unit of thread computation that all threads in a warp are executed simultaneously. Each SM manages a number of specified warps, and the maximum number of threads in an SM is the number of warps times that of threads. The maximum number of threads executed in parallel can be yielded from the number of threads in an SM times that of SMs. For example, the specification of GT200 includes 32 warps and 30 SMs; thus, the thread number is 30,720. The SM specifications in different versions of CUDA hardware are shown in Table 1 [29].

TABLE 1. SM specifications in different CUDA hardware versions.

Hardware Version	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Num. of SP / SM	8				32	96	
Max Blocks / SM	8					16	
Max Warps / SM	24		32		48	64	
Max Threads / SM	768		1024		1536	2048	
Num. of Registers / SM	8K		16K		32K	64K	

Each SM only fetches one instruction for its SPs. Accordingly, the execution model of warps is single instruction multiple thread (SIMT), where only the threads in a warp fetching the same instruction is practically executed. Therefore, the performance can benefit from preventing the occurrence of conditional branch instructions to activate more threads in parallel. Both thread scheduling in a warp and warp scheduling in a block are similarly determined by CUDA GPU.

A CUDA GPU is equipped with a number of registers, e.g. 32,768 registers in GF110. These registers have low access latencies and are dispatched to concurrently executing threads. The CUDA complier automatically arranges and saves programmer-declared variables into registers until the registers are insufficient. The remainder variables are stored in the relatively slow local memory. To improve the computing performance, programmers should minimize the number of temporary variables to keep them in registers.

B. MEMORY

The main memory of a CUDA device is global memory, which can be accessed by both CPU and GPU without caching support. Each access to global memory exhibits the longest access latency in the CUDA memory hierarchy. The long access latency can be improved by binding a portion of global memory as texture memory to enable texture caching. Texture memory is accessed only when a texture cache miss occurs; thus, texture memory usually has a much better

performance than global memory for the data with temporal locality. However, the content of texture memory cannot be modified in runtime in order to maintain data consistency between SMs.

A CUDA device also supports several types of fast memory. Constant memory is another type of read-only memory during thread execution. It supports caching but usually too small to save a complete data structure [24]. Each SM has a shared memory, which supports read/write operations. Shared memory is slower than texture cache and may incur extra cost to maintain data consistency between different SMs.

Data allocated in the CUDA memory system is a crucial factor for the overall performance because increasing the cache hit rate can alleviate the memory competition between SMs. In the past, the placement of operated data was difficult to handle. A well-designed data structure is thus important for CUDA architecture.

IV. ARCHITECTURE REFINEMENT

CUDA has an inherent hazard that one copy of all data and at least one device function must be acquired from the main memory of the computer system via PCIe bus. When the computing task is accomplished, the results are duplicated and transmitted reversely. When the data must be retrieved from other components like NICs, the PCIe bus becomes a bottleneck for transmitting the data for CUDA computing. In order to solve this problem, GPUDirect remote direct memory access (RDMA) technology is contrived to enable third party endpoints directly communicate each other [30]. CUDA GPUs implement a base address register (BAR) which allows devices directly access the internal memory of a GPU device without passing through system memory or CPU. Several hard real-time and high availability systems using CUDA are equipped with the GPUDirect RDMA feature, e.g. GE IPN 250 and IPN 251 for military and aerospace [31], [32]. Although the GPUDirect RDMA technology avoids unnecessary memory copies and reduce the transmitting cost, the transmission rate is still limited since the bandwidth of the PCIe bus is much less than that of GPU memory, e.g. 8 GB/s of PCIe 2.0 versus 57.6 GB/s of GeForce 9800GT or 288 GB/s of Tesla K40. To fully utilize the computation capability of CUDA, we propose an architecture of CUDA-based forwarding engine as a stand-alone device with NICs, which is similar to GE IPN 250 and IPN 251.

A. PROPOSED ARCHITECTURE

We show the conceptual architecture of the proposed GPU-based IP forwarding engine in Fig. 1, where the GPU is only responsible for performing IP address lookups. In this architecture, NICs directly transfer headers of packets to the global memory of GPU by the fast on-board memory bus. Although transmitting IP addresses via a PCIe bus is possible, the bus bandwidth (8 GB/s per direction with 16 lanes) is not sufficient, as demonstrated in our experiments in Section 6.

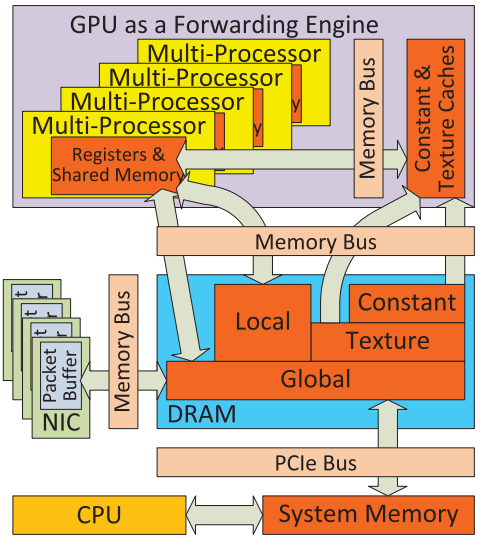


FIGURE 1. GPU-based IP forwarding engine.

The CPU constructs and transmits the data structure to the global memory via the PCIe bus. By instructing the CUDA compiler, the searchable data structure is stored in the texture memory to benefit from the texture cache and shorten the access latency. Both packet headers and the searchable data structure are accessed through the CUDA memory bus.

We determine the number of threads which execute the IP address lookup procedures simultaneously according to the packet buffer size. For instance, each gigabit Ethernet (GbE) port is equipped with 42 MB buffer for Juniper L3 switch [34]. The buffer is increased to 512 MB for each 10 GbE port. Cisco XR 12000 series routers are equipped with 512 MB for both one-port 10 GbE and 10-port GbE line cards [35]. These line cards use buffers to transmit and receive packets. According to these specifications, more than 350K packets can be stored in a router per 10 Gbps throughput. Considering the number of packets is significantly higher than the number of allowable GPU threads, we maximize the number of threads in our experiments. We are also aware that the packet latency may be increased because of the long lists of packets awaiting for the results of IP address lookups. However, our experimental results show that a GPU can achieve sufficient throughput to support more than one thousand GbE ports. Thus, the packet latency can be reduced significantly to make the configuration feasible. We discuss the issue of packet latency in Section 6.5.

We describe the packet path in the proposed architecture in the data plane and the control plane. The path of data plane is listed below.

1) The incoming packets received by a NIC are stored in the buffer.
2) A batch of packet headers is transmitted to a pre-assigned address of the global memory.
3) SMs read the headers from the global memory, where each header is assigned to one thread.
4) SMs search the data structure in the texture cache/memory to determine the next-hops of all headers.

5) The lookup results for each batch are stored in another pre-assigned address of the global memory.
6) The lookup results in the global memory are pushed back to the corresponding NICs to accomplish packet forwarding.

The bus of global memory is shared by all NICs for transmitting packet headers and their next-hops. As discussed above, the global memory has enough bandwidth to fulfill the packet path.

Besides packet forwarding, routers also need to provide control plane and management plane functions. For example, a router needs to ensure that the contents of the forwarding table reflect the current network topology by receiving route update messages from neighboring routers. When the NICs receive packets for control or management plane, these packets are forwarded to the CPU for further processing. The CPU may also transmit control messages to other routers. These messages are transmitted to one or more NICs through the global memory of CUDA. The packet path of the control messages is listed below.

1) NICs identify that the received packets are control messages.
2) The control packets in NICs are transferred to a designated region in the global memory.
3) The control packets in the global memory are moved to the system memory through PCIe bus.
4) The CPU processes control packets and generates response packets in the system memory.
5) The response packets are delivered to the global memory via PCIe bus and transmitted to the packet buffer of NICs for forwarding as a general data packet.

We note that the next-hops of the control packets are determined by the CPU itself to avoid interfering the procedure of data plane. Since only a small percentage of packets are control packets, the lookup overhead of CPU and the bandwidth consumption of both PCIe bus and global memory are acceptable.

B. BATCH PROCESSING FOR UPDATES

Although a lookup data structure may support incremental updates, the conventional update procedure is not suitable for GPU execution because of two reasons. First, the memory space allocated to a device function cannot be modified by other device functions. Thus, both update procedure and lookup procedure must reside in the same device function. Second, the update procedure in a device function can be executed only by one thread to avoid race conditions between different SMs. The limitation may cause the update procedure too slow to support frequent updates. Therefore, we use batch processing to update the data structure via a CPU in our implementation.

The CPU receives route updates and renews the data structure while performing IP address lookups in the GPU. To keep both operations work concurrently, a GPU has two copies of searchable data structures, as shown in Fig. 2. When a copy is used for IP address lookup, the other copy is uploaded from the renewed data structure generated by the CPU. Therefore, there are two device functions for the two copies data structures, respectively. When the updated copy is ready, the CPU stops the device function with the old data structure and starts another device function with the new data structure. In the next iteration, the CPU will update the old data structure and activate it for lookups. Our current implementation uploads the entire data structure to simplify the upload procedure. Although a redundant copy of the data structure consumes extra memory space, the overhead is acceptable since our experiments in Section 6 show that the proposed data structure occupies fractional memory in a CUDA-enabled device.

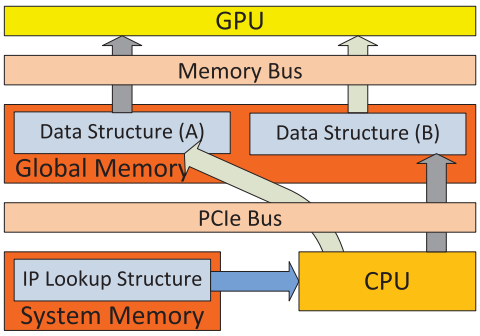


FIGURE 2. **Batch updates with two data structures.**

V. IP ADDRESS LOOKUP ALGORITHM

In this section, we describe the data structure used in the GPU for the IP address lookup routine. Given that the GPU is designed to process streaming data and has several programming limitations. For example, the GPU hardware architecture supports recursive function after hardware version 2.0 but is unlike a traditional CPU platform that can use a recursive function to implement the lookup routine easily. Recursive functions are thus transformed into codes with loops to achieve a better performance. Static variables for state maintenance are also forbidden in our CUDA codes.

After investigating the hardware characteristics of CUDA-based GPUs, we conclude that there are several prerequisites for designing the data structure of IP address lookup. First, the number of different element types in the searchable data structure should be minimized. According to the SIMT execution mode, an SM only performs one instruction for all threads. Since the instructions for processing each element type are different, fewer element types can increase the probability of executing the same instruction in all threads. The instructions for processing each element should also be minimized. As a result, complex techniques for compressing a data structure are not suitable for the GPU implementation. The second requirement is the support of IPv6 since its increasing importance caused by the depletion of IPv4 addresses. Third, the data structure should be updateable to avoid a cost-ineffective reconstruction.

We present a multi-bit trie data structure, which can satisfy the prerequisites mentioned above. By comparing multiple

bits in each access of an intermediate node, a multi-bit trie usually has a lower depth and lookup time. The memory space of a general multi-bit trie could be high because of the high-cost prefix expansion. In Fig. 3(a), a general binary tree is built for a set of prefixes. We set the stride to 4 to construct a multi-bit trie by a simple top-down manner. As shown in Fig. 3(b), the memory usage of the multi-bit trie is high because of numerous redundant fields. The problem can be alleviated by reducing the number of leaf nodes using the proposed multi-bit trie data structure. The detail of constructing an efficient multi-bit trie is described in the following subsection.

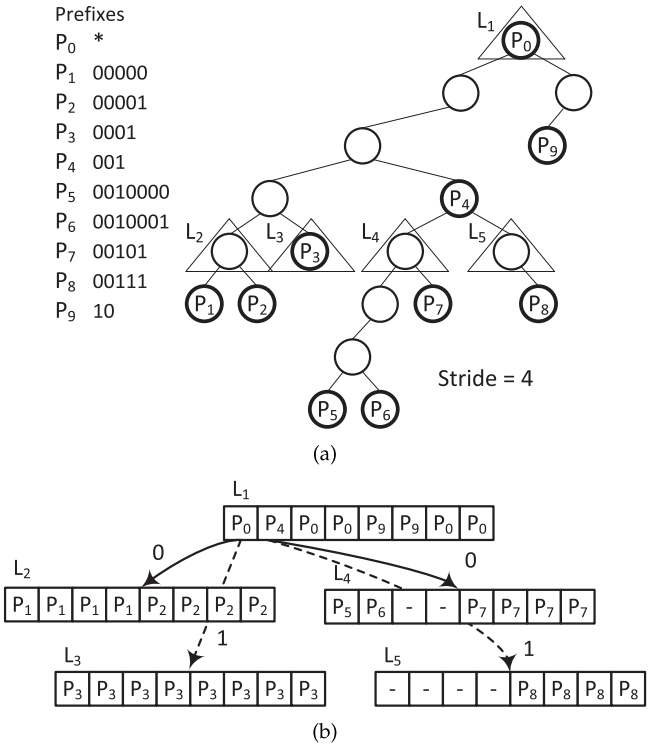


FIGURE 3. A native multi-bit trie. (a) Tree representation. (b) Array representation.

A. MULTI-BIT TRIE

The proposed multi-bit trie consists of two components, multi-bit node and multi-bit leaf. The stride size of the multi-bit node and multi-bit leaf are denoted as α and β, respectively. Both α and β are parameters for adjusting the tradeoff between speed and storage performance. The differentiation of parameters is depicted in Section 6. Our algorithm starts by generating a binary tree from an original routing table. We then traverse the binary tree with DFS to build the proposed multi-bit trie.

Several operations are executed while visiting each node of the binary tree. The first operation checks whether the depth of the successive subtrie is smaller than or equal to β. If the condition is meet, a multi-bit leaf is generated corresponding to the currently visited node. Otherwise, we further check if the depth of the current node is a multiple of α. If the answer is positive, a multi-bit node is generated for the current node. In Fig. 4, we show how the multi-bit trie is generated for the same set of prefixes in Fig. 3, where the values of α and β are 2 and 3, respectively. In Fig. 4(a), the multi-bit nodes are generated in a top-down manner, and the multi-bit leaves are generated in a bottom-up manner. In the multi-bit trie, multi-bit leaves are always covered by multi-bit nodes and all multi-bit leaves are disjoint to each other. For each multi-bit node, a 2^{α}-entry array is generated to store the all information within an α-bit stride in the original binary trie. Each multi-bit leaf corresponds to a 2^{β}-entry next-hop array. The content of both multi-bit nodes and leaves is shown in Fig. 4(b). As compared to the general multi-bit trie, our multi-bit trie has better space efficiency because of fewer leaves.

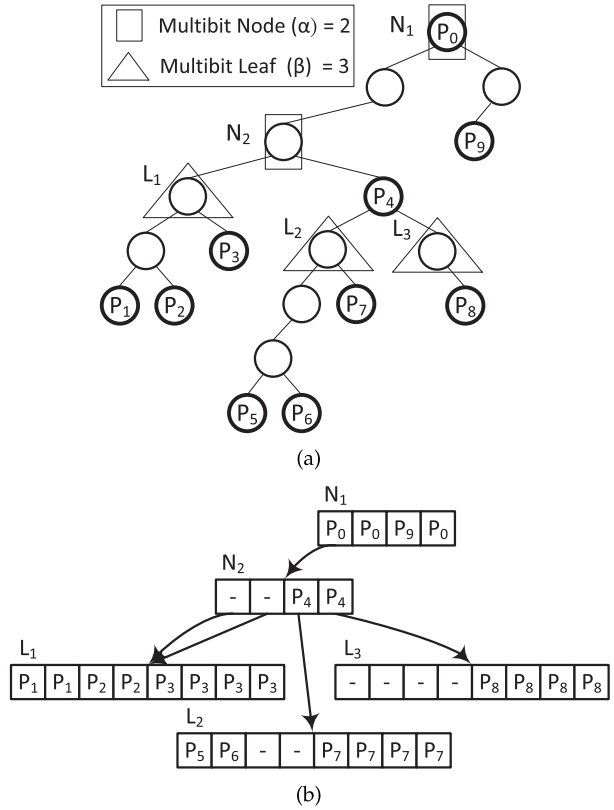


FIGURE 4. The proposed multi-bit trie. ($\alpha = 2$, $\beta = 3$). (a) Tree representation. (b) Array representation.

The performance of the previous multi-bit trie can be further improved with the introduction of overlapping multi-bit leaves. The pseudo code of the improved algorithm is shown in Fig. 5. The function "*Subtrie_Depth()*" returns the depth of the subtrie whose root is specified in the input argument. Another function called "*Depth()*" returns the depth of the trie node specified in the input argument. In the improved algorithm, a multi-bit leaf is generated only when the depth of the successive subtrie is equal to β. Once a multi-bit leaf is generated, the successive subtrie is removed

```
Input: Root of the binary trie constructed from the routing table.
Output: Multi-bit nodes and leaves.
Constructor (Current_Node) BEGIN
1. IF (Subtrie_Depth(Current_Node) == β)
2.     Current_Node→Type = MULTIBITLEAF;
3. ELSE
4.     IF ((Depth(Current_Node)%α) == 0)
5.         Current_Node→Type = MULTIBITNODE;
6.     Constructor(Current_Node→Left);
7.     Constructor(Current_Node→Right);
8.     IF (Subtrie_Depth(Current_Node) == β)
9.         Current_Node→Type = MULTIBITLEAF;
END
```

FIGURE 5. Multi-bit trie construction with overlapping multi-bit leaves.

from the binary tree. This enhancement could generate multi-bit leaves with overlapping regions. Consequently, the algorithm could benefit from the flexibility to further decrease the number of multi-bit leaves. The example in Fig. 6 shows the multi-bit trie generated by the improved algorithm with the same values of α and β. Since the first multi-bit leaf, L_1, covers prefixes P_5 to P_7, only one multi-bit leaf can cover the other prefixes from P_1 to P_8. As a result, the number of leaves is reduced to achieve better space efficiency.

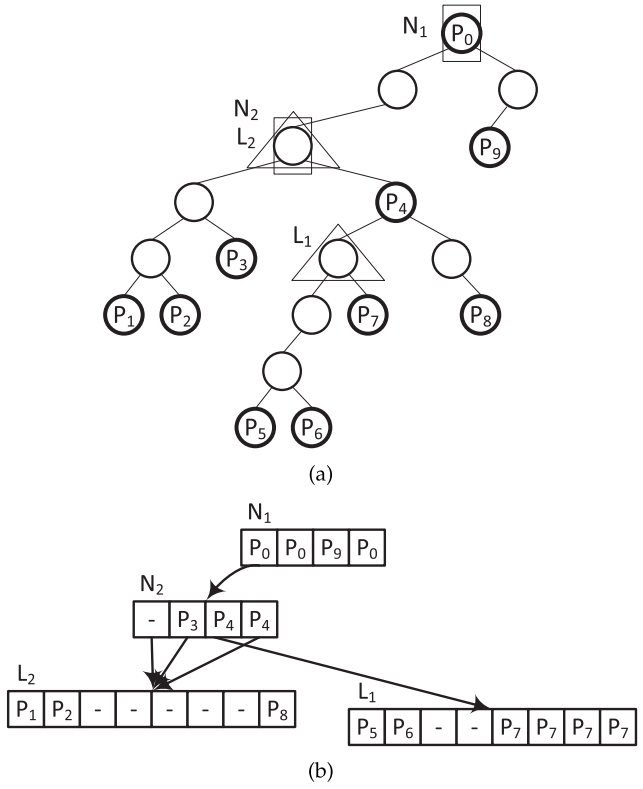


FIGURE 6. The refined multi-bit trie. ($\alpha = 2$, $\beta = 3$). (a) Tree representation. (b) Array representation.

Since the multi-trie algorithm only uses two different types of data elements, the number of instructions for processing these data elements in an IP lookup procedure is reduced. Moreover, there is only one conditional branch in processing the multi-bit node. Since each address lookup accesses multiple trie nodes and at most one leaf, the instructions of parsing a trie node are usually executed simultaneously. The number of idle threads can be reduced to improve the computation throughput. The trie-based data structure also improves the cache hit ratio. The multi-bit nodes nearby the trie root are usually resident in the cache memory because these nodes are accessed frequently. The procedure of constructing the data structure is implemented in the host function since the construction procedure cannot be accelerated by using parallel computation.

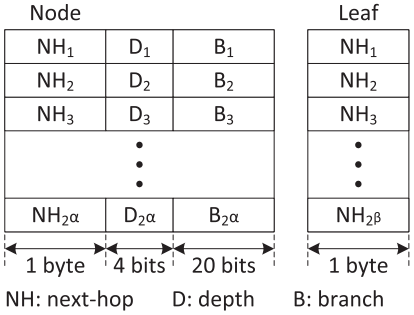


FIGURE 7. Data structures of multi-bit node and leaf.

B. DATA STRUCTURE

We describe the data structures of multi-bit node and multi-bit leaf in this section. Both data structures are depicted in Fig. 7. The multi-bit node includes an array of 2^α entries with α-bit stride size. Each entry of the node array comprises three fields: next-hop, node depth, and branch pointer. The next-hop field (one byte) indicates the next-hop of the prefix corresponding to the entry. This field can avoid exhausted route updates caused by leaf pushing [36]. The depth field shows the relative depth of the corresponding multi-bit leaf. The last field, branch pointer (two bytes), specifies the address of either a multi-bit node or a multi-bit leaf. The addresses of multi-bit nodes and multi-bit leaves are independent. We use the depth field to identify the address type. If the value of the depth field equals $(\alpha + 1)$, the branch pointer stores a node address; otherwise, this field branch stores a leaf address. Since the depth field has $(\alpha + 2)$ distinct values and requires at least $\lceil log_2(\alpha + 2) \rceil$ bits, we reserve four bits for the depth field and set the maximal value of α to 14. Each entry of a multi-bit node has four bytes. The data structure of multi-bit leaf is a next-hop array with 2^β elements, where each entry has only one byte.

For the generated multi-bit trie shown in Fig. 6, the content of all multi-bit nodes and multi-bit leaves are illustrated in Fig. 8. We use NH(P_i) to indicate the next-hop of prefix P_i. Since the first entry of node N_1 stores a branch pointer to N_2, the value of its depth field is 3 $(= \alpha + 1)$. The last three entries of N_1 do not point to any multi-bit node or leaf; thus their branch fields are assigned with NULL. There is no prefix

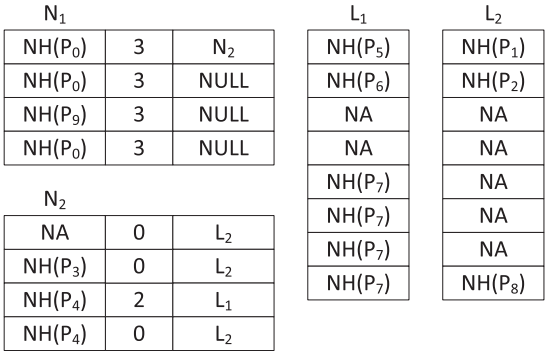


FIGURE 8. Multi-bit nodes and leaves of the multi-bit trie in Fig. 6.

corresponding to the first entry of node N_2; thus, the next-hop value is filled with not available (NA). We also show that the branch fields of N_2 can correctly point to L_2 by setting the depth field to zero.

To perform IP address lookup within the proposed data structure, the destination IP address is split into α-bit chunks. These chunks are used to traverse a path of multi-bit nodes until an address of a multi-bit leaf or a NULL branch value is fetched. In the first case, the corresponding depth value is less than $(\alpha + 1)$. The relative depth of the multi-bit leaf is used to generate the suitable β-bit chunk for indexing the entry in the multi-bit leaf. In the later case, the search procedure is terminated directly and returns the last matching next-hop in the traversed multi-bit nodes. The number of memory accesses is thus equal to $[(W - \beta)/\alpha + 1]$ in the worst case, where W is the IP address length.

The proposed data structure also supports incremental updates. Since the proposed data structure does not use the technique of leaf pushing, it can be updated without reconstructing the data structure. In the worst case, $\lceil W/\alpha \rceil$ multi-bit nodes are generated for a new prefix to result in $(\lceil W/\alpha \rceil \times 2^{\alpha})$ updated entries.

VI. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our scheme. We use five IPv4 and three IPv6 routing tables downloaded from Route Views Project [37] and APNIC [38]. The IPv4 routing tables have 354K to 366K prefixes, whereas the IPv6 routing tables have only 5200 to 5700 prefixes. We use four sets of (α, β) parameters, (2, 4), (4, 4), (4, 8), and (8, 8), to construct the proposed multi-bit trie. We also use ten batches of randomly generated 100 million IP addresses to calculate the lookup throughput in MPPS or GPPS. We execute prefix updates simultaneously while performing IP address lookups. Three types of GPUs, G92, GT200 and GF100, are used in our experiments. G92 is embedded in a display adapter with the lowest specification. Both GT200 and GF110 designed for pure computations provide relatively powerful computing capabilities. Detailed specifications of these GPUs are listed in Table 2.

The evaluation consists of five parts. In the first part, we show the storage performance of the proposed multi-bit trie.

TABLE 2. Specifications of G92, GT200, and GF110.

GPU	G92(A2)	GT200(B1)	GF110(A1)
Device Name	GeForce 9800GT	Tesla C1060	Tesla C2075
Hardware Version	1.1	1.3	2.0
Number of SP	112	240	448
Number of SM	14	30	14
Frequency of SP	1.50 GHz	1.30 GHz	1.15 GHz
Clock Rate	600 MHz	602 MHz	450 MHz
Mem. Size	512 MB	4096 MB	6144 MB
Mem. Bus Width	256 bits	512 bits	384 bits
Mem. Bandwidth	57.6 GB/s	102.4 GB/s	144 GB/s
Registers Per SM	8192	16384	32768
Per-SM Threads	768	1024	1536
Per-chip Threads	10752	30720	21504
Constant Memory	64 KB	64 KB	64 KB
Bus Interface	PCIe 2.0 × 16	PCIe 2.0 × 16	PCIe 2.0 × 16

The speed performance of our scheme with different GPU implementations is represented in the second part. The third part depicts the update performance. The performance for IPv6 routing tables is shown in the fourth part. We discuss the issue of packet latency in the last part.

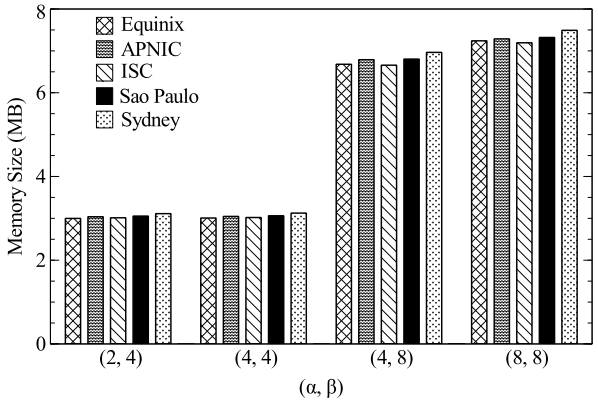


FIGURE 9. Memory requirements of IPv4 routing tables.

A. STORAGE PERFORMANCE FOR IPv4 ROUTING TABLES

Fig. 9 shows the memory requirements of the proposed scheme for different routing tables. The memory size of the generated multi-bit trie based on parameters (4, 4) is about half of that of parameters (4, 8) because of the finer granularity of the small stride of the multi-bit leaf. However, the small stride of the multi-bit leaf also result in more multi-bit nodes. Thus, the average depth (or IP lookup latency) of the multi-bit trie with $\beta = 4$ is larger than that with $\beta = 8$ (see Fig. 10). Similarly, the number of internal nodes for parameters (4, 8) is larger than that for parameters (8, 8), and the latter consumes slightly more memory than the former. The similar trend can also be observed for another two sets of parameters, (2, 4) and (4, 4).

B. SPEED PERFORMANCE FOR IPv4 ROUTING TABLES

Next, we show the packet throughput using different GPUs with four sets of parameters. We include the results

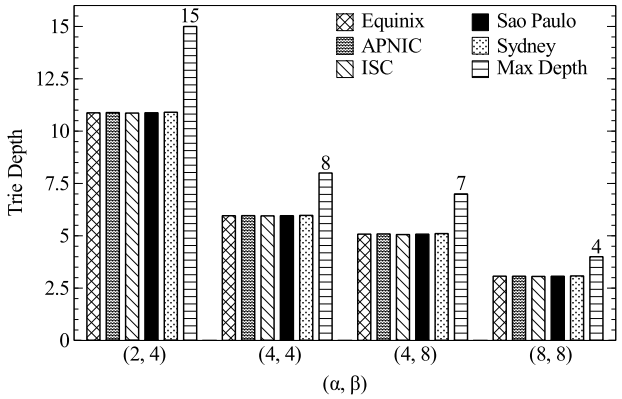


FIGURE 10. **Average and maximum trie depths of IPv4 routing tables.**

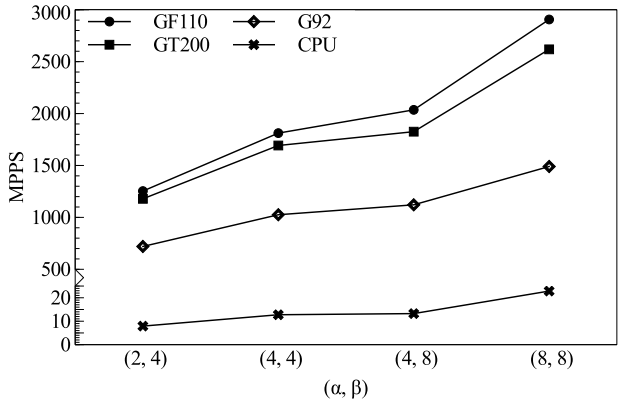


FIGURE 11. **IPv4 throughput with different stride size (Sydney).**

of a 2.33 GHz CPU, Intel Xeon E5410, for comparison. Fig. 11 depicts the throughput for Sydney routing table. A larger stride setting usually leads to a greater throughput due to the decreased tree depth. The result shows that GF110 with stride parameters, (8, 8), can achieve the throughput with more than 2.9 GPPS, while G92 and GT200 with the same stride parameters can achieve more than 1.49 GPPS and 2.62 GPPS, respectively. GF110 achieves better performance due to its 448 SPs. GF110 also has a better hardware architecture, which supports more threads and registers in a SM, to achieve over twice throughput of G92. We list the detailed performance for Sydney in Table 3. It also shows that our CPU implementation only achieves approximately 0.8 % of the throughput of GF110 in the case of parameters (8, 8).

TABLE 3. **Numerical results for IPv4 routing table (Sydney).**

(α, β)	(2,4)	(4,4)	(4,8)	(8,8)
Number of Nodes	72882	16692	13810	813
Number of Leaves	58264	66899	12761	14007
Avg./Max. Depth	10.9/15	6.0/8	5.1/7	3.1/4
Mem. size(MB)	3.11	3.12	6.96	7.49
(Speed Performance in MPPS)				
GF110	1253.48	1811.59	2035.62	2906.04
GT200	1178.83	1692.33	1825.48	2618.49
G92	719.94	1025.43	1121.08	1490.09
CPU	7.91	12.76	13.25	22.86

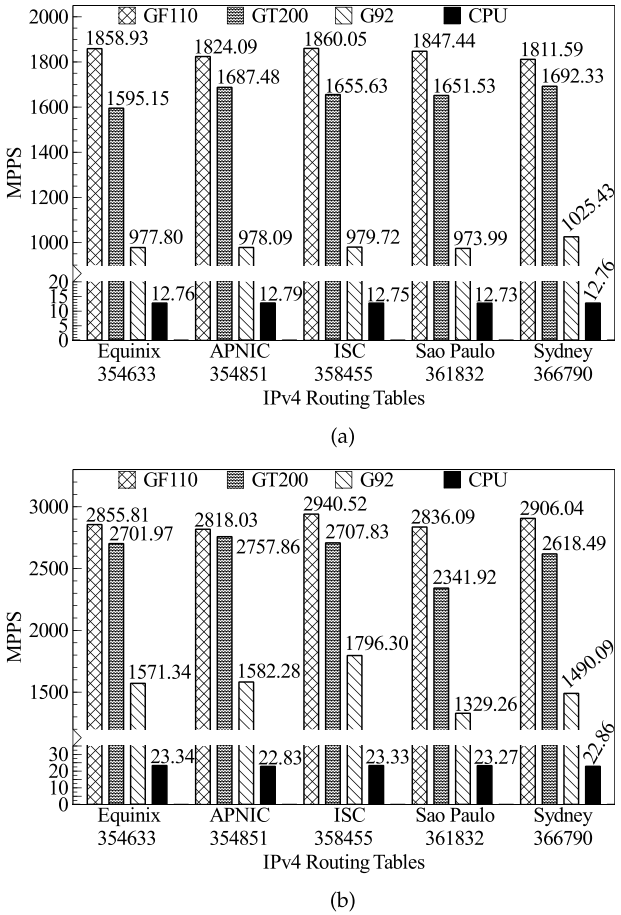


FIGURE 12. **Comparison of IPv4 throughput for different routing tables. (a) ($\alpha = 4$, $\beta = 4$). (b) ($\alpha = 8$, $\beta = 8$).**

Fig. 12 shows the throughput of our scheme for five routing tables with two different sets of parameters, (4, 4) and (8, 8). The results in both Fig. 12(a) and (b) reveal that the trie depth determines the speed performance. As shown in Fig. 10, the depth of the generated multi-bit trie using parameter set (4, 4) is about two times than that using parameter set (8, 8). However, the difference between different parameter sets is not as large as the trie depth because the large stride size also leads to a large data structure to lower the cache hit ratio. Accordingly, we can observe that the throughput difference between different routing tables is less significant for small strides. In Fig. 12(a), the largest throughput difference between two routing tables is about 50 MPPS between ISC and Sydney, where Sydney requires the largest data structure among all routing tables. For another parameter set, (8, 8), the throughput difference between two routing tables is increased to 122.5 MPPS between APNIC and ISC, where ISC requires the smallest data structure among all routing tables.

C. UPDATE PERFORMANCE FOR IPv4 ROUTING TABLES

Before showing the update performance, we first describe the measurement methodology. We have introduced our

approach of batch processing for updating in Section 4.2. We divide the update performance into two parts. The first part considers the switching frequency between two data structures, whereas the second part shows the number of updated prefixes within the time interval of switching data structures. The switching frequency can be determined by the number of processed IP addresses in a batch or by the number of updated prefixes. Since the lookup function is executed by GPU, we choose the former to maximize the GPU utilization in this study. Each batch of the lookup function processes 10 million IP addresses. In Table 4, we show the performance metrics related to the update performance based on GF110. We measure the time period of processing a lookup batch. We also measure the required time period for transmitting the updated data structure to the GPU device. The difference between the lookup and transmission time is then used to update the data structure by CPU. This time period is defined as the update time. We further measure the number of updated prefixes in the update time to obtain the number of updated prefixes per second.

TABLE 4. Update performance for IPv4 routing table (Sydney).

(α, β)	(2,4)	(4,4)	(4,8)	(8,8)
Trans. Time (ms)	0.57	0.57	1.20	1.34
Update Time (ms)	7.41	4.95	3.71	2.10
Update TimePer Prefix (ns)	712	727	732	748
Batch Lookup Time (ms)	7.98	5.52	4.91	3.44
Average Lookup TimePer Packet (ns)	0.80	0.55	0.49	0.34
Million UpdatesPer Second	1.30	1.23	1.03	0.82

For parameters (2, 4), a batch of IP addresses lookup time consumes 7.989 milliseconds with GF110. We also discover that transmitting the lookup data structure (about 3.112 MB) through PCIe 2.0 × 16 bus requires 0.571 milliseconds from CPU to GPU. Thus, 7.408 milliseconds is left for the CPU to update the data structure. The CPU can update 10,404 prefixes in the update time. Therefore, the average update time for each prefix is 712 ns. Theoretically, we can update about 1.3 million prefixes because we can switch the data structures 125 times per second. For another set parameters (8, 8), the lookup and transmitting time are 3.441 and 1.336 milliseconds, respectively. The frequency of swapping data structures is 290 times per second. The update time is reduced to 2.101 milliseconds. The CPU can update 2,809 prefixes within the update time, and each update requires a time which is slightly longer than 748 ns because of the large stride size. As a result, 0.816 million prefixes can be updated per second. Basically, a set of larger stride parameters usually leads to poor update performance since the shorter lookup time and larger transmitting time (due to larger memory size). However, the switching frequency can be decreased to further increase the update performance.

D. PERFORMANCE FOR IPv6 ROUTING TABLES

We show the performance of our scheme for IPv6 routing tables. We downloaded three IPv6 routing tables, denoted as Equinix, ISC, and Sydney. Fig. 13 shows the throughput for the routing table, Sydney. Although the numerical results for IPv4 show that the throughput of our scheme decreases for a higher multi-bit trie, it is not the case while comparing the packet throughput between IPv4 and IPv6. As shown in Fig. 13, GF110 achieves better performance for IPv6 routing tables even though fourfold-depth multi-bit tries are generated. Since IPv6 routing tables have much fewer prefixes than IPv4, the cache hit ratio of GF110 is high enough to override the extra memory accesses caused by a longer search path in a multi-bit trie. G92 and GT200 cannot produce the same results because of their lower cache hit ratios. Fig. 14 shows the packet throughput for different IPv6 routing tables using the proposed multi-bit trie with stride size (8, 8). GF110 achieves the best throughput with 3.6 GPPS, which is 155 times faster than that of our CPU implementation.

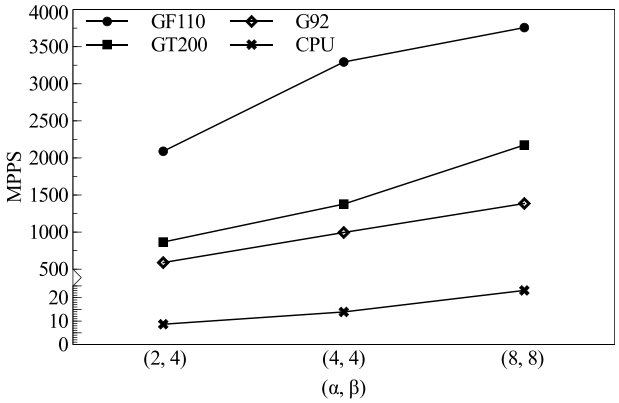


FIGURE 13. IPv6 throughput with different stride size (Sydney).

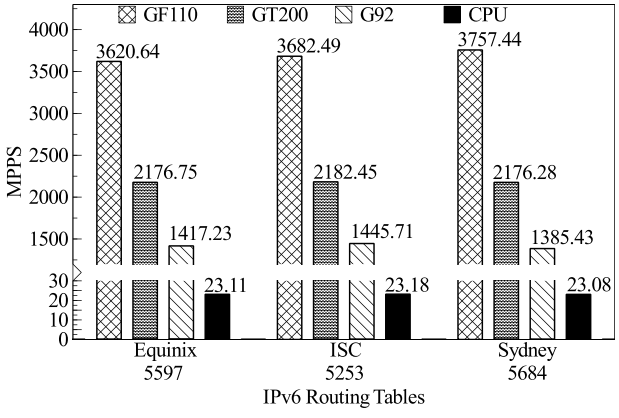


FIGURE 14. IPv6 throughput with ($\alpha = 8, \beta = 8$) stride.

Table 5 shows both storage and update performance for the IPv6 routing tables. Since the performance difference among these routing tables is negligible, we only show the numerical results of Sydney. The storage requirement is less than 0.5 MB for stride sizes (2, 2) and (4, 4). However, raising the stride size to (8, 8) significantly increases the storage to 2.4 MB. The increased storage is caused by the depth

TABLE 5. Update performance for IPv6 routing table (Sydney).

(α, β)	(2,2)	(4,4)	(8,8)
Trans. Time (ms)	0.08	0.13	0.45
Update Time (ms)	4.71	2.91	2.21
Update TimePer Prefix (ns)	924	982	1026
Batch Lookup Time(ms)	4.79	3.04	2.66
Average Lookup TimePer Packet (ns)	0.48	0.30	0.27
Million UpdatesPer Second	1.06	0.98	0.81
Mem. Size(MB)	0.31	0.45	2.42

of the longer IPv6 prefixes. We use the same methodology to measure the update performance. For the parameter set, (8, 8), the measured lookup and transmitting time are 2.661 and 0.450 milliseconds, respectively. The frequency of swapping data structures is 375 times per second. The update time is 2.211 milliseconds, which allows the CPU to update 2,154 prefixes. Therefore, 0.809 million prefixes can be updated per second. The experimental results show that GPUs have superior scalability for the current IPv6 routing tables.

E. PACKET LATENCY

According to the previous results, we can measure the extra packet latency to fully utilize the computing capability of GPU. We use the implementation with GF110 and the stride sizes (8, 8) as an example for calculation. For the IPv4 routing table of Sydney, the throughput of our implementation is 2.906 GPPS, which can be translated to a transmission rate of 1.487 terabits per second with 64-byte packets. We assume that the total transmission rate is shared by 1487 GbE ports in a core router. Since GF110 supports up to 21504 threads, each batch of IP address lookup accommodates about 15 packets from each port. Thus, the extra latency is merely 7.68 μs (15×512 bits/1 Gbps). Therefore, the extra latency for address lookups is negligible. The buffer in GPU for storing these packets is less than 20 MB (21504×960 bytes), which is reasonable for the state-of-the-art core routers. We can conclude that the configuration is feasible for a core router. A GPU with fewer SMs would be rather suitable to fully utilize the computation capability for edge or corporation routers with fewer ports.

VII. CONCLUSIONS AND FUTRUE WORK

In this paper, we present a CUDA-based IP forwarding engine for maximizing the packet throughput. We investigate the properties of the CUDA-based GPU, including memory architectures and SIMT, which can benefit the lookup performance. The refined data structure for IP address lookup is resilient and can get rid of the restriction of prefix-length to support IPv4 and IPv6 lookup in high performance. By using a GF110 GPU, we can achieve throughputs more than 2.8 GPPS for IPv4 routing tables and 3.6 GPPS for IPv6 routing tables. G92 and GT200 can achieve throughputs more than 1.3 and 2.3 GPPS, respectively. The results demonstrate that the CUDA-based IP forwarding engine with the proposed data structure is capable of forwarding more than one billion packets per second. We also present the architecture with dual data structures to support several hundred thousand updates per second.

In our experiments, IP addresses are stored in the GPU memory directly to avoid transmissions via the relatively slow PCIe bus. Currently, GPU memory has sufficient bandwidth to fully utilize the computation capability of SMs. We suggest the provision of extra channels for broadening the transmission rate between NICs and GPU to make the GPU-based packet forwarding engine practical and efficient. Due to the incoming adoption of IPv6, our future work will also focus on the improvements of both update and storage performance for IPv6 routing tables.

ACKNOWLEDGMENT

The authors want to express our sincere gratitude to Dr. Chao-Tung Yang at the Department of Computer Science in TungHai University for his generous support to facilitate our experiments.

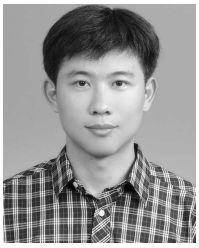
REFERENCES

[1] Y. Rekhter and T. Li, *An Architecture for IP Address Allocation With CIDR*, document RFC 1518, Sep. 1993.
[2] M. Gao, K. Zhang, and J. Liu, "Efficient packet matching for gigabit network intrusion detection using TCAMs," in *Proc. IEEE AINA*, vol. 1. Apr. 2006, pp. 249–254.
[3] P. M. Reddy, "Fast updating algorithm for TCAMs using prefix distribution prediction," in *Proc. ICEIE*, Aug. 2010, pp. V1-400–V1-404.
[4] T. Mishra and S. Sahni, "PETCAM—A power efficient TCAM architecture for forwarding tables," *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 3–17, Jan. 2012.
[5] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems," in *Proc. 19th USENIX Conf. Secur.*, 2010, pp. 1–16.
[6] T. Banerjee-Mishra and S. Sahni, "Consistent updates for packet classifiers," *IEEE Trans. Comput.*, vol. 61, no. 9, pp. 1284–1295, Sep. 2012.
[7] L.-C. Hung and Y.-C. Chen, "Parallel table lookup for next generation Internet," in *Proc. 32nd Annu. IEEE COMPSAC*, Jul./Aug. 2008, pp. 52–59.
[8] Y. Qu and V. K. Prasanna, "High-performance pipelined architecture for tree-based IP lookup engine on FPGA," in *Proc. 27th IEEE IPDPSW*, May 2013, pp. 114–123.
[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[10] NVIDIA. (Jul. 19, 2013). *CUDA C Programming Guide*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed Dec. 30, 2013.
[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. 13th ACM SIGPLAN PPOPP*, Feb. 2008, vol. 96, no. 5, pp. 73–82.
[12] T.-H. Li, H.-M. Chu, and P.-C. Wang, "IP address lookup using GPU," in *Proc. IEEE 14th Int. Conf. HPSR*, Jul. 2013, pp. 177–184.
[13] P.-C. Wang, C.-T. Chan, W.-C. Tseng, and Y.-C. Chen, "Fast trie-based routing lookup with tiny searchable core," in *Proc. IEEE GLOBECOM*, Nov. 2002, pp. 2328–2332.
[14] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Trans. Netw.*, vol. 14, no. 2, pp. 397–409, Apr. 2006.
[15] P.-C. Wang, C.-T. Chan, and Y.-C. Chen, "A fast IP routing lookup scheme," *IEEE Commun. Lett.*, vol. 5, no. 3, pp. 125–127, Mar. 2001.
[16] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: Hardware/software IP lookups with incremental updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[17] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, 1960.
[18] H. Lim and J. H. Mun, "An efficient IP address lookup algorithm using a priority trie," in *Proc. IEEE GLOBECOM*, Nov./Dec. 2006, pp. 1–5.
[19] H. Lim, H.-G. Kim, and C. Yim, "IP address lookup for Internet routers using balanced binary search with prefix vector," *IEEE Trans. Commun.*, vol. 57, no. 3, pp. 618–621, Mar. 2009.
[20] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for network packet signature matching," in *Proc. IEEE ISPASS*, Apr. 2009, pp. 175–184.
[21] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu, "Exploiting graphics processors for high-performance IP lookup in software routers," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 301–305.
[22] J. Zhao, X. Zhang, X. Wang, and X. Xue, "Achieving O(1) IP lookup on GPU-based software routers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 429–430, 2010.
[23] X. Yao, Y. Lin, G. Wang, and G. Hu, "A dynamic IP lookup architecture using parallel multiple hash in GPU-based software router," *J. Comput. Inf. Syst.*, vol. 9, no. 3, pp. 967–976, 2013.
[24] Y. Lee, M. Jeong, S. Lee, and E.-J. Im, "Fast forwarding table lookup exploiting GPU memory architecture," in *Proc. Int. Conf. ICTC*, Nov. 2010, pp. 341–345.
[25] 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. 9th ACM/IEEE Symp. ANCS*, Oct. 2013, pp. 1–12.
[26] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, "IP routing processing with graphic processors," in *Proc. DATE*, Mar. 2010, pp. 93–98.
[27] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 195–206, 2010.
[28] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM*, Mar./Apr. 1998, pp. 1240–1247.
[29] NVIDIA. (Jul. 2013). *CUDA GPUs*. [Online]. Available: https://developer.nvidia.com/cuda-gpus/, accessed Dec. 30, 2013.
[30] NVIDIA. (Jul. 19, 2013). *RDMA for GPUDirect*. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma/, accessed Dec. 30, 2013.
[31] GE. (2011). *IPN 250 Data Sheet*. [Online]. Available: http://defense.ge-ip.com/products/ipn250/p3514, accessed Dec. 30, 2013.
[32] GE. (2013). *IPN 251 Data Sheet*. [Online]. Available: http://defense.ge-ip.com/products/ipn251/p3709, accessed Dec. 30, 2013.
[33] RIPE. (Oct. 7, 2013). *Ripe Routing Information Service (RIS)*. [Online]. Available: http://www.ripe.net/data-tools/stats/ris/, accessed Dec. 30, 2013.
[34] Juniper Networks. (2011). *EX8200 Ethernet Line Cards*. [Online]. Available: http://www.juniper.net/us/en/local/pdf/datasheets/1000262-en.pdf, accessed Dec. 30, 2013.
[35] Cisco. (2011). *Cisco XR 12000 and 12000 Series SPA Interface Processors*. [Online]. Available: http://www.cisco.com/en/US/prod/collateral/routers/ps167/product_data_sheet0900aecd80465682.pdf, accessed Dec. 30, 2013.
[36] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, 1999.
[37] Advanced Network Technology Center. (May 2011). *Route Views Project*. [Online]. Available: http://www.routeviews.org/, accessed Dec. 30, 2013.
[38] APNIC. (May 2011). *BGP Routing Table Analysis*. [Online]. Available: http://thyme.apnic.net/, accessed Dec. 30, 2013.

HUNG-MAO CHU received the M.S. degree in computer science and information engineering from Asia University, Taiwan, in 2008. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, National Chung Hsing University. His research interests include IP address lookup, metro Ethernet networks, and software-defined networking.

TSUNG-HSIEN LI received the M.S. degree in computer science and engineering from National Chung Hsing University, in 2011. He is currently a Senior Engineer with Hiyoung Telecommunication Technology Company, Ltd., where he works on smartphone modem protocol development.



PI-CHUNG WANG received the M.S. and Ph.D. degrees in computer science and information engineering from National Chiao Tung University, Taiwan, in 1997 and 2001, respectively. He was with Telecommunication Laboratories, Chunghwa Telecom, from 2002 to 2006. He joined the Department of Computer Science and Engineering, National Chung Hsing University, Taiwan, in 2006, where he has been a Professor since 2014. His research interests include IP lookup and classification algorithms, scheduling algorithms, protocols in local area networks, and wireless sensor networks.