Journal of The Korea Society of Computer and Information
Vol. 21 No. 5, pp. 49-56, May 2016
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2016.21.5.049

High Performance IP Address Lookup Using GPU

Junghwan Kim*, Jinsoo Kim**

Abstract

Increasing Internet traffic and forwarding table size need high performance IP address lookup engine which is a crucial function of routers. For finding the longest matching prefix, trie-based or its variant schemes have been widely researched in software-based IP lookup. As a software router, we enhance the IP address lookup engine using GPU which is a device widely used in high performance applications. We propose a data structure for multibit trie to exploit GPU hardware efficiently. Also, we devise a novel scheme that the root subtrie is loaded on Shared Memory which is specialized for fast access in GPU. Since the root subtrie is accessed on every IP address lookup, its fast access improves the lookup performance. By means of the performance evaluation, our implemented GPU-based lookup engine shows 17~23 times better performance than CPU-based engine. Also, the fast access technique for the root subtrie gives 10% more improvement.

▸ Keyword : IP address lookup, Router, GPU, Multibit trie, CUDA

I. Introduction

지속적으로 증가하는 인터넷 트래픽에 대응하기 위해서는 물리적 전송선의 대역폭 증가도 필요하지만, 라우터의 성능도 고도화하는 것이 요구된다. 가령, OC-3072의 경우 159.252 Gbps의 대역폭을 갖는데, 이 경우 라우터에서는 최소 이더넷 프레임 (64 bytes) 기준으로 약 초당 3.1억 개 이상의 패킷 처리 성능이 요구된다. 즉, 라우터에서 IP address lookup을 초당 3억 번 이상 처리할 수 있는 성능을 갖춰야 한다.

IP address lookup은 라우터의 주요 기능으로써, 주어진 패킷의 목적지 IP address에 대해 nexthop 정보를 찾아내는 작업이다[1]. CIDR(Classless Inter-Domain Routing)[2]이 도입된 이후 IP address lookup은 LMP(Longest Matching Prefix)를 찾는 과정이 되어 시간이 많이 걸리는 작업이 되었으며, 라우팅 테이블의 prefix의 개수 또한 지속적으로 증가하고 있는 추세이다[3].

IP address lookup에 사용되는 포워딩 테이블(forwarding table)은 <prefix, nexthop> 쌍들로 구성되며, lookup 과정에서 주어진 목적지 IP address에 대한 LMP를 찾게 된다. 그동안 IP address lookup의 성능을 향상시키기 위한 많은 연구들이 진행되어 왔는데, 크게 하드웨어에 기반하거나[4-6], 또는 소프트웨어에 기반한 자료구조와 알고리즘들[7-9]가 연구되어 왔다.

한편, 그래픽 처리에 사용되던 GPU(Graphic Processing Unit)는 근래에 그래픽에 국한되지 않고 범용적으로 고성능 컴퓨팅에 사용되기 시작했으며, 이를 GPGPU(General Purpose GPU)라고 부른다[10, 11]. GPU는 대규모의 병렬 unit들을 갖고 있어, 데이터 병렬성이 많은 작업을 처리하기에 적합하다. GPU는 호스트 컴퓨터에 장착되어 있는 디바이스이므로 호스트의 CPU와는 다른 방식의 프로그램 작성이 필요하다. NVIDIA의 CUDA 플랫폼은 기존의 프로그래밍 언어를 확장하여 사용할 수 있게 함으로써, GPU 프로그래밍을 용이하게 하였으며[12], CUDA 기반의 많은 GPU 응용들이 출현하였다.

라우터에 들어오는 수많은 IP address들에 대한 lookup은 대규모의 데이터 병렬성을 갖고 있으므로, GPU를 활용하기에 적합하

• First Author: Junghwan Kim, Corresponding Author: Jinsoo Kim
*Junghwan Kim (jhkim@kku.ac.kr), Dept. of Computer Engineering, Konkuk University
**Jinsoo Kim(jinsoo@kku.ac.kr), Dept. of Computer Engineering, Konkuk University
• Received: 2016. 02. 19, Revised: 2016. 03. 21, Accepted: 2016. 04. 26.
• This paper was written as part of Konkuk University's research support program for its faculty on sabbatical leave in 2014.

다. 본 연구에서는 CUDA 기반의 GPU를 사용하였으며, 기존 연구와 차별하여 효율적인 multibit trie 구조와 GPU의 shared memory를 활용한 빠른 접근 방법을 제시하였다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구를 소개하고, 3절에서는 multibit trie를 기반으로 하는 IP address lookup에 대해 설명한다. 4절에서는 NVIDIA 사의 GPU를 중심으로 구조와 성능 관점을 소개한다. 5절과 6절은 제안하는 기법과 성능 평가 결과를 각각 기술한다. 마지막으로 7절에서 결론을 맺는다.

II. Related Works

고성능 IP address lookup에 대한 연구는 하드웨어에 기반하거나 또는 소프트웨어에 기반한 방식들이 있다. 하드웨어 기반으로는 TCAM을 이용한 병렬 lookup 방식[4, 13], FPGA로 전용 하드웨어를 설계한 방식[5], 파이프라인 방식의 하드웨어[6] 등 여러 연구가 있다.

소프트웨어 방식은 IP address lookup을 위한 전용 하드웨어 대신 일반적인 프로세서-메모리 구조에 기반한 처리에 의존한다. 이러한 방식에서는 메모리 접근 시간 또는 접근 회수를 최소화할 수 있도록 자료구조와 알고리즘을 설계한다. 이러한 연구로는 multibit trie[7], LC-trie[8], tree bitmap[9]를 포함한 수많은 자료구조가 제안되어 왔다.

GPU는 범용 컴퓨터는 아니지만, CUDA를 통해 통상적인 프로그램을 작성, 적용할 수 있게 되어 많은 병렬 컴퓨팅에 사용되고 있다. IP address lookup은 라우터에 입력으로 들어오는 수많은 패킷들에 대해 병렬적으로 적용할 수 있으므로, GPU의 병렬 프로그래밍을 사용하기에 적합하다. 이러한 연구로는 GALE, PacketShader, GAMT 등이 있다[14-17].

GALE[14]는 prefix들이 24 bits로 확장된 거대한 하나의 포워딩 테이블을 사용한다. 한 번의 메모리 접근으로 lookup을 완료할 수 있는 장점이 있지만, 길이가 25 bits 이상인 prefix들이 드물지만 존재함에도 불구하고 처리할 수 없다. 또한 24 bits로 확장되었기 때문에 포워딩 테이블의 갱신 부담이 매우 크고, 갱신을 병렬화할 수도 없다.

PacketShader[15]에서는 GPU를 이용한 고속의 소프트웨어 라우터를 제시하였는데, IP address lookup 뿐 아니라 다른 라우터의 기능들에 대해서도 고려하였다. 또한 Master/Worker 기반하여 CPU 코어와의 병렬성도 고려하였다. 그러나, 포워딩 테이블이 24-8 multibit trie에 기반하고 있어 테이블 갱신 부담이 매우 크다는 문제를 갖고 있다.

GAMT[16]은 임의의 stride로 multibit trie를 구성할 수 있지만, 32 bits 엔트리에서 nexthop 정보를 위해 불필요하게 많은 24 bits를 사용하고, 매 단계에서 다음 level로 진행하는게 명확함에도 불구하고 불필요하게 level 값으로 8 bits를 저장하고 있다. 이러한 것들은 GPU에서 shift 연산과 불필요한 읽기 연산을 유발하게 된다. 또한 32 bits 크기에서 이러한 정보를 담기 위해 disjoint multibit trie를 사용하는데, 이는 각 subtrie가 독립적으로 갱신되지 않기 때문에 갱신 부담이 크다.

Chu 등이 제안한 기법[17]은 multibit trie를 저장할 때 internal subtrie와 leaf subtrie의 구조를 달리함으로써, 메모리를 절약하고 접근 속도를 높인다. 즉, leaf subtrie의 경우, 각 엔트리에서 자식에 대한 index를 저장할 필요가 없으므로 nexthop 정보 8 bits만 저장하게 된다. Internal subtrie의 경우, 엔트리 크기는 32 bits인데, 이 중 자식을 나타내는 index는 20 bits이므로 2^{20} words로 주소 공간이 한정된다. 또한 internal stride의 경우 level 별로 다양하게 구성할 수 없고 고정되어야 하는 문제가 있다.

III. Multibit Trie for IP Lookup

1. IP address lookup

많은 소프트웨어 방식의 IP address lookup 기법들은 trie 구조에 기반을 두고 있다. 그림 1은 a, b, c, d, e 등 5개 prefix로 구성되는 prefix table과 그에 대응되는 binary trie를 보여준다. 그림에서 보는 바와 같이, 주어진 입력의 각 bit가 0 또는 1이냐에 따라, binary trie의 각 노드에서 왼쪽 또는 오른쪽으로 따라 내려가며 prefix를 찾을 수 있다. 예를 들어, prefix a = 0*는 trie의 root에서 0을 나타내는 왼쪽 자식 노드에서 찾을 수 있다.

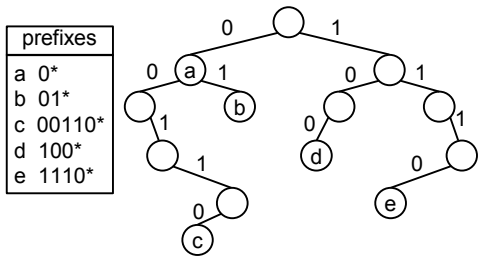


Fig. 1. Prefix Table and Binary Trie

IP address lookup은 matching되는 여러 개의 prefix 중 LMP를 최종 검색 결과로 결정한다. 예를 들어 그림 1의 binary trie에 목적지 IP address로 ip_1 = 001100인 패킷이 들어왔다고 하자(편의상 IP address는 32 bits가 아닌 6 bits로 가정한다). 그러면 이 ip_1에 matching되는 prefix는 a와 c이며 각각 1 bit와 5 bits 길이로 matching되므로 LMP는 c가 된다. 이 경우 최소 5번의 메모리 접근이 필요하다. IPv4의 경우 prefix의 최대 길이는 32이므로, 최대 32번의 메모리 접근이 요구된다.

2. Multibit trie

Binary trie는 하나의 bit에 대해 한 번의 메모리 접근이 필

요한데, 만일 여러 개의 bit를 한꺼번에 indexing할 수 있다면, 메모리 접근 회수를 크게 줄일 수 있다. 이러한 구조가 multibit trie이다. 그림 2는 그림 1에 대해 stride 3으로 multibit trie를 구성한 예를 보여 준다. 그림에서 보는 바와 같이 모든 노드는 $8(=2^3)$개의 자식 노드를 가질 수 있다. 3 bits가 한 개의 level로 압축되었기 때문에, 그 중간에 있는 prefix들은 모두 길이가 3의 배수로 확장된다. 그림에서 prefix a와 b의 경우, 각각 a_1, a_2와 b_1, b_2로 확장된다. 또한 prefix c와 e는 각각 c_1, c_2와 e_1, e_2, e_3, e_4로 확장되었다. 목적지 IP address ip_1 = 001100에 대한 LMP인 prefix c(=c_1)는 2번의 메모리 접근으로 찾을 수 있다.

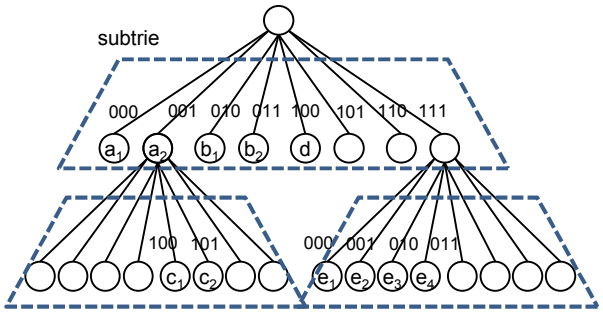


Fig. 2. Multibit Trie

일반적으로 k-bit trie의 경우, 최대 2^k개의 자식 노드를 가질 수 있고, 한 번의 메모리 접근으로 k-bit를 indexing할 수 있기 때문에 접근 회수를 $1/k$로 줄여 속도를 향상시킬 수 있다. 그러나 prefix 확장에 따라 메모리 사용량이 늘어날 수 있다. 포워딩 테이블에 prefix가 추가되거나 삭제되는 갱신이 일어나는 경우, multibit trie의 여러 노드가 영향을 받지만, 그 범위는 subtrie로 한정된다. 그림 2에서 subtrie는 총 3개가 있으며, 갱신은 subtrie 단위로 독립적으로 처리될 수 있다.

모든 intermediate prefix를 leaf까지 확장시키면, 하나의 목적지 IP address에 대해 항상 한 개의 prefix만 매치된다. 이러한 기법을 *leaf pushing*이라고 하며[7], 그림 3은 leaf pushing을 통해 생성된 multibit trie를 보여준다.

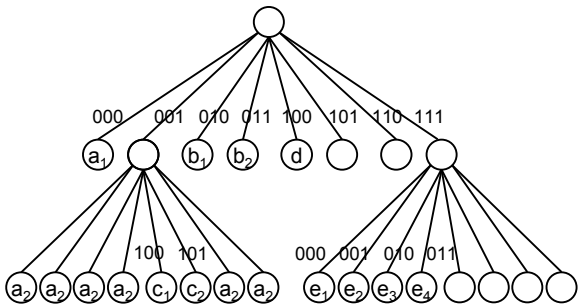


Fig. 3. Multibit Trie (disjoint)

그림 2의 첫 번째 level의 subtrie에 있는 prefix a_2는 그림 3에서 두 번째 level의 subtrie로 확장되었다. 그림 2에서 ip_1 = 001100은 두 개의 prefix a와 c_1이 matching되었지만, 그림 3은 한 개 prefix c_1 만이 matching된다. 그림 2에서는 LMP를 찾아야 하기 때문에 각 level에서 matching되는 prefix의 nexthop 정보를 기억해야 하지만, 그림 3은 그렇지 않다. 그림 3과 같이 leaf-pushing한 trie는 모든 prefix가 leaf node이므로 어떤 prefix도 다른 prefix의 prefix가 될 수가 없기 때문에 *disjoint* trie라고 하며, 임의의 IP address에 대해 오직 1개의 prefix만 matching된다.

그림 2에서 prefix a_2를 나타내는 노드는 prefix의 nexthop을 나타내는 정보와 자식 subtrie에 대한 포인터 정보를 모두 가지고 있어야 하지만, 그림 3에서는 자식 subtrie에 대한 포인터만 갖고 있으면 된다. Disjoint trie의 모든 노드는 nexthop 또는 포인터 정보 중 한 가지만 저장하면 되므로 메모리 소요량을 줄일 수 있다.

그러나 disjoint multibit trie는 생성 시 leaf-pushing을 위한 부수적인 작업이 필요할 뿐 아니라, prefix 갱신에 따른 오버헤드가 매우 크다는 단점을 갖고 있다. 예를 들어 prefix a의 삭제는 그림 2에서 한 개의 subtrie에 국한되지만, 그림 3의 경우 두 개의 subtrie 갱신이 필요하다.

IV. CUDA-enabled GPU

1. NVIDIA GPU architecture

NVIDIA는 GPU 컴퓨팅을 지원하기 위한 CUDA(Compute Unifed Device Architecture) 환경을 제공한다. CUDA에서 사용자 프로그램은 호스트 수행 부분과 GPU 수행 부분으로 나뉘며, GPU 수행 부분을 특히 kernel이라고 부른다. Kernel은 수많은 thread들에 의해 병렬 수행된다.

GPU 구조는 그림 4와 같이 다수의 Multiprocessor와 Device Memory로 구성된다. 한 개의 Multiprocessor 안에는 다시 수십 개의 실행 코어(그림에서 P_0 ~ P_k)가 들어 있는데, 이들 코어에 의해 thread가 수행된다. 따라서 전체적으로 수백 개의 thread가 병렬적으로 수행된다.

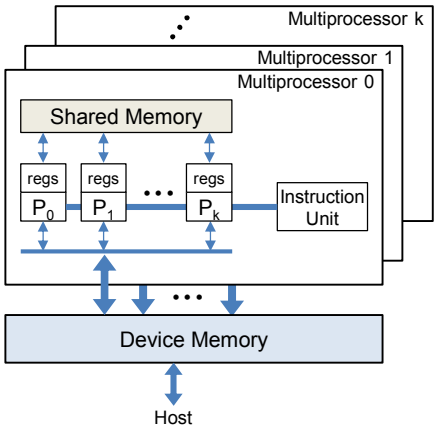


Fig. 4. NVIDIA GPU Architecture

Thread들은 block이라는 단위로 묶여 실행되는데, 한 개의 block은 여러 개의 Multiprocessor에 걸쳐 할당될 수 없다. Instruction Unit이 지시하는 동일한 instruction을 수행하는 thread들의 단위를 warp라고 하며, 이 크기는 32로써 block 크기는 warp 크기의 배수가 되는 것이 좋다.

프로그래밍 모델 관점에서 Global Memory는 모든 thread block에서 접근 및 공유할 수 있는 영역이며, 물리적으로는 그림 4의 Device Memory에 위치한다. 반면, Local Memory는 개별 thread에 의해 접근되는 영역으로, 물리적으로는 역시 Device Memory에 위치한다. Global Memory는 GPU의 thread 뿐 아니라 호스트에 의해서도 접근된다. 호스트 프로그램은 필요한 데이터 영역을 Global Memory에 할당하고 송수신하게 된다.

Shared Memory는 Global Memory와 달리 하나의 block 내의 thread들에 의해서만 공유될 수 있지만, 매우 빠른 접근이 가능하다. 따라서 반복적으로 사용되는 데이터는 Global Memory 보다는 Shared Memory에 두는 것이 성능에 좋다. 그러나, Shared Memory는 cache와 달리 프로그램에서 직접 데이터를 적재하고 명시적으로 접근해야 한다. 또한 공유할 수 있는 범위가 block 내의 thread들로 한정되어 있으므로, 매 block이 시작될 때마다 필요한 데이터를 Global Memory에서 Shared Memory로 적재해주어야 한다.

IP address lookup에 GPU를 사용하기 위해서는 포워딩 테이블을 사전에 Global Memory에 적재해야 하는데, 이는 cudaMemcpy()와 같은 CUDA 함수를 호스트에서 실행함으로써 완료된다. 포워딩 테이블의 적재는 lookup 성능에는 영향을 미치지 않는다. Lookup을 수행할 대상 IP address들은 역시 cudaMemcpy() 함수에 의해 Global Memory로 옮겨지고, 이후 GPU kernel 프로그램을 실행함으로써 lookup이 진행된다. 마지막으로 수행 결과인 nexthop 정보들을 cudaMemcpy() 함수에 의해 Global Memory에서 호스트 메모리로 옮기게 된다.

2. Performance optimization

GPU 프로그램은 CUDA에 의해 자동으로 병렬 수행되지만, 프로그래머는 좀 더 낮은 관점의 사항들을 고려해야 고성능을 얻을 수 있다.

첫째, Multiprocessor 내의 thread 병렬성을 최대화해야 하는데, 이를 위해서는 레지스터 이용과 실행 코어 이용을 극대화할 수 있는 block 크기를 찾아야 한다.

둘째, instruction 수준의 이용률을 극대화하기 위해서 제어문 사용을 최소화하고, 수행 연산 개수를 최적화한다.

마지막으로 memory 접근 시간을 줄이는 것으로써, 성능 매우 큰 영향을 미친다. Global Memory는 모든 thread에 의해 접근 가능하긴 하지만, 접근 시간이 길기 때문에 접근 회수를 최소화할 수 있도록 coalescing과 alignment가 되어야 한다. 또한 반복적으로 사용되는 데이터는 Shared Memory를 사용하여 접근 시간을 줄이는 것이 성능에 큰 영향을 미친다.

V. Proposed Scheme

1. Efficient multibit trie structure

Disjoint multibit trie는 각 엔트리에서 자식 subtrie 또는 nexthop 정보, 둘 중의 하나만 저장하면 되지만, 갱신의 부담이 매우 크다. 따라서 본 논문에서는 non-disjoint multibit trie를 사용하지만, 여전히 32 bits로 엔트리를 표현하고 이를 효과적으로 접근하는 방법을 제시한다.

그림 5는 본 논문에서 multibit trie 기반의 IP address lookup을 수행하기 위한 알고리즘을 보여준다. 하나의 엔트리는 <child, nexthop>의 두 개 필드로 구성된다. child는 자식 subtrie를 가리키는데, 포인터 대신 정수형 인덱스를 사용하여 배열 내의 위치를 지정한다.

```
Algorithm IP_Lookup
Input: mtrie, ip
Output: nexthop
 1 level = 0;
 2 child_subtrie = 0;
 3 do {
 4   Extract an index idx from ip for current level ;
 5   subtrie = address of child_subtrie in mtrie;
 6   if (subtrie[idx].nexthop != INVALID)
 7     nexthop = subtrie[idx].nexthop;
 8   child_subtrie = subtrie[idx].child;
 9   level++;
10 } while (child_subtrie != NO_CHILD);
11 return nexthop;
```

Fig. 5. IP Address Lookup in Multibit Trie

결국, 문장 3에서 문장 10까지를 빠르게 처리하는 것이 lookup 성능을 결정할 것이다. 먼저, 문장 4에서 목적지 ip로부터 현재 level에 해당하는 인덱스부분을 추출하기 위해서는 ip를 우측 shift한 후, bitwise-and를 통한 masking 연산이 필요하다. 예를 들어 ip = 0xa23b5501 이고, stride가 8-8-8-8의 4개 level로 구성되어 있다고 하자. 현재 두 번째 level을 진행하고 있다면, (ip & 0x00ff0000) >> 24 와 같은 연산을 수행해서 인덱스 0x3b를 얻을 수 있다. 이러한 연산을 고속으로 수행하기 위해서는 사전에 level 별 shift 크기와 masking bits를 배열에 보관하는 것이 바람직하며, 따라서 문장 4는 다음과 같이 구현된다.

```
idx = ( ip & mask [ level ] ) >> shift_amt [ level ];
```

이때 배열 mask와 shift_amt는 Shared Memory에 적재함으로써 매우 빠르게 접근할 수 있다.

다음으로 고려해야 할 부분은 mtrie의 각 엔트리에 대한 접근이다(알고리즘에서 문장 5~8). mtrie는 multibit trie를 담고 있는 배열로 Global Memory에 사전에 적재된다. 그림 6은 multibit trie를 GPU에서 효율적으로 사용할 수 있도록 설계된 구조를 나타낸다. 각 엔트리는 32 bits 크기를 가지며, 이중 16 bits는 child_subtrie를, 8 bits는 nexthop를 저장한다. nexthop은 특별한 용도의 1 bit를 제외한 나머지 7 bits를 사용하며, INVALID 값을 포함하여 총 128개를 표현할 수 있다. child_subtrie를 지정하기 위해 24 bits를 사용할 수도 있지만, 이 경우 alignment되지 않은 값을 사용하는데 따른 부가 연산을 유발하므로 16 bits로 한정하였다. 이때, child_subtrie는 전체 multibit trie에서의 엔트리 인덱스가 아니라, level 안에서의 subtrie 번호이다. 그림 6에서 각 level의 stride는 2로 가정하였다. 따라서 매 4 개 엔트리마다 하나의 subtrie 번호가 부여된다. 16 bits를 사용할 수 있으므로 한 level 안에는 최대 65536 개의 subtrie가 존재할 수 있다.

그리고 child_subtrie는 level 안에서 subtrie 번호이므로, 실제 child subtrie의 위치를 찾기 위해서는 level의 시작 위치와 subtrie 한 개의 크기를 알아야 한다. 그림 6에서 배열 first_in_level은 각 level 별 시작 위치를 나타낸다. Root subtrie의 시작 위치는 0으로 지정된다. 그림 5 알고리즘을 배열 first_in_level을 사용하도록 수정하면, 문장 5의 subtrie(= child subtrie의 실제 시작 위치)는 다음과 같이 계산된다.

```
subtrie = & mtrie[ first_in_level [ level ]
        + child_subtrie * size_of_subtrie ];
```

여기서 size_of_subtrie는 level의 stride에 따라 달라지므로 pow(2, stride[level])로 계산될 수 있을 것이다. 그러나, 매번 size_of_subtrie를 계산하는 대신 미리 계산된 값을 배열에 저장하여 사용한다.

배열 first_in_level을 사용함으로써, child_subtrie는 level의 시작 위치로부터 상대적인 subtrie 개수만을 나타내면 되어 16 bits 크기만으로 표현 가능하게 되었다.

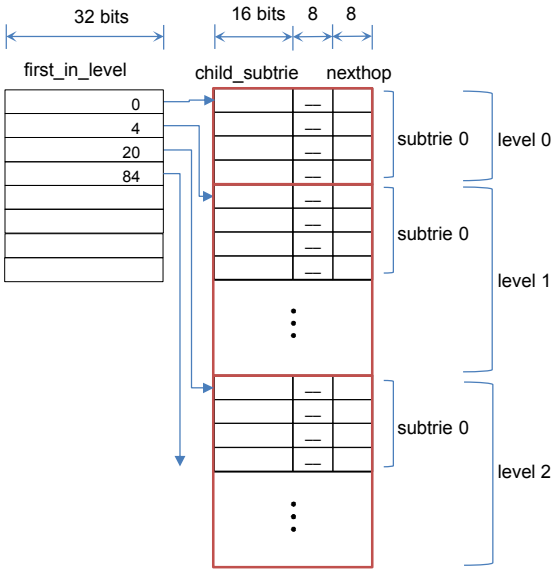


Fig. 6. Multibit Trie for GPU

2. Fast access using shared memory

매 IP address lookup에서 빈번하게 사용되는 데이터들은 GPU의 Shared Memory에 적재함으로써, 보다 빠르게 처리할 수 있다. 본 연구에서 제안한 구조에서 배열 mask, shift_amt, first_in_level, size_of_subtrie는 매 level 마다 빈번하게 사용되는 데이터이므로 Shared Memory에 적재한다. 또한 root subtrie(level 0의 subtrie)는 매 lookup에서 반드시 접근되는 구조이므로 역시 Shared Memory에 적재한다.

구현에 사용한 GPU에서 1개 thread block 당 사용할 수 있는 최대 Shared Memory 용량은 48 KB이다. 이 용량 하에서 root subtrie의 최대 엔트리 개수는 12 K이고, 따라서 level 0의 stride는 13이하 이어야 한다. 매 thread block 시작 시마다 Shared Memory에 데이터를 새로 적재해주어야 하므로, root subtrie가 커지면, 적재 부담도 따라서 커진다. 제안하는 구조에서는 level 0의 stride를 8로 고정하였으며, 이때의 root subtrie는 256 개의 엔트리로 Shared Memory에 적재하기에 충분하다.

Global Memory는 32-, 64-, 128-byte 단위로 트랜잭션을 처리할 수 있으므로, root subtrie를 Global Memory로부터 Shared Memory로 옮기는 데는 최소 8번의 트랜잭션이 필요하다. 이러한 부담을 줄이기 위해 root subtrie에 대해서는 그림 7과 같이 각 엔트리를 32 bits 대신 16 bits로 새롭게 구성한다. nexthop은 그대로 8 bits를 유지하고, child_subtrie를 16 bits에서 8 bits로 줄인다. Root subtrie의 엔트리 개수가 256이므로 다음 level에서 가능한 subtrie 개수는 256개를 넘지 않는다. 따라서, 8 bits이면 child subtrie를 지정할 수 있다.

다만, child subtrie가 아예 없는 경우를 표현해야 하는데, 이 경우 nexthop의 최상위 1 bit를 사용한다.

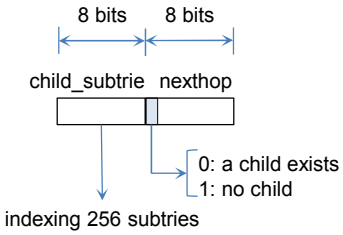


Fig. 7. Root Subtrie Entry

VI. Performance Evaluation

1. Experiment environment

성능 평가를 위해 구현된 시스템의 사양은 표 1 및 표 2와 같다. 호스트 시스템은 리눅스 운영체제를 사용하며, 장착된 GPU의 코어 개수는 192개이다.

객관적인 성능 평가를 위해 실제 포워딩 테이블을 Route View Project[18]의 2015년 1월 1일 0시 RIB(Routing Information Base)로부터 추출하여 사용하였다. 포워딩 테이블의 총 prefix 개수는 552,981이다. 또한 실험에 사용된 trace는 지역성을 내포하도록 패킷들을 랜덤하게 발생하여 얻었으며, 사용된 총 패킷은 2,647,663 개 이다.

Table 1. Host System Specification

Item	Specification
CPU	Intel(R) Core(TM) i7-2600
CPU Clock Speed	3.4 GHz
Main Memory	4 GB

Table 2. GPU Specification

Item	Specification
GPU	GeForce GTX 550 Ti
GPU Clock	1.8 GHz
CUDA Capability	2.1
Global Memory	1 GB
Multiprocessors	4
CUDA Cores	192 (= 4 * 48)
Total Shared Memory / Block	48 KB
Max. Threads / Block	1024

2. Experiment result

주어진 포워딩 테이블을 갖고 8-8-8-8, 8-10-6-8, 8-16-8의 3가지 stride 조합에 대해 multibit trie를 생성하여 비교하였다. 각 stride 조합에 대해 level 별 subtrie 수는 표 3과 같다. 괄호 안은 실제 차지하는 메모리 용량(KB)을 나타낸다. 길이가 8보다 작은 prefix는 없고, level 0에서 Shared Memory를 이용한 빠른 접근을 수행하기 위해 level 0의 stride는 8로 고정하였다. 따라서, level 0의 subtrie가 갖고 있는 총 엔트리 개수는 256개이며, 용량은 0.5 KB가 된다. 각 level의 엔트리 크기는 32 bits이지만, 본 연구에서 level 0의 엔트리 크기는 그림 7과 같이 16 bits로 제안되었다. 0.5 KB는 Shared Memory에 적재되기에 충분히 작을 뿐 아니라, 매 thread block이 시작될 때마다 적재 작업을 하기에도 적합하다.

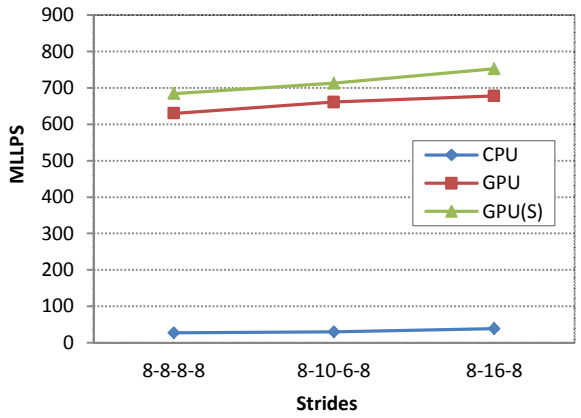
Table 3. Number of Subtries

Stride	Level 0	Level 1	Level 2	Level 3
8-8-8-8 (Total KB)	1 (0.5)	211 (211)	21295 (21295)	4430 (4430)
8-10-6-8 (Total KB)	1 (0.5)	211 (844)	52815 (13204)	4430 (4430)
8-16-8 (Total KB)	1 (0.5)	211 (54016)	4430 (4430)	N/A

전체 multibit trie가 차지하는 용량은 8-8-8-8의 경우, 25,936.5 KB, 8-10-6-8은 18,478.5 KB, 그리고 8-16-8은 58,446.5 KB로써, 1 GB 용량을 갖는 Global Memory에 적재하기에 충분히 작다.

그림 8은 각 stride 조합에 대해 IP address lookup의 성능을 보여준다. 실험에 사용한 GPU는 low-end임에도 불구하고 lookup 성능은 CPU에 비해 약 17~23배로 월등히 높다. 또한 그림에서 GPU(S)는 level 0의 subtrie를 Shared Memory에 적재하는 방식을 나타내며 본 논문에서 고안한 기법이다. GPU(S)는 그렇지 않은 경우에 비해 약 10% 정도 추가 향상된 성능을 보여주며, 최고 752.2 MLPS(Million Lookups Per Second)의 성능에 도달하고 있다.

Stride 조합별로 비교하였을 때, 8-16-8이 가장 우수한 성능을 보여준다. 이는 다른 stride 조합과 비교하였을 때 level 수가 하나 적을 뿐 아니라, 대부분의 prefix 가 24 bits이하이어서 두 번째 level에서 lookup이 완료되기 때문이다. 그러나, 16-bit stride는 한 개의 prefix 갱신 시에 최대 65,536 개의 엔트리 수정을 유발할 수 있어 부담이 크다. 8-8-8-8이 8-16-8과 비교하여 많은 성능 차이가 나지 않으면서도 갱신 부담이 작은 stride 조합으로 생각된다.



MLPS = Million Lookups Per Second
CPU = lookup in host only
GPU = lookup in GPU w/o Shared Memory
GPU(S) = lookup in GPU w/ Shared Memory

Fig. 8. Comparison of Throughput

그림 9는 제안하는 기법인 GPU(S)에 대해 thread block 크기를 달리했을 때 성능 변화를 보여준다. 전반적으로 한 block 당 thread 개수가 많을수록 높은 성능을 보이며, 8-8-8-8 stride의 경우 최대 block 크기인 1024에서 가장 높은 성능인 684.2 MLPS에 도달한다. 매 block을 수행할 때마다 level 0의 subtrie를 Shared Memory에 적재해야 하기 때문에, block 크기가 클수록 이 부담이 작아지는 것으로 분석된다.

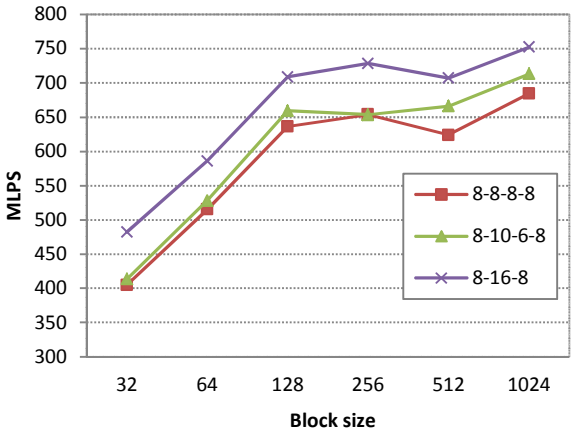


Fig. 9. Throughput over Block Size

VII. Conclusion

IP address lookup의 성능은 라우터의 성능에 큰 영향을 끼치는 매우 중요한 기능으로써, 급증하는 인터넷 트래픽과 포워딩 테이블의 크기 증가에 따라 그 중요성은 더 커지고 있다.

본 논문에서는 최근 저비용 고성능 컴퓨팅 장비로 널리 활용되는 GPU를 사용하여 IP address lookup 엔진을 구현하였다. GPU의 병렬처리를 극대화하기 위해서는 Global Memory 접근을 최소화해야 하며, 이를 위해 multibit trie에서 각 subtrie를 level별로 구분하여 인덱스하는 방법을 사용하였다. 성능 평가를 통해 호스트 컴퓨터에서 구현했을 때와 GPU에서 구현했을 때 최고 23배의 성능 차이가 있음을 확인했다. 또한 본 논문에서 고안한 root subtrie의 Shared Memory 적재 기법을 적용했을 때, 성능이 추가적으로 10% 향상되었다.

제안된 기법은 다양한 stride 크기에 대해 비교 분석되었다. 8-16-8 stride가 lookup 성능 자체만 고려하면 가장 우수하였다. 한편 8-8-8-8 stride는 8-16-8 stride에 비해 prefix 갱신은 매우 용이하면서도 성능 저하는 아주 작음이 실험을 통해 확인되었다. 또한 이때의 성능은 684.2 MLPS로써, OC-3072에서 최대 요구되는 311 MLPS를 만족하기에 충분하다.

포워딩 테이블의 갱신과 IP address lookup의 병행 처리, 그리고 호스트 컴퓨터에서 lookup의 일부 기능을 병행 처리함으로써 성능을 극대화하는 것은 향후 과제로 남는다.

REFERENCES

[1] Mi. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," IEEE Network, March/April 2001.

[2] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," RFC1519, Sep. 1993.

[3] G. Huston, "BGP routing table analysis reports," http://bgp.potaroo.net/.

[4] Y. Sun, H. Liu, and M. S. Kim, "Using TCAM efficiently for IP route lookup," Proc. of 8th IEEE Consumer Communications and Networking Conference, pp. 816-817, 2011.

[5] Hoang Le, Weirong Jiang, and Viktor K. Prasanna, "Memory-Efficient IPv4/v6 Lookup on FPGAs Using Distance-Bounded Path Compression," Proc. of IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 242-249, 2011.

[6] Weirong Jiang, and Viktor K. Prasanna, "Parallel IP Lookup Using Multiple SRAM-Based Pipelines," Proc. of 22nd International Parallel and Distributed Processing Symposium (IPDPS), pp. 1-14, 2008.

[7] V. Srinivasan, and G. Varghese, "Fast address

lookups using controlled prefix expansion," ACM Transactions on Computer Systems (TOCS): Vol. 17, Issue 1, pp. 1-40, Feb. 1999.

[8] S. Nilsson, and G. Karlsson, "IP-Address Lookup Using LC-Tries," IEEE Journal on Selected Areas in Communications, Vol. 17, Issue 6, pp. 1083-1092, 1999.

[9] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," ACM SIGCOMM Computer Communication Review, Vol. 34, Issue 2, pp. 97-122, April 2004.

[10] John Nickolls and William J. Dally "The GPU Computing Era," IEEE Micro, Vol. 30, Issue 2, March-April 2010.

[11] J. Kim and J. Kim, "Implementation of Efficient Power Method on GPU," Journal of The Korea Society of Computer and Information, Vol. 16, No. 2, pp. 9-16, Feb. 2011.

[12] Tom R. Halfhill, "Parallel Processing with CUDA," Microprocessor Report, Jan. 2008.

[13] J. Kim and J. Kim, "Fast Prefix Deletion for Parallel TCAM-Based IP Address Lookup," Journal of The Korea Society of Computer and Information, Vol. 15, No. 12, pp. 93-100, Dec. 2010.

[14] Jin Zhao, Xinya Zhang, Xin Wang, Yangdong Deng, and Xiaoming Fu, "Exploiting Graphics Processors for High-performance IP Lookup in Software Routers," In Proceedings of INFOCOM, pp. 301-305, 2011.

[15] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon, "PacketShader: A GPU-Accelerated Software Router," ACM SIGCOMM Computer Communication Review, Vol. 40, Issue 4, pp. 195-206, 2010.

[16] Yanbiao Li, Dafang Zhang, Alex X. Liu, and Jintao Zheng, "GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers," In Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 1-12, 2013.

[17] Hung-Mao Chu, Tsung-Hsien Li, and Pi-Chung Wang, "IP Address Lookup by Using GPU," IEEE Transactions on Emerging Topics in Computing, Vol. PP, Issue 99, 2015.

[18] University of Oregon Route Views Project, http://www.routeviews.org/

Authors

Junghwan Kim received the B.S., M.S. and Ph.D degrees from Seoul National University, Seoul, in 1991, 1993 and 1999, respectively, all in computer science.

Dr. Kim joined Samsung Electronics as a senior researcher in 1999. He joined the faculty of Konkuk University in 2001. His research interests are in the areas of parallel computing, communication networking, GPU computing, and design of efficient algorithms.

Jinsoo Kim received the B.S. degree from Seoul National University, Seoul, in 1983, and the M.S. and Ph.D degrees from Korea Advanced Institute of Science and Technology (KAIST), in 1985 and 1998, respectively, all in computer engineering.

Dr. Kim joined Korea Telecom, where he was a senior researcher in 1985. He has been a professor of Konkuk University since 2000. His research interests include parallel computing architectures, high-speed networking, wireless sensor networks, and packet processing systems.