Wire Speed Name Lookup: A GPU-based Approach

Yi Wang[†], Yuan Zu[‡], Ting Zhang[†], Kunyang Peng[‡], Qunfeng Dong[‡©], Bin Liu[†©],
Wei Meng[†], Huichen Dai[†], Xin Tian[‡], Zhonghu Xu[‡], Hao Wu[†], Di Yang[‡]
[†]*Tsinghua National Laboratory for Information Science and Technology,*
Department of Computer Science and Technology, Tsinghua University
[‡]*Institute of Networked Systems (IONS) & School of Computer Science and Technology,*
University of Science and Technology of China

Abstract

This paper studies the name lookup issue with longest prefix matching, which is widely used in URL filtering, content routing/switching, etc. Recently Content-Centric Networking (CCN) has been proposed as a clean slate future Internet architecture to naturally fit the content-centric property of today's Internet usage: instead of addressing end hosts, the Internet should operate based on the identity/name of contents. A core challenge and enabling technique in implementing CCN is exactly to perform name lookup for packet forwarding at wire speed. In CCN, routing tables can be orders of magnitude larger than current IP routing tables, and content names are much longer and more complex than IP addresses. In pursuit of conquering this challenge, we conduct an implementation-based case study on wire speed name lookup, exploiting GPU's massive parallel processing power. Extensive experiments demonstrate that our GPU-based name lookup engine can achieve 63.52M searches per second lookup throughput on large-scale name tables containing millions of name entries with a strict constraint of no more than the telecommunication level 100μs per-packet lookup latency. Our solution can be applied to contexts beyond CCN, such as search engines, content filtering, and intrusion prevention/detection.

©Prof. Qunfeng Dong (qunfeng.dong@gmail.com) and Prof. Bin Liu (lmyujie@gmail.com), placed in alphabetic order, are the correspondence authors of the paper. Yi Wang and Yuan Zu, placed in alphabetic order, are the lead student authors of Tsinghua University and University of Science and Technology of China, respectively.

This paper is supported by 863 project (2013AA013502), NSFC (61073171, 61073184), Tsinghua University Initiative Scientific Research Program(20121080068), the Specialized Research Fund for the Doctoral Program of Higher Education of China(20100002110051), the Ministry of Education (MOE) Program for New Century Excellent Talents (NCET) in University, the Science and Technological Fund of Anhui Province for Outstanding Youth (10040606Y05), by the Fundamental Research Funds for the Central Universities (WK0110000007, WK0110000019), and Jiangsu Provincial Science Foundation (BK2011360).

1 Introduction

Name lookup is widely used in a broad range of technological fields, such as search engine, data center, storage system, information retrieval, database, text processing, web application, programming languages, intrusion detection/prevention, malware detection, content filtering and so on. Most of these name lookup applications either perform exact matching only or operate on small-scale data sets. The recently emerging Content-Centric Networking (CCN) [12] proposes to use a content name to identify a piece of data instead of using an IP address to locate a device. In CCN scenario, every distinct content/entity is referenced by a unique name. Accordingly, communication in CCN is no longer address-based, but name-based. CCN routers forward packets based on the requested content name(s) carried in each packet header, by looking up a forwarding table consisting of content name prefixes.

CCN name lookup complies with longest prefix matching (LPM) and backbone CCN routers can have large-scale forwarding tables. Wire speed name lookup presents a research challenge because of stringent requirements on memory occupation, throughput, latency and fast incremental update. Practical name lookup engine design and implementation, therefore, require elaborate design-level innovation plus implementation-level re-engineering.

1.1 Names and Name Tables

Content naming, as recently proposed in the Named Data Network (NDN) project [28][1], is hierarchically structured and composed of explicitly delimited name components, such as reversed domain names followed by directory-style path. For in-

[1]CCN refers to the general content-centric networking paradigm; NDN refers to the specific proposal of the NDN project. However, we shall use them interchangeably in the rest of the paper.

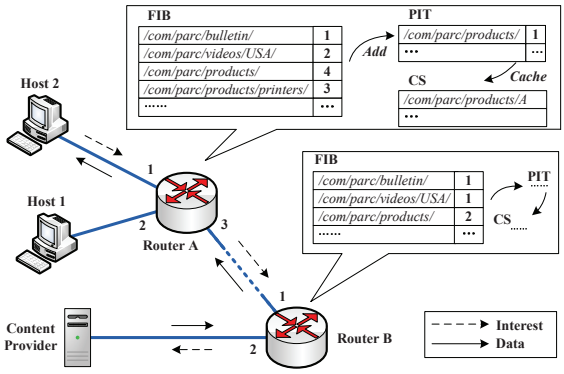

Figure 1: NDN communication example[3].

stance, `com/parc/bulletin/NSDI.html` is an example NDN content name, where `com/parc/` is the reversed domain name `parc.com` of the web site and `/bulletin/NSDI.html` is the content's directory path on the web site. '`/`' is the component boundary delimiter and not a part of the name; `com`, `parc`, `bulletin` and `NSDI.html` are four components of the name.

The format of *Forwarding Information Base (FIB)*[2] of NDN routers is shown beside Router A and B in Figure 1. Each FIB entry is composed of a name prefix and the corresponding outgoing port(s). NDN name lookup also complies with *longest prefix matching (LPM)*. For example, suppose the content name is `/com/parc/products/printers/hp`, which matches the third and fourth entries in Router A; the fourth entry as the longest matching prefix determines that the packet should be forwarded through port 3.

1.2 Challenges

To implement CCN routing with large-scale FIB tables in high speed networks, a core challenge and enabling technique is to perform content name lookup for forwarding packets at wire speed. In particular, a name lookup engine is confronted with the following difficulties.

First, content names are far more complex than IP addresses. As introduced above, content names are much longer than IPv4/IPv6 addresses; each name is composed of tens, or even hundreds, of characters. In addition, unlike fixed-length IP addresses, content names have variable lengths, which further complicates the name lookup.

Second, CCN name tables could be far larger than today's IP forwarding tables. Compared with the current IP routing tables with up to 400K IP prefix entries, CCN name tables could be orders of magnitude larger. Without elaborate compression and implementation, they can by far exceed the capacity of today's commodity devices.

Third, wire speeds have been relentlessly accelerating. Today, OC-768 (40Gbps) links have already been deployed in Internet backbone, and OC-3072 (160Gbps) technology is emerging at the horizon of Internet.

Fourth, in addition to network topology changes and routing policy modifications, CCN routers have to handle one new type of FIB update — when contents are published/deleted, name prefixes may need to be inserted into or deleted from FIBs. This makes FIB update much more frequent than in today's Internet. Fast FIB update, therefore, must be well handled for large-scale FIBs.

1.3 Our work

In pursuit of conquering these challenges, we conduct an implementation-based case study of wire speed name lookup in large-scale name tables, exploiting GPU's massive parallel processing power.

1) We present the first design, implementation and evaluation of a GPU-based name lookup engine. Through this implementation-based experimental study, we demonstrate the feasibility of implementing wire speed name lookup with large-scale name tables at low cost, using today's commodity GPU devices. We have released the implementation code, data traces and documents of our work [3].

2) Our GPU-based name lookup engine is featured by a new technique called *multiple aligned transition arrays (MATA)*, which combines the best of two worlds. On one hand, MATA effectively improves lookup speed by reducing the number of memory access. On the other hand, MATA as one-dimensional arrays can substantially compress storage space. Due to these unique merits, MATA is demonstrated through experiments to be able to compress storage space by two orders of magnitude, while promoting lookup speed by an order of magnitude, compared with two-dimensional state transition tables.

3) GPU achieves high processing throughput by exploiting massive data-level parallelism — large amounts of input data (i.e., names) are loaded into GPU, looked up in GPU and output results from GPU together to hide GPU's DRAM access latency. While effectively boosting processing throughput, this typical GPU design philosophy easily leads to extended per-packet latency. In this work, we take on this throughput-latency dilemma by exploiting the *multi-stream* mechanism featured in NVIDIA's Fermi GPU family. Our stream-based pipeline solution ensures practical per-packet latency (less than 100μs) while keeping high lookup throughput.

4) We employ data interweaving [32] technique for optimizing the storage of input names in GPU memory.

[2] In this paper, we shall use three terms — *FIB*, *FIB table* and *name table* — interchangeably.

[3] In the NDN proposal, there are three kinds of tables, FIB, PIT and CS. Only if the CS and PIT both fail to match, name lookup in FIB is performed. When we evaluate our lookup engine, we assume this worst case where every name has to be looked up in FIB.

As a result, memory access efficiency is significantly improved, further boosting name lookup performance.

We implement our name lookup engine on a commodity PC installed with an NVIDIA GeForce GTX590 GPU board. Using real world URL names collected from Internet, we conduct extensive experiments to evaluate and analyze the performance of our GPU-based name lookup engine. On large-scale name tables containing up to 10M names, the CPU-GPU lookup engine obtains 63.52M searches per second (SPS) under average workload, enabling an average line rate of 127 Gbps (with 256-byte average packet size). Even under heavy workload, we can still obtain up to 55.65 MSPS, translating to 111 Gbps wire speed. Meanwhile, lookup latency can be as low as 100μs. In fact, if the PCIe bus bandwidth between CPU and GPU were not the system bottleneck, the lookup engine core running on the GPU could achieve 219.69 MSPS! Besides, experiments also show that our name lookup engine can support fast incremental name table update.

These results advocate our GPU-based name lookup engine design as a practical solution for wire speed name lookup, using today's off-the-shelf technologies. The results obtained in this work, however, will have broad impact on many technological fields other than CCN.

2 Algorithms & Data Structures

In this section, we present the core algorithmic and data structure design of our GPU-based name lookup engine. The entire design starts with name table aggregation in Section 2.1, where name tables are aggregated into smaller yet equivalent ones. After that, we present in Section 2.2 aligned transition array (ATA), which subsequently evolves into multi-striding ATA in Section 2.3 and multi-ATA (MATA) — the core data structure for high speed and memory efficient name lookup — in Section 2.4. Finally in Section 2.5, we demonstrate how name table updates can be handled with ease in our lookup engine design.

2.1 Name table aggregation

The hierarchical structure of NDN names and the longest prefix matching property of NDN name lookup enable us to aggregate NDN name tables into smaller ones. For example, consider Router A's name table in Figure 1. If the third entry and the fourth entry map to the same next hop port, they can be aggregated into one, by removing the fourth entry. After this aggregation, names originally matching the fourth entry will now match the third one. Since the two entries are hereby assumed to map to the same port, it is safe to perform this aggregation.

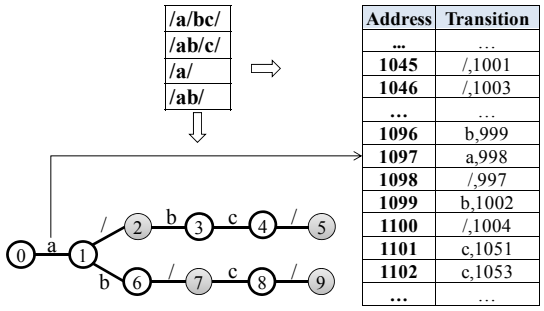

Figure 2: Aligned transition array.

To safely aggregate the two name table entries, they need to comply with two simple principles: (1) One of them is the shortest prefix of the other in the name table; (2) They must map to the same next hop port(s).

2.2 Aligned transition array (ATA)

A natural approach to implementing NDN name lookup is to build a character-trie [9], which is essentially a finite state machine (FSM) for matching incoming names against a name table, as shown in the left part of Figure 2. Each node in the character-trie is implemented as a state in the FSM, and transitions to its child nodes (i.e., states) on specified input characters. To start with, we assume the FSM processes one input character on each state transition. In such a 1-stride FSM, each state has 256 transitions, and each transition corresponds to a distinct input character. The entire FSM is essentially a two-dimensional state transition table, where the ith row implements the ith state s_i and the jth column corresponds to the jth character c_j in the input alphabet Σ, which in this case is the ASCII character set. The table entry at the intersection of the ith row and jth column records the destination state we should transit to, if the current state is s_i and the input character is c_j; an empty entry indicates failing to match. Using current state ID as row number and input character as column number, one memory access is sufficient to perform every state transition.

However, this standard solution does not work in practice. To see that, we experiment with a real name table consisting of 2,763,780 names (referred as "3M name table" for brevity). After aggregation, the constructed 1-stride FSM consists of 20,440,366 states; four bytes are needed for encoding state ID, and $256 \times 4 = 1,024$ bytes are thus needed for each row of the state transition table. The entire state transition table takes 19.49 GB memory space. In fact, the largest name table used in our experiments is even several times larger than the 3M name table. To fit such a large-scale name table into commodity GPU devices, the FSM has to be compressed by at least 2-3 orders of magnitude, while still has to perform name

lookup at wire speed, meeting stringent lookup latency requirement and supporting fast incremental name table update. This is a key challenge in the design and implementation of a practical name lookup engine, which we take on in this work.

As we can observe from Figure 2, the FSM for name lookup demonstrates a key feature — most states have valid transitions on very few input characters. For example in Figure 2, state 0 only has a valid transition on character `a`. This intuitive observation is also verified through experiments. For example, in the 1-stride FSM constructed from the 3M name table, more than 80% of states have only one single valid transition, plus more than 13% of states (which are accepting states) that have no valid transition at all. The state transition table is thus a rather sparse one.

In light of this observation, we store valid transitions into what we call an *aligned transition array (ATA)*. The basic idea is to take the sum of current state ID and input character as an index into the transition array. For example, if the current state ID is 1,000 and the input character is `a` (whose ASCII code is 97), we shall take the transition stored in the 1,097th transition array element as our valid transition[4]. To properly implement, we need to assign each state s a unique state ID such that no two valid transitions are mapped to the same transition array element. For that, we first find the smallest input character on which state s has a valid transition; suppose the character is the kth character in input alphabet Σ. Then, we find the lowest vacant element in the transition array, and suppose it is the ℓth element in the transition array. The number ℓ-k, if previously unused as a state ID, is considered as a candidate state ID for state s. To avoid possible storage collision, we need to check every input character c on which state s has a valid transition. If no collision is detected on any valid transition of state s, ℓ-k is assigned as the state ID of state s. Otherwise, if the (ℓ-k+c)th transition array element is already occupied, a collision is detected and ℓ-k is not good as the state ID for state s. The next vacant elements in the transition array are probed one by one until finding the available element.

Another mistake that can potentially happen here is that, even if current state s has not a valid transition on the current input character, (state ID + input character) may mistakenly refer to a stored valid transition belonging to another state. To handle this problem, we store for each valid transition not only its destination state ID, but also its input character for verification.

With this aligned transition array design, the example

[4]This basic idea of storing a sparse table as a one-dimensional array is introduced by Robert Endre Tarjan and Andrew Chi-Chih Yao [23] back in 1979. With our new techniques proposed in subsequent sections, we shall be able to effectively boost name lookup speed while further reducing storage space.

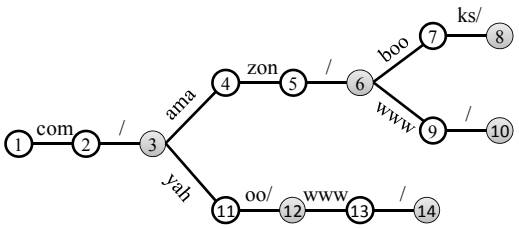

Figure 3: A 3-stride FSM.

name table in the left part of Figure 2 is implemented as the aligned transition array in the right part of Figure 2. State transition is as efficient as using two-dimensional state transition table. We simply take the sum of current state ID and input character, and read the transition array element indexed by the sum. If the stored character is the same as input character, the obtained transition directly gives us the destination state; otherwise, failing to match is detected. One single memory access is still sufficient for every state transition. Meanwhile, only valid transitions need to be stored, leading to significant compression of storage space.

Thus far, this basic ATA design looks perfect. However, scale can completely change the nature of problems. Given the limited resource and capacity of today's commodity devices, as we shall observe in subsequent sections, performing name lookup at 10Gbps in a name table containing hundreds of thousands name prefixes is one thing, while performing name lookup at 100Gbps in a name table consisting of millions of name prefixes is totally different. Especially, we also have to meet stringent per-packet lookup latency requirement and support fast incremental name table update. In subsequent sections, we shall unfold and solve various technical issues, as we proceed towards wire speed name lookup with large-scale name tables.

2.3 Multi-striding

The storage efficiency and lookup speed of aligned transition array can be further improved with multi-striding — instead of processing one character per state transition, d characters are processed on each state transition. The same algorithm for constructing 1-stride FSMs can be used to construct multi-stride FSMs. To ensure proper matching of name prefixes and to facilitate name table update, component delimiter '`/`' can only be the last character we read upon each state transition. Thus, the d-stride FSM we construct is actually an FSM of variable strides, which processes *up to* d characters ($8d$ bits) per state transition. For example, Figure 3 shows the 3-stride FSM constructed from the following three names: `/com/amazon/books/`, `/com/amazon/www/` and `/com/yahoo/www/`. Upon state transition, we keep reading in d input characters unless '`/`' is encountered,

where we stop. These input characters are transformed into an integer, which is taken as the *input number*.

By processing multiple bytes per state transition, multi-striding effectively accelerates name lookup. Even better, multi-striding also helps reduce storage space. Because a large number of intermediate states and transitions in the 1-stride FSM will be consolidated.

2.4 Multi-ATA (MATA)

While multi-striding is very effective on boosting lookup speed and reducing storage space, trying to further expand its power using larger strides leads us to an inherent constraint of the basic ATA design — its multi-striding is limited by available memory. For example, in the 3-stride FSM in Figure 3, it takes two state transitions to match the first-level name component `'com/'`, making us naturally think about using a 4-stride FSM so that one state transition will be enough. Unfortunately, in a d-stride FSM, a state can have 2^{8d} transitions at most, so the distance between a state's two valid transitions stored in the aligned transition array can be as large as 2^{8d}-1. Thus, with the 3GB memory space available on the NVIDIA GTX590 GPU board used in our experiments, 3-stride has been the maximum stride that can be implemented with basic ATA; 4-stride is not a practical option.

We break this constraint of basic ATA by defining a maximum ATA length L ($L < 2^{8d}$). For a state with state ID x, its valid transition on input number y can be stored in the ((x+y) mod L)th transition array element instead of the (x+y)th element. However, suppose state x has another valid transition on input number z. If y-z is a multiple of L, the two valid transitions will be mapped to the same transition array element and hence cause a storage collision.

For solving the above problem, we shall use a set of prime numbers L_1, L_2, ..., L_k, such that $L_1 \times L_2 \times \ldots \times L_k \geq 2^{8d}$. Accordingly, instead of creating one huge ATA, we create a number of small ATAs, each ATA using one of the prime numbers as its maximum length. Then, we first try to store the two valid transitions on y and z into an ATA with prime number L_1. (There can be multiple ATAs having the same maximum length.) If the two valid transitions do not collide with each other but collide with some valid transition(s) previously stored in that ATA, we shall try another ATA with the same maximum length; if the two valid transitions collide with each other, we shall move on trying to store state x into an ATA with a different maximum length, until ATAs with all different maximum lengths have been tried. It is guaranteed that, there must be at least one prime number L_i that can be used to store the two valid transitions without any collision. To prove by contradiction, assume the two valid transitions collide with all prime numbers L_1, L_2, ..., L_k as the maximum length. That means, y-z is a multiple of all these prime numbers L_1, L_2, ..., L_k, and hence a multiple of $L_1 \times L_2 \times \ldots \times L_k$; this in turn means $y\text{-}z \geq L_1 \times L_2 \times \ldots \times L_k \geq 2^{8d}$, which is impossible.

For each state in the FSM, as part of its state ID information, we record the small ATA that is used to store its valid transition(s). Thus, one single memory access is still sufficient to perform every state transition.

To handle cases where a state has multiple pairs of valid transitions colliding with each other, the above design can be simply augmented with more prime numbers.

In addition to breaking the constraint on maximum stride, the above described multi-ATA (MATA) design also has two other merits.

First, ATAs can leave elements unused in vacancy, due to storage collision or insufficient number of valid transitions to store. By defining maximum ATA length, we now have better control over the amount of ATA elements that are wasted in vacancy.

Second, constructing the MATA optimally is NP-hard. One single large basic ATA can take prohibitively long time to construct, even employing a heuristic algorithm. By breaking into a number of small ATAs, the entire FSM takes much less time to store into these ATAs. For example, the construction time of heuristic algorithm for the 3M name table can be reduced from days (for basic ATA) to minutes (for MATA), with an appropriate set of prime numbers.

2.5 Name table update

There are two types of name table updates: insertion and deletion. In this subsection, we demonstrate how these updates can be handled with ease in our design. First of all, it is worth reminding the reader that our name lookup mechanism is composed of two components.

(1) The name table is first organized into a character-trie, which is essentially a finite state machine.

(2) The character-trie is then transformed into MATA.

Accordingly, insertion and deletion of names are first conducted on the character-trie, in order to determine the modifications that need to be made to the character-trie structure. Then, we carry out the modifications in MATA. Following this logic, we explain name insertion and deletion as follows.

2.5.1 Name deletion

To delete a name P from the name table, we simply conduct a lookup of name P in the name table. If it is not matched, we determine that the proposed deletion operation is invalid, as name P does not actually exist in the name table.

Once name P is properly matched and comes to the leaf node representing itself, we then simply backtrack

towards the root. (This can be done by remembering all the nodes we have traversed along the path from the root to the leaf node.) Every node with one single valid transition will be deleted from the trie (each trie node contains a field that records the number of children nodes), till when we encounter the first node with next-hop information or more than one valid transition.

It is equally simple to carry out character-trie node deletion in MATA. Since every node to be deleted has one single valid transition, deleting the node is equivalent to deleting its stored valid transition in MATA. It takes one single memory access to locate the transition array element storing that transition, and mark that transition array element as vacant.

2.5.2 Name insertion

To insert a name P into the name table, we also conduct a lookup of name P in the name table, where we traverse the character-trie in a top-down manner, starting from the root. At each node on our way down the character-trie, if an existing transition is properly matched by P, we need to do nothing about the node. Otherwise, suppose we read in a number of characters from name P, which is converted into an integer x. We add a new transition on x to the current node, pointing to a new node we create for this new transition. This process continues until lookup of name P is completed.

To add an existing node's new transition on x into MATA, we directly locate the transition array element in which the new transition should be stored. If that element is vacant, we simply store the new transition into that element, and we are done. Otherwise, suppose the existing node needs to be relocated to resolve storage collision. This is done as if the node is a new node to be stored into MATA, following the algorithm described in Section 2.2 and 2.4. (One minor difference is that, here we also need to update the upstream transition pointing to the relocating node. This can be easily handled by always remembering the parent node of current node, during the lookup process for name P.)

3 The CPU-GPU System: Packet Latency and Stream Pipeline

GPU achieves high processing throughput by exploiting massive data-level parallelism — a large batch of names are processed by a large number of GPU threads concurrently. However, this massive batch processing mode can lead to extended per packet lookup latency[5]. Figure 4 presents a 4-stride MATA's throughput and latency

[5]Name lookup engine latency: the aggregate time from a packet being copied from host CPU to GPU device till its lookup result being returned from GPU to CPU.

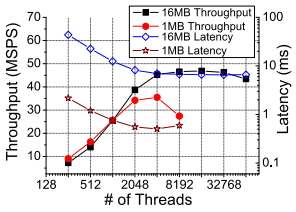

Figure 4: Throughput and latency of MATA without pipeline (3M name table, average workload).

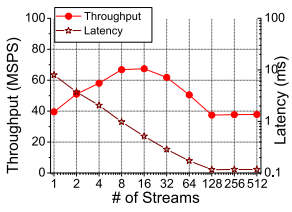

Figure 5: Throughput and latency of MATA with pipeline (3M name table, average workload).

obtained on the 3M name table, where names are processed in 16MB batches. As we can see, per-packet lookup latency can be many milliseconds. While in practice, telecommunication industry standards require that the entire system latency should be less than 450 μs[6]; name lookup as one of the various packet processing tasks should take no longer than 100 μs.

This extended lookup latency results from concurrent lookups of multiple names, due to contention among concurrent lookup threads processing different names. That said, a straightforward thought, therefore, is to reduce per-packet lookup latency by reducing batch size. Figure 4 also presents the above MATA's throughput and latency obtained with 1MB batch size. As we can see, small batch size leads to reduced lookup throughput, due to reduced data-level parallelism. But it is insufficient to hide off-chip DRAM access latency, causing throughput decline accordingly. Essentially, this latency-throughput dilemma is rooted in the GPU design philosophy of exploiting massive data-level parallelism. Unfortunately, previous proposals on GPU-based pattern matching (e.g. [29, 16]) have not taken latency requirements into account.

In this work, we resolve this latency-throughput dilemma by exploiting the multi-stream mechanism featured in NVIDIA's Fermi GPU architecture. In CUDA programming model, a *stream* is a sequence of operations that execute in issue-order. For example, in our design, each stream is composed of a number of lookup threads, each thread consisting of three tasks. (1) `DataFetch`: copy input names from host CPU to GPU device (via PCIe bus); (2) `Kernel`: perform name lookup inside GPU; (3) `WriteBack`: write lookup results back from GPU device to host CPU (via PCIe bus). Among them, `DataFetch` and `WriteBack` tasks are placed into one queue, executed by the *copy engine*; `Kernel` tasks are organized into another queue, executed by the *kernel engine*. Tasks in the same queue are executed in the order they enter the queue. In our design, each batch of input names is divided into m subsets; the kth subset is assigned to the kth stream for lookup.

[6]Here we refer to the specifications of the ISDN switch.

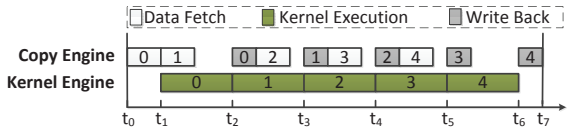

Figure 6: Multi-stream pipeline solution.

By pipelining these concurrent streams, lookup latency can be effectively reduced while keeping high lookup throughput. Algorithm 1 shows the pseudo code description of this optimized scheduling.

As shown in Figure 6, the `Kernel` task of stream *i* runs (on the kernel engine) in parallel with the `WriteBack` task of stream *i*-1 followed by the `DataFetch` task of stream *i*+1 (both running on the copy engine).

Figure 5 presents MATA's throughput and latency obtained on the 3M name table with 16MB batch size organized into 1∼512 streams, using 2,048 threads. This multi-stream pipeline solution successfully reduces lookup latency to 101μs while maintaining lookup throughput (using 128 or more streams).

Throughput: As described in Section 6.2.2, the copy engine is the throughput bottleneck. Then, the time T to finish processing an input name batch can be calculated by Formula (1) [15].

$$T = 2t_{start} * N + \frac{M_{batch} + M_{result}}{S_{PCIe}} + \frac{M_{batch}}{N * S_{kernel}} \quad (1)$$

Here, M_{batch} and M_{result} are the name batch size and the corresponding lookup results size, respectively. S_{PCIe} is the PCIe speed, S_{kernel} is the name lookup speed in GPU kernel, t_{start} is the warm up time of the copy engine, and N is the number of streams. Maximizing throughput means minimizing T. According to Fermat's theorem [2], T gets the minimal value at the stationary point $f'(N) = 0$, where $N = \sqrt{\frac{M_{batch}}{2t_{start} * S_{kernel}}}$. In Figure 5, our CPU-GPU name lookup engine has M_{batch}=16MB, t_{start}≈10μs, S_{kernel}≈8GB/s (200MSPS × 40B/packet). So the engine gets the maximal throughput with N=16 streams.

Algorithm 1 Multi-stream Pipeline Scheduling

```
1: procedure MultiStreamPipelineScheduling
2:    i ← 0;
3:    offset ← i*data_size/m;
4:    DataFetch(offset, streams[i]);
5:    Kernel(offset, streams[i]);
6:    for i:  0 → m-2 do
7:       offset ← (i+1)*data_size/m;
8:       DataFetch(offset, streams[i+1]);
9:       Kernel(offset, streams[i+1]);
10:       wb_offset ← i*data_size/m;
11:       WriteBack(wb_offset, streams[i]);
12:    end for
13:    WriteBack(offset, streams[m-1]);
14: end procedure
```

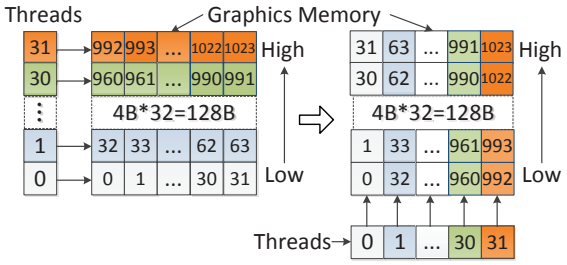

Figure 7: Input name storage layout.

Latency: Lookup latency $T_{latency}$ equals to the period from a stream's `DataFetch` task launched to the corresponding `WriteBack` task finished, i.e.,

$$T_{latency} = 2t_{start} + \frac{1}{N} * \left(\frac{M_{batch} + M_{result}}{S_{PCIe}} + \frac{M_{batch}}{S_{kernel}} \right) \quad (2)$$

Obviously, lookup latency decreases as the increasing of the stream number N.

4 Memory Access Performance

Like in all other modern computing architectures, memory access efficiency has a significant impact on GPU application performance. One practical approach to boosting performance is to reduce the amount of slow DRAM accesses, by exploiting GPU's memory access coalescence mechanism. In NVIDIA GeForce GTX GPU devices, the off-chip DRAM (e.g. global memory) is partitioned into 128-byte memory blocks. When a piece of data is requested, the entire 128-byte block containing the data is fetched (with one memory access). When multiple threads simultaneously read data from the same block, their read requests will be coalesced into one single memory access (to that block).

In our design, we employ an effective technique for optimizing memory access performance called input *interweaving* [32], which stores input names in an interweaved layout. In NVIDIA GeForce GTX GPUs, every 32 threads (with consecutive thread IDs) are bundled together as a separate *warp*, running synchronously in a single-instruction multiple-data (SIMD) manner — at any time, the 32 threads execute the same instruction, on possibly different data objects. In common practice, input data (i.e., names) are simply stored contiguously, as shown in the left part of Figure 7. For ease of illustration, each name is 128-byte long and occupies a whole memory block (one row). The 32 names parallel processed by the 32 threads in a *warp* reside in 32 different 128-byte blocks. Therefore, when the 32 threads simultaneously read the first piece of data from each of the names they are processing, resulting in 32 separate memory accesses that cannot be coalesced.

Here, memory access performance can be substantially improved by storing input names in an interweaved layout. Suppose the name lookup engine employs N_{thread} concurrent GPU threads. (1) Host CPU distributes input names into N_{thread} queues (i.e., name sequences), in the order of their arrival[7]; (2) Then, every 32 adjacent name sequences are grouped together, to be processed by a GPU warp consisting of 32 threads; each thread in the warp locates one of the 32 name sequence using its thread ID; (3) Finally, each group of 32 name sequences are interweaved together. Iteratively, CPU takes a 4-byte data *slice* from the head of each name sequence and interweaves them into a 128-byte memory block (one row), as shown in the right part of Figure 7. After interweaving, the 32 data slices of one name are stored in 32 different blocks (one column), and the 32 data slices belonging to 32 different names are stored in one block. The interweaved memory blocks are then transmitted to GPU's DRAM. Now, when the 32 threads in the same warp each requests for a slice from its own name sequence simultaneously, the 32 requested slices reside in the same 128-byte block. Therefore, the 32 memory requests are coalesced into one single memory access to that block. Interweaving thus significantly reduces the total amount of memory accesses to DRAM and hence substantially boosts overall performance.

5 Implementation

In this section, we describe the implementation of our GPU-based name lookup engine, including its input/output. First in Section 5.1, we introduce the hardware platform, operating system environment and development tools with which we implement the name lookup engine. Then in Section 5.2, we present the framework of our system. The lookup engine has two inputs: name tables and name traces. We introduce how we obtain or generate the name tables and name traces in Section 5.3 and Section 5.4, respectively.

5.1 Platform, environment and tools

We implement and run the name lookup engine on a commodity PC installed with an NVIDIA GeForce GTX 590 GPU board. The PC is installed with two 6-core CPUs (Intel Xeon E5645×2), with 2.4GHz clock frequency. Relevant hardware configuration is listed in Table 1.

Table 1: Hardware configuration.

Item	Specification
Motherboard	ASUS Z8PE-D12X (INTEL S5520)
CPU	Intel Xeon E5645×2 (6 cores, 2.4GHz)
RAM	DDR3 ECC 48GB (1333MHz)
GPU	NVIDIA GTX590 (2×512 cores, 2×1536MB)

The PC runs Linux Operating System version 2.6.41.9-1.fc15.x86_64 on its CPU. The GPU runs CUDA NVIDIA-Linux operating system version x86_64-285.05.09.

The entire lookup engine program consists of about 8,000 lines of code, and is composed of two parts: a CPU-based part and a GPU-based part. The CPU part of the system is developed using the C++ programming language; the GPU part of the system is developed using NVIDIA CUDA C programming language's SDK 4.0.

5.2 System framework

Figure 8 depicts the framework of our name lookup engine. Module 1 takes a name table as input and builds a character-trie for aggregating and representing that name table; this character-trie serves as the control plane of name lookup. To implement high speed and memory efficient name lookup, Module 2 transforms the character-trie into MATA, which serves as the data plane of name lookup, and hence will be transferred to and operated in the GPU-based lookup engine. Module 3 is the lookup engine operating in GPU, which accepts input names, performs lookup in the MATA and outputs lookup results. Meanwhile, GPU takes the generated name traces as input to search against the name table, which is implemented as the MATA. The lookup result in GPU is output to a module running on CPU, which obtains next-hop interface information based on GPU's output. There is also a 5-th module that is responsible for handling name table updates. We measure the core latency between point *A* and *B*, that is from the sending buffer to the receiving buffer of CPU.

5.3 Name Tables

The two name tables used in our experiments contain 2,763,780 entries and 10,000,000 entries, respectively. For brevity, we refer to them as the "3M name table" and the "10M name table", respectively. Each name table entry is composed of an NDN-style name and a next hop port number. As NDN is not yet standardized and there is no real NDN network, the name tables are obtained through the following five-step process.

[7]When appending names into sequences, we transform each input name (and name sequence) from a sequence of characters into a sequence of numbers. In our implementation and experiments, we implement 4-stride FSMs and hence each input name is transformed into a sequence of 32-bit `unsigned int` type integers. To transform an input name, CPU keeps reading up to 4 characters from that name unless a '/' is encountered; the characters read out are then transformed into an `unsigned int` integer. Each name sequence is thus transformed into a sequence of 32-bit integers.

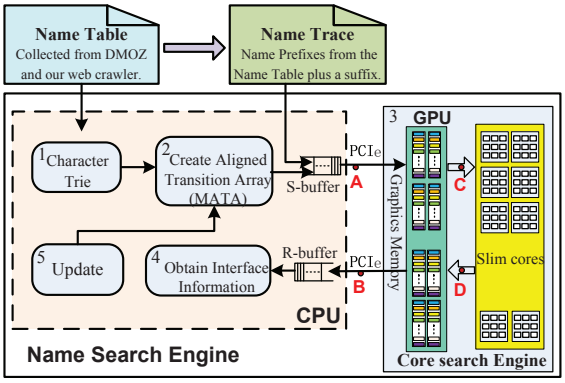

Figure 8: Framework of the name lookup engine.

Step 1: We collect Internet domain names in two ways. (1) We obtain existing domain name information from DMOZ [1], which is later used to generate the 3M name table. (2) We use a web crawler program to collect domain names, which are later used to generate the 10M name table. To achieve good geographic coverage, we ran web crawler programs on three servers located in North America, Europe and Asia, respectively. The web crawler programs kept collecting URLs from October 1st, 2011 to March 31st, 2012. At last, the crawler collected 7M domain names different from that collected from DMOZ. Consequently, we obtain 10 million non-duplicate domain names in total with our maximum efforts.

Step 2: We convert the domain names into NDN-style names, by putting the components in reverse order. For example, domain name `www.parc.com` is transformed into `/com/parc/www/`.

Step 3: For each NDN-style name, we map its corresponding domain name to an IP address resolved by DNS.

Step 4: For each NDN-style name, we obtain its next hop port number by performing longest prefix matching on its IP address obtained in Step 3, using an IP FIB downloaded from www.ripe.net.

Step 5: We map each NDN prefix to its obtained next hop port number, which gives us the final name table.

5.4 Name Traces

The name traces, which are generated from name tables, simulate the destination names carried in NDN packets. The names are formed by concatenating name prefixes selected from the name table and randomly generated suffixes. We generate two types of name traces, simulating average lookup workload and heavy lookup workload, respectively. Each name trace contains 200M names, and is generated as follows.

For each name table used in our experiments, its average workload trace is generated by randomly choosing names from the name table; its heavy work load trace is generated by randomly choosing from the top 10% longest names in the name table. Intuitively, the longer the input names are, the more state transition operations the GPU will perform for their lookup, meaning heavier workload.

Names chosen from name tables do not have a directory path. From the URLs we collected from Internet, we randomly choose a directory path for each name in the traces and append that path to the name.

6 Experimental Evaluation

We compare the performance of four lookup methods we have discussed in Section 2. The baseline method is using a two-dimensional state transition table (denoted by STT), which is largely compressed by ATA. Then, we upgrade ATA into 4-stride MATA. Finally, we improve MATA with interweaved name storage (denoted by MATA-NW).

First, in Section 6.1, we evaluate the memory efficiency of these methods. Then in Section 6.2, we conduct comprehensive evaluation and analysis of the throughput and latency of our name lookup engine — both the entire prototype engine and the core GPU part. The scalability of our lookup engine is evaluated in Section 6.3. Finally in Section 6.4, we evaluate its performance on handling name table updates.

6.1 Memory Space

If implemented with STT, the 3M and 10M name tables will take 19.49GB and 69.62GB, respectively. Compared with state transition table, ATA compresses storage space by 101× (on the 3M name table) and 102× (on the 10M name table), respectively. By constructing multiple smaller ATAs, MATA further compresses storage space — **MATA compresses storage space by 130× (on the 3M name table) and 142× (on the 10M name table), respectively**, easily fitting into the GTX590 GPU board of our prototype system, which has 3GB off-chip DRAM on board. (Note that input name interweaving changes storage layout but does not change storage space requirement.)

6.2 Lookup Performance

6.2.1 CPU-GPU System Performance

We now proceed to compare the lookup throughput and latency that can be achieved by the methods in comparison. Due to the excessive memory space requirement of STT, we hereby implement two small subsets of the 3M and 10M name tables, respectively, each containing 100,000 name entries. To explore the best achievable

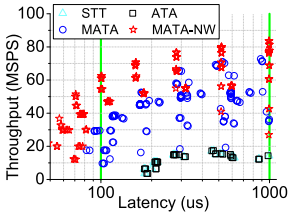

Figure 9: Throughput and latency on the 3M table's subset (average workload).

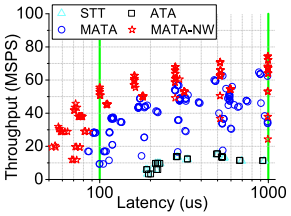

Figure 10: Throughput and latency on the 3M table's subset (heavy workload).

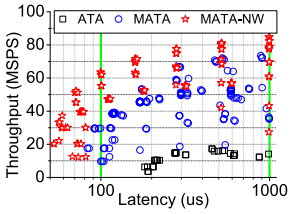

Figure 13: Throughput and latency on the 3M table (average workload).

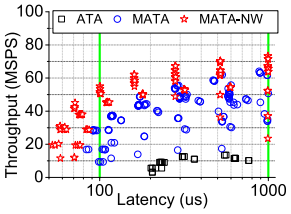

Figure 14: Throughput and latency on the 3M table (heavy workload).

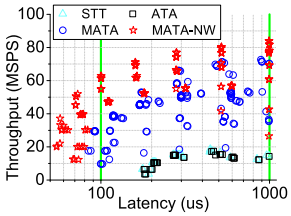

Figure 11: Throughput and latency on the 10M table's subset (average workload).

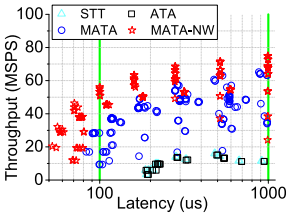

Figure 12: Throughput and latency on the 10M table's subset (heavy workload).

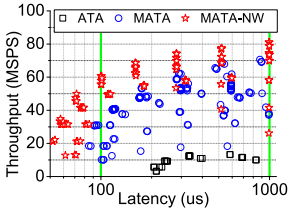

Figure 15: Throughput and latency on the 10M table (average workload).

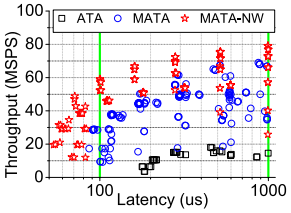

Figure 16: Throughput and latency on the 10M table (heavy workload).

lookup performance of each method, we run experiments with a wide range of parameter settings: doubling number of CUDA thread blocks from 8 to 4096, doubling number of threads per CUDA thread block from 32 to 1024, and doubling CUDA stream count from 1 to 4096. The measured lookup throughput and latency of the four methods are plotted in Figure 9-12, in which one point means the throughput and latency of the method with one parameter setting. (For legibility, we have only plotted results with less than 1ms latency.)

As we expected, STT and ATA have nearly identical performance, although ATA uses two orders of magnitude less memory. With multi-striding, MATA significantly outperforms ATA. STT and ATA have not been able to meet the 100μs latency requirement; in contrast, MATA can achieve up to 29.75 MSPS under average worload and 28.52 MSPS under heavy workload, while keeping latency below 100μs. With input name interweaving, MATA-NW further raises lookup throughput to 61.40 MSPS under average workload and 56.10 MSPS under heavy workload. The minimum lookup latency that can be achieved by MATA-NW is around 50μs (with about 20 MSPS lookup throughput), while the minimum achievable latency of STT and ATT is around 200μs. With a 200μs latency requirement, the maximum throughput that can be achieved by STT, ATA, MATA and MATA-NW are 6.59, 6.56, 52.99 and 71.12 MSPS under average workload and 6.02, 6.01, 49.04 and 63.32 MSPS under heavy workload, respectively; **MATA-NW achieves over 10× speedup.**

As the above two subsets are relatively small, we then conduct experiments based on the 3M and 10M name tables (without STT), and plot the results in Figure 13-16. The results are similar to what we observe on the two subsets. With 100μs latency requirement, MATA-NW can achieve 63.52 MSPS under average workload and 55.65 MSPS under heavy workload, translating to 127 Gbps under average workload and 111 Gbps under heavy workload, respectively.

6.2.2 GPU Engine Core Performance

The above experiments have not answered the following question — *which part of the prototype system is the performance bottleneck?* To answer this question, we conduct the following experiments for comparison[8]. (1) In experiment I, the GPU does not perform lookup on any input name and directly returns. The measured throughput represents the raw bandwidth of PCIe bus connecting CPU and GPU, and is plotted as "PCIe" in Figure 17-18; (2) In experiment II, the GPU performs one lookup on every input name. The measured throughput represents the normal throughput of our prototype system and is plotted as "MATA-NW". The experiment results reveal that system throughput is tightly bounded by PCIe bus bandwidth; although with overly large stream counts, system throughput starts dropping due to insufficient number of threads per stream[9].

As the PCIe bus limits the lookup throughput achievable with our prototype system, we conduct another set of experiments to evaluate the lookup throughput that can be achieved with our core algorithmic design (MATA-

[8]Here, lookup throughput is obtained with 16 concurrent CUDA thread blocks and 32 threads per CUDA thread block running in parallel. Because the above experiment results show that they produce the best lookup throughput under 100μs latency requirement.

[9]Recall that every warp of 32 threads must execute synchronously.

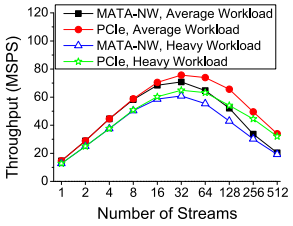

Figure 17: System throughput on the 3M table.

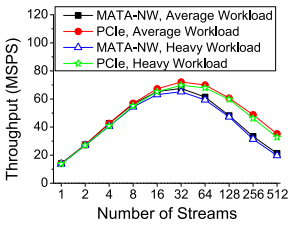

Figure 18: System throughput on the 10M table.

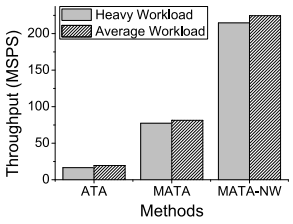

Figure 19: Kernel throughput on the 3M table.

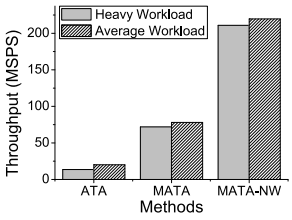

Figure 20: Kernel throughput on the 10M table.

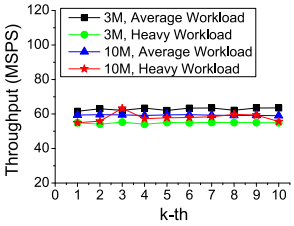

Figure 21: Growth trend of lookup throughput.

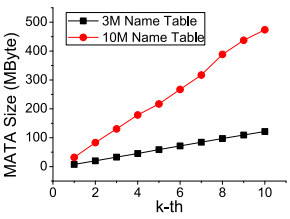

Figure 22: Growth trend of MATA size.

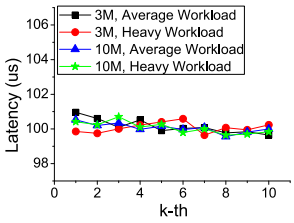

Figure 23: Growth trend of lookup latency.

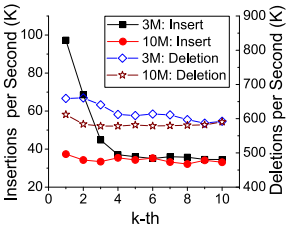

Figure 24: Growth trend of update performance.

NW) running on GPU. (1) First, we transmit input names from CPU to GPU's global memory in advance; (2) Then, we perform name lookup on GPU. Lookup results are not written back to CPU (via PCIe). When calculating lookup throughput, we do not include the time taken in (1). The calculated throughput represents the GPU's kernel throughput, without the bottleneck of PCIe bandwidth. As we can see in Figure 19-20, GPU's kernel throughput is more than two times higher than the entire engine throughput, reaching up to 219.69 MSPS, which is 96× of CPU-based implementation (MATA can perform 2.28 MSPS with one thread in CPU-based platform). These results demonstrate the real potential of our GPU-based name lookup engine design; this potential can be realized by high speed routers, which do not have the PCIe bandwidth problem.

6.3 Scalability

While our GPU-based name lookup engine is demonstrated to perform well on the 3M and 10M name tables, we are also interested in foreseeing its performance trend as name table size grows. For that, we partition each name table into ten equal-sized subsets, and progressively generate ten name tables for each of them; the kth generated name table consists of the first k equal-size subsets. Experiments are then conducted on these 20 generated name tables derived from the 3M name table and 10M name table. Measured results on lookup throughput, memory space requirement and lookup latency are presented in Figure 21-23, respectively.

As name table size grows, lookup throughput and lookup latency tend to stabilize around 60 MSPS and 100μs, respectively. The memory space requirement, represented by MATA size, tends to grow with linear scalability, which is consistent with our intuition.

6.4 Name table update

Finally, we measure the performance of our design on handling name table updates, both insertions and deletions. The results are reported in Figure 24. The general trend is that, the larger the name table, the more difficult it is to handle updates. Just like what we have observed on the growth trend of throughput and latency, update performance also tends to stabilize at a certain performance level. On both name tables, we can consistently handle more than 30K insertions per second. As we have described in Section 2, deletions are much easier to implement than insertions; we can steadily handle around 600K deletions per second. Compared with the current IP networks, which have an average update rate of several thousand per second, our name updating mechanism runs one order of magnitude faster.

7 Related Work

7.1 GPU-based Packet Processing

GPU as a high throughput parallel processing platform is attracting proliferating interest, and is being studied as a potential platform for high speed packet processing, such as IP lookup [10, 30, 19], packet classification [14, 20] and pattern matching [17, 7, 29, 24, 32, 16].

Pattern matching is a much simpler special form of name lookup, whose technical core is longest prefix string matching. While the initial study on name-based routing [22, 11, 25, 26] has revealed the feasi-

bility of routing based on hierarchical names instead of IP addresses, there has been lacking a comprehensive implementation-based study to address the following practical problems: (1) With large-scale name tables containing millions of names, how and to what extent can name tables be compressed for practical implementation; (2) What name lookup throughput can be reached under practical latency constraints; (3) What update performance can be obtained.

The existing methods for GPU-based pattern matching [17, 7, 29, 24, 32, 16] are designed targeting IDS-like systems where packet latency is not a first priority, and have all ignored the important issue of packet latency. As we have analyzed in Section 3 and demonstrated through experiments in Section 6, the latency-throughput tradeoff is rooted in GPU's design philosophy; optimizing throughput without considering latency constraints leads to overly optimistic results, and are not practically competent for high speed routers. By employing the multi-stream mechanism featured by NVIDIA Fermi architecture, which has been proven effective in other fields (e.g. [21, 13, 6]), our design is able to achieve wire speed lookup throughput with 50-100μs packet latency.

In fact, the existing GPU-based pattern matching methods have not even considered practical performance issues specific to such CPU-GPU hybrid systems, such as data transmission (e.g. via PCIe bus). Meanwhile, the existing methods have also not paid deserved attention to update performance, which is an important issue in high speed router design. The pattern sets used in their study are also multiple orders of magnitude smaller than what we target and have adopted in our experiments. On one hand, this requires more advanced compression techniques; on the other hand, high speed lookup can become even more challenging in the presence of more sophisticated compression. In contrast, our work is the first system-based study on GPU-based large-scale pattern matching and addresses all these performance issues: lookup throughput, latency, memory efficiency, update performance and CPU-GPU communication.

IP lookup is much simpler than name lookup. For example, in our implementation-based study, average name length is around 40 bytes, 10$\times$ longer than IP addresses used in GPU-based IP lookup research (e.g. PacketShader [10]). Unlike fixed length IP addresses, names are also variable in length, making it even more complex to implement efficient lookup. Moreover, name tables are 1-2 orders of magnitude larger than IP forwarding tables in terms of entry count, and 2-3 orders of magnitude larger in terms of byte count.

Packet classification is more complex than IP lookup in that packet classification rules typically check five packet header fields (13 bytes in total for IPv4) including two IP addresses. Nevertheless, packet classification rules are still much shorter than names, and are also fixed in length. In fact, packet classification rule sets are typically 1-3 orders of magnitude smaller than IP forwarding tables, let alone name tables.

In summary, GPU-based IP lookup and packet classification are not comparable/applicable to the large-scale name lookup problem studied in our work. Meanwhile, almost all of these GPU-based packet processing techniques (except PacketShader [10]) ignore the important issue of packet processing latency.

7.2 Algorithm & Data Structure

The technical core of name lookup is longest prefix matching. Before determining the matched longest prefix, hash-based methods have to perform multiple hash computations, which significantly degrade lookup performance [27]. In trie-based methods, to quickly determine the correct branch to transfer, hash techniques have been intensively studied. B. Michel et al. designed an incremental hash function called Hash Chain [18], improved by Zhou et al. [31], to aggregate URLs sharing common prefixes, while minimizing collisions between prefixes. These hash-based algorithms all have a common drawback — false positives due to hash collisions. More importantly, hash collision during trie traversal can lead name lookup to a wrong branch, causing packets to be forwarded to wrong ports; this undermines routing integrity — a fundamental property of NDN routers. So remedies for false positives are required in these systems. For eliminating false positive, we [26] proposed to encode all the name components for efficient traversal. When balancing hash collision probability and memory space requirement, we may not necessarily gain memory efficiency.

Tarjan and Yao proposed in their pioneer work [23] a method for compressing two-dimensional (state transition) tables into compact one-dimensional arrays — the original idea underlying basic ATA. In a follow-up work, Suwaiyel and Horowitz [5] proved that producing an optimal array is NP-hard, and proposed approximation algorithms for producing arrays. Our work develops beyond prior arts on the following aspects: (1) We propose a multi-stride ATA approach that can significantly improve storage efficiency, matching speed and lookup latency; (2) We propose the more advanced multi-ATA (MATA) design. Compared with the basic ATA design, MATA liberates multi-stride ATA from the constraint of available memory space. As a result, matching speed, latency, storage space and array construction time are all optimized substantially. We demonstrate through experimental evaluation that, while the naïve ATA design can be inadequate for large-scale, high throughput and low latency name lookup, the innovated MATA design is suf-

ficient for tackling such real applications.

7.3 Software Routers

Compared with hardware routers, software routers are cost-effective and much easier to program. Software routers can also provide extensible platforms for implementing CCN router prototypes.

PacketShader [10] exploits the GPU's massively-parallel processing power to overcome the CPU bottleneck in the current software IP routers. However, name lookup is more complex than IP lookup, for the variable and unbounded length of names as well as the much larger name tables. Compared with the IP FIB in [10], name FIB in our engine needs elaborate data structures to reduce memory consumption and speed up name lookup. Besides, PacketShader has not described any detail about how to balance lookup throughput and latency in PacketShader. However, the high performance packet I/O engine in PacketShader may help improve the performance of a whole CCN router.

In backbone routers, a high speed interface (e.g. 40 Gbps OC-768) is usually processed by a single data plane. RouteBricks [8] bundles multiple PCs together to handle packets, but one single PC is hard to handle the traffic from a high speed interface. Our work focuses on wire speed name lookup with a single data plane, which is different from RouteBricks. If more capacity or a larger number of ports is needed, we can apply a multi-machine approach in [8].

8 Discussion and Conclusion

8.1 Discussion

As described in Section 6.2.2, PCIe is the bottleneck of our name lookup engine, which has to use a GPU board installed on a PC via PCIe. However, it does not rule out the possibility of embedding GPUs into a high-end router through other non-PCIe means. Moreover, note that, although GTX590 GPU has two processor cores on chip, we have only used one of them in our experiments; using both processor cores can potentially boost performance as well.

Since CCN has not been standardized yet, its FIB table size and name length bound are still unknown. The 10M name table, collected with our maximum efforts, has only consumed one-sixth of the memory resource of GTX590. Thus we estimate our name lookup engine at least can handle a name table with 60M prefixes while keeping the lookup throughput. However, the name table would be orders of magnitude vaster when the prefixes cannot be aggregated effectively. As the first step, in this paper we demonstrate the feasibility of implementing wire speed name lookup on a table of substantial size. Scaling name tables to the larger conceivable size will be our future work.

Routing changes, including network topology changes, routing policy modifications and content publish/deletion, will cause FIB updates. Given no CCN/NDN network is deployed today, FIB update frequency cannot be accurately measured. We will estimate the update frequency from the current Internet. On one hand, the frequency of network topology changes and routing policy modifications in CCN can be inferred from the current IP network, which makes up to several thousand updates per second. On the other hand, content publish/deletion will not necessarily lead to FIB updates, since the name prefixes in FIBs are aggregated. If we assume the addition and deletion of domains will cause FIB updates, there are only a few tens of FIB updates per second, according to the top-level domain statistics [4] of the current Internet. Therefore, in this case, our approach can meet update performance requirements. Whether our FIB update mechanism meets practical performance requirements in real CCN networks needs to be further studied in future work.

8.2 Conclusion

Name lookup is one of the fundamental functions underlying a great variety of technological fields, among which wire speed CCN name lookup in large-scale name tables is the most challenging. Thus far, implementation-based study on real systems has been lacking. How much memory will CCN name tables consume? How fast can name lookup engine be implemented, from both technical and economic point of view? These are still unknown. In this paper, we propose a multi-stride character-trie algorithm, implemented in our GPU-based name lookup engine with a number of new techniques. Extensive experiments demonstrate that our GPU-based lookup engine can achieve up to 63.52 MSPS on name tables containing millions of names under a delay of less than 100μs. Our GPU-based implementation results answer that large-scale name lookup can not only run at high speed with low latency and fast incremental update, but also be cost-effective with today's off-the-shelf technologies.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd Michael Walfish for their help and invaluable comments.

References

[1] DMOZ - Open Directory Project. http://www.dmoz.org/.

[2] Fermat's theorem (stationary points). http://en.wikipedia.org/wiki/fermat's_theorem_(stationary_points).

[3] http://s-router.cs.tsinghua.edu.cn/namelookup.org/index.htm.

[4] Internet Statistics, http://www.whois.sc/internet-statistics/.

[5] AL-SUWAIYEL, M., AND HOROWITZ, E. Algorithms for trie compaction. *ACM Trans. Database Syst. 9*, 2 (June 1984), 243–263.

[6] BHATOTIA, P., RODRIGUES, R., AND VERMA, A. Shredder: GPUaccelerated incremental storage and computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)* (2012).

[7] CASCARANO, N., ROLANDO, P., RISSO, F., AND SISTO, R. iNFAnt: NFA pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev. 40*, 5, 20–26.

[8] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP'09, ACM, pp. 15–28.

[9] FREDKIN, E. Trie memory. *Commun. ACM 3*, 9 (Sept. 1960), 490–499.

[10] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 195–206.

[11] HWANG, H., ATA, S., AND MURATA, M. A Feasibility Evaluation on Name-Based Routing. In *IP Operations and Management*, G. Nunzi, C. Scoglio, and X. Li, Eds., vol. 5843 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 130–142.

[12] JACOBSON, V., SMETTERS, D. K., THORNTON, J. D., PLASS, M. F., BRIGGS, N. H., AND BRAYNARD, R. L. Networking Named Content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (2009), CoNEXT '09, ACM, pp. 1–12.

[13] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2011).

[14] KANG, K., AND DENG, Y. Scalable packet classification via GPU metaprogramming. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011* (march 2011), pp. 1 –4.

[15] L. HENNESSY, J., AND A. PATTERSON, D. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers, 2011.

[16] LIN, C.-H., LIU, C.-H., CHANG, S.-C., AND HON, W.-K. Memory-efficient pattern matching architectures using perfect hashing on graphic processing units. In *INFOCOM, 2012 Proceedings IEEE* (march 2012), pp. 1978 –1986.

[17] LIN, C.-H., TSAI, S.-Y., LIU, C.-H., CHANG, S.-C., AND SHYU, J.-M. Accelerating String Matching Using Multi-Threaded Algorithm on GPU. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE* (dec. 2010), pp. 1 –5.

[18] MICHEL, B., NIKOLOUDAKIS, K., REIHER, P., AND ZHANG, L. URL forwarding and compression in adaptive Web caching. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2000), vol. 2, pp. 670 –678.

[19] MU, S., ZHANG, X., ZHANG, N., LU, J., DENG, Y. S., AND ZHANG, S. IP routing processing with graphic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2010), DATE '10, pp. 93–98.

[20] NOTTINGHAM, A., AND IRWIN, B. Parallel packet classification using GPU co-processors. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists* (New York, NY, USA, 2010), SAICSIT '10, ACM, pp. 231–241.

[21] RENNICH, S. C/C++ Streams and Concurrency. http://developer.download.nvidia.com/.

[22] SHUE, C., AND GUPTA, M. Packet Forwarding: Name-based Vs. Prefix-based. In *IEEE Global Internet Symposium, 2007* (May 2007), pp. 73 –78.

[23] TARJAN, R. E., AND YAO, A. C.-C. Storing a sparse table. *Commun. ACM 22*, 11 (Nov. 1979), 606–611.

[24] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E., AND IOANNIDIS, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. vol. 5230. 2008, pp. 116–134.

[25] WANG, Y., DAI, H., JIANG, J., HE, K., MENG, W., AND LIU, B. Parallel Name Lookup for Named Data Networking. In *IEEE Global Telecommunications Conference (GLOBECOM)* (dec. 2011), pp. 1 –5.

[26] WANG, Y., HE, K., DAI, H., MENG, W., JIANG, J., LIU, B., AND CHEN, Y. Scalable Name Lookup in NDN Using Effective Name Component Encoding. In *IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)* (june 2012), pp. 688–697.

[27] WANG, Y., PAN, T., MI, Z., DAI, H., GUO, X., ZHANG, T., LIU, B., AND DONG, Q. NameFilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters. In *INFOCOM mini-conference 2013. Proceedings. IEEE* (2013).

[28] ZHANG, L., ESTRIN, D., JACOBSON, V., AND ZHANG, B. Named Data Networking (NDN) Project. http://www.named-data.net/.

[29] ZHAO, J., ZHANG, X., WANG, X., DENG, Y., AND FU, X. Exploiting graphics processors for high-performance IP lookup in software routers. In *INFOCOM, 2011 Proceedings IEEE* (april 2011), pp. 301 –305.

[30] ZHAO, J., ZHANG, X., WANG, X., AND XUE, X. Achieving O(1) IP lookup on GPU-based software routers. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 429–430.

[31] ZHOU, Z., SONG, T., AND JIA, Y. A High-Performance URL Lookup Engine for URL Filtering Systems. In *Communications (ICC), 2010 IEEE International Conference on* (may 2010), pp. 1 –5.

[32] ZU, Y., YANG, M., XU, Z., WANG, L., TIAN, X., PENG, K., AND DONG, Q. GPU-based NFA implementation for high speed memory efficient regular expression matching. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2012).