GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers

Yanbiao Li*, Dafang Zhang*, Alex X. Liu† and Jintao Zheng*
*College of Information Science and Engineering, Hunan University, Changsha, China
{lybmath_cs, dfzhang, zhengjintao}@hnu.edu.cn
†Department of CSE, Michigan State University, East Lansing, USA
alexliu@cse.msu.edu

ABSTRACT

Recently, the Graphics Processing Unit (GPU) has been proved to be an exciting new platform for software routers, providing high throughput and flexibility. However, it is still a challenging task to deploy some core routing functions into GPU-based software routers with anticipatory performance and scalability, such as IP address lookup. Existing solutions have good performance, but their scalability to IPv6 and frequent updates are not so encouraging.

In this paper, we investigate GPU's characteristics in parallelism and memory accessing, and then encode a multi-bit trie into a state-jump table. On this basis, a fast and scalable IP lookup engine called GPU-Accelerated Multi-bit Trie (GAMT) has been presented. According to our experiments on real-world routing data, based on the multi-stream pipeline, GAMT enables lookup speeds as high as 1072 and 658 Million Lookups Per Second (MLPS) for IPv4/6 respectively, when performing a $16M$ traffic under highly frequent updates ($70,000$ *updates/s*). Even using a small batch size, GAMT can still achieve 339 and 240 *MLPS* respectively, while keeping the average lookup latency below 100 μs. These results show clearly that GAMT makes significant progress on both scalability and performance.

1. INTRODUCTION

Due to the ever-increasing link rate, modern routers must process packets with a high throughput. Besides, new protocols and applications have been widely used in network nowadays, such as IPv6, network virtualization, streaming media and so on. Thus, modern routers are also required to be re-configurable and easily programmable, which prevents hardware-based solutions being not so adaptable due to the flexibility. Furthermore, with the research progress in future networks, some emerging architectures [1, 3] come to the fore. To provide smooth transitions to these new architectures in near future, defining and controlling the network on basis of software are very important.

1.1 Summarize of Prior Arts

General software routers have suffered from serious challenge in performance. The throughput bottlenecks of these routers are always caused by some core routing functions, such as IP address lookup, which needs to compare the input address against all prefixes stored in the Forwarding Information Base (FIB) to make a Longest Prefix Matching (LPM).

Major software-based solutions to LPM fall into two categories. Generally, hash-based solutions [6, 9] provide relative high throughput. However, the prohibitive requirements for high-bandwidth memory, false positive rates and problems resulted by hash conflicts have impeded their applications in practice. The other type of solutions improves flexibility by employing some tree-like data structures [18], such as a trie. Although optimized by many techniques [21, 7, 22], they are still difficult to reach the speed level provided by the Ternary Content Addressable Memory (TCAM)-based table lookup [23] or Static Radom Access Memory (SRAM)-based pipeline architectures [10, 11].

Fortunately, the GPU is becoming an emerging platform for high performance general-purpose computing [5]. Some GPU-based software routers have been proposed to achieve very high throughput. To work on such routers, the IP lookup engine still faces enormous challenges in terms of high performance, scalability to large tables, new protocols and also new applications. Most previous studies have focused on either entire framework of software routers [8, 28] or comprehensive performance of multiple routing functions [14]. They all treat the routing table as static and fail in dealing with update overhead. However, the peak of real-world update frequency has exceeded $20,000$ *updates/s* [1] and is still increasing. Such frequent updates lead to competition of computing resources with lookup process. As a result, the lookup throughput may be affected. Particularly, in some new applications, such as the virtual router platform [12] and the OpenFlow switch [13], the update is more frequent. Accordingly, update overhead must be considered in the lookup engine design. In view of this, J. Zhao et al. [27] presented a GPU-Accelerated Lookup Engine (GALE), providing fast lookup and efficient update. However, the proposed engine is only applicable for IPv4, and its throughput declines sharply with the increase of update frequency.

1.2 Our Approach

In this paper, we aim to design a high performance and more scalable IP lookup engine for GPU-based software routers. Specifically, three goals will be reached: 1) Scale to IPv6 smoothly. 2) Keep stable lookup throughput under highly frequent updates. 3) Improve lookup performance with latency controlled.

In order to address the above issues, we present a fast and scalable IP lookup engine, GPU-Accelerated Multi-bit Trie (GAMT). Our basic idea is as follows: Given an FIB, we first build a multi-bit trie, and then encode it into a state-

[1]This is based on our experimental data collected from the RIPE RIS Project [2].

978-1-4799-1640-5/13/$31.00 ©2013 IEEE

jump table, which can be easily deployed onto GPU's global memory as a 2-D array to provide fast IPv4/6 lookup.

Meanwhile, a multi-bit trie is maintained in main memory of the CPU for off-line updates. Then, we introduce an efficient mechanism for completely parallel on-line updates. The mechanism reduces the disruption of the update to the lookup, and makes GAMT keep stable throughput under frequent updates.

Furthermore, the performance of GAMT is enhanced by employing a multi-stream pipeline for efficient batch processing. Meanwhile, to achieve a reasonable latency, a small batch size is required. Even in this case, GAMT also works well.

1.3 Key Contributions

This paper makes three key contributions. Firstly, we propose novel approaches to encode a multi-bit trie into a GPU-based state-jump table, and to optimize its structure on the basis of GPU's characteristics. According to the experiments, our proposed scheme, GAMT, is proved faster than GALE in practice. Besides, being further accelerated by the multi-stream technique, GAMT can also achieve a desirable throughput even with a small batch size.

Secondly, an efficient update mechanism for GAMT is introduced. It reduces update's disruption to lookup and promotes the update parallelism on the GPU. Then, the system architecture is designed as an efficient IP lookup engine, which can be deployed into GPU-based software routers as an additional plug-in.

At last, the performance of GAMT is evaluated by using real-world FIBs and corresponding update traces. Meanwhile, we have compared our proposed scheme with GALE and the multi-bit trie implemented on the many-core CPU. We investigate the influence on throughput exerted by several metrics, involving the batch size, the level of multi-bit trie, update frequency and also GPU's kernel configuration. Finally, the superiorities of GAMT are demonstrated in a comprehensive view.

The rest of this paper is organized as follows. Section 2 introduces some background knowledges and reviews related works. Section 3 presents the detail of GAMT. Section 4 proposes some optimization techniques. Section 5 describes the evaluation methodology and experimental results. Finally, Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 Multi-bit Trie

As shown in Fig. 1(b), the LPM on the multi-bit trie is performed by traversing from the root to some leaf step by step. One or more bits of the input address may be processed in each step according to the stride array[2]. In order to implement a simple pipeline, each level of the multi-bit trie is mapped onto a stage of an SRAM-based pipeline.

A. Basu et al. [4] construct a balanced multi-bit trie and then deploy it onto the Filed-Programable Gate Array (FPGA) platform for pipeline implementation. Based on characteristics of such platform, an efficient update mechanism, named *Write Bubble*, is also proposed to support fast incremental updates. The mechanism packages all route updates into a group of bubbles. Each bubble contains a sequence of $(stage, location, value)$ triples, with at most one triple for one stage of the pipeline. After off-chip updates, all produced bubbles are sent to the pipeline one by one. Since different SRAMs can be accessed independently, all triples in a bubble can be processed in parallel.

As the GPU has very different characteristics from the FPGA, in this paper, we structure the multi-bit trie in a different shape, and propose a different update mechanism to utilize GPU's vast computing powers. But some of their basic ideas are also useful for us. Such as representing memory modifications by triples, and reducing update's disruption to the lookup through off-line updates.

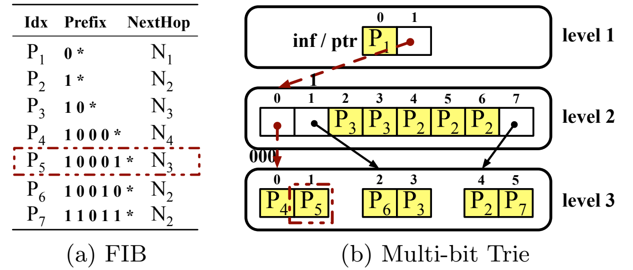


Figure 1: Perform LPM on the multi-bit trie, which has 3 levels and its corresponding stride array is $\{1, 3, 1\}$.

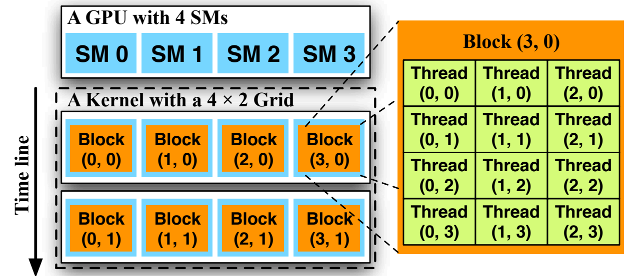


Figure 2: A kernel configured with 8 threads blocks is executed on a GPU with 4 SMs.

2.2 CUDA Programming Model

Compute Unified Device Architecture (CUDA), as the basic platform for our approach, is a scalable parallel computing platform and programming model for NVIDIA GPUs [16].

As shown in Fig. 2, GPU's hundreds of cores are organized as an array of Streaming Multiprocessors (SMs). Each SM which consists of several Streaming Processors (SPs) works in Single Instruction with Multiple Threads (SIMT). In CUDA, the function called by the CPU but executed on the GPU is called *kernel*, which will be executed in one or more SMs, and is always configured with a grid of threads blocks.

2.2.1 Coalescence of Global Memory Accesses

Global Memory, as GPU's device memory, is used to exchange data between the GPU and the CPU. It can be accessed by all executing kernels. Moreover, It is always accessed via 32-, 64-, or 128-byte memory transactions. All

[2]A stride is the number of bits should be processed in a step. The stride array is composed of all strides for all steps.

memory accesses produced by threads of a *warp*[3] may coalesce into a few memory transactions. Since accessing the global memory results in a long latency, it is a useful strategy to optimize performance on the GPU by reducing the number of produced memory transactions.

2.2.2 Overlapping Behaviors on the GPU

Thanks to GPU's massive parallelism, overlapping more behaviors may exploit greater computing power. Actually, active warps in a SM are naturally overlapped. When a warp is waiting for memory access, other active warps can be scheduled to perform some computing tasks. Such a warp schedule is done by hardware, incuring zero context-switch overhead. What we should do is to assign tasks carefully. Besides, by using the page-locked memory on the CPU, data transfers and kernel executions within different streams[4] can be overlapped. On some GPUs, such as the one we used, even different kennel executions can be overlapped.

2.3 GPU-Accelerated IP Lookup Engine

Recently, some novel GPU-based software routers have been proposed to provide very high performance. Such as *PacketShader* [8], the first to demonstrate the potentials of GPUs in the context of multi-10G software routers, and *Hermes* [28], an integrated CPU/GPU micro-architecture for QoS-aware high speed routing. With their contributions on the general architecture design of software routers, in this paper, we only focus on the detail of IP lookup engine.

As a high-speed IP lookup engine, *GALE* [27] has pushed the speed of IPv4 address lookup to the top level in theory (the lookup's time complexity is O(1)), by means of a large *direct table*, which stores all possible prefixes no longer than 24. On the other hand, it also provides efficient algorithms to map update operations to GPU's parallel architecture. However, in *GALE*, without additional operations, breaking updates' order may result in uncertain problems. Therefore, table modifications produced by different updates can not be processed in parallel, that's why *GALE*'s throughput declines sharply with increasing update frequency.

3. GPU-ACCELERATED MULTI-BIT TRIE

3.1 Encoding Rules and Lookup Approach

Multi-bit trie [21, 19] is a good choice for fast IPv4/6 lookup with memory controlled. In order to compress data structure and to simplify lookup logic for efficient implementation on the GPU, we transform a multi-bit trie into a state-jump table, by encoding each unit of a trie node[5] into a 32-bit integer, according to the following rules:

1. The code of each unit takes 4 bytes, ensuring that multiple units can just fulfill a single memory transaction. As shown in Fig. 3, the upper 3 bytes of a unit's code represent an *inf* code, while the least significant byte represents a *jump* code.

2. For a unit that stores a next hop, we set its *jump* code to 0, and its *inf* code is set as the index number of the stored next hop information in the next hop table.

[3]A warp is the basic schedule unit and consists of 32 threads.
[4]A stream is a sequence of operations executed in order.
[5]A trie node with stride s has 2^s units in a leaf-pushed multi-bit trie.

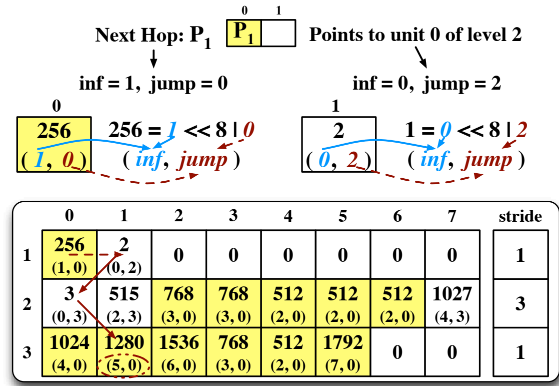


Figure 3: By encoding each trie node into several units, we can get a state-jump table from the multi-bit trie shown in Fig. 1(b). The process of looking up 10001* on it is then transformed into 3 jumps.

Accordingly, our scheme supports at most $2^{24} = 16M$ different next hop information.

3. For a unit that stores a child pointer, it is encoded on basis of the child it points to. Its *jump* code is set as the index number (starts from 1) of the level this child locates in, while its *inf* code is set as this child's offset in that level (in units). Therefore, our scheme supports at most $2^8 - 1 = 255$ levels, where each level consists of $16M$ units at most.

Based on above rules, a multi-bit trie can be transformed into a simple state-jump table (Fig. 3), in which a state is just a unit. Given an IP address, the lookup is simplified as a series of state jumps. In each step, we read several bits of the address according to the stride corresponding to this step, and add them as an integer to the *inf* code of the current unit's code (use the default code 1 (0, 1) in the first step), producing a new code as a result. Then, this new code guides us jump to another unit. This process repeats until encountering a unit whose *jump* code is 0. Then, the next hop index represented by its *inf* code is returned as the result of this lookup.

For instance, to lookup address 10001$*$, since the strides array is $\{1, 3, 1\}$, we read the first bit of the address (1), and add it to the default code's *inf* code in the first step. A new code $(0 + 1,\ 1)$ is generated, which guides us jump to unit 1 of level 1, whose code is 2 (0, 2). Then, we read next 3 bits of the address (000), and add them to the *inf* code of the current unit. The generated code $(0 + 0,\ 2)$ guides us jump to unit 0 of level 2, whose code is 3 (0, 3). With its *inf* code (0) added by the last bit of the address (1), a new code $(0 + 1,\ 3)$ is generated and guides us jump to unit 1 of level 3. Since its code is 1280 (5, $\mathbf{0}$), the lookup is terminated with a matched result of P_5.

To implement this algorithm on GPU and to achieve higher performance, a batch of destination addresses will be processed by a kernel in parallel. Each address is mapped to a thread (but a thread may receive two ore more addresses.). Take the address locality in real network traffic into account, the tasks are assigned in a "jump" way to make continuous threads perform continuous requests. This algorithm is described in pseudo-code in Algorithm 1 as a GPU kernel.

Algorithm 1: Lookup kernel.

```
/*input a batch of destination addresses.*/
Input: AddrArray, BatchSize
Output: ResArray
/*index of the first request processed in this thread.*/
idx = blockIdx.x * blockDim.x + threadIdx.x;
idx_step = gridDim.x * blockDim.x;
while idx < BatchSize do
    addr = AddrArray[idx];/*read request*/
    len = 32;/*for IPv6, it should be 64*/
    jump = 1;
    inf = 0;
    while 0 ≠ jump and len > 0 do
        len −= StridesArray[jump];
        inf += addr >> len;
        addr &= (1 << len) − 1;
        code = GAMT.Array[jump × ArrayWidth + inf];
        jump = code & 0xff;  inf = code >> 8;
    end
    ResArray[idx] = inf;/*write result*/
    idx += idx_step;/*go to the next request.*/
end
```

Algorithm 2: Prepare bubbles for on-line updates.

```
/* number of bubbles in a batch.*/
Input: BatchSize
Output: BubbleList
BubbleNum = BatchSize;
for s = 1 to TreeLevel do
    if BubbleNum=0 then
        break;
    end
    s_num =bubblesForStage[s];
    for i = s_num to 1 do
        bubble =WB[s][i − 1];
        bubble.value=Trie[bubble.stage][bubble.location];
        BubbleList[− − BubbleNum] = bubble;
        if BubbleNum=0 then
            break;
        end
    end
    bubblesForStage[s] = i;
end
return BubbleList;
```

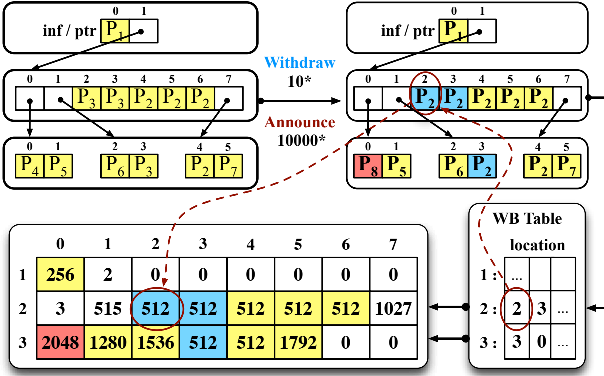


Figure 4: Issue grouped bubbles to the GPU.

3.2 Update Mechanism

Write Bubble is an efficient update mechanism that minimizes the disruption to the lookup. But it's designed for the SRAM-based pipeline on FPGA, in which the parallelism is realized by concurrent accesses to different SRAMs. In that case, all memory modifications grouped in a bubble can be processed in parallel.

However, a GPU has very different characteristics. Firstly, its hundreds and thousands of cores provide massive parallelism, but it works in the SIMT mode. So, to achieve higher update performance, a batch of update requests should be performed on the GPU by a group of threads, which calls for more parallelism of the update. Besides, due to the coalescence of global memory access on the GPU, accessing adjacent memories by threads of a warp always leads to higher performance. In view of this, a novel update mechanism is designed for GAMT.

To reduce the disruption of update to the lookup, a backup of the main structure should be reserved for off-chip updates. In our case, the original multi-bit trie is still maintained on the CPU after being encoded and deployed onto the GPU. As shown in Fig. 4, when a route update arrives, we first update this original trie, and collect all produced memory modifications for later on-chip updates.

In our scheme, memory modification is also represented by a triple $< stage, location, value >$, named a bubble. Obviously, it's not available for batch processing that two bubbles toward the same unit. Thus, the overlap of any two bubbles should be eliminated to ensure completely parallel on-line updates. Besides, performing adjacent memory modifications, but not those toward different stages (as in *WriteBubble*), by threads of a warp will benefit more from GPU's global memory coalescence.

Therefore, only the locations of all units which need to be modified are stored uniquely (possible overlap is eliminated) in the Write Bubble Table (WB Table), and are grouped by the level index. Then, as shown in Fig. 4, in order to form entire bubbles, the latest values for all bubbles will be fetched from the original multi-bit trie before these bubbles being sent to the GPU. Algorithm 2 describes the algorithm of preparing bubbles in pseudo-code.

3.3 Architecture Overview

As shown in Fig. 5, our system architecture is based on CUDA, in which, all of the program codes are divided into two cooperative parts: the *Host* and the *Device*, which are executed respectively on CPU and GPU.

In the *Host*, a control thread, as the system heart, manages CPU's working threads to deal with route updates and lookup requests, by unitizing computing resources of both the CPU and the GPU. On the other hand, the encoded multi-bit trie is stored on the *Device* (GPU) as a 2-D array, leaving a backup on the CPU in tree shape. Besides, a Next-Hop Table (NH Table) that contains all entire next hop information is stored on the CPU. The purpose is to avoid storing complicated next hop information (such as multi-next-hop [26]) on the GPU. What's more, in this way, any route update that requires to modify an existing prefix will only modify the NH Table, which reduces the disruption of update to the lookup as well.

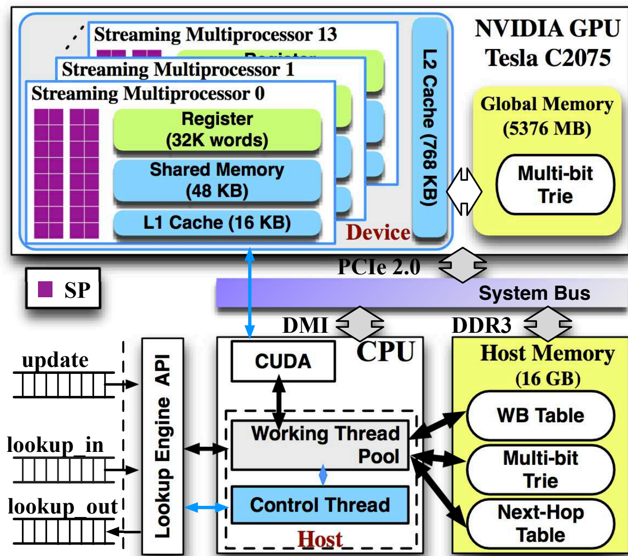


Figure 5: System architecture of GAMT.

As mentioned above, the lookup is performed on the GPU. Meanwhile, the route updates are executed on the CPU firstly. The produced memory modifications are then reflected to the main structure on the GPU. Moreover, to achieve higher performance, lookups and on-chip updates are performed in parallel through batch processing.

Actually, the details of a task should be determined before assigning to threads. It can be realized by the control thread through obeying some user-defined configurations or receiving some specific orders from the host router with *Lookup Engine APIs*. The task detail include what (lookup or on-chip update), when and how many (the batch size). Then, the control thread activates an idle thread to handle this task. After that, the activated thread will prepare data and call a kernel to finish the task on the GPU in data parallel way.

4. PERFORMANCE OPTIMIZATION

4.1 Possibility of being Faster than GALE

In GALE, one lookup requires only one global memory access. While in GAMT, one lookup is transformed into several state jumps. Each step consists of a global memory access and a few operations on integers. It seems obvious that GALE is faster than GAMT. However, this may not actually be the case due to the coalescence of global memory accesses on GPU. In this section, we will show the possibility of GAMT being faster than GALE with some analysis and a simple example. Actually, in section 5, our experiment results demonstrate that GAMT is truly faster than GALE in most cases.

The operations on integers are far faster (almost multi-hundred times [15]) than global memory access on GPU. Therefore, only the performance of memory access will be discussed in this section. Actually, all memory accesses by threads of a wrap are coalesced into a few transactions by the device. These transactions will be orderly processed. Generally, considering random accesses, a larger access range always means a more scattered access pattern, which will result in more transactions.

For 32 simultaneous lookups within a warp, GALE will produce y memory transactions. While GAMT produces x memory transactions in each step of concurrent state jumps. Since the size of each level of the trie in GAMT is far less than that of the direct table in GALE, x may be smaller than y. What's more, the sum of transactions produced in all steps ($\sum x$) may even be smaller than y in some cases. Figure 6 shows a simple example.

In this example, the memory size of a single transaction is supposed to be 32 bytes[6] (that's to say 8 units can be accessed in a single transaction), and all produced memory accesses are supposed to be distributed in a balanced way. As shown in Fig. 6, four memory transactions are produced for 32 concurrent lookups in GALE. While in GAMT, although one lookup requires 3 jumps, 32 concurrent lookups only produce one memory transaction in each step. Consequently, GAMT only produces three transactions in total. That is to say, performing these 32 lookups by GAMT will cost less time than that by GALE. Although the reality is more complicated than the case for only one warp, this example truly shows the possibility of GAMT's being faster than GALE. What's more, it also offers useful guidance for optimizing the structure.

4.2 Optimized Multi-bit Trie

For a multi-bit trie, it is critical to calculate the stride array after the number of trie levels is specified. With different goals, we can calculate stride array in different manners: minimizing the total memory consumption [21] or minimizing the maximal level to reduce update overhead with memory controlled for SRAM-based pipeline on the FPGA [4]. In our case, we have different requirements.

Given the number of levels for the target trie, the array width (the size of the maximal level in units) should be minimized for memory optimization and improvement of cache hits in memory access. As shown in Fig. 6, the performance of lookup may be affected by the size of each tree level. Therefore, the number of memory transactions possibly produced in each level[7] should also be taken into account. The secondary optimal objective is to minimize the sum of possible produced memory transactions in all levels.

Supposing that four units can be accessed in a single memory transaction, let's compare two multi-bit tries shown in Fig. 1(b) and Fig. 7(a) respectively. As depicted, for these two tries, the maximal levels have the same size (8 units), but the total numbers of possible produced memory transactions are different. For the one shown in Fig. 7(a), it is only 4, while for the other it is 5. In another word, the multi-bit trie shown in Fig. 7(a) may have better lookup performance on the GPU.

Actually, there are many choices of stride arrays to achieve our first objective (minimize the array width). Among all these choices, what we chose should make it easier to realize our secondary objective. Such a algorithm is described in Algorithm 3.

4.3 Delete in Lazy Mode

[6]In practice, it is 128 bytes for the GPU we used.

[7]Suppose the size of this tree level is x (in units), and y units can be accessed in a single memory transaction, then, this number can be calculated as $\lceil x/y \rceil$

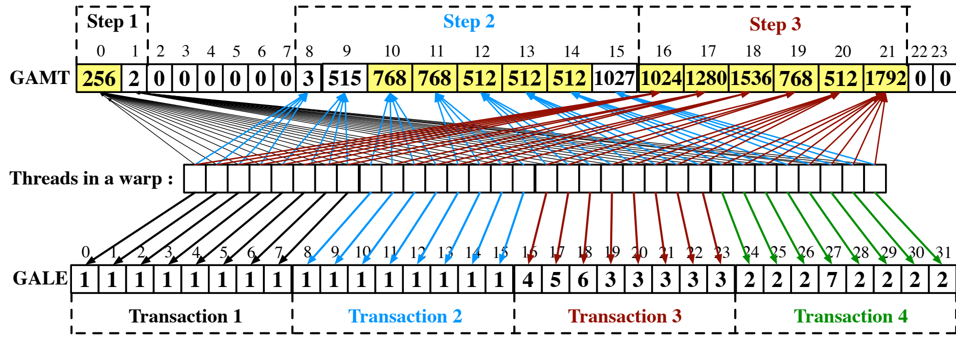
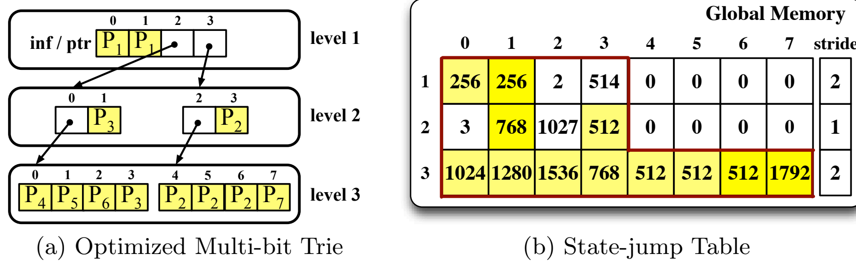


Figure 6: Memory transactions produced by processing 32 lookups within a warp.



(a) Optimized Multi-bit Trie (b) State-jump Table

Figure 7: The multi-bit trie built according to Algorithm 3 and its corresponding state-jump table.

Algorithm 3: Minimize the array width with total possible transactions controlled.

Input: NodesInLevel
Output: StridesArray

/*H is the objective trie level*/
/*L is the level of the original uni-bit trie.*/
for $s = 1$ **to** H **do**
  for $l = 1$ **to** L **do**
    $opt_trans = opt_size =$ INF;
    for $k = 1$ **to** $L - 1$ **do**
      $cur_size =$ NodesInLevel$[k] \times 2^{l-k}$;
      $cur_trans = \lceil cur_size/T \rceil$;
      $max_size =$ Max$(f[s-1][k], cur_size)$;
      if $max_size < opt_size$ **then**
        $opt_size = max_size$;
        $opt_trans = g[s-1][k] + cur_trans$;
      end
      else
        if $opt_size = max_size$ **then**
          $opt_trans =$ **min**$(opt_trans, g[s-1][k] + cur_trans)$;
        end
      end
    end
    $f[s][l] = opt_size$; $g[s][l] = opt_trans$;
  end
end
/*calculate the stride array through a backtrack*/
StrideArray = CalStrideArray(f, g, H, L)
return *StrideArray*;

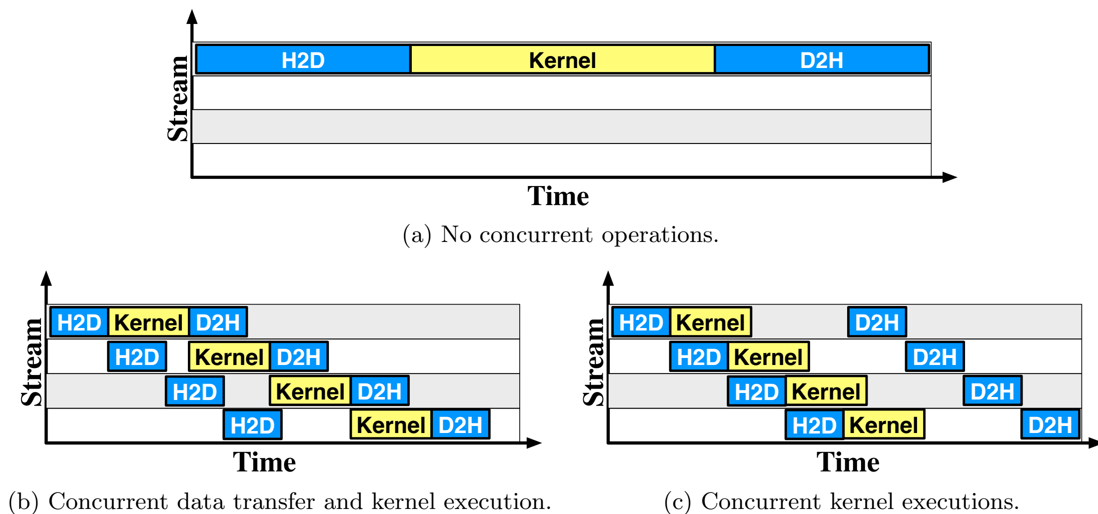
In order to optimize the performance of update, the prefix deletions will be performed in *lazy mode* [20] without any backup modifications. In this way, only the unit corresponding to the deleting prefix is modified. However, it is potentially dangerous. Some levels of the multi-bit trie on the GPU may reach the array bound after a long term of updates. It will pose a demand to rebuild the whole structure at that time.

Consequently, some "head room" are reserved for each level and a threshold to the array width is pre-defined. Once it has been reached by any level, the multi-bit trie is rebuilt off-line, and the generated structure is then sent to the GPU as several bubbles. To minimize rebuilding's influence on the lookup, we store two multi-bit tries on the GPU, one for lookup and the other for rebuilding. Interchange of their roles may not incur any problem, as the right one is chosen when a specified kernel is launched to process lookups. Actually, according to our experiments, GAMT's memory cost on the GPU is small enough to allow us store an additional backup for it. Besides, its memory growing rates is also very slow, which ensures the rebuilding process quite infrequent.

4.4 Multi-Stream Pipeline

For GPU-accelerated applications, batch processing is the basic rule for performance optimization. But the cost is extra delay resulted by waiting for enough requests to fulfill a batch. Fortunately, such a throughput-latency dilemma can be resolved to some extent, by the multi-stream technique [25].

Three sequential steps are required to perform a batch of lookups on basis of CUDA, which are stated as follows: 1) Transfer requests from the Host to the Device (H2D). 2) Kernel executions. 3) Transfer results from the Device to the Host (D2H). However, if two or more streams are used,



(a) No concurrent operations.

(b) Concurrent data transfer and kernel execution.

(c) Concurrent kernel executions.

Figure 8: Two types of concurrent operations in the multi-stream pipeline.

some behaviors within different streams can be overlapped. It can not only improve the throughput but also reduce the average latency (as shown in Fig. 8).

By means of page-locked memory in the host, data transfer and kernel execution within different streams can be overlapped. To benefit from this, all lookup requests should be divided into several groups, each of which should be sent to independent kernels that are assigned to different streams. Figure. 8(b)) shows an example in this way: four streams work as a 3-stage pipeline to improve the whole performance.

Actually, two kernels in different streams can also be executed in parallel. As shown in Fig. 8(c), kernel executions are overlapped, which helps us to take more advantages of GPU's parallel architecture. However, memory copies from the device to the host are blocked, which makes the processing time even longer than that without concurrent kernel executions (Fig. 8(b)).

Although activating multiple streams is an efficient strategy for performance optimization, it's really device sensitive. Both concurrent data transfer and kernel execution, and concurrent kernel executions are only supported by some devices with specified properties [16]. Furthermore, the blocking problem in our example can be solved by utilization a device with the computing capability higher than 3.0. Therefore, we implement this optimization strategy into several versions, a proper one among which is selected according to the device specification in a self-adaption way.

5. EXPERIMENTAL EVALUATION

5.1 Evaluation Methodology

In this section, we evaluate GAMT's lookup performance, update overhead and the comprehensive performance, and then demonstrate its superiorities in comparison with GALE and a CPU-based solution, Many-core Accelerated Multi-bit Trie (MAMT). Furthermore, we also evaluate GAMT's performance for IPv6 and on other devices to show its scalability.

5.1.1 *FIB, Update Trace and Traffic*

We collected 4 public BGP routing data sets from the RIPE RIS Project [2], each having an IPv4 FIB, an IPv6 FIB, and a whole day's update traces. For $rrc12$, we also collected a week's update traces. Table 1 shows characteristics of all data sets. To measure lookup performance, we take the similar method as [27] to generate traffics from FIBs. Moreover, we also generate traffics in a completely random way.

5.1.2 *Evaluation Platform*

We implement MAMT on basis of OpenMP 2.0, while GALE and GAMT are implemented based on CUDA 5.0. Most of the experiments are running on a Dell T620 server, and two extra experiments are conducted respectively on a desktop PC and a notebook. In this way, the comprehensive performance on different platforms can be evaluated. Table 2 shows some specifications of these platforms.

5.1.3 *Major Metrics to Measure*

To evaluate lookup performance, five factors are chosen for different experimental configurations. In each case, we measure the lookup performance with Million Packets Per Second (MLPS). These metrics are listed as follows. 1) *batch size*, it denotes the number of requests in a batch and ranges from 2^0 to 2^{25}. 2) *routing data*, we have 4 data sets and in total 8 FIBs for IPv4/6. 3) *traffic type*, we have two types of traffics, the traffic generated completely randomly is called *Radom* in short, while the traffic generated from a FIB is called $Table$. 4) *tree level*, it denotes the height of the multibit trie and ranges from 4 to 20 for IPv4 and ranges from 8 to 30 for IPv6. 5) *CUDA configuration*, it involves the number of streams ($1 \sim 24$), the number of threads blocks ($16 \sim 64$) for one kernel and the number of threads ($128 \sim 1024$) in each block.

For update overhead, the update mechanism of GAMT is evaluated in lazy mode. The mechanisms whether separate the next hop table are called GAMT_S and GAMT_T respectively. Two metrics are measured: 1) *off-line update speed*, the speed of performing off-line updates on the CPU. 2) *on-line memory accesses*, the number of memory loads/writes produced when performing on-line updates on the GPU.

Table 1: Routing data

Data			Route Table[a]		Update Trace[b]			
name	location	type	prefix	length	stamp	total	max	avg
rrc11	New York (NY), USA	v4	442176	32	56207	1177425	6031	20.95
		v6	11656	64	37080	207287	439	5.59
rrc12	Frankfurt, Germany	v4	450752	32	63524	4049260	19854	63.74
		v6	11841	64	54727	1260126	3520	23.03
rrc13	Moscow, Russia	v4	456580	32	61128	2025239	15104	33.13
		v6	11635	64	107244	23102	774	4.64
rrc14	Palo Alto, USA	v4	446160	32	78175	1388217	5210	17.76
		v6	11719	64	55858	247228	334	4.43

[a] collected on Jan.1, 2013. *prefix* denotes the total number of prefixes and *length* represents the maximum length of all prefixes.
[b] collected on Jan.1 2013 from 0:00 a.m. to 23:55 p.m. *stamp* denotes the total number of time stamps. *total* denotes the number of updates in this day. *max* denotes the maximum number of updates in one stamp. *avg* denotes the average number of updates per stamp.

Table 2: Experiment Platform Specification

	Item	Specification	Cost [a]
Server	CPU	1 Intel Xeon E5-2630 (2.30GHz, 6Cores)	$640.00
	RAM	2 RDIMM 8GB (1333 MHz)	$138.00
	GPU	1 NVIDIA Tesla C2075 (1.15GHz, 5376MB, 14×32 Cores, Capability 2.0)	$1999.00
PC	CPU	1 AMD Athlon(TM)II X2 240 (2.80GHz, 2Cores)	$49.00
	RAM	1 DDR3 4GB (1333 MHz)	$36.00
	GPU	1 NVIDIA GeForce GTS 450 (1.57GHz, 512MB, 4×48 Cores, Capability 2.1)	$95.00
NoteBook	CPU	1 Intel Core(TM) i7-2630QM (2.00GHz, 4Cores)	*unavailable*
	RAM	2 DDR3 4GB (1333 MHz)	$72.00
	GPU	1 NVIDIA GeForce GT 550M (1.48GHz, 512MB, 2×48 Cores, Capability 2.1)	*unavailable*

[a] all prices are from *http://www.newegg.com*. and the prices of CPU and GPU for notebook are unavailable.

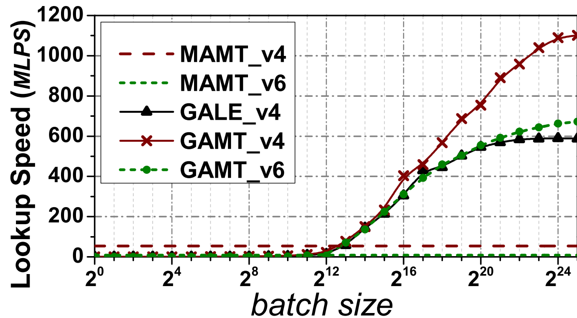


Figure 9: Lookup speed VS. batch size

At last, the comprehensive performance is evaluated, which includes lookup throughputs with controllable latencies, lookup throughputs under increasing update frequencies, memory efficiency on the GPU, performance for IPv6 FIBs and on other two GPUs.

5.2 Lookup Performance

5.2.1 Batch Size

Firstly, a 6-level (for IPv6 FIBs, we chose level 20) multibit trie and GALE's direct table are constructed on $rrc12$. Then, with the batch size increasing from 2^0 to 2^{25}, we measure lookup performance by randomly generated traffics with different CUDA configurations, and report the highest speed for each configuration.

Figure 9 shows clearly that lookup speeds of GPU-based solutions will be seriously affected by the batch size. For IPv4 FIBs, if the batch size is below $8K$, both GAMT and GALE are slower than MAMT, whose speed is almost 50 *MLPS* accelerated by 6 cores and 12 threads. However, GAMT enables higher speed than GALE in most cases. After the batch size beyond $256K$, GALE's lookup speed rises slower and slower, and then waves around 580 *MLPS*. By contrast, GAMT's lookup speed also increases sharply until beyond 1100 *MLPS*, which achieves speedups by 1.9 and 20 to GALE and MAMT respectively.

For IPv6 FIBs, GALE doesn't work. While GAMT can also achieve a lookup speed as high as 680 *MLPS* with a big batch size ($16M$). On the other hand, IPv6 address lookup is always slower than IPv4 address lookup in the same condition. That's because an IPv6 address takes 8 bytes[8], as twice as for an IPv4 address. So, it costs more time to copy IPv6 requests to GPU. What's more, a multibit trie built on an IPv6 FIB always requires more levels to control the memory consumption.

From the above results, a big batch size is required to make full use of the powerful computing capability of GPU. So, we chose $16M$ as the batch size to evaluate the lookup performance of GALE and GAMT.

[8] An IPv6 address takes 16 bytes, but only 8 bytes are used in forwarding. [24]

5.2.2 Routing Data and Traffic Type

As shown in Fig. 10, unlike the CPU-based solution, GAMT and GALE are all obviously sensitive to the type of traffic. Actually, they achieve higher speeds if $Table$ is used for test. It is because the address locality is more obvious, which produces more adjacent global memory accesses on the GPU. As a result, less memory transactions will be produced. On the other hand, GAMT always has better performance than GALE in the same condition, achieving speedups at most 2.0 and 1.2 for *Random* and $Table$ respectively. Besides, from $Table$ to *Random*, GAMT's performance is more stable. In fact, the lost of speed is only 11% $\sim$ 14%, which is almost 50% for GALE.

5.2.3 Tree Level

In this section, some experiments are conducted to evaluate the effects of the structure to the lookup performance. We construct the Multi-bit Trie (MT) on $rrc12$'s IPv4 FIB with different tree levels, and implement them on the CPU (MAMT) and GPU (GAMT) respectively. The lookup speed and the memory cost per prefix are measured in each case. Certainly, GALE's speed and memory efficiency are all constants in these cases.

Figure 11 shows an interesting scenario: for the CPU-based implementation, more tree levels result in lower lookup speed, while GAMT's highest speed is achieved by the 6-level multi-bit trie. Generally, it requires more memory accesses to finish one lookup on a tree with more levels. However, on the GPU, due to the characteristic of memory coalescence, the size of each tree level (in bytes) will also affect the performance in each step of parallel lookups. Therefore, for GAMT, less tree levels and smaller array width will result in higher lookup speed. As a result, the memory cost per prefix[9] will have an approximate opposite curve as the lookup speed, which is also demonstrated in Fig. 11.

5.2.4 CUDA Configuration

Figure 12 shows the lookup speeds of GAMT with different CUDA configurations. If only one stream is used, 88% of the measured speeds fall into an interval of 300 $\sim$ 400 *MLPS*. But, if accelerated by two streams, 55% of speeds are between 400 *MLPS* and 500 *MLPS*, and other 34% fall into the next higher interval. What's more, the highest speed (522 *MLPS*) is 1.5 times as using only one stream.

On the other hand, no matter how many streams are used, a light weight configuration (say $block < 32$ and $thread < 256$) makes the speeds lower than half of the maximum speed. In this case, either adding blocks or enlarging the block size can effectively improve the performance. However, if a kernel has been configured with enough blocks and each block has been fulfilled with enough threads, adding blocks or threads continuously always makes the performance fluctuates. Therefore, despite activating more streams, it is also very important to choose a proper configuration for each kernel.

5.3 Update Overhead

To support update on the GPU, both GALE and GAMT require to process updates on the CPU first. We replay a weeks' update traces of $rrc12$'s IPv4 FIB and a whole day's update traces for all IPv4 FIBs, to measure their on-line update speeds with Million Updates Per Second (MUPS).

Figure 13 shows that the performance of off-line update for our scheme is not so good as that of GALE. It is caused by additional time that is spent on managing the WB Table. However, with a separated NH Table (GAMT_S), the speed of off-line update is promoted, even being higher than GALE's in some cases. Actually, GAMT_S enables an off-line update as high as 2.7 *MUPS*, which is far faster than peak update frequencies of these 4 FIBs (see Table 1).

The global memory access is measured for on-line updates. It reflects the speed of on-line updates and the power consumption of GPU as well. In fact, GALE requires global memory loads and writes to perform on-line updates. But all bubbles produced in our mechanism represent only memory writes. As shown in Fig. 14, GAMT_T produces far less memory accesses than that of GALE with a reduction by 94.7% $\sim$ 96.6%. The reduction is still 90.3% $\sim$ 92.5% even if GALE's memory loads are ignored. Such a superiority is enhanced by processing all prefix modifications in a separated next hop table in GAMT_S, the on-line update overhead is further reduced by 78.7% $\sim$ 88.2%.

5.4 Comprehensive Performance

5.4.1 Scalability to IPv6

The flexible data structure makes GAMT scale well to IPv6. It is one of the most important features of GAMT superior to GALE. Figure 15 demonstrates the same scenario as for IPv4: as the tree level is increasing, the memory cost per prefix decreases first and then starts increasing, while GAMT's lookup speed curve varies in an opposite way. In fact, the 20-level GAMT achieves the highest speed (658 *MLPS*).

5.4.2 Scalability to Frequent Updates

Since the frequency of route update is continuously increasing, it's important for an IP lookup engine to maintain a high lookup performance under highly frequent updates, especially in virtual routers [12] or the OpenFlow switch [13].

Figure 16 shows that the lookup speed of GALE will decrease by more than 80% if the update frequency reaches $70K$ *updates/s*. By contrast, the lookup speed of GAMT just decreases by 4% and its speed can achieve 972 *MLPS* under so frequent updates. For the IPv6 FIB, the descent of GAMT's lookup speed is only 8%. Consequently, it is clear that GAMT scales to frequent updates very well with the help of our efficient update mechanism. This is GAMT's another characteristic superior to GALE.

5.4.3 Performance on Other Devices

Figure 17 shows the lookup speed of GALE and GAMT for $rrc12$'s IPv4 FIB on other two GPUs. On the desktop PC (the GPU is GTS450, 196 cores), the highest speeds of GALE and GAMT are 235 *MLPS* and 218 *MLPS* respectively. But on the notebook (the GPU is GT550M, 96 cores), their highest speeds are only 107 *MLPS* and 111 *MLPS* respectively. Although their performance vary a lot on different GPUs, the stability of GAMT under highly frequent updates is still a significant superiority.

5.4.4 Controllable Latency

[9] It's calculated as $tree_level \times array_width/total_prefix$.

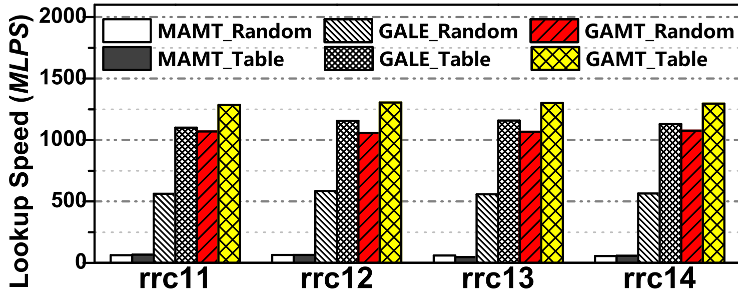


Figure 10: Lookup speed VS. routing data with two types of traffics.

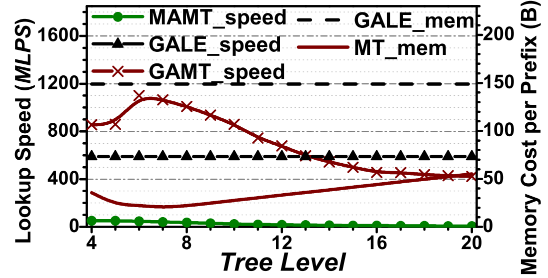
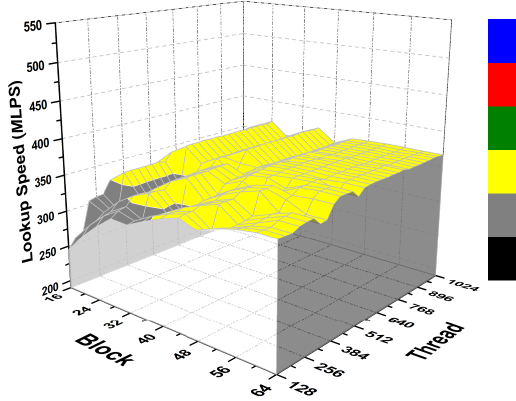
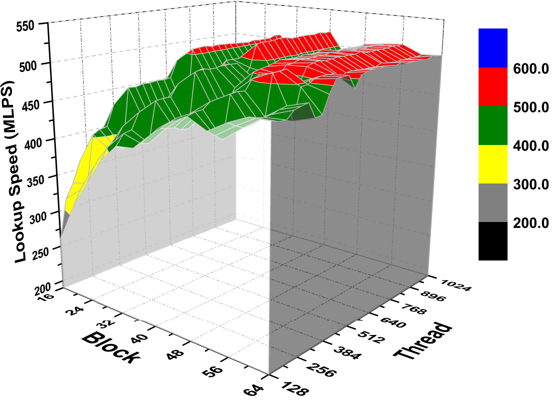


Figure 11: Lookup speed and memory efficiency VS. tree level.



(a) Using only one stream.



(b) Accelerated by two streams.

Figure 12: Lookup speed with different CUDA configurations.

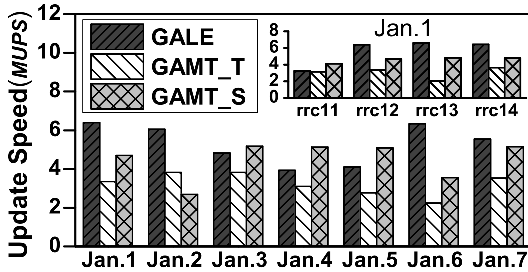


Figure 13: Off-line update speed.

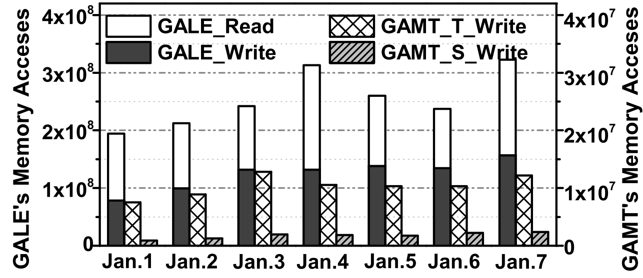


Figure 14: On-line update overhead.

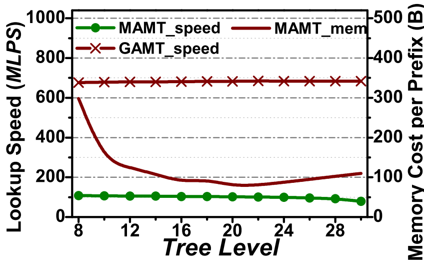


Figure 15: Lookup speed for IPv6.

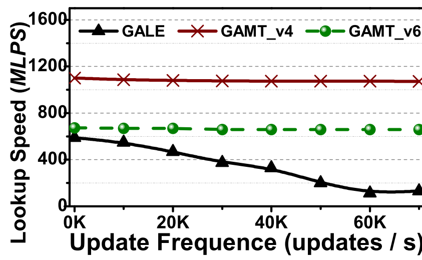


Figure 16: Lookup speed under frequent updates.

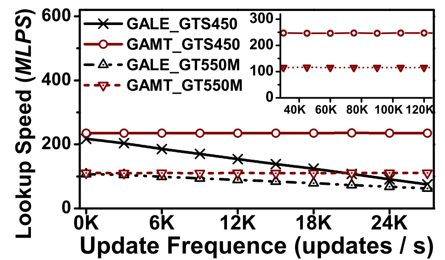


Figure 17: Lookup speed on other GPUs.

As mentioned in Section 4.4, batch processing on the GPU improves performance at the cost of extra delays, and such a dilemma can be resolved by the multi-stream pipeline to some extent. To keep a reasonable latency (measured as the elapsed time between a destination address being transferred to GPU until its lookup result has returned to CPU),

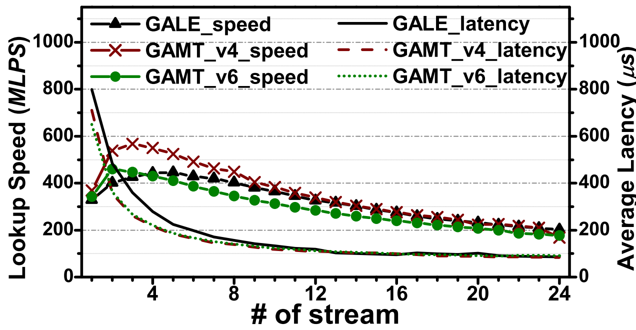


Figure 18: Lookup speed with controlled latency.

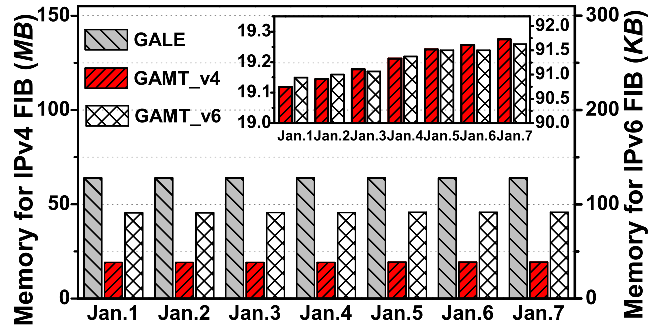


Figure 19: Memory consumptions.

say below 100 μs (as in [25]), we chose a small batch size ($256K$). Up-to 24 streams are activated, and each kernel is configured to have 64 blocks with each block having 1024 threads. Besides, lookup speeds and latencies are measured and analyzed based on the detail timeline generated by the NVIDIA Visual Profiler tool [17].

Figure 18 shows that GALE and GAMT (for IPv4/6) have similar behaviors. As expected, the average latency is consistently decreased with an increasing number of GPU streams. However, each stream has its own data transfers and kernel executions (see Fig. 8). As a result, although activating more streams can benefit more from behavior overlaps, it also results in cumulation of some fixed costs, such as warming up copy engines and starting kernels. Accordingly, the lookup speed will increase first and then decrease with the increasing number of streams.

In fact, to keep latency below the baseline (100 μs), GALE requires 18 streams, enabling a lookup speed of 248 *MLPS* with an average latency of 98.8 μs. In our system, for IPv4, 12 streams can achieve a speed as high as 339 *MLPS*, with an average latency controlled below our baseline. For IPv6, 16 streams will achieve 240 *MLPS* with an average latency of 99.5 μs.

5.4.5 Memory Consumptions

On the GPU, GALE consumes 64 *MB* (its direct table has $2^{24} = 16M$ units, each taking 4 bytes.) global memories, which is a constant in all cases. While GAMT's memory cost is determined by the shape of the multi-bit trie (shown as Fig. 11 and Fig. 15). For $rrc12$'s IPv4 FIB, the 6-level GAMT requires $22B$ to store each prefix in average, achieving a reduction to GALE by a factor of 70.3%.

In our system, to keep hierarchical lookup on the GPU, we should reserve some "head room" for each stage to support later updates. Due to the lazy-mode deleting (see section 4.3), we need rebuild GAMT after continuously updates. A backup of GAMT is stored on the GPU for rebuilding to avoid its disruption to the lookup. Even in this case, as shown in Fig. 19, the total memory consumptions of our system is still only $19.1MB$, as less as 29.8% of GALE's memory cost on the GPU. Furthermore, after a week's updates, the increasing rate of GAMT's memory cost (which for the array width is the same) is below 8%, ensuring the rebuilding process quite infrequent even with a little "head room".

6. CONCLUSION

In this paper, we have presented GPU-Accelerated Multi-bit Trie (GAMT), a fast and scalable IP lookup engine for GPU-based software routers. With a carefully designed multi-bit trie and an efficient multi-stream pipeline, the proposed scheme exploits GPU's characteristics in coalescence of memory access and concurrent operations, to provide very high performance in IPv4/6 address lookup. The speed of IPv4/6 address lookup has exceeded 1000/650 *MLPS* respectively, when an NVIDIA Tesla C2075 GPU is nearly fully utilized. Even a small batch size is used, GAMT can also achieve 339 and 240 *MLPS* for IPv4 and IPv6 respectively, with the average latency controlled below 100 μs.

On the other hand, GAMT scales well to frequent updates by employing an efficient update mechanism. In fact, it enables a stable throughput, which decreases only 4% and 8% for IPv4 and IPv6 respectively, even if update frequency has increased to $70K$ *updates/s*. Our experiments on different GPUs also demonstrate that GAMT works well on them in a self-adaptive way, and also provides higher performance and better scalability to frequent updates than that of GALE in most cases.

7. ACKNOWLEDGMENTS

This work is supported by the National Basic Research Program of China (973) under Grant 2012CB315805, the National Natural Science Foundation of China under Grant Number 61173167, 61272546, and 61370226, and the National Science Foundation under Grant Numbers CNS-1017598, CNS-1017588, CNS-0845513, and CNS-0916044. The corresponding authors of this paper are Dafang Zhang and Alex X. Liu.

8. REFERENCES

[1] Named data networking (NDN). http://www.named-data.net.

[2] RIPE network coordination centre [on line]. Available: http://www.ripe.net.

[3] Software-defined networking (SDN). https://www.opennetworking.org.

[4] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. *IEEE/ACM Transactions on Networking (TON)*, 13(3):690–703, 2005.

[5] D. Blythe. Rise of the graphics processor. *Proceedings of the IEEE*, 96(5):761–778, 2008.

[6] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve ip lookups. In *INFOCOM 2001.*

Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3, pages 1454–1463. IEEE, 2001.

[7] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software ip lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[8] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[9] Z. Huang, J. Peir, and S. Chen. Approximately-perfect hashing: improving network throughput through efficient off-chip routing table lookup. In *INFOCOM, 2011 Proceedings IEEE*, pages 311–315. IEEE, 2011.

[10] W. Jiang, Q. Wang, and V. K. Prasanna. Beyond tcams: An sram-based parallel multi-pipeline architecture for terabit ip lookup. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 1786–1794. IEEE, 2008.

[11] H. Le, W. Jiang, and V. K. Prasanna. A sram-based architecture for trie-based ip lookup using fpga. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 33–42. IEEE, 2008.

[12] L. Luo, G. Xie, Y. Xie, L. Mathy, and K. Salamatian. A hybrid ip lookup architecture with fast updates. In *INFOCOM, 2012 Proceedings IEEE*, pages 2435–2443. IEEE, 2012.

[13] R. McGeer. A safe, efficient update protocol for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 61–66, New York, NY, USA, 2012. ACM.

[14] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang. Ip routing processing with graphic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 93–98. European Design and Automation Association, 2010.

[15] NVIDIA Corporation. *NVIDIA CUDA C Best Practices Guide, Version 5.0*, Oct. 2012.

[16] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide, Version 5.0*, Oct. 2012.

[17] NVIDIA Corporation. *NVIDIA CUDA Profiler User Guide, Version 5.0*, Oct. 2012.

[18] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *Network, IEEE*, 15(2):8–23, 2001.

[19] S. Sahni and K. S. Kim. Efficient construction of multibit tries for ip lookup. *IEEE/ACM Transactions on Networking (TON)*, 11(4):650–662, 2003.

[20] S. Sahni and H. Lu. Dynamic tree bitmap for ip lookup and update. In *Networking, 2007. ICN'07. Sixth International Conference on*, pages 79–79. IEEE, 2007.

[21] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999.

[22] Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis. Smalta: practical and near-optimal fib aggregation. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 29. ACM, 2011.

[23] G. Wang and N.-F. Tzeng. Tcam-based forwarding engine with minimum independent prefix set (mips) for fast updating. In *Communications, 2006. ICC'06. IEEE International Conference on*, volume 1, pages 103–109. IEEE, 2006.

[24] M. Wang, S. Deering, T. Hain, and L. Dunn. Non-random generator for IPv6 tables. In *Proc. IEEE Symposium on High Performance Interconnects, Hot Interconnects*, pages 35–40, 2004.

[25] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire speed name lookup: A gpu-based approach. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 199–212, 2013.

[26] C. Wenping, Z. Xingming, Z. Jianhui, and W. Bin. Research on multi next hop rip. In *Information Technology and Applications, 2009. IFITA'09. International Forum on*, volume 1, pages 16–19. IEEE, 2009.

[27] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu. Exploiting graphics processors for high-performance ip lookup in software routers. In *INFOCOM, 2011 Proceedings IEEE*, pages 301–305. IEEE, 2011.

[28] Y. Zhu, Y. Deng, and Y. Chen. Hermes: an integrated cpu/gpu microarchitecture for ip routing. In *Proceedings of the 48th Design Automation Conference*, pages 1044–1049. ACM, 2011.