Lucchesi *et al. Journal of Internet Services and Applications* (2018) 9:3
DOI 10.1186/s13174-017-0075-y

Journal of Internet Services and Applications

RESEARCH Open Access

High-performance IP lookup using Intel Xeon Phi: a Bloom filters based approach

Alexandre Lucchesi*, André C. Drummond and George Teodoro

Abstract

IP lookup is a core operation in packet forwarding, which is implemented using a Longest Prefix Matching (LPM) algorithm to find the next hop for an input address. This work proposes and evaluates the use of parallel processors to deploy an optimized IP lookup algorithm based on Bloom filters. We target the implementation on the Intel Xeon Phi (Intel Phi) many-core coprocessor and on multi-core CPUs, and also evaluate the cooperative execution using both computing devices with several optimizations. The experimental evaluation shows that we were able to attain high IP lookup throughputs of up to 182.7 Mlps (Million packets per second) for IPv6 packets on a single Intel Phi. This performance indicates that the Intel Phi is a very promising platform for deployment of IP lookup. We also compared the Bloom filters to an efficient approach based on the Multi-Index Hybrid Trie (MIHT) in which the Bloom filters was up to 5.1× faster. We also propose and evaluate the cooperative use of CPU and Intel Phi in the IP lookup, which resulted in an improvement of about 1.3× as compared to the execution using only the Intel Phi.

Keywords: Longest prefix matching, Software router, Intel Xeon Phi

1 Introduction

The use of software-based routers is motivated by their extensivity, programmability, and good cost-benefit. However, these routers need to attain high packet forwarding rates, which may be achieved with efficient algorithms and/or with the use of high-performance parallel computing devices. The next hop calculation for input packets is a core operation in the forwarding phase of routers, and is implemented via Longest Prefix Matching (LPM) since the development of CIDR (Classless Inter-Domain Routing).

In this work, we investigate the use of the Intel Xeon Phi processor as a platform for efficient execution of LPM algorithms for IP lookup. The Intel Phi is a highly parallel platform that supports up to 72 computing cores and 4-way hyperthreading. It is also equipped with 512-bit SIMD instructions and has a high-bandwidth memory. The Intel Phi may be attached as a coprocessor in a computer through PCIe or it can be deployed as an independent or standalone system in the newest Knights Landing (KNL) generation.

The design of the Intel Phi as a standalone system was a critical aspect that motivated its use in this work. The use of PCIe to connect coprocessors (Graphics Processing Unit (GPU) or Intel Phi) to the CPU has shown to be a major bottleneck for attaining high performance in data intensive applications, limiting the application throughput to that of the PCIe channel used. Previous work that used GPU for IP lookup reported this limitation [1, 2]. Thus, even though the number of computing cores of current Intel Phi processors is smaller than the one found in GPUs, the Intel Phi is likely to emerge as a major platform for the practical deployment of parallel and efficient IP lookup algorithms.

We have designed a parallel algorithm that uses Bloom filters (BFs) and hash tables (HTs) to efficiently find the LPM for both IPv4 and IPv6. Our implementation leverages the Intel Phi capabilities to mitigate the main drawback of the algorithm – the high costs of computing hash functions during lookup/store operations. This is achieved with the use of vectorization to reduce the costs of hashing computations and thread-level parallelism to increase throughput.

In order to evaluate our propositions, we have compared the BFs-based algorithm to a parallel version of the Multi-Index Hybrid Trie (MIHT). The MIHT is an efficient sequential IP lookup algorithm that has been shown

*Correspondence: lucchesi@aluno.unb.br
Department of Computer Science, University of Brasília, Brasília, Brazil

to attain better performance than well-known tree/trie-based algorithms commonly used for IP lookup: the Binary Trie, the Prefix Tree, the Priority Trie, the DTBM, the 4-MPT and the 4-PCMST [3]. The experimental evaluation has shown that our optimized BFs-based algorithm was able to outperform MIHT in both sequential and parallel settings on the Intel Phi. In a parallel execution using IPv4 and IPv6 prefix datasets, our BFs algorithm was, respectively, up to 3.8× and 5.1× faster than MIHT. The results show that, although the MIHT is a very memory-efficient algorithm, it benefits less from the Intel Phi. For instance, the use of vector SIMD instructions available in most of the modern device architectures can be used to improve the BFs approach, but it is not effective for MIHT because of the irregular nature of the data structures used.

The main contributions of this paper can be summarized as:

- We design and implement an optimized BFs-based LPM algorithm that fully exploits the Intel Phi and modern CPU capabilities, such as SIMD instructions. This algorithm performs better than MIHT even in a sequential executions, and attains higher scalability in a parallel setups.
- We propose a novel approach combining dynamic programming and Controlled Prefix Expansion [4] (DPCPE) to enhance the performance of IPv6 lookups. This optimization resulted in performance gains of up to 5.1× in our BFs-based algorithm.
- We propose and evaluate a cooperative execution model for IP lookup that uses Intel Phi and multi-core CPUs available in the system. After optimizing PCIe data transfers, we were able to achieve a speedup of about 1.33× vs. the execution using only the Intel Phi.
- We show that the most efficient sequential algorithm may not be the best solution in a parallel setting. The results also show that the Intel Phi is a promising platform for high-performance IP lookup. To the best of our knowledge, this is the first work to systematically evaluate the Intel Phi using multiple algorithms and device architectures for IP lookup.

This paper is built on top of our previous work [5], and has extended it with the introduction of techniques to cooperatively use the CPU and Intel Phi, a thorough evaluation using a larger number of datasets, the use of the new KNL Intel Phi that is faster and is deployed as a standalone processor, and a detailed description of the optimization approaches used.

The rest of this document is organized as follows. Section 2 describes the Intel Phi and related work. Section 3 presents the use of BFs to solve the IP lookup problem. Section 4 details of the Bloom Filters algorithm design, optimizations, parallelization strategies, and relevant implementation details. The MIHT algorithm used for comparison purposes is presented in Section 5. We experimentally evaluate our solution in Section 6, and we conclude and present future directions in Section 7.

2 Background and related work

This section describes the Intel Xeon Phi accelerator used to speedup the IP Lookup, and presents the closest related work in the IP lookup domain.

2.1 Intel Xeon Phi

The Intel Phi processor is based on the Intel Many Integrated Core (MIC) architecture, which consists of many simplified, power efficient, and in-order computing cores equipped with a 512-bit vector processing unit (SIMD unit). In this architecture, the computing cores are replicated to create multicore processors with up to 72 cores (model 7290), which are placed in a high performance bidirectional ring network with fully coherent L2 caches. The Intel Phi runs specific versions of the CentOS, SuSE, and RedHat Linux OSs.

The MIC architecture combines features of general-purpose CPUs and many-core processors or accelerators to provide an easy to program and high-performance computing environment [6]. It is based on the x86 instruction set and supports traditional parallel and communication programming models, such as OpenMP (Open Multi-Processing), Pthreads (POSIX Threads Programming), and MPI (Message Passing Interface).

The characteristics of the two Intel Phi used in this work are presented in Table 1. As shown, the 7250 is part of the newest KNL generation and, as a consequence, has better computing capability and memory bandwidth. The 7120P is only deployed as a coprocessor attached to the CPU through a PCIe channel. The 7250, on the other hand, is executed in a standalone mode in which it is the machines' bootable processor. The standalone mode brings improvements for data-intensive applications because it removes the communication overheads of the PCIe communication among CPU and coprocessors to offload computations, which exists in the 7120P and is still a major limitation with GPUs. The table also presents the prices for the processor. Although the 7250 is more

Table 1 Characteristics of the Intel Phi processors used

Processor	Name	Cores	Freq.	Mem. bandwidth	Exec. mode	Price ($)
7120P	KNC	61	1.33 GHz	352 GB/s (GDDR5)	Coprocessor	1500
7250	KNL	68	1.60 GHz	500 GB/s (MCDRAM)	Standalone	2400

expensive than the 7120P, the first one does not need to be deployed with a host machine — thus reducing the cost of the entire system.

2.2 Previous work

The trie/tree-based is a popular class of IP lookup algorithms [3, 11–13], and finding the LPM in these algorithms usually consists of sequentially traversing a sequence of tree nodes. Generally, these algorithms strive to reduce the number of required memory accesses as a means to speed up the lookup process. For instance, in order to achieve that, the Multi-Index Hybrid Trie (MIHT) [3] employs space-efficient data structures, such as B^+ trees and Priority Tries [14]. Recently, the use of compressed trie data structures have also been proposed [15, 16]. Nevertheless, trie/tree-based schemes commonly share the characteristic of being memory-intensive and irregular. Another class of algorithms for IP lookup is based on Bloom filters [17, 18]. These algorithms are more compute-intensive and may require many hash calculations during each lookup/store operation. Hashing is used within Bloom filters as a means to avoid unnecessary memory accesses to hash tables (where IP prefixes and next hop information are actually stored).

A wide range of hardware architectures have been used to implement IPv4 lookup algorithms, including CPU, FPGA, GPU and many-cores [7–10]. While some GPU-based implementations present, in specific configurations, high IP lookup rates, they suffer from a high lookup latency because of the required data transfers between host and GPU. Table 2 roughly compares the main characteristics of previous works using GPU accelerators. Although the authors have used different hardware, data, and testing methodologies, we have made our best effort to provide an overall comparison between them. The table presents the algorithm used in each work, the computing device employed, the maximum throughput attained for both IPv4 and IPv6 using randomize input querying addresses (worst case scenario), and whether they consider the data transfers among CPU and coprocessor in the experiments.

As shown in the table, a variety of proposals exist for IP lookup using GPUs. The first two approaches have developed algorithms specifically for IPv4, whereas the other approaches can deal with both IPv4 and IPv6. However, in [1] different algorithms are used in each IP configuration. Among other algorithms, the Multi-bit Trie based solution presented in [9] attained by far the highest throughput. However, this solution does not consider the time spent in the data transfers among CPU and GPU, which makes the results attained unrealistic. The PCIe communication cost can not be ignored, because it may be the performance limiting factor. As compared to the remaining approaches [1, 10], it is noticeable that our approach is more efficient than [1], whereas the throughput of GAMT [10] is higher. However, the performance of GAMT is reached with the cost of transferring data through PCIe, which increases the delays in the packet forwarding. Also, the performance of GAMT may be higher because it employs a compacting scheme that greatly reduces the routing table, and could also be used in our solution because it is a preprocessing phase. In our approach with the Intel Phi 7250, the data transfers using the PCIe to offload computation to the coprocessor do not exist, because the 7250 is deployed as the bootable or standalone processor. However, the PCIe will still be used by the NIC to receive and transmit packets in a complete router solution. Therefore, although the PCIe may continue to be a bottleneck in the case of very high forwarding rates, it will saturate in higher forwarding rates for the 7250 as compared to other coprocessor based solutions. Further, with the increase in the number of computing cores of the Intel Phi, we expect that the computing capabilities of this processor will improve rapidly, and it may be able to attain similar performance to the best GPU algorithm and keep the advantage of not requiring PCIe data transfers.

Table 2 Comparison between previous work that employed accelerators in the IP lookup problem. We present the algorithm used, the processor, and the maximum performance for IPv4 and IPv6. Finally, we also shown if the performance collected considered the PCIe data transfers, which are not necessary on our algorithm when using the KNL Intel Phi

Algorithm	Computing device	IPv4 (Mlps)	IPv6 (Mlps)	Considers PCIe transfer?
Radix Tree [7]	GTX280	0.035	–	Yes
SAIL_L [8]	Tesla C2075	547	–	Yes
Multi-bit Trie [9]	Tesla C2075	2,900	3,600	No
DIR-24-8-BASIC (IPv4)	2x GTX480	76.17	74.22	Yes
Binary search (IPv6) [1]				Yes
GAMT [10]	Tesla C2075	1,072	658	Yes
Bloomfwd	Phi 7250	169.6	182.7	–

A Parallel Bloom Filter (PBF) was implemented in the Intel Phi [19]. PBF was proposed to reduce synchronization overhead and improve cache locality. However, the proposed implementation was not specialized for IP lookup and, as such, our approach is different both in the algorithmic and implementation levels. Our approach is built on top of [17] and includes several optimizations targeting efficient execution on the Intel Phi. As presented in the experimental evaluation, these optimizations are crucial to attain high performance.

Other interesting related work include Click [20], RouteBricks [21], and Open vSwitch [22]. Click proposes a modular architecture to deploy software routers. In Click, routers are built on top of fine-grain components chained along the processing path, which provides the ability to quickly implement extensions by creating new components and connecting them to the computation workflow. These components only have to implement a common interface. RouteBricks is a solution based on Click, which has been constructed with the goal of maximizing performance of software routers. It proposed a number of optimizations to minimize the costs of processing packets and has also employed parallelism. Such as Click, Open vSwitch is an interesting software-based switching platform, which we will investigate in our future effort as a target to integrate our Bloom Filters based approach into a complete software router solution, as discussed in Section 7.

3 Bloom filters for IP lookup

The use of BFs coupled with HTs for computing IP lookups has been proposed in [17]. The naïve algorithm uses 32 and 64 pairs of BFs and hash tables, respectively, for IPv4 and IPv6 lookups. Dharmapurikar et al. have also described the use of Counting Bloom Filters (CBFs) [23] to enable dynamic forwarding tables (FIBs) and Controlled Prefix Expansion (CPE) [4] to reduce the number of BFs and HTs pairs for IPv4 to 2 pairs and a direct lookup array (DLA). In Section 3.1, we present the naïve Bloom Filters based algorithm in detail, whereas Section 3.2 details the CPE and other optimizations for IPv4 that are incorporated into the Baseline version we use for performance comparisons.

3.1 The Naïve Bloom filters algorithm

A BF is a data structure for membership queries with tunable false positive errors [24] commonly used in web caching, intrusion detection, and LPM [17]. In essence, a BF is a bit-vector that represents a set of values. The BF is programmed by computing hash functions on each element it stores, and by setting the corresponding indices in the bit-vector. Further, to check if a value is in the set, the same hash functions are computed on the input value and the bits in the bit-vector structure addressed by the hash values are verified. The value is said to be contained in the set with a given probability only if all bits addressed by the hash values are set. Note that the actual prefix values are stored in HTs, which are searched only when a match occurs on their associated BF. A CBF is a BF variant that adds a counter associated to each bit in the bit-vector, such that each counter is incremented or decremented when an element is added or removed, respectively.

The lookup operations in the naïve Bloom filters algorithm are executed within multiple sets of filters and hash tables — one for each possible IP prefix length. As network addresses in IPv4 are 32-bit long, they require the algorithm to employ 32 Bloom filters with their respective 32 hash tables. Each hash table stores their corresponding [prefix, next hop] pairs and any other relevant routing information, such as metric, interface, etc. If a default route exists, it is stored in a separate field in the forwarding table data structure. Let $F = \{(f_1, t_1), (f_2, t_2), \ldots, (f_{32}, t_{32})\}$ be the set of Bloom filters (f_i) and associated hash tables (t_i) that form an IPv4 forwarding table, where (f_1, t_1) corresponds to the data structures that store 1-bit long prefixes, (f_2, t_2) corresponds to the data structures that store 2-bit long prefixes, and so on. In addition, let $len(f_i)$ be the length of the bit-vector of the i-th Bloom filter, where $1 \leq i \leq 32$. The forwarding table construction is as follows. For every network prefix p of length l to be stored, k hash functions are computed, yielding k hash values: $H = \{h_1, h_2, \ldots h_k\}$. The algorithm uses H to set the k bits corresponding to the indices $I = \{h_i \bmod len(f_l) \mid 1 \leq i \leq 32\}$ in the bit-vector of the Bloom filter f_l. It also increments the corresponding counters in the array of counters of f_l.

Algorithm 1: STORE(*FIB*, *prefix*, *nextHop*): Storing a network prefix in the Naïve BFs algorithm

Input: *FIB* {forwarding table}, *prefix* {network prefix}, *nextHop* {next hop address}.
Result: Updated FIB.

```
// FIB.BF - array of bit-vectors or Bloom filters.
// FIB.CT - array of counters.
// FIB.HF - array of sets of hash functions.
// FIB.HT - array of hash tables.
1 l := prefix.length
2 foreach h ← FIB.HF[l] do
3     i := h(prefix)
      // Ensure i is within BFs size.
4     i := i mod FIB.BF[l].size
5     FIB.BF[l][i] := True
      // Increment associated counter (CBF).
6     FIB.CT[l][i] ++
  // Store routing information in the hash table.
7 FIB.HT[l].store(prefix, nextHop)
```

The process of storing prefixes into the forwarding table data structure of the Bloom filters algorithm is detailed in Algorithm 1. The algorithm receives as input a triple

[*FIB*, *prefix*, *nextHop*], and iterates over the set of hash functions associated to the Bloom filter that stores prefixes whose length (l) is equal to that of the supplied prefix (Lines 2-6). Each hash function is applied to the prefix yielding indexes i used to set the corresponding bits in the associated Bloom Filter ($B[l][i]$ - Line 5) and to increment the counting Bloom Filter (Line 6). Finally, the prefix contents, including its next hop (*nextHop*) and any other relevant information, are stored in the associated hash table (Line 7).

The lookup process is carried out as follows. Given an input destination address DA, the algorithm first extracts its segments or prefixes. Let $S_{DA} = \{s_1, s_2, \ldots, s_{32}\}$ be the set of all the segments of a particular address DA, where s_i is the segment corresponding to the first $1 \leq i \leq 32$ bits of DA.

For each $s_i \in S_{DA}$, k hash functions are computed, yielding k hash values for each segment: $H = \{(h_1, h_2, \ldots h_k)_1, (h_1, h_2, \ldots h_k)_2, \ldots, (h_1, h_2, \ldots h_k)_{32}\}$. The element $H'_i \in H$ is used to query the Bloom filter $f_i \in F$. The algorithm checks the k bits in the bit-vector of f_i using the indices $I = \{h_j \bmod len(f_i) \mid h_j \in H'_i, 1 \leq j \leq k \text{and} 1 \leq i \leq 32\}$. The result of this process is a *match vector* $M = \{m_1, m_2, \ldots, m_{32}\}$ containing the answers of each Bloom filter, i.e., each $m_i \in M$ indicates whether a match occurred or not in f_i. The match vector M is used to query the associated hash tables. The search begins by sequentially performing queries to the associated hash tables by traversing M backwards, i.e., starting in m_{32}. This is because we are interested in the LPM. If the algorithm finds the next hop (a true match) for a given DA in the pair (f_i, t_i), we have found the LPM and there is no need to continue looking into smaller prefix sizes. As Bloom filters may produce false-positives but never false negatives, when a filter does not match a segment, i.e., $m_i \in M$ indicates a mismatch, the algorithm can safely skip to the next Bloom filter f_{i-1} (if $i \geq 2$) without touching its associated hash table t_i. This process continues until the LPM is found or all pairs (f_i, t_i) are unsuccessfully searched. Please, note that false-positives will only lead to extra hash table searches, and the actual result of the algorithm will remain the same regardless of that ratio.

The actual lookup is presented in Algorithm 2. Starting from the Bloom filter associated to the largest address segment, the algorithm iterates on the filters backwards (Lines 2 to 13). Within an iteration, it extracts the most significant i bits of DA (Line 3), and checks whether that segment is in the corresponding Bloom filter (Lines 4 to 8). In this phase, a set of hash functions is applied to the segment p, and the resulting hash values are used to address the i-th BF (Line 7). If the value in the filter is false for any of the hash functions, then that particular segment is certainly not stored in the associated hash table and the algorithm can leave this phase (Line 8) and continue to the next iteration checking for shorter segments of DA. When the filter responds that the segment is in the set, the search continues in the hash table associated (Line 10), which may or may not contain the searched routing information (e.g., it may a false positive). Further, if the next hop address is not found in any hash table, the default route is returned (Line 14).

Algorithm 2: LOOKUP(*FIB*, *DA*): The lookup phase of Naïve IP lookup Bloom Filters algorithm

```
   Input: FIB {forwarding table}, DA {destination address}.
   Result: The next hop address.
   // FIB.BF - array of bit-vectors or Bloom
      filters.
   // FIB.HF - array of sets of hash functions.
   // FIB.HT - array of hash tables.
   // FIB.BF.n - Number of BFs/HTs (32 for IPv4).
   // FIB.g - Default route.
1  i := FIB.BF.n − 1
2  while i ≥ 0 do
       // Extract the most significant i bits of DA
          to p.
3      p := DA[0 : i]
       // Loop on hash functions associated to i-th
          BF.
4      foreach h ← FIB.HF[i] do
5          j := h(p)
           // Ensure i is within the BF size.
6          j := j mod FIB.BF[i].size
           // Check the j-th bit of i-th BF.
7          if ¬FIB.BF[i][j] then
8              break
       // If p is in associated Bloom Filter.
9      if FIB.BF[i][j] then
           // Search hash table associated to i-th
              BF.
10         nextHop := FIB.HT[i].lookup(p)
           // Check if segment was found in HT.
11         if nextHop ≠ ∅ then
12             return nextHop
       // Try a shorter prefix.
13     i := i − 1
14 return FIB.g
```

3.2 Baseline Bloom filters algorithm

The Baseline Bloom Filters lookup algorithm we developed is built on top of the Naïve version and includes optimizations to (i) reduce the amount of memory used in each Bloom filters structure and to (ii) decrease the number of Bloom filters employed. The first optimization is implemented with the use of *asymmetric* Bloom filters [17], which implies that $len(f_i)$ may be different of $len(f_j)$, for $1 \leq i,j \leq 32$ and $i \neq j$. The goal of asymmetric Bloom filters is to optimally allocate memory for each data structure according to the expected number of elements to be stored. Also, for each distinct prefix length in the FIB, the algorithm allocates a CBF and a HT. As previously discussed, the CBF is intended to provide the ability of removing addresses from the Bloom filter.

The optimization CPE allows the expansion of shorter prefixes into multiple equivalent larger prefixes. Before building an IPv4 FIB, we use this technique to ensure there are only prefixes of length 20, 24, and 32. The first group is stored in a direct lookup array (DLA), while the other two are stored in separate sets consisting of one BF and one HT (G_1 and G_2, respectively). The DLA is a flat array with 2^{20} entries that stores the next hops associated to 20-bit prefixes. When using this structure, the lookup algorithm will sequentially search G_2 and G_1 (starting from G_2, since it stores the longest prefixes) and, if the LPM is not found, the next hop stored in the DLA position indexed by the first 20 bits of the input address is returned (it may be the default route).

For IPv6, previous work has reported that CPE was inefficient because of the longer "strides" between hierarchical boundaries of addresses, which would result in a very high memory use after expansion. It was suggested to use 64 sets of CBFs and HTs, one for each possible prefix length. However, even though most lengths in realistic IPv6 FIBs indeed are either empty or contain few prefixes, we have proposed an algorithm (DPCPE) that uses dynamic programming to group prefixes by length and perform the expansions with limited additional memory demands. The details of this algorithm are presented in Section 4.2.

4 Bloom filters optimizations and parallelization

This section describes the optimizations proposed in this work that are implemented on top of the baseline BFs IP lookup algorithm as well as its parallelization targeting the Intel Phi. The parallel CPU version employs similar parallelization strategies, but differs with respect to the instruction-level parallelism that used auto-vectorization. The baseline implementation on which our work is built incorporates the following optimizations: the use of CBF to allow FIB updates, asymmetric memory allocation proposed in [17], and CPE to reduce the number of required data structures.

4.1 Optimizing the hash calculations

Hashing is an important aspect of the algorithm because it impacts the efficiency of BFs and HTs. In the BFs, it affects both the false positive ratio (FPR) and the memory utilization. With respect to the associated HTs, the better the quality of the hash, the less collisions are likely to happen and, as a consequence, the lookup process will also be faster.

In order to improve the algorithm, we have (i) accelerated the hash calculations with the use of instruction-level parallelism or vectorization, as discussed in detail in Section 4.3; (ii) reduced the cost of hashing by combining the output of two hash calculations to generate more hashes; and (iii) implemented and evaluated the reuse of hash values between BFs and HTs to minimize the overall number of hash calculations. The reuse affects both the lookup and update operations. The generation of extra hashes was performed through the use of a well-known technique that employs a simple linear combination of the output of two hash functions $h_1(x)$ and $h_2(x)$ to derive additional hash functions in the form $g_i(x) = h_1(x) + i \times h_2(x)$. This technique results in faster hash calculations and can be effectively applied in the BFs and HTs without affecting the asymptotic false positive probabilities [25]. We have also proposed the reuse of one of the hashes calculated to search or store a key in the BF to address its associated HT, which avoids the calculation of another hash whenever a HT is visited. This is possible because we calculate the hash without taking into account the size of the BF. Thus, during the actual access to the BF or the HT, we compute the rest of the division of the hash value to the data structures size.

4.2 The new dynamic programming CPE (DPCPE)

Another crucial optimization we have implemented for IPv6 is the use of CPE to reduce the number of required sets of Bloom filters and hash tables in the algorithm. This technique consists of expanding every prefix of a shorter length to multiple, equivalent, prefixes of a greater length, so that the number of distinct prefix lengths and, consequently, filters and hash tables, is reduced. In IPv4, as previously discussed, we used CPE to expand prefixes into two groups: $G_1 \in [21–24]$ and $G_2 \in [25–32]$. After the CPE, G_1 has only 24-bit prefixes and G_2 has only 32-bit prefixes, and two sets of Bloom filters and hash tables are allocated to store these prefixes.

A Direct Lookup Array (DLA) is allocated to store the next hops of the remaining prefixes, whose lengths are ≤ 20 bits, using the prefixes themselves as the indices. In this way, we are able to bound the worst-case lookup scenario to two queries (G_1, G_2) and one memory access (DLA), as detailed in [17]. Note that the trade-off of CPE is faster search on the cost of increased memory footprint, as shown in Table 4. It was also mentioned that this technique also not viable for the IPv6 case [17], but, in this work, we proposed the DPCPE algorithm that builds the CPE for IPv6.

The DPCPE algorithm works as follows. Let $L = \{l_1, l_2, \ldots, l_{64}\}$ be the prefix distribution of an IPv6 FIB, where l_i is the number of unique prefixes of length i (in bits). Given a desired number of expansion levels n (or target number of BFs), DPCPE uses dynamic programming to compute the set of lengths to be used so that the total number of prefixes in the resulting FIB is minimized. DPCPE always starts by picking the length 64, since it is the largest prefix length for IPv6 and, as such, its inclusion is required for correctness (i.e., every IPv6 prefix can, theoretically, be expanded to one or more 64-bit prefixes). Let $S = \{64\}$ represent the initial set of resulting lengths

and $C = \{1, 2, \ldots, 63\}$ represent the initial set of candidate lengths. While $|S| < n$, the algorithm removes an element $l \in C$ and inserts it into S. In each iteration, the length l is selected by mapping a cost function f over all possible sets of lengths and choosing the length associated with the smaller cost. For instance, in the second iteration (assuming $n \geq 2$), f is mapped over the set $Q = \{\{l, 64\} \mid l \in C\}$ and the value l from the set that resulted in the minimum cost is selected. The cost function f takes as input L and a set of expansion levels $Q' \in Q$. It then computes the resulting number of prefixes after expanding L to Q'. The (maximum) number of prefixes, resulting from expanding a prefix of length $l_i \in L$ to $q \in Q'$ (such that, $i < q$), is defined as $2^{q-i} \times l_i$. Note that f does not take into account the problem of prefix capture [4], which happens whenever a prefix is expanded to one or more existing prefixes in the database. In this case, the existing longer prefix "captures" the expanded one, which is ignored. Therefore, although DPCPE is not guaranteed to return the optimal solution, it usually returns solutions that work better in practice for the BFs algorithm than directly using the database with no preprocessing (Section 6.6).

4.3 Parallelization

Our parallelization strategy employs both TLP and ILP to fully utilize the Intel Phi computing power. These parallelism strategies is described in the following sections.

4.3.1 Thread-level parallelism (TLP)

Due to its regular data structures, the Bloom filters algorithm exposes multiple opportunities for parallelism. For instance, in [17] it was suggested a parallel search over the two sets of Bloom filters/hash tables and the DLA (associated with the different prefix lengths) for a given input address, which is mentioned to be appropriate for hardware implementations. In this strategy, a final pass is performed to verify if a match occurs in any of these data structures and to select the next hop. The same approach could be used for a software-based parallelization by dispatching a thread to search each data structure. However, IPv4 prefix databases have the well-known characteristic that prefixes are not uniformly distributed in the range of valid prefix lengths and, as a consequence, it is more likely that a match occurs to prefixes within lengths that concentrate most of the addresses, i.e., the set of Bloom filter and hash table that stores 24-bit prefixes. Therefore, computing all Bloom filters in parallel may not be efficient because, most of the times, the results from the data structures associated with prefix lengths smaller or greater than 24 bits will not be used. Instead, it is more compute efficient to sequentially query the Bloom filters and the DLA. The other option for TLP, which is used in our approach, is to perform the parallel lookup computation for multiple addresses by assigning one or multiple addresses to each computing thread available. In this way, we can carry out the processing of each address using the compute efficient algorithm, while we are still able to improve the system throughput by computing the lookup for multiple addresses concurrently. This is possible because the processing of addresses is independent and, as such, there is no synchronization across the computation performed for different addresses. The implementation of the parallelization at this level employed the Open Multi-Processing API (OpenMP) [26], which was used to annotate the main algorithm loop that iterates over the input addresses to find their next hops. The specific OpenMP settings used, which led to the better results, were the *dynamic* scheduler with *chunk size* of one.

4.3.2 Instruction-level parallelism (ILP)

The use of ILP is important to take full advantage of the Intel Phi, which is equipped with a 512-bit vector processing unit (see Section 2.1). We used its SIMD instructions to efficiently compute the hash values for multiple input addresses at the same time. The ILP optimization focused on the hashing calculations because it is the most compute intensive stage of the algorithm. The original work [27] and previous implementations of algorithms employing Bloom filters to the LPM problem [18, 19] do not discuss their decisions and reasons on the hash functions used.

Thus, we have decided to implement, vectorize, and evaluate three hash functions: MurmurHash3 [28] (Murmur), Knuth's multiplicative method [29] (Knuth), and a hash function named to here as H2 [30]. Murmur is widely used in the context of Bloom filters, but its original version takes as input a variable-length string. In order to improve its efficiency, we have derived versions of it specialized to work on 32-bit (for IPv4) and 64-bit (for IPv6) integer keys. Knuth is a simple hash function of the form: $h(x) = x \times c \bmod 2^l$, where c should be a multiplier in the order of the hash size 2^l that has no common factors with it.

H2 takes as input a key and mixes its bits using a series of bitwise operations, as shown in Algorithm 3. Although simple, the H2 hash function has been shown to be effective in practice [30].

Algorithm 3: Definition of the H2 hash function

Input : x {A 32-bit unsigned integer key}.
Output: The computed hash value.

1 $x := ((x \gg 16 \oplus x) \times$ 0x45d9f3b
2 $x := ((x \gg 16 \oplus x) \times$ 0x45d9f3b
3 $x := ((x \gg 16 \oplus x)$
4 **return** x

The hash implementations employed the low-level Intel Intrinsics API [31] to perform a manual vectorization of all the hash functions. We have also evaluated the use of automatic vectorization available with the Intel C Compiler, but the manually generated code has proved to be more efficient.

4.4 Cooperative execution and efficient data transfers

In this section, we present a variant of our algorithm that cooperatively uses the CPU and Intel Phi when it is deployed as a coprocessor (7120P) attached to the PCIe. The use of hybrid machines equipped with CPUs and accelerators has raised quickly in the recent years [32]. However, utilizing these systems adequately may require the use of complex software stacks and scheduling strategies. As such, a number of works have already been developed to provide techniques that simplify the use of such machines [33–37].

In the case of IP lookup studied in this work, the IP addresses are stored in the host memory and are transferred in batches through the PCIe channel to the Intel Phi for processing. After the lookup algorithm is executed on the coprocessor, the computed next hops are copied back from the coprocessor device memory to the host. The data transfer times in this process may be significant to the overall execution, and strategies to reduce transfer costs should be used. Also, the availability of the CPU creates opportunities for leveraging this processor as an additional computing device to carry out IP lookups. These two optimizations are briefly described below. We argue that the cooperative execution would benefit any application domains in which the load on the router is higher than the throughput delivered by a single processor.

4.4.1 Efficient CPU-Intel Phi data transfers

The limited bandwidth between CPU and Intel Phi may represent a major bottleneck to the use of the coprocessor, especially for data intensive applications [32]. The data transfers necessary to run a computation in the Intel Phi are typically carried out synchronously (*sync*), by pipelining the input data transfer (host to device), the computation in the coprocessor, and the output data transfer (device to host).

A common approach used for reducing the impact of such transfer costs to performance is to overlap data transfer operations with the execution of other tasks. This technique receives the name of double buffering [38, 39] and has been used in several domains. In our problem, it can be used to reduce both the CPU and Intel Phi idle times during the transfers, and may be implemented using an asynchronous data transfer (*async*) mechanism available in the Phi. In this strategy, while the Intel Phi is in the computation stage of the sync pipeline, we concurrently launch the input data transfer of a batch of IP addresses while a second buffer is being processed in the Intel Phi. The same also occurs for the output data transfers. This will allow for the overlapping of computation and data transfers.

4.4.2 Cooperative execution on CPU and Intel Phi

To cooperatively use both devices, we divide the input IP addresses into two sets that are independently and concurrently processed by the CPU and Intel Phi. The work division must be computed in a way to minimize the load imbalance between them. Otherwise, a processor may take much longer to process its lookups, and this may offset the potential performance gains of this approach. In order to carry out this partitioning, we take into account the relative performance among the processors, which should be the same as the relative sizes of the sets of IP addresses assigned for computation with each of the devices. The relative performance is computed in a profiling phase before the execution. Additionally, in order to avoid the CPU threads computing IP lookups to interfere with the CPU thread responsible for managing the Intel Phi, an entire CPU physical core is allocated to the latter. As such, in our system configuration, 15 CPU cores are used for IP lookup computations and 30 threads are launched in these cores. Similar strategies to cooperatively use CPUs and coprocessors have been employed in other application domains [33, 40]. In this work, we use these techniques and demonstrate their performance in the IP lookup domain.

5 MIHT algorithm and implementation

This algorithm uses a data structure named Multi-Index Hybrid Trie (MIHT) [3]. The MIHT was built by combining the advantages of B^+ trees and priority tries [14] to design dynamic forwarding tables. It consists of one B^+ tree and multiple priority tries. A B^+ tree is a generalization of a binary search tree in that a node can have more than two children. A B^+ tree of order m is an ordered tree that satisfies the following properties: (i) each node has at most m children; (ii) each node, except the root, has at least $\frac{m}{2}$ children; (iii) the root has at least 2 children; (iv) all leaves occur on the same level; and (v) the satellite information is stored in the leaves and only keys and children pointers are stored in the internal nodes. Although priority tries may be used alone to build dynamic forwarding tables, MIHT uses them as auxiliary substructures to build a more efficient algorithm. A priority trie is similar to a binary trie in that each node has at most two children and the branch is based on the address bits. However, priority tries have two main advantages over binary tries in the context of IP lookup. First, in a priority trie, prefixes are reversely assigned, i.e., longer prefixes are associated with higher levels nodes and shorter prefixes are associated with lower level nodes, allowing the search

to finish immediately whenever a match occurs (a binary trie always requires the traversal until a leaf). Second, as opposed to the binary trie, there are no empty internal nodes in a priority trie — every node stores routing information to improve memory usage.

5.1 Algorithm

In order to build the forwarding table in the MIHT, each network prefix is split into two parts: a prefix (the key) and a suffix. Let $p = p_0 p_1 \dots p_{l-1}$ be a network prefix and $q = p_i p_{i+1} \dots p_{l-1}$ for $0 \leq i \leq l-1$ be a suffix of p. The length of a prefix p is denoted $len(p)$. For example, $len(p_0 p_1 \dots p^*_{l-1}) = l$. For an integer $k \leq l$, the k-prefix key of p, denoted $prefix\_key_k(p)$, is the value of $(p_0 p_1 \dots p_{k-1})_2$. The k-suffix of p, denoted by $suffix_k(p)$, is defined as $suffix_k(p) = p_k p_{k+1} \dots p_{l-1}$, where $0 \leq k \leq l$. For example, $prefix\_key_4(00010^*) = prefix\_key_4(00011^*) = (0001) = 1$ and $suffix_4(00010^*) = 0^*$ [3]. For all network prefix p whose $len(p) \geq k$, its k-prefix (or key) is stored in the B^+ tree and a priority trie is allocated. Remember that, in a B^+ tree, data are stored only in external nodes (or leafs). In MIHT, the data consist of pointers to priority tries. If the key already exists in the B^+ tree, the pointer to the associated priority trie, which was previously allocated, is retrieved and used instead. The k-suffixes of all prefixes, along with their corresponding next hops and other routing information, are stored in priority tries. Network prefixes whose length is less than k are "keyless", i.e., they are stored directly as suffixes in a separate priority trie, named $PT[-1]$. All the suffixes stored in a particular priority trie share the same prefix. The root of MIHT has two pointers: one to a B^+ tree and another one to $PT[-1]$.

To search for a destination address DA, the algorithm extracts its key by applying $prefix\_key_k(DA)$ and searches the B^+ tree in a top-down manner, starting from the root. A tree traversal from the root to a leaf is performed with a binary search using the key value to search each visited node. If a match occurs in a leaf, then the algorithm searches $suffix_k(DA)$ in the corresponding priority trie using the pointer stored in that node. If we find the best matching of $suffix_k(DA)$ in some priority trie, then it is the LPM, and the search is terminated. If the $prefix\_key_k(DA)$ is not found in the B^+ tree, the algorithm searches DA in $PT[-1]$.

IP lookup operations can be performed by associating each network prefix with a key value of length k in the MIHT. By associating each prefix with a key value, the problem of searching for the longest matching prefix was transformed into a problem of searching for a corresponding index. Based on this transformation, the height of the MIHT is less than W (the length of input addresses), which accelerates the lookup speed. There are two parameters that affect the MIHT's performance: the length k of the keys and the order m of the B^+ tree. A (k, m)-MIHT is a data structure combining a B^+ tree of order m and priority tries, which contains two types of nodes: index nodes (i-node) and data nodes (d-node). An i-node can be either *internal* or *external*. An internal i-node is a node in which each child is also an i-node. An external i-node is a leaf node in the B^+ tree, which stores keys and pointers to d-nodes. Finally, a d-node is a priority trie which stores the next hops. It has been shown in [3] that the best lookup performance in IPv4 is obtained setting $k = m = 16$, which is used in our evaluations.

This algorithm is already highly optimized for implementation in software, as described into greater detail in [3]. Thus, our implementation of the baseline sequential algorithm incorporated the optimizations originally proposed by the algorithm as well as we developed a few optimizations targeting Phi. For instance, in order to optimize the throughput of memory, we have aligned all the B^+ tree nodes in addresses multiple of 64 bytes. Such optimization is useful because it leverages the fact that each node stores sixteen 32-bit integers, which matches exactly the coprocessor's word size. This allows an entire node to be fetched from memory in one memory access, accelerating the traversal of the structure. As suggested in the original algorithm, the sixteen keys in a node are ordered and we perform a binary search on each visited node in order to quickly find the next child or data node.

5.2 Parallelization

This section describes the parallelization of the MIHT for IP lookup.

5.2.1 Thread-Level Parallelism (TLP)

The MIHT is built out of tree-based data structures and the lookup process consists basically of traversing these structures. Therefore, similarly to the the Bloom filters approach, we have parallelized the execution of multiple address lookups in MIHT. The implementation of the parallelization at this level also employed the OpenMP programming interface, and each computing thread is responsible for independently processing one or multiple input messages.

5.2.2 Instruction-Level Parallelization (ILP)

The MIHT is built from tree data structures, which are irregular in nature and make the use of SIMD operations very challenging and inefficient. Our analysis of the algorithm shows an opportunity for using SIMD vector instructions in the search performed in each node of the B^+ tree. However, since this search is very quick because of the small number of elements in a node and the fact they are ordered, the MIHT performance was not improved with the use of vector instructions.

6 Performance evaluation

This section evaluates the performance of our optimized BFs algorithm both for IPv4 and IPv6. We perform the lookups using pseudo-random generated IP addresses and addresses from a real packet trace in the "CAIDA Anonymized Internet Traces 2016" dataset [41].

6.1 Experimental setup and databases

The CPU runs were performed in a machine equipped with a dual socket Intel Xeon E5-2640v3 CPU (16 CPU cores with Hyper-Threading), 64 GB of main memory, and CentOS 7. This machine also hosts the Intel Xeon Phi 7120P, and is deployed in a local machine in our laboratory. The source codes were developed using C11 and compiled with the Intel C Compiler 16.0.3 for both the CPU and the Intel Phi using the -O3 optimization level.

We have used 7 real prefix databases for IPv4, whose characteristics are summarized in Table 3. The databases AS65000 and SYDNEY were obtained from [42, 43], respectively. The remaining databases are from [44]. Table 3 presents the amount of addresses in each database and the total number of prefixes before and after performing the CPE to group them into sets of 24-bit and 32-bit long prefixes.

For IPv6, we use the AS65000-V6 [42], EQUINIX, LINX, and NWAX datasets. Because IPv6 is still not widely used, the available datasets have a small number of prefixes: AS65000-V6 has 31,645 unique prefixes distributed in 34 distinct prefix lengths; EQUINIX has 42,663 unique prefixes distributed in 43 prefix lengths; LINX has 44,103 unique prefixes distributed in 45 prefix lengths; and NWAX has 42,227 unique prefixes distributed in 43 prefix lengths. Table 4 shows the effects of DPCPE on these datasets. The expansion with 3 levels was not possible because of the high memory requirements. Because the algorithm ignores the prefix capture problem (Section 4.2) when computing the levels, its estimates are half of the actual results for most of the configurations. We want to highlight that the main goal of the algorithm was not to provide an accurate estimate, but to reduce the number of filters instantiated and, as a consequence, improve the algorithm performance. Therefore, although not precise, the provided estimation will serve as a lower bound for the number of prefixes after expansion, and it can be used to decide whether it is worth computing the expansion.

Table 3 Characteristics of the IPv4 prefix datasets used

Dataset	Location	Original		
		≤20	21 −24	25 −32
AS65000	-	104,283	516,699	1625
SYDNEY	Sydney	102,696	553,811	10,862
DE-CIX	Frankfurt	102,984	535,074	9287
LINX	London	100,331	519,503	354
MSK-IX	Moscow	102,555	528,728	9529
NYIIX	New York	102,085	528,455	3637
IX.br/SP	Sao Paulo	103,733	544,703	4095
		After CPE		
		≤20	21 −24	25 −32
AS65000	-	1,048,576	971,555	11,3397
SYDNEY	Sydney	1,048,576	1,037,247	67,397
DE-CIX	Frankfurt	1,048,576	1,007,513	209,488
LINX	London	1,048,576	982,940	19,863
MSK-IX	Moscow	1,048,576	1,004,073	203,360
NYIIX	New York	1,048,576	1,000,128	151,391
IX.br/SP	Sao Paulo	1,048,576	1,024,899	147,201

The experimental results are organized as follows:

- From Sections 6.2 to 6.5, we evaluate the performance of the Bloomfwd as the hash configurations, input query data characteristics, and prefix dataset sizes are varied, and we compare our approach to the MIHT algorithm. These analyses are intended to stress the algorithm under different scenarios in order to understand the aspects that affect its performance. The experiments in this phase use the Intel Xeon CPU and Intel Phi 7120P processors only, because these processors are deployed in a local machine to which we have unlimited access.
- Further, in Section 6.6, we analyze the performance of the Bloomfwd and MIHT for multiple IPv4 and IPv6 prefix datasets in the Intel Phi. This evaluation uses the best parameters found in the previous experiments, and also compares the performance of the Intel Phi 7120P and 7250 devices. The 7250 Intel Phi used is part of the Stampede 2 supercomputer deployed in the Texas Advanced Computing Center (TACC), which we had access through the The Extreme Science and Engineering Discovery Environment (XSEDE) program.
- Finally, the benefits of cooperative execution using the CPU and Intel Phi 7120P are assessed in Section 6.7. This evaluation has not included the Intel Phi 7250 because it is a standalone bootable device and, as such, it is not attached to a CPU.

6.2 The effect of the hash function and false positive ratio

The false positive ratio (FPR or f) is a key aspect for the effectiveness of a Bloom filter because it affects the memory requirements and the number of hash calculations per lookup. We highlight that the FPR does not affect the results of the algorithm, but only the number of times that a value is informed to be in the associated hash

Table 4 Results of performing CPE in four real IPv6 prefix datasets collected from Routeviews [43]

Dataset	CPE level	Expansion levels suggested by algorithm	Estim. # of prefix. ×1000	Actual # of prefix. ×1000
AS65000-V6	CPE8	{24, 29, 33, 38, 40, 44, 48, 64}	61	60
	CPE7	{29, 33, 38, 40, 44, 48, 64}	77	76
	CPE6	{29, 33, 38, 44, 48, 64}	104	103
	CPE5	{33, 38, 44, 48, 64}	385	380
	CPE4	{33, 44, 48, 64}	1185	1166
	CPE3	{44, 48, 64}	650,417	—
EQUINIX	CPE8	{30, 35, 41, 45, 49, 51, 57, 64}	137	236
	CPE7	{30, 35, 41, 45, 51, 57, 64}	218	387
	CPE6	{30, 35, 41, 45, 51, 64}	368	648
	CPE5	{35, 41, 45, 51, 64}	962	1823
	CPE4	{35, 45, 51, 64}	1586	2702
	CPE3	{45, 51, 64}	672,373	—
LINX	CPE8	{17, 30, 33, 37, 43, 49, 57, 64}	194	216
	CPE7	{17, 33, 37, 43, 49, 57, 64}	271	367
	CPE6	{17, 33, 37, 43, 49, 64}	1079	942
	CPE5	{17, 37, 43, 49, 64}	2550	3818
	CPE4	{17, 37, 49, 64}	4301	6523
	CPE3	{17, 37, 64}	60,364,260	—
NWAX	CPE8	{30, 35, 41, 45, 49, 51, 57, 64}	137	235
	CPE7	{30, 35, 41, 45, 51, 57, 64}	217	386
	CPE6	{30, 35, 41, 45, 51, 64}	338	587
	CPE5	{35, 41, 45, 51, 64}	950	1763
	CPE4	{35, 45, 51, 64}	1574	2640
	CPE3	{45, 51, 64}	691,485	—

table by a Bloom filter without being. When this occurs, the algorithm will unsuccessfully search in the hash table. Probing a hash table consists in traversing a linked list, which may become expensive as the FPR increases. On the other hand, a very low FPR requires a larger number of hash calculations and a high memory utilization. The FPR is determined by three parameters: the number n of entries stored in the filter, the size m of the filter, and the number k of hash functions used to store/query the filters [24]. As detailed in [17], when FPR is minimized with respect to k, we get the following relationship:

$$k = \frac{m}{n} ln\, 2 \tag{1}$$

At this point, FPR is given by:

$$f = \left(\frac{1}{2}\right)^k \tag{2}$$

For a desired false positive probability f, and knowing in advance the total number of prefixes n to be stored in the forwarding table for each prefix length, we compute the size m of each Bloom filter using the following equation, derived by substituting 1 in 2.

$$m = \frac{n \lg \frac{1}{f}}{\ln 2} \tag{3}$$

Then, we use m and n to calculate k from 1. Table 5 summarizes how these parameters are affected for each IPv4 prefix dataset presented in Table 3.

The trade-off between increasing the hash calculations and the application memory footprint in order to avoid the extra cost of a false positive is complex. Therefore, we have evaluated it experimentally by measuring the execution times for various FPRs and hash function configurations. Hash functions are used in the Bloom filters algorithms for querying the Bloom filters and to search the hash tables associated to each filter. As such, we are able to use combinations of hash functions to compute the multiple hashes within a Bloom filter or the single hash for a particular hash table. The hash functions used were presented in Section 4, and we employ the AS65000 prefix

Table 5 Bloom filters parameters as FPR is varied. Two bloom filters (G_1 and G_2) are created

f			1%	10%	30%	60%	90%
AS65000	G_1	*m*	9,312,412	4,656,206	2,434,631	1,032,974	213,057
		k	7	4	2	1	1
	G_2	*m*	1,086,917	543,459	284,163	120,566	24,868
		k	7	4	2	1	1
SYDNEY	G_1	*m*	9,942,074	4,971,037	2,599,250	1,102,819	227,463
		k	7	4	2	1	1
	G_2	*m*	646,005	323,003	168,891	71,658	14,780
		k	7	4	2	1	1
DE-CIX	G_1	*m*	9,657,071	4,828,536	2,524,739	1,071,205	220,942
		k	7	4	2	1	1
	G_2	*m*	2,007,955	1,003,978	524,959	2,22,732	45,940
		k	7	4	2	1	1
LINX	G_1	*m*	9,421,538	4,710,769	2,463,161	1,045,079	215,553
		k	7	4	2	1	1
	G_2	*m*	190,389	95,195	49,775	21,119	4356
		k	7	4	2	1	1
MSK-IX	G_1	*m*	9,624,099	4,812,050	2,516,119	1,067,548	220,188
		k	7	4	2	1	1
	G_2	*m*	1,949,218	974,609	509,603	216,216	44,596
		k	7	4	2	1	1
NYIIX	G_1	*m*	9,586,286	4,793,143	2,506,233	1,063,353	219,323
		k	7	4	2	1	1
	G_2	*m*	1,451,092	725,546	379,373	160,962	33,200
		k	7	4	2	1	1
IX.br/SP	G_1	*m*	9,823,717	4,911,859	2,568,307	1,089,690	224,755
		k	7	4	2	1	1
	G_2	*m*	1,410,931	705,466	368,873	156,507	32,281
		k	7	4	2	1	1

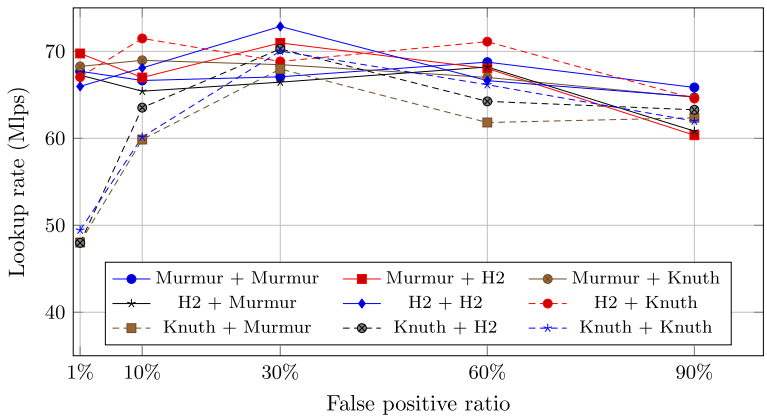

Fig. 1 Execution times for multiple hash functions and FPRs using 244 threads in the Intel Phi. The "Murmur + H2" entry means Murmur was used within the Bloom filters and H2 was used to address the hash tables

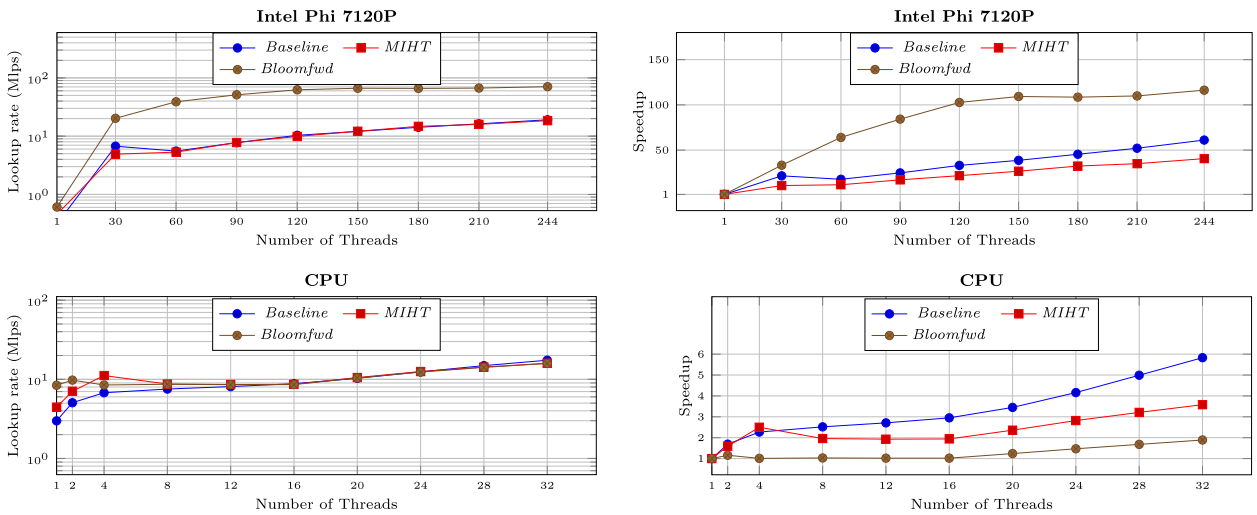

Fig. 2 Lookup rate (Mlps) and scalability of the IPv4 algorithms on Intel Phi 7120P and CPU using the AS65000 prefix database

database and an input IP address dataset with 2^{26} random IP addresses.

The results presented in Fig. 1 show that the performance of the application is affected by the FPR and hash functions. As presented, the use of Knuth resulted in a lower average performance as compared to other methods. The reason for the observed results is that this hash function preserves divisibility, e.g., if integer keys are all divisible by 2 or by 4, their hash values will also be. This is a problem in Bloom filters or hash tables in general, where many values will address the same bits in the bit-vector and only a half or a quarter of the buckets will end up being used, respectively.

On the other hand, Murmur and H2 are more sophisticated functions that provide a smaller number of collisions, hence all the configurations using any combination of them attained similar execution times. However, the best average performance was reached with 30% of FPR, where the results are less scattered for all hash functions. Furthermore, the best performance was attained when H2 was used in both stages of the algorithm. This occurs, in part, because we are able to reuse the hash calculated to probe the Bloom filter to address the hash table and, as a consequence, hash calculations are saved. Therefore, we use the configuration of 30% of FPR and H2 for both stages in the remaining experiments for IPv4. Because H2 only works for 32-bit unsigned integers and Murmur also resulted in good performance, we use 30% of FPR and MurmurHash3 (64 bits) for IPv6. As shown, the compromises of having a small or high FPR are complex. A small FPR will perform less hash calculations and use smaller data structures, whereas a high FPR will perform more hash computations and employ larger data structures. As such, the first is less expensive in the filter access, but may result in extra HT accesses, whereas the high FPR has an opposite compromise.

6.3 The scalability of BFs and MIHT

This section evaluates the optimized BFs algorithm for IPv4, which we refer here to as Bloomfwd, as the number

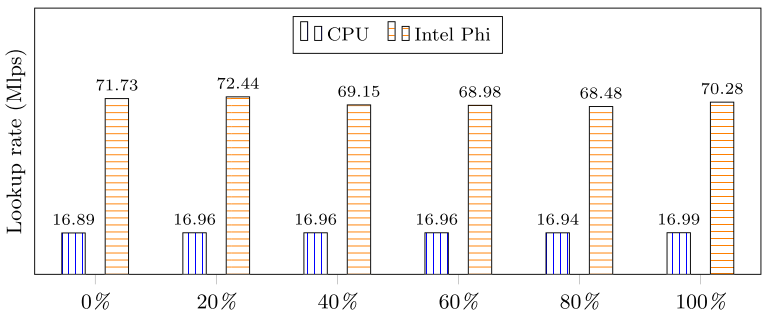

Fig. 3 Performance as matching ratio is varied. The entry 80% means that this percentage of the addresses match with equal probability a prefix in the forwarding table, while 20% of them end up in the default route

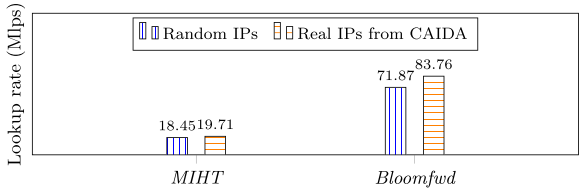

Fig. 4 Bloomfwd vs. MIHT using random and real IPs on AS65000 prefix dataset

of computing cores used is increased on the 7120P Intel Phi and on the CPU. We compare Bloomfwd to a Baseline implementation of the algorithm, introduced in Section 4.1, and to the MIHT. The difference between Baseline and Bloomfwd is on the hash functions, i.e., Baseline uses the standard C *rand()* function without vectorization [3, 17]. The MIHT is an algorithm that has outperformed several other IP lookup algorithms [18]. Our implementation of MIHT for IPv4 was tuned according to the original work for the (16,16)-MIHT. The speedups for the IPv6 dataset are similar and were omitted because of space constraints.

The lookup rates (in log scale) and speedups for both algorithms and processors are presented in Fig. 2. As shown, the performance of MIHT ($\approx$ 0.46 Mlps) is better than Baseline ($\approx$ 0.31 Mlps) for the sequential execution on the Intel Phi. However, as the number of computing threads used increases, the performance gap reduces quickly due to the better scalability of the BFs approach. For instance, the maximum speedup of Baseline as compared to its sequential counterpart is about 61×, whereas MIHT attains a speedup of only up to 40× when compared to its sequential version. The Bloomfwd, on the other hand, is the fastest algorithm on a single core and is still able to attain better scalability on the Intel Phi (116×). Also, it is at least 3.7× faster than the other algorithms. The differences between the lookup rates of Bloomfwd and Baseline highlights the importance of the use of vectorization and the hash function choice to performance.

The analysis of the CPU results show that all algorithms attained very similar lookup rates at scale in a multi-threaded setup, though they attained different speedups. We attribute the similar performance of the algorithms on the CPU to the fact that the memory bandwidth of this processor is much smaller than that of Intel Phi, which limits the scalability of the solutions [45]. Because the Bloomfwd is the fastest sequential algorithm, it reaches the memory bandwidth limits earlier as the number of cores used increases.

6.4 The impact of input address (queries) characteristics on performance

In order to investigate the effects of the input addresses on the performance, we performed the lookups using pseudo-random generated datasets containing 2^{26} IP addresses with different matching ratios and the AS65000 prefix database. We call *matching ratio* the relation between the number of addresses that matches at least one prefix in the database and the total number of addresses, thus a matching ratio of 80% implies that 20% of the input addresses do not match any prefix in the database and, as such, end up being forwarded to the default route. This evaluation intended to vary the characteristics of the input data and evaluate the algorithms under different configurations.

We ensure that a given address has the same probability to match any prefix stored in the database, and we also filter out all the IETF/IANA reserved IP addresses. Please note that a workload for forwarding could include other characteristics, such as the arrival of packets in bursts. We use random IP input addresses because it may be considered the worst-case scenario and it is the most commonly method used in previous works.

The lookup rates obtained for the Bloomfwd algorithm on both the CPU and on the Intel Phi are shown in Fig. 3.

As presented in Fig. 3, the matching ratio has little impact in the overall performance of the application. The reason for that is that the case of an address matching some prefix in the database *is not necessarily faster* than the case where the address end up in the default route, and vice-versa. For example, consider an address that does not match any prefix in the forwarding table. If no false positives occur, i.e., the two Bloom filters correctly answer not to look in their associated hash tables, the search quickly finishes with one additional memory access to the DLA.

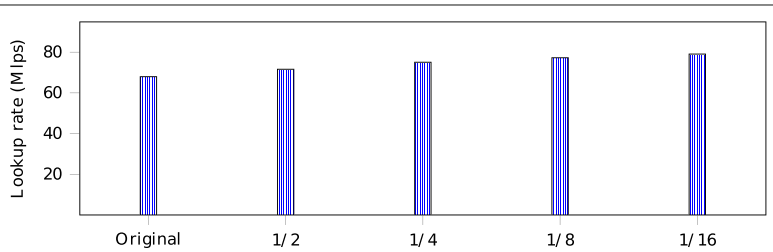

Fig. 5 Performance of Bloomfwd as the number of prefixes in the forwarding table decreases

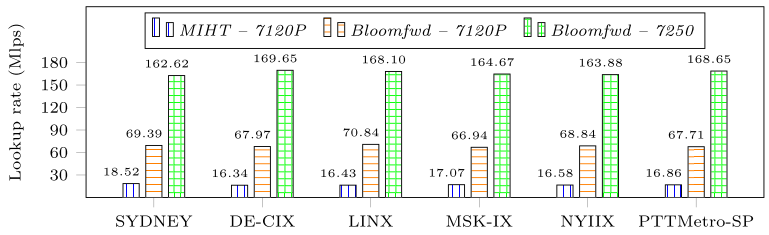

Fig. 6 Performance of Bloomfwd and MIHT for 5 IPv4 prefix datasets and both Intel Phi processors: 7120P and 7250

However, if for another prefix a false positive occurs in the first, but not in the second Bloom filter, or if there are plenty of values stored in the searched hash table buckets, then this case of matching will likely be *slower* than the former. As such, the speed and good statistical distribution of the hash function to control the FPR and also minimize the number of collisions in the hash tables is an important aspect to limit variations in performance as a result of dataset characteristics.

Further, in Fig. 4, we present the performance of Bloomfwd and MIHT for the AS65000 dataset using both random and real IP addresses from CAIDA traces. This comparison is interesting because it shows the impact, for instance, of the cache on performance. In the case of IPs from real trace, the same IP will be queried multiple times as a consequence, for instance, of a continuous communication flow among pairs of nodes. In the Bloomfwd case, the performance using the real IPs is about 1.16× higher than the case that uses random IPs.

6.5 The impact of the lookup table size to the performance

This section evaluates the impact of the lookup table size to the performance of our Bloomfwd algorithm into the 7120P Intel Phi processor. For sake of this analysis, we performed the lookups using pseudo-random generated datasets containing 2^{26} IP addresses and. Further, to evaluate different table sizes we have used the AS65000 prefix database as a reference (original) and have randomly removed prefixes from this dataset to create smaller tables.

The experimental results are presented in Fig. 5. As expected, the performance of the algorithm improves as the size of the table used is reduced. In this evaluation, the lookup rate has increased from about 68 Mlps using the original dataset for about 79 Mlps (1.16×) when the number of prefixes is 1/16 of the original size.

6.6 The IP lookup performance for IPv4 and IPv6

This section evaluates the BFs algorithms on IPv4 and IPv6 prefix datasets in both 7120P and 7250. First, we discuss the performance for the 5 remaining IPv4 prefix databases presented in Table 3 using a querying input dataset with 2^{26} random IP addresses. The results, presented in Fig. 6, show that the performance gains of our Bloomfwd as compared to the MIHT is about 4× for 7120P, regardless of the dataset used. Further, we execute the Bloomfwd in the 7250 processor, and the algorithm attained an additional speedup of about 2.4× as compared to the execution in 7120P and a throughput of up to 169.65 Mlps. This significant performance gap between the processors could not be directly derived from their different characteristics, presented in Section 2.1. Therefore, we have benchmarked the processors using the STREAM benchmark (data-intensive application) and we have observed that, in practice, the 7250 attains a memory bandwidth that is about 2.7× higher than the 7120P, which explains the gains of the Bloomfwd with the 7120P.

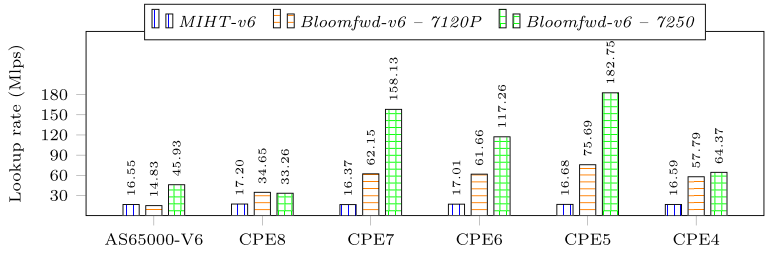

Fig. 7 Performance of Bloomfwd-v6 and MIHT-v6 for the AS65000-V6

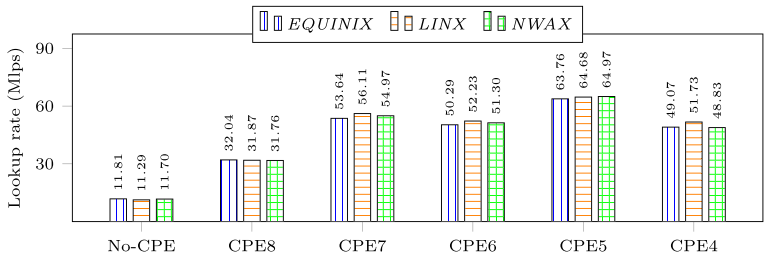

Fig. 8 Performance of Bloomfwd-v6 for 3 real IPv6 datasets from Routeviews [43]

We further evaluate the performance of the Bloom filters algorithm for IPv6 and the impact of using our DPCPE optimization. This experiment compares the lookup rates of our implementation (Bloomfwd-v6) with the corresponding version of MIHT for IPv6 (MIHT-v6 – the (32,32)-MIHT [18]).

Figure 7 shows the results for the AS65000-V6 with and without the use of DPCPE for multiple expansion levels and 2^{26} random input addresses. As presented, the performance of the Bloomfwd-v6 is greatly improved by the use of DPCPE, and the expansion with 5 levels is about 5.1× faster than the performance without CPE in the 7120P. This version is also 4.5× faster than the MIHT-v6 algorithm.

The performance of MIHT-v6 is similar in all cases because it groups routes by keys in Priority Tries (PTs), rather than prefix lengths (as in the Bloom filters approach). In other words, the number of distinct prefix lengths in the forwarding table does not directly affect the performance of MIHT. The Bloomfwd-v6 on the Intel Phi 7250 attained a throughput of up to 182.75 Mlps, and as presented there is a strong variation in performance as the number of CPE levels used is varied. The choice of the CPE levels is complex and involves many aspects such as caching capabilities and its effects on FPR, which requires an experimental evaluation for its adequate choice.

We have also evaluated the performance of MIHT-v6 and Bloomfwd-v6 using IP addresses from a real trace from CAIDA and the AS65000-V6 dataset. The MIHT-v6 and Bloomfwd-v6 achieved lookup rates of 19.7 Mlps and 83.76 Mlps, respectively, on the 7120P. We further executed the Bloomfwd-v6 in 7250 and it attained a lookup rate of 179.6 Mlps. These results are consistent with the ones using random queries and confirm the gains of our propositions. Finally, we executed the Bloomfwd-v6 in the Intel Phi 7120P for three other real IPv6 prefix datasets collected from Routeviews [43]. As depicted in Fig. 8, for each CPE configuration the algorithm attained similar lookup rates regardless of the prefix dataset. Furthermore, the performance pattern matches the one obtained for AS65000-V6 (Fig. 7), in which CPE5 was the best configuration.

6.7 Cooperative execution and data transfers strategies

This section evaluates the performance gains resulting from an implementation of Bloomfwd that employs an asynchronous strategy to fill and process two IP address buffers concurrently (double buffering) using the CPU and the Intel Xeon Phi 7120P cooperatively. The results from the synchronous execution were omitted because the performance of the asynchronous version was better in all cases, reaching speedups of up to 1.13× in comparison to the former. In order to exploit double buffering, the async strategy must choose an appropriate size for the buffers, which we have defined experimentally focusing on maximizing the IP lookup throughput. Figure 9 shows the impact of the buffer size on the performance of the application.

Figure 9 shows that the best performance is obtained using a 4M (*mebipackets*) buffer. However, despite the

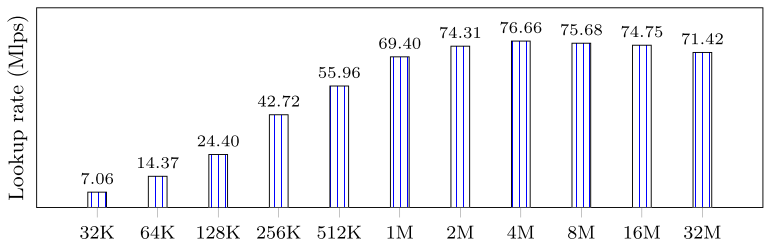

Fig. 9 Performance of *async* for different buffer sizes and a input containing 2^{30} addresses

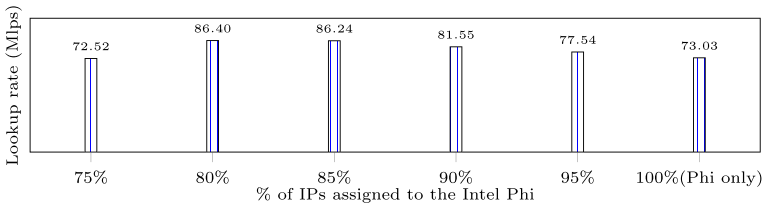

Fig. 10 Cooperative execution between CPU and Intel Phi 7120P for the AS65000 and 2^{30} input addresses for the Bloomfwd algorithm

throughput, another aspect that must be taken into consideration when choosing the buffer size is the trade-off between the buffer size and the average latency to compute lookups. Because the lookup rate attained with buffers of 2M was very close to the best performance (achieved with buffers of 4M), and considering that the time to process a 2M buffer (about 28ms including lookup and data transfers) is nearly 2× smaller than the 4M case, we have chosen the 2M configuration in the next experiments, which evaluates the cooperative execution using the CPU and the Intel Phi 7120P to process 2^{30} random IPv4 addresses.

In the cooperative evaluation, the amount of work (% of IP addresses) assigned for computation in the coprocessor is varied. The rest of the IP addresses in each case are processed by the CPU. The previous experiments show that the Intel Phi is about to 4× faster than the CPU execution. As such, our workload division strategy assigns about 80% of the IP addresses to the Intel Phi and the rest to the CPU. The results presented in Fig. 10 confirm that this is the best configuration, leading to a speedup of 1.18× on top of the Intel Phi only execution. The combined gain with async and cooperative execution is about 1.33× vs. the Intel Phi only execution.

7 Conclusions and future directions

In this work, we have designed, implemented, and evaluated the performance of efficient algorithms for IP lookup (MIHT and BFs approach) in multi-/many-core systems. The MIHT is known to be an efficient sequential algorithm [3]. However, it is also irregular, which typically leads to reduced opportunities for optimized execution on parallel systems. The baseline BFs algorithm, on the other hand, is a more compute intensive and regular algorithm with a less efficient sequential version. Nevertheless, it offers more opportunities for optimizations, for instance, due to SIMD instructions, and it is more scalable. As presented, the optimized BFs algorithm significantly outperformed the MIHT on the Intel Phi, and it was able to compute up to 169.6 Mlps (84.8 Gbps for 64B packets) and 182.7 Mlps (119.9 Gbps for 84B packets). The recent improvements in the Intel Phi, such as larger memory bandwidth, higher number of computing cores, and the possibility of its use as a standalone processor, makes it a very attractive and promising platform for the development of high-performance software routers.

Finally, as a future work, we would like to evaluate the use of our Bloom filters based approach in Open Flow networks to perform Ethernet and TCP/UDP lookup (in addition to IP). We argue this is a promising approach because Bloom filters is capable of performing different kinds of pattern matching algorithms [46]. Further, we also want to integrate our accelerated lookup into a complete software router solution as DPDK [47] or ClickOS [20] running on top of Xeon Phi hardware to investigate the efficiency of a low cost solution.

Acknowledgments
The authors thank the anonymous reviewers for helpful comments that increased the quality of the article.

Funding
This work was supported in part by CNPq and CAPES/Brazil. This research used resources of the XSEDE Science Gateways program under grant TG-ASC130023.

Availability of data and materials
The source codes of our implementation of Bloomfwd and MIHT are publicly available in the following repositories: https://bitbucket.org/gteodoro/bloomfwd and https://bitbucket.org/gteodoro/miht. The datasets were not shared because they are also public and are available in other websites.

Authors' contributions
Conceived and designed the experiments: AL, AD, and GT. Performed the experiments: AL. Analyzed the data: AL, AD, and GT. Wrote the manuscript: AL, AD, and GT. All authors read and approved the final manuscript.

Ethics approval and consent to participate
No need. Only secondary and public data used.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 30 August 2017 Accepted: 30 November 2017
Published online: 19 February 2018

References

1. Han S, Jang K, Park K, Moon S. PacketShader: a GPU-accelerated software router. ACM SIGCOMM Comput Commun Rev. 2011;41(4):195–206.
2. Kalia A, Zhou D, Kaminsky M, Andersen DG. Raising the Bar for Using GPUs in Software Packet Processing. In: NSDI. Washington, DC: IEEE; 2015. p. 409–23.

3. Lin CH, Hsu CY, Hsieh SY. A multi-index hybrid trie for lookup and updates. IEEE Trans Parallel Distributed Syst. 2014;25(10):2486–98.
4. Venkatachary S, Varghese G. Faster IP lookups using controlled prefix expansion. Perform Eval Rev. 1998;26:1–10.
5. Lucchesi A, Drummond A, Teodoro G. Parallel and Efficient IP Lookup using Bloom Filters on Intel® Xeon Phi™. In: Proceedings of the XXXV Brazilian Symposium on Computer Networks and Distributed Systems (SBRC). Porto Alegre: SBC; 2017. p. 229–42.
6. Jeffers J, Reinders J. Intel Xeon Phi Coprocessor High-performance Programming. Amsterdam, Boston (Mass.): Elsevier Waltham (Mass.); 2013.
7. Mu S, Zhang X, Zhang N, Lu J, Deng YS, Zhang S. IP Routing Processing with Graphic Processors. In: Proceedings of the Conference on Design, Automation and Test in Europe. Washington, DC: European Design and Automation Association; 2010. p. 93–8.
8. Yang T, Xie G, Li Y, Fu Q, Liu AX, Li Q, Mathy L. Guarantee IP lookup performance with FIB explosion. ACM SIGCOMM Comput Commun Rev. 2015;44(4):39–50.
9. Chu H-M, Li T-H, Wang P-C. IP Address Lookup by using GPU. IEEE Trans Emerg Top Comput. 2016;4:187–98.
10. Li Y, Zhang D, Liu AX, Zheng J. GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers. In: Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems. Washington, DC: IEEE Press; 2013. p. 1–12.
11. Ruiz-Sanchez M, Biersack EW, Dabbous W, et al. Survey and Taxonomy of IP Address Lookup Algorithms. IEEE Netw. 2001;15(2):8–23.
12. Hsieh SY, Yang YC. A classified Multisuffix Trie for IP lookup and update. IEEE Trans Comput. 2012;61(5):726–31.
13. Sahni S, Lu H. Dynamic Tree Bitmap for IP Lookup and Update. In: Networking, 2007. ICN'07. Sixth International Conference On. Washington, DC: IEEE; 2007. p. 79–9.
14. Lim H, Yim C, Swartzlander Jr EE. Priority tries for IP address lookup. IEEE Trans Comput. 2010;59(6):784–94.
15. Rétvári G, Tapolcai J, Kőrösi A, Majdán A, Heszberger Z. Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond. In: ACM SIGCOMM Computer Communication Review, vol. 3. New York: ACM; 2013. p. 111–22.
16. Asai H, Ohara Y. Poptrie: a compressed trie with population count for fast and scalable software IP routing table lookup. In: ACM SIGCOMM Computer Communication Review, vol. 45. New York: ACM; 2015. p. 57–70.
17. Dharmapurikar S, Krishnamurthy P, Taylor DE. Longest Prefix Matching Using Bloom Filters. IEEE/ACM Trans Netw. 2006;14(2):397–409.
18. Lim H, Lim K, Lee N, Park KH. On Adding Bloom Filters to Longest Prefix Matching Algorithms. IEEE Trans Comput. 2014;63(2):411–23.
19. Ni S, Guo R, Liao X, Jin H. Parallel bloom filter on xeon phi many-core processors. In: International Conference on Algorithms and Architectures for Parallel Processing. Cham: Springer; 2015. p. 388–405.
20. Kohler E, Morris R, Chen B, Jannotti J, Kaashoek MF. The Click Modular Router. ACM Trans Comput Syst (TOCS). 2000;18(3):263–97.
21. Dobrescu M, Egi N, Argyraki K, Chun BG, Fall K, Iannaccone G, Knies A, Manesh M, Ratnasamy S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In: Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York: ACM; 2009. p. 15–28.
22. Open vSwitch: An Open Virtual Switch. 2017. http://openvswitch.org/. Accessed 25 Aug 2017.
23. Fan L, Cao P, Almeida J, Broder AZ. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. IEEE/ACM Trans Netw (TON). 2000;8(3):281–93.
24. Bloom BH. Space/time trade-offs in hash coding with allowable errors. Commun ACM. 1970;13(7):422–6.
25. Kirsch A, Mitzenmacher M. Less hashing, same performance: building a better bloom filter. Random Struct Algorithms. 2008;33(2):187–218.
26. OpenMP API for Parallel Programming, Version 4.0. 2016. http://openmp.org/. Accessed 25 Aug 2017.
27. Dharmapurikar S, Krishnamurthy P, Taylor DE. Longest prefix matching using bloom filters. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. New York: ACM; 2003. p. 201–12.
28. Appleby A. MurmurHash3 Hash Function. 2011. https://code.google.com/p/smhasher/wiki/MurmurHash3. Accessed 25 Aug 2017.
29. Knuth DE. The Art of Computer Programming: Sorting and Searching, vol. 3. Boston: Pearson Education; 1998.
30. Mueller T. H2 Database Engine. 2006. http://h2database.com. Accessed 25 Aug 2017.
31. Intel. Intel Intrinsics Guide. 2015. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 25 Aug 2017.
32. Sundaram N, Raghunathan A, Chakradhar ST. A framework for efficient and scalable execution of domain-specific templates on GPUs. In: IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing. Washington, DC: IEEE; 2009. p. 1–12. doi:10.1109/IPDPS.2009.5161039.
33. Luk CK, Hong S, Kim H. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 42nd International Symposium on Microarchitecture (MICRO). Washington, DC: IEEE; 2009.
34. Augonnet C, Thibault S, Namyst R, Wacrenier PA. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing. Berlin: Springer-Verlag Publisher; 2009. p. 863–74. doi:10.1007/978-3-642-03869-3_80.
35. Teodoro G, Sachetto R, Sertel O, Gurcan MN, Meira W, Catalyurek U, Ferreira R. Coordinating the Use of GPU and CPU for Improving Performance of Compute Intensive Applications. In: IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER'09. Washington, DC: IEEE; 2009. p. 1–10.
36. Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J. OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett. 2011;21(02):173–93.
37. Teodoro G, Pan T, Kurc TM, Kong J, Cooper LA, Podhorszki N, Klasky S, Saltz JH. High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS). Washington, DC: IEEE; 2013. p. 103–14.
38. Sancho JC, Kerbyson DJ. Analysis of Double Buffering on two Different Multicore Architectures: Quad-core Opteron and the Cell-BE. In: International Parallel and Distributed Processing Symposium (IPDPS). Washington, DC: IEEE; 2008.
39. Teodoro G, Kurc T, Kong J, Cooper L, Saltz J. Comparative Performance Analysis of Intel(R) Xeon Phi(tm), GPU, and CPU: a Case Study from Microscopy Image Analysis. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. Washington, DC: IEEE; 2014. p. 1063–72.
40. Teodoro G, Kurc T, Andrade G, Kong J, Ferreira R, Saltz J. Application performance analysis and efficient execution on systems with multi-core CPUs, GPUs and MICs: a case study with microscopy image analysis. Int J High Perform Comput Appl. 2017;31(1):32–51.
41. The CAIDA UCSD Anonymized Internet Traces 2016. 2016. http://www.caida.org/data/passive/passive_2016_dataset.xml. Accessed 25 Aug 2017.
42. BGP Potaroo. 2016. http://bgp.potaroo.net/. Accessed 25 Aug 2017.
43. University of Oregon Route Views Project. 2017. http://www.routeviews.org/. Accessed 25 Aug 2017.
44. RIPE Network Coordination Centre. 2016. http://data.ris.ripe.net/. Accessed 25 Aug 2017.
45. Gomes JM, Teodoro G, de Melo A, Kong J, Kurc T, Saltz JH. Efficient irregular wavefront propagation algorithms on Intel (r) Xeon Phi (tm). In: 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). Washington, DC: IEEE; 2015. p. 25–32.
46. Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. Internet Math. 2004;1(4):485–509.
47. DPDK. 2017. http://dpdk.org/. Accessed 25 Aug 2017.