Multi-Layer Packet Classification with Graphics Processing Units

Matteo Varvello[†][*], Rafael Laufer[◇], Feixiong Zhang[□], T.V. Lakshman[◇]

[†]Telefonica Research, matteo.varvello@telefonica.com
[◇]Bell Labs, {firstname.lastname}@alcatel-lucent.com
[□]Rutgers University, feixiong@winlab.rutgers.edu

ABSTRACT

The rapid growth of server virtualization has ignited a wide adoption of software-based virtual switches, with significant interest in speeding up their performance. In a similar trend, software-defined networking (SDN), with its strong reliance on rule-based flow classification, has also created renewed interest in multi-dimensional packet classification. However, despite these recent advances, the performance of current software-based packet classifiers is still limited, mostly by the low parallelism of general-purpose CPUs. In this paper, we explore how to accelerate packet classification using the high parallelism and latency-hiding capabilities of graphic processing units (GPUs). We implement GPU-accelerated versions for both linear and tuple search, currently deployed in virtual switches, and also introduce a novel algorithm called *Bloom search*. These algorithms are integrated with high-speed packet I/O to build *GSwitch*, a GPU-accelerated software switch. Our experimental evaluation shows that GSwitch is at least 7x faster than an equally-priced CPU classifier and is able to reach 10 Gbps with minimum-sized packets and a rule set containing 128K OpenFlow entries with 512 different wildcard patterns.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Designs**]: Network communications; C.2.6 [**Internetworking**]: Routers; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Design

Keywords

Packet classification; Software-defined networking; OpenFlow; Software switch; GPU; CUDA

[*]Work done while at Bell Labs.

CoNEXT'14, December 2–5, 2014, Sydney, Australia.
Copyright 2014 ACM 978-1-4503-3279-8/14/12 ...$15.00.
http://dx.doi.org/10.1145/2674005.2674990.

1. INTRODUCTION

Recent trends in virtualization are stimulating the wide adoption of *virtual switches*, *i.e.,* software programs that run within the hypervisor to enable the communication among virtual machines. Commercial virtual switches, such as Open vSwitch [1] and Cisco Nexus 1000V [4], are already deployed in datacenters, mostly because of the tight integration they provide with virtualization software suites.

For high performance, virtual switches must sustain rates of at least 10 Gbps without incurring much overhead at the server. This is a major challenge, because it requires not only high-speed packet I/O, but also efficient packet classification algorithms. Recent advances in software-based packet I/O have shown that wire speed is achieved with optimizations in the data path [7, 10, 20]. The same, however, cannot be said about packet classification. In fact, the recent emergence of software defined networking (SDN) and OpenFlow [17], with its large rule tables and long multi-dimensional tuples, have been imposing unforeseen challenges to current packet classifiers [6, 18]. As a result, new directions to accelerate software-based switching are required in order to support the ever-increasing throughput demands.

Due to their high parallelism, graphics processing units (GPUs) have been recently proposed to accelerate several software functions, such as IP lookup [10, 14, 26], pattern matching [3, 13, 25], and packet classification [10, 11]. However, the performance evaluation of recent GPU-based packet classifiers [10, 11] shows that they still cannot sustain high packet I/O rates as the size and dimension of forwarding tables grow, e.g., the case of SDN. Exploiting the full potential of GPUs is particularly challenging because of their unique parallel programming model, which demands in-depth knowledge of the underlying hardware architecture to support high speeds. Consequently, existing software-based packet classification schemes designed for sequential general-purpose CPUs model cannot directly exploit the GPU parallelism.

In this paper, we show that much faster packet classifiers are attainable with a careful design aimed at GPU hardware architectures. In particular, our design incorporates several GPU-specific features, including (1) memory accesses are always coalesced to significantly reduce latency to off-chip memory; (2) on-chip SRAM memory is heavily utilized and shared among threads to speed up rule matching; and (3) all available GPU cores are exploited to their full extent, *i.e.,* 100% occupancy is guaranteed. Using these directives, we introduce highly efficient GPU implementations for three packet classification algorithms: linear and tuple search, currently deployed in virtual switches, and *Bloom search*, an optimization of tuple search that takes advantage of Bloom filters to avoid expensive table lookups. The three proposed algorithms are integrated with high-speed packet I/O to build *GSwitch*, a GPU-accelerated software switch.

GSwitch is evaluated using both synthetic and real rule sets [24]. We provide an experimental evaluation of each packet classification algorithm, considering only its GPU performance at first to fully understand its tradeoffs. The analysis is composed of several GPU microbenchmarks, an in-depth comparison between GPU and CPU performance using OpenCL, and a study on the classification throughput for different rule sets. Experiments with high-speed packet I/O are also performed to evaluate the forwarding throughput and packet latency of GSwitch under different scenarios.

The major outcome of this work is that a single GPU, if exploited correctly, is capable of attaining high enough throughput to shift the performance bottleneck in the data path from packet classification back to packet I/O once again. Additional outcomes of this work are summarized in the following:

- Depending on the algorithm, packet classification in our GPU is up to 7x (linear search), 11x (tuple search), and 12x (Bloom search) faster than an OpenCL implementation in an equally-priced 8-core CPU;
- Bloom search is our fastest and most scalable algorithm, supporting up to 10 Gbps with minimum-sized 64-byte Ethernet frames and a large table with 128K OpenFlow rules and 512 different wildcard patterns;
- Under a real rule set [24], GSwitch is able to forward 64-byte frames at 30 Gbps and a maximum per-packet latency of 500 μs, and achieves higher throughputs with more relaxed delay requirements.

The remainder of this paper is organized as follows. We present the related work in Section 2, and provide a brief overview of the GPU architecture in Section 3. Section 4 describes the design and implementation of the linear, tuple, and Bloom search algorithms for GPU. In Section 5 we introduce GSwitch, a GPU-accelerated software switch with high-speed packet I/O. Section 6 describes our experimental evaluation, whose results are discussed in Section 7. Finally, Section 8 concludes the paper.

2. RELATED WORK

Graphics processing units (GPUs) have been applied to several computation-intensive applications, such as IP lookup [10, 14, 26] and pattern matching [3, 13, 25]. More recently, GPUs have also been adopted to improve the performance of general packet classifiers [10, 11].

Han *et al.* [10] introduce a GPU-accelerated software router, PacketShader, where GPUs are used for packet classification and provide major performance gains. The system focus is mostly on improving packet I/O performance at the operating system level as well as efficiently offloading packet batches to GPU. To classify packets, the authors adopt an implementation of linear search. Performance results show that PacketShader is able to classify incoming packets at 7 Gbps with two GPUs, assuming a rule set of 1M exact matching rules and 1K wildcard matching rules[1].

Comparing our work with [10] is challenging due to the following reasons. First, our GPU has 32 additional cores than the GPU used in [10]. Second, the results in [10] are derived assuming two GPUs working in parallel. Third, it was not possible for the authors of [10] to open source the GPU code of PacketShader. Nevertheless, for the same scenario of 1K wildcard rules, our linear search implementation achieves 20 Gbps, cf. Figure 6(c). Assuming that the bottleneck in [10] is the wildcard matching algorithm, this indicates a 2.8x throughput gain despite we only rely on a single GPU, or 512 cores versus 960 cores. This throughput improvement is due to (1) our careful implementation that fully exploits the GPU parallelism and enforces memory coalescing, and (2) the heavy utilization of fast on-chip SRAM memory for speeding up rule matching.

[1]PacketShader implements the full OpenFlow matching algorithm, *i.e.,* exact matching plus wildcard matching in case of a miss.

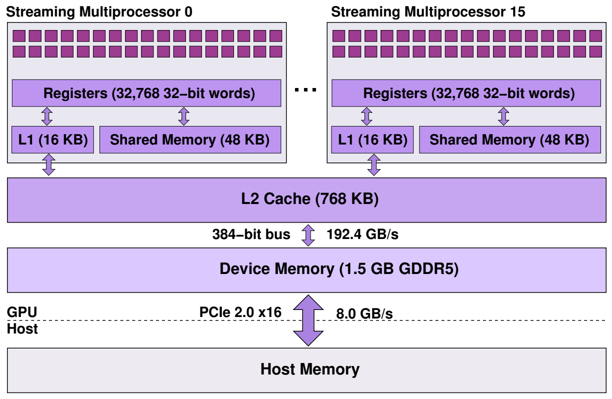

Streaming Multiprocessor 0
Streaming Multiprocessor 15
Registers (32,768 32–bit words)
L1 (16 KB)
Shared Memory (48 KB)
L2 Cache (768 KB)
384–bit bus
192.4 GB/s
Device Memory (1.5 GB GDDR5)
GPU
Host
PCIe 2.0 x16
8.0 GB/s
Host Memory

Figure 1: Architecture of the NVIDIA GTX 580 GPU.

Kang *et al.* [11] explore *metaprogramming*, *i.e.,* generating source or executable code dynamically from a more abstract program, to implement linear search in GPUs. Under this premise, rules are embedded in the code as compilation-time constants, which are locally stored in a read-only cache.

Compared to [11], our linear search implementation provides comparable performance figures. In fact their metaprogramming approach is likely to achieve caching of classifiers in SRAM memory, similarly to our design rationale. Nonetheless, linear search is our slowest solution; both tuple and Bloom search are able to classify packets up to 7x faster.

3. GPU BACKGROUND

This section reviews the architecture and operation of a graphics processing unit (GPU) as well as few metrics used to measure its performance.

Figure 1 shows the architecture of the NVIDIA GTX 580 GPU used in this work. It has a total of 16 streaming multiprocessors (SMs), each with 32 stream processors (SPs) running at 1544 MHz. Memory is hierarchically organized with higher positions reflecting faster response times as well as lower storage capacities. At the bottom of the hierarchy, the device memory (1.5 GB) and the L2 cache (768 KB) are shared among the SMs and are off-chip. In contrast, the small L1 cache (16 KB) and the shared memory (48 KB) are on-chip and guarantee low latency and high bandwidth. Finally, at the top of the hierarchy, registers (32K per SM) provide the fastest access speed, but limited storage space.

All threads in the GPU execute the same function, called *kernel*. A kernel execution consists of three steps: (1) copying the input dataset from the host CPU memory to the GPU device memory; (2) kernel execution; and (3) copying the results back from device to host memory. Although significant, the overhead of these data transfers can be amortized by pipelining via concurrent copy and execution [21]. As a result, kernel execution is usually the bottleneck and must be optimized for high performance (cf. Section 4).

The degree of parallelism of a kernel is controlled by two parameters, namely, the number of *blocks* and the number of *threads* *per block*. Blocks are sets of concurrently executing threads that collaborate with each other using shared memory and barrier synchronization primitives. At run-time, each block is assigned to a SM that first allocates the resources for the block, such as registers and shared memory (both known at compile-time), and then activates it for execution.

During execution, each SM follows the single instruction multiple thread (SIMT) parallel programming model, with all 32 SPs executing the same instruction in lockstep for a *warp* (*i.e.,* a set of 32 threads). Threads in a warp use unique identifiers to access different memory addresses, and thus perform the same task for a large dataset in parallel.

After activating a block for execution, the SM partitions it into 32-thread warps, that are independently scheduled by a dual-warp scheduler. The scheduler selects two warps at a time and issues one instruction from each. The block resources remain allocated in the SM for as long as the block is active, resulting in zero context-switch overhead for the scheduler. If a warp stalls on an instruction fetch, a synchronization barrier, or a data dependency (e.g., memory reads), other eligible warps are immediately scheduled in an attempt to hide latency.

There are several metrics to evaluate the performance of a kernel. We now describe the two most important metrics which motivate our design of packet classification in GPU. A full description of other GPU metrics can be found in [15].

Load efficiency: Cache lines in the NVIDIA GTX 580 GPU have a 128-byte granularity, which incurs a high loading overhead if threads only require a handful of these bytes. However, if all threads in a warp are instructed to read/write memory addresses within the same cache line, then the GPU efficiently maps these reads/writes into a single transaction from device memory. This memory access pattern is called *coalesced*. The load efficiency is a metric to measure the amount of coalesced memory accesses, and it is defined as the ratio between the number of requested bytes and the total number of bytes fetched from device memory.

Occupancy: The occupancy is defined as the fraction of time that the threads in the SMs are actually working, and it is limited by two factors. First, SMs have a hard limit on the number of blocks and threads concurrently executing. As a result, kernels must be carefully dimensioned to respect these limitations. Second, registers and shared memory must be available in the SM for each block before (and during) its execution. Therefore, the required per-block resources should be low to allow more blocks to run in parallel.

In the NVIDIA GTX 580, a maximum of 8 blocks and 48 warps (1536 threads) can be concurrently allocated and executing within each SM. In addition, no block is allowed to have more than 1024 threads. It follows that only a few $\langle$blocks per SM, threads per block$\rangle$ configurations are able to achieve 100% occupancy, namely $\langle 8, 192\rangle$, $\langle 6, 256\rangle$, $\langle 4, 384\rangle$, $\langle 3, 512\rangle$, and $\langle 2, 768\rangle$. In addition, each thread must not use more than $\lfloor 32\text{K}/1536 \rfloor = 21$ registers and more than $48\text{ KB}/1536 = 32$ bytes of shared memory[2] to have all 1536 threads concurrently running.

4. DESIGN AND IMPLEMENTATION

This section introduces the design and implementation of linear, tuple, and Bloom search for GPUs. Before doing so, we briefly summarize the adopted terminology.

A *tuple* is defined as the set of header fields used for packet classification. A *rule* is the ensemble of a value, a mask, an action, and a priority [9]. The rule value specifies the bits required in the header of an incoming packet to have a match, with wildcards allowed. The mask specifies the position of the wildcarded fields within the rule value. The action is the operation to perform on a packet whose tuple matches the corresponding rule. Finally, the priority defines the importance of a rule and it is used to handle the case of overlapping rules, *i.e.,* multiple rules that match the same tuple. We call a collection of rules a *rule set*.

Packet classification consists of identifying the highest-priority rule that matches the tuple of an incoming packet. After classification, the action of the matching rule is then applied to the packet.

4.1 Linear Search

Linear search consists of checking the tuple of an incoming packet against all entries in the rule set. For each rule, the tuple is first masked using the wildcard bits from the rule mask and then compared with the rule value.

Algorithm 1 shows our linear search kernel. As input, it requires the arrays R and T that contain the rules and tuples to be looked up, respectively. The kernel computes the index of the matching rule for each tuple $t \in T$ and stores them in the index array I, which is later used by the CPU to find the proper action for the packet. This design choice of storing only the rule value, mask, and priority in GPU while keeping the rule action in CPU allows us to avoid dealing with variable-sized actions, and to more efficiently exploit the memory space in GPU. The R, T, and I arrays reside in the GPU device memory. Each block also keeps a copy of n rules that are loaded at run-time to the R' array in shared memory to improve the matching speed. In GPU programming, a few variables are available to threads at run-time: blockIdx and threadIdx, corresponding to the block and the thread indexes, respectively, as well as blocks and threads, corresponding to the number of blocks and threads per block, respectively.

The kernel is designed to maximize parallelism by splitting the rule set among the several blocks, such that each block is responsible for checking the incoming tuples only against part of the set. Accordingly, line 1 computes the maximum number of rules per block, respecting the shared memory limit of n rules, and line 2 calculates the offset of the block within the rule set.

Lines 3–24 are the core of linear search. In each iteration, at most n rules are copied to shared memory and then compared to all tuples in T. The while loop in line 3 runs until the current offset exceeds the number of rules $|R|$. Line 4 computes the actual number of rules to copy. Line 5 performs coalesced memory accesses to load the subset of rules to shared memory. The READAOS call is showed in Algorithm 2, where each thread copies a 32-bit word at a time from an array of structures (AoS). Threads with consecutive indexes access consecutive addresses, guaranteeing coalesced memory accesses. The SYNCTHREADS() call in line 6 imposes a synchronization barrier to ensure that all threads in the block have finished copying after this line.

After loading rules to shared memory (lines 5-6), matching is then performed (lines 8–20). Each thread uses its threadIdx (line 7) to read a tuple t from the array T using the READSOA() call showed in Algorithm 3. In this case, the array T is organized as a structure of arrays (SoA) with 32-bit granularity, *i.e.,* the first 32-bit word of each tuple is stored sequentially in memory, then the second word of each tuple, and so on. The SoA format allows each thread to load a unique tuple while still ensuring that memory accesses are coalesced. In lines 12–18, the tuple t is checked against all rules in shared memory. If, after proper masking, t matches a rule with higher priority than any previous rule, then the priority (line 15) and global index (line 18) of this rule are saved. In case of a match,

[2]Shared memory is allocated per block, hence the limit is in practice $32 \times T$ bytes, where T is the number of threads per block.

Algorithm 1: Linear search kernel

```
Input: rules $R$, tuples $T$
Output: rule indexes $I$ for each tuple $t \in T$
Data: rules $R'[n]$ in shared memory
1  $rulesPerBlock \leftarrow$ MIN($\lceil |R|/\mathsf{blocks} \rceil, n$)
2  $rulesOffset \leftarrow rulesPerBlock \times \mathsf{blockIdx}$
3  while $rulesOffset < |R|$ do
4      $rules \leftarrow$ MIN($|R| - rulesOffset, rulesPerBlock$)
5      $R' \leftarrow$ READAOS($R, rulesOffset, rules$)
6      SYNCTHREADS()
7      $tid \leftarrow \mathsf{threadIdx}$
8      while $tid < |T|$ do
9          $t \leftarrow$ READSOA($T$, $tid$)                    // Get a tuple
10         $rIdx \leftarrow$ NIL
11         $rPri \leftarrow$ NIL
12         for $i \leftarrow 0$ to $rules - 1$ do            // Look for a match
13             $val \leftarrow$ GETVALUE($R'[i]$)
14             $mask \leftarrow$ GETMASK($R'[i]$)
15             $pri \leftarrow$ GETPRIORITY($R'[i]$)
16             if $(t \,\&\, mask) = val$ and $pri > rPri$ then
17                 $rPri \leftarrow pri$
18                 $rIdx \leftarrow rulesOffset + i$
19         if $rIdx \neq$ NIL then                          // Write to memory
20             ATOMICMAX($I[tid], (rPri \ll 24) \mid rIdx$)
21         $tid \leftarrow tid +$ threads
22     SYNCTHREADS()
23     $rulesOffset \leftarrow rulesOffset + rulesPerBlock \times$ blocks
```

Algorithm 2: Read from an array of structs (READAOS)

```
Input: array $A$, offset $o$, entries $n$
Output: array $A'$ containing entries $A[o, \ldots, o+n-1]$
Data: pointer $B$ to access $A$ as an array of 32-bit words
      pointer $B'$ to access $A'$ as an array of 32-bit words
$size \leftarrow$ WORDSPERENTRY()
$tid \leftarrow$ threadIdx
while $tid < n \times size$ do
    $B'[tid] \leftarrow B[o \times size + tid]$
    $tid \leftarrow tid +$ threads
```

lines 19–20 write the index of the matching rule with highest priority to device memory. In line 19, the ATOMICMAX(u, v) call stores the value of v into u only if v is higher than the value of u. This call is atomic to prevent potential racing conditions across all SMs. The rule priority is an 8-bit integer stored in the most significant bits to ensure that a higher-priority rule has precedence regardless of its index value.

Finally, SYNCTHREADS() is called in line 22 to ensure that all threads have finished with the current rule subset in shared memory before moving on to the next. The rule offset is then increased in line 23 and matching is performed on the next rule subset.

Algorithm 3: Read from a struct of arrays (READSOA)

```
Input: array $A$, entry index $eIdx$
Output: entry $e$ containing entry $A[eIdx]$
Data: pointer $B$ to access $A$ as an array of 32-bit words
      pointer $B'$ to access $e$ as an array of 32-bit words
$size \leftarrow$ WORDSPERENTRY()
for $i = 0$ to $size - 1$ do
    $B'[i] \leftarrow B[eIdx + i \times |A|]$
```

4.2 Tuple Search

Although simple, linear search takes $O(RT)$ time and does not scale well as the number of rules increases. Therefore, we now introduce a GPU-accelerated algorithm for tuple space search [23], which is more efficient for large rule sets. Tuple search is the packet classification algorithm adopted by Open vSwitch [1].

Tuple search uses the notion of a *class* or a set of rules with the same mask, *i.e.,* wildcards in the same header bits. Rules belonging to a class are stored in the same *class table*, which is typically implemented as a hash table for efficient lookup. In this case, the rule value is the key being hashed and the rule, represented by the set $\langle$value, action, priority$\rangle$, is stored in the table. Each class has a *class mask* which is the same for all rules. Matching in tuple search consists of, for each class, masking the tuple using the class mask and looking the result up in the class table. The rule with the highest priority among rules of all classes is the final match.

We implement class tables using cuckoo hashing [19], which provides constant lookup time and is provably more efficient than chaining. In cuckoo hashing, the table is composed of s equally-sized subtables, each with d slots. Let $R = \{r_1, r_2, \ldots, r_n\}$, with $n \leq s \times d$, be the set of rules to be inserted into the class table. To insert a rule r_i, the values $h_1(r_i), h_2(r_i), \ldots, h_s(r_i)$ are computed from a set of independent and uniform hash functions $\{h_j \mid j = 1, \ldots, s\}$ and used as indexes in each of the corresponding subtables. If one of these slots is empty, then rule r_i is stored at this slot and the insertion terminates. Otherwise, a subtable q must be selected and the rule r_j, stored at the required slot $h_q(r_i) = h_q(r_j)$, is evicted to make space for r_i. The same procedure is then repeated to reinsert r_j back into the table. If, during this procedure, a given number of tries is reached, then a new set of hash functions is selected and all rules are reinserted. Rehashes, however, are very rare if the tables are well dimensioned. The lookup of a tuple t consists of two steps. First, the class mask is applied to the tuple t, deriving the masked tuple u. Then, the slots $h_1(u), h_2(u), \ldots, h_s(u)$ in each of the corresponding subtables are checked, which is done in $O(s)$ in the worst case.

For hashing, we implement and analyze the GPU computation performance of three non-cryptographic hash functions: universal, MurmurHash3, and SuperFastHash. Due to their simplicity, universal hash functions provide by far the highest computation throughput and are chosen for our kernel. The universal hash functions used are defined as $h_i(x) = [(a_i\, x + b_i) \bmod p] \bmod d$, where p is a large prime and both a_i and b_i are randomly selected integers modulo p with $a_i \neq 0$. The parameters a_i and b_i are loaded to constant memory at the GPU initialization and stored at a read-only cache during the kernel execution for faster computations.

Algorithm 4 shows our tuple search kernel. As input, it requires arrays H, M, and T that contain the cuckoo hash tables, class masks, and tuples to be looked up, respectively. As before, the kernel returns the indexes of the highest priority rule for each tu-

Algorithm 4: Tuple search kernel

```
Input: cuckoo hash tables $H$, class masks $M$, tuples $T$
Output: rule indexes $I$ for each tuple $t \in T$
Data: cuckoo hash table $H'[s,d]$ in shared memory
      class mask $M'$ in shared memory
1  $H' \leftarrow$ READAOS($H$, blockIdx, 1)
2  $M' \leftarrow$ READAOS($M$, blockIdx, 1)
3  SYNCTHREADS()
4  $tid \leftarrow$ threadIdx
5  while $tid < |T|$ do
6      $t \leftarrow$READSOA($T$, $tid$)                    // Get a tuple
7      $u \leftarrow t \,\&\, M'$                           // Get masked tuple
8      $x \leftarrow$TUPLEHASH($u$)                         // Compute tuple hash
9      $rIdx \leftarrow$ NIL
10     $rPri \leftarrow$ NIL
11     for subtable $i \leftarrow 0$ to $s-1$ do
12         $j \leftarrow h_i(x)$                            // Get subtable slot
13         if $u = H'[i,j]$ then
14             $rPri \leftarrow$ GETPRIORITY($H'[i,j]$)
15             $rIdx \leftarrow$ blockIdx $\times s \times d + i \times d + j$
16             break
17     if $rIdx \neq$ NIL then                              // Write to memory
18         ATOMICMAX($I[tid], (rPri \ll 24) \mid rIdx$)
19     $tid \leftarrow tid +$ threads
```

ple $t \in T$ in array I. In this design, each block is responsible for matching all tuples against a single class. For improved matching performance, each block keeps a copy of the class table and mask in H' and M', which are loaded at run-time to shared memory.

Lines 1–2 load the class table and mask to shared memory using coalesced accesses. Note that, since each block is responsible for a class, we use blockIdx as the offset in both cases. A SYNCTHREADS() call is then used to ensure that threads in the block have finished copying after this line. Lines 5–16 perform the table lookup. In line 6, each thread loads a tuple $t \in T$ and a bitwise AND with the class mask is performed in line 7, deriving u. Then, line 8 computes the 32-bit tuple hash x to speed up the slot computations in line 12. The for loop (line 11) goes over each subtable, computes the candidate slot for u based on x (line 12), compares it with the rule value in that slot (line 13). If so, the priority and the global index of this rule are saved (lines 14–15) for later writing them to memory (lines 17–18). The tuple index is then increased in line 19 to load the next tuple.

4.3 Bloom Search

Tuple search requires checking a tuple against each class table, which translates to a large number of table lookups in the case of many classes. We therefore propose to use Bloom filters as a compact representation of the class tables to quick identify the matching classes and perform table lookups only when there is a filter match. This technique is similar to solutions proposed for high-speed IP lookup in SRAM [8, 22], but not yet explored in GPUs.

A Bloom filter [2] is a space-efficient data structure that consists of an array with m bits and k independent hash functions $g_1, g_2, \ldots, g_k$ whose outputs are uniformly distributed over the discrete range $\{0, 1, \ldots, m-1\}$. We use a Bloom filter to represent

Algorithm 5: Bloom search kernel

```
Input: Bloom filters $B$, tables $H$, masks $M$, tuples $T$
Output: rule indexes $I$ for each tuple $t \in T$
Data: Bloom filter $B'$ in shared memory
      cuckoo hash table $H'[s,d]$ in shared memory
      class mask $M'$ in shared memory
1  $B' \leftarrow$ READAOS($B$, blockIdx, 1)
2  $H' \leftarrow$ READAOS($H$, blockIdx, 1)
3  $M' \leftarrow$ READAOS($M$, blockIdx, 1)
4  SYNCTHREADS()
5  $tid \leftarrow$ threadIdx
6  while $tid < |T|$ do
7      $t \leftarrow$READSOA($T$, $tid$)                    // Get a tuple
8      $u \leftarrow t \,\&\, M'$                           // Get masked tuple
9      $x \leftarrow$TUPLEHASH($u$)                         // Compute tuple hash
10     $match \leftarrow$ TRUE
11     for $i \leftarrow 0$ to $k-1$ do                     // Check BF bits
12         $j \leftarrow g_i(x)$
13         if $B'[j] =$FALSE then
14             $match \leftarrow$ FALSE
15             break
16     if $match =$ TRUE then                               // Find matching rule
17         $rIdx \leftarrow$ NIL
18         $rPri \leftarrow$ NIL
19         for subtable $i \leftarrow 0$ to $s-1$ do
20             $j \leftarrow h_i(x)$                        // Get subtable slot
21             if $u = H'[i,j]$ then
22                 $rPri \leftarrow$ GETPRIORITY($H'[i,j]$)
23                 $rIdx \leftarrow$ blockIdx$(s \times d) + (i \times d) + j$
24                 break
25         if $rIdx \neq$ NIL then                          // Write to memory
26             ATOMICMAX($I[tid], (rPri \ll 24) \mid rIdx$)
27     $tid \leftarrow tid +$ threads
```

the set $R = \{r_1, r_2, \ldots, r_n\}$ of n rules within a class. Initially, all bits in the filter are set to 0. Then, for each rule $r_i \in R$, the bits of positions $g_1(r_i), g_2(r_i), \ldots, g_k(r_i)$ are set to 1. The lookup for a tuple t in a class C works as follows. First, we mask t with the class mask and derive u, then we check whether the bits of positions $g_1(u), g_2(u), \ldots, g_k(u)$ are set to 1. If at least one bit is 0, then no rule $r_i \in R$ applies to t. Otherwise, a matching rule exists in C with high probability and, with a low probability $(1 - e^{-kn/m})^k$, this is a false positive. This probability can be maintained low in practice by properly choosing parameters k and m for a given number n of rules. After a match is found in the filter, for each candidate matching class, we check the corresponding tables to either rule out eventual false positives or to retrieve the rule priority.

Algorithm 5 shows our Bloom search kernel. This kernel is similar to the tuple search kernel (cf. Algorithm 4). The main differences between the two kernels are that Bloom filters also need to be loaded to shared memory at run-time (line 1) and its bits must be checked for every incoming tuple (lines 10-14). Also, the table lookup is only performed if a rule is found by the filter (line 15). For the filter hash functions, we use universal hash functions defined as

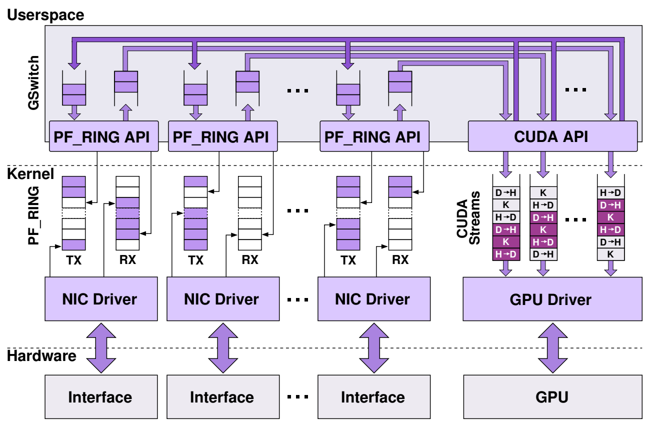


Figure 2: The software architecture of GSwitch.

$h_i(x) = (a_i\, x + b_i) \gg (32 - \log_2 m)$, where the filter size m is a power of two to avoid modular operations on a per-packet basis. The lookup in the cuckoo hash table is only performed if a match is found in the filter.

5. GSWITCH

We now integrate the previous algorithms with high-speed packet I/O. GSwitch is a multi-threaded application running in userspace and capable of classifying packets from multiple 10GbE ports simultaneously. It is built on top of publicly available APIs and composed of roughly 7K lines of C++ code. High-speed packet I/O is realized using PF_RING [7], a network socket optimized for capturing minimum 64-byte packets at wire speed. PF_RING directly exposes the NIC ring buffer to applications in userspace, thus reducing the high overhead of the OS network stack. Compared to alternative packet I/O acceleration engines, such as netmap [20] and PacketShader [10], PF_RING provides higher configuration flexibility and the same high-speed performance. For interfacing with the GPU, GSwitch uses the compute unified device architecture (CUDA) API provided by NVIDIA.

Figure 2 shows the software architecture of GSwitch. At the bottom, the NIC driver interfaces with the TX/RX ring buffers of PF_RING, and manages the hardware. When the incoming traffic rate is low, packets are received via direct memory access (DMA) initiated by regular interrupt calls. As the rate grows, packet I/O interrupts are disabled to avoid receive livelock, and periodic device polling is used for efficient interrupt coalescing. Similarly, packet transmissions are initiated by DMA requests when the number of enqueued packets in the TX ring buffer is above a minimum threshold of 128 packets.

PF_RING modifies the NIC driver to enable the NIC to share its circular buffer with an application such as GSwitch. When a packet is received at the NIC, the driver advances the writer pointer forward in the RX ring buffer; GSwitch then reads the packet and advances the reader pointer. Transmissions are performed using the TX ring buffer, with GSwitch having the role of the writer and the driver as the reader. Since each ring only has a single reader/writer, I/O operations are lock-free in the OS kernel.

GSwitch assigns a separate thread to each TX/RX ring buffer; these threads are hard-affinitized to a CPU core to eliminate processing bottlenecks and migration costs. Packets are read from the RX ring buffer either when the buffer contains enough packet to form a batch or when a timeout occurs (cf. Section 7).

GSwitch takes advantage of CUDA *streams*, which are FIFO task queues that allow threads to asynchronously send jobs to the GPU in a guaranteed execution order. Each thread has its own stream, to which it issues a sequence of commands to (1) copy a batch of tuples (header fields) from host to device memory, (2) run the packet classification kernel in GPU, (3) copy the results from device to host memory, and (4) register the batch completion using a CUDA *event*. Streams allow the CPU and GPU to overlap in time, with the CPU receiving packets of the next batch while the GPU classifies the packets of the current batch. Streams also improve overall GPU utilization with concurrent copy and execution, *i.e.,* while a kernel from one stream is running, a data copy from another stream may occur in parallel.

After dispatching a batch to the GPU, the RX thread inserts the batch information (e.g., memory address and length) into a buffer shared with the TX thread of the same NIC. The TX thread then uses a synchronous call to wait for the corresponding CUDA event indicating the batch completion. Once the GPU is finished with the batch, its packets are forwarded to the selected output TX queues. To prevent race conditions, each TX queue has a read-write lock which must be acquired before the queue can be updated.

6. EXPERIMENTAL METHODOLOGY

We perform an experimental evaluation of packet classification in GPU and GSwitch. In this paper, we assume that classifiers are statically stored in the device memory of GPU, *i.e.,* classifiers are already in the GPU before the packets arrive. Experiments are conducted both with and without packet I/O in order to isolate performance bottlenecks in the packet classification and in the packet I/O. When packet I/O is turned off, the batch of incoming packets is generated at 10 Gbps per interface directly at the switch. When packet I/O is active, the software switch is connected to a software-based traffic generator based on PF_RING, which acts both as packet source and sink.

GSwitch runs on a server equipped with two Intel Nehalem quad-core Xeon X5550 2.66 GHz processors, a Super Micro X8DAH+F motherboard, 12 GB DDR3 memory, and a NVIDIA GTX580 GPU [16] (cf. Section 3). We choose this configuration since it only includes inexpensive commodity hardware, as the whole system costs about $2,000. Additionally, this configuration allows us to make a fair comparison between the GPU and CPU performance, since the total cost of the two Intel Nehalem quad-core Xeon X5550 2.66 GHz processors is about $400, which is approximately the same price of the NVIDIA GTX 580 GPU. The traffic generator runs on a machine equipped with an 8-core AMD FX-8350 processor and 16 GB DDR3 memory. Each machine is further equipped with two Intel X520-DA2 dual-port 10GbE NICs for a theoretical aggregate rate of 40 Gbps.

Each experiment consists of 1,000 runs for which we report average and standard deviation by mean of error bars. To quantify the speed of each classification algorithm when packet I/O is turned off, we measure the number of tuples per second classified by the GPU and then derive the throughput in gigabits per second (Gbps) assuming, unless otherwise stated, 84-byte Ethernet frames, *i.e.,* the minimum 64-byte payload size plus 20-byte Ethernet overhead. With packet I/O enabled, we experiment with several different packet sizes (cf. Section 7.4) and measure the throughput as the rate at which GSwitch is able to forward the received packets.

Rule sets and incoming tuples are sampled from both real and synthetic traces. We consider two scenarios:

Realistic scenario: This scenario quantifies the performance gains of GPU acceleration for packet classification in a real setting (Sec-

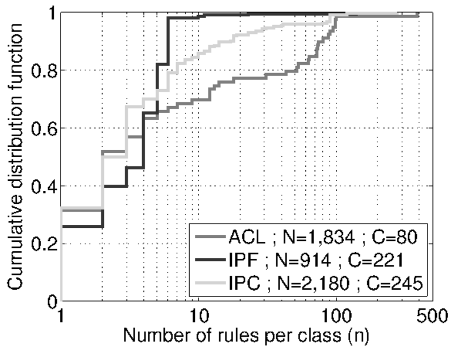


Figure 3: The cumulative distribution function (CDF) of the number of rules per class for ACL, IPF, and IPC.

tion 7.2 and 7.4). We use real rule sets from ClassBench [24], a publicly available tool for benchmarking packet classification algorithms. ClassBench comes with three real rule sets: Access control list (ACL), containing 1834 rules and 80 classes; IP forwarding (IPF), with 914 rules and 221 classes; and IP chaining (IPC), with 2180 rules and 245 classes. Each set only contains classic 5-tuple rules[3]. Figure 3 plots the cumulative distribution function (CDF) of the number of rules per class for the ACL, IPF, and IPC rule sets, respectively. Overall, the distribution of rules per class is skewed; the three sets exhibit a high number of classes (about 25-30%) with a single rule, and only few classes (about 1-2%) with more than 100 rules. The largest class is found in the ACL rule set and contains 384 rules. The trace containing the packets to be classified are also provided by ClassBench along with each rule set.

Synthetic scenario: This scenario explores the impact of system parameters and rule set characteristics on each classification algorithm (Sections 7.1 and 7.3). For this purpose, we generate synthetic rule sets varying features like the number of rules and classes. Rule generation is done by mean of a simple tool we wrote that takes a tuple with f fields as input and generates a rule set composed of N rules, each with f fields, organized as C classes. The tool uses the input tuple to generate C masks by replacing its fields with unique combinations of wildcards. Rules in a class are created by randomly varying the non-wildcarded fields. The input tuples to be classified are derived from the synthetic rule set by randomly setting masked fields.

7. EVALUATION

This section experimentally evaluates our packet classification algorithms in GPU. Section 7.1 presents the results from a series of microbenchmarks conducted to properly set system parameters. Section 7.2 then compares the throughput achieved by the GPU and our multi-core CPU. Next, Section 7.3 analyzes the impact of system parameters and rule set characteristics on the throughput achieved by each classification algorithm. Finally, Section 7.4 analyzes the performance of GSwitch focusing on throughput and delay.

7.1 GPU Microbenchmarks

Batch size: Data transfers between host and device memory occur before and after each kernel execution, and may incur a significant overhead. We are interested in determining if these transfers could affect the classification throughput.

[3]To our knowledge, there are no public OpenFlow rule sets.

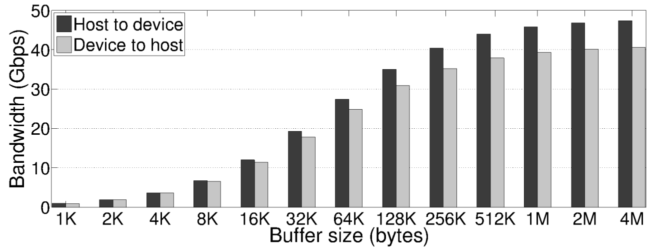


Figure 4: Data transfer rates between host and device.

Figure 4 plots the bandwidth measured for a 1-GB transfer from host to device, CPU to GPU, and vice-versa, using different buffer sizes. The buffer size determines the amount of data transferred in each transaction. Although host and device memory are connected via PCIe 2.0 x16 (64 Gbps), we measure a maximum of 48 Gbps from host to device, and 40 Gbps in the opposite direction, assuming buffer sizes of few MB. This bandwidth difference compared to the theoretical 64 Gbps is due to system call, DMA, and PCIe overheads. The asymmetry between the bandwidth measured in the two directions is expected as detailed in [10]. The figure also shows that for buffers of 1 MB or less, these overheads are even higher, which further reduces the bandwidth. Nonetheless, a 10-Gbps link carries a maximum of 14.8 million 64-byte packets; assuming the largest tuple size, *i.e.,* a 288-bit OpenFlow tuple per packet, a bandwidth of only 4.3 Gbps is required between host and device memory. Figure 4 shows that this bandwidth can be achieved with buffers of at least 8 KB, or equivalently, a batch of at least $\lfloor 8\ \text{KB}*8/288 \rfloor = 227$ packets.

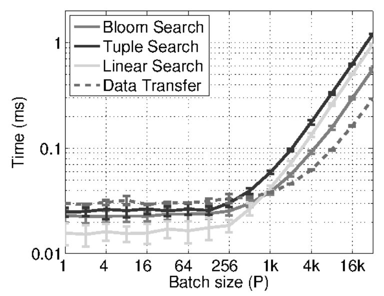
In addition to significantly reducing the data transfer overhead, batches also impose an interesting tradeoff. On the one hand, the transfer time should be lower than the kernel execution time in order for the data transfer to overlap with a previous kernel execution (cf. Section 3). On the other hand, we aim to offload enough tuples to the GPU in order to fully exploit its parallelism. Figure 5(a) and 5(b) analyze this tradeoff by plotting, respectively, the kernel execution time versus the data transfer time, and the throughput of each classification algorithm for different batch sizes P. We use a synthetic rule set with 1K rules equally distributed among 128 classes, which is representative of a realistic trace (cf. Figure 3).

Figure 5(a) shows that, for $P \leq$ 1K, the transfer time is longer than each kernel time, indicating that it cannot be effectively hidden by a concurrent kernel execution. Conversely, when $P >$ 1K, the transfer time is shorter than the execution time of even the fastest kernel, e.g., Bloom search requires 170 μs for $P =$ 8K whereas the transfer time takes only 100 μs. However, Figure 5(b) shows that, independent from the algorithm, the batch must contain at least 8K tuples to maximize throughput, and $P >$ 8K provides only a marginal throughput increase. As a result, in the remainder of the evaluation we set $P =$ 8K.

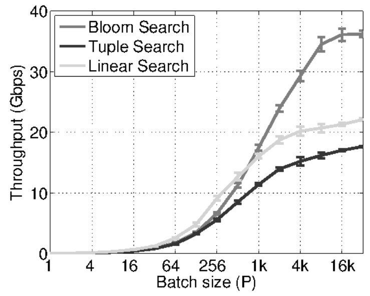
Blocks and threads per block: The number of blocks and threads also have to be properly chosen in order to maximize GPU occupancy. Table 1 shows the configurations that achieve 100% occu-

Blocks per SM (B)	2	3	4	6	8
Total number of blocks	32	48	64	96	128
Threads per block	768	512	384	256	192
Shared memory per block	24 KB	16 KB	12 KB	8 KB	6 KB

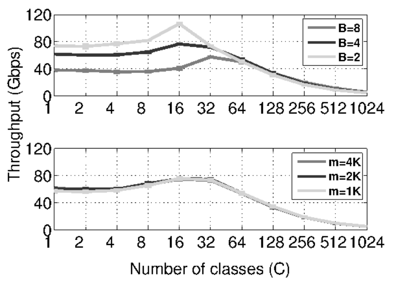
Table 1: Kernel configurations with 100% occupancy.



(a) $N = 1\text{K}, C = 128$.



(b) $N = 1\text{K}, C = 128$.



(c) Variable rules per class, see Table 2.

Figure 5: Microbenchmarks to evaluate (a) the kernel execution time versus the data transfer time, (b) the throughput as a function of the batch size, and (c) the throughput of Bloom search as a function of the number C of classes.

pancy for each possible number B of blocks per SM (cf. Section 3). We now investigate the impact of B on the proposed classification algorithms.

In linear search, each block compares all tuples against the rules in its shared memory (cf. Section 4.1). Since shared memory is equally partitioned among the B blocks, a smaller B results in a larger shared memory size per block. In this case, each block is responsible for a larger subset of rules and, due to the smaller B, tuples are read less often from device memory, significantly increasing throughput. Hence, for linear search, we set $B = 2$, as this is the best setting. We confirm this with experiments (omitted here due to space).

For tuple and Bloom search, each block handles one class at a time (cf. Sections 4.2 and 4.3), and performance depends on the number of classes in the rule set. Figure 5(c) (top plot) shows the throughput of Bloom search for different numbers C of classes assuming that (1) B equals either 2, 4, or 8; (2) the Bloom filter size is 1/6th of the available shared memory per block; and (3) each class holds the maximum number of rules (cf. Table 2). We only evaluate Bloom search as the tradeoffs are the same as those of tuple search.

Figure 5(c) shows that for $C < 64$, $B = 8$ blocks per SM provides the lowest throughput. This occurs because several blocks remain idle as there are not enough classes to occupy all blocks. As the number of classes increases, there are no idle blocks regardless of the value of B and the performance gap between the three curves is reduced. Despite being hard to see, four and eight blocks per SM guarantees the highest throughput for $50 \leq C < 200$ and $C \geq 200$ classes, respectively.

Blocks per SM	2			**4**			8		
Shared memory per block (KB)	24			**12**			6		
Tuple search									
Cuckoo hash subtables	3			**3**			3		
Slots per subtable	56			**113**			227		
Max. rules per class	168			**339**			681		
Bloom search									
Cuckoo hash subtables	3			**3**			3		
Bloom filter hash functions	2			**2**			2		
Bloom filter size (KB)	0.5	1	2	1	**2**	4	2	4	8
Slots per subtable	51	47	37	103	**94**	75	208	189	151
Max. rules per class	153	141	111	309	**282**	225	624	567	453
False positive rate	6e-3	1e-3	1e-4	6e-3	**1e-3**	1e-4	6e-3	1e-3	1e-4

Table 2: Partition of the shared memory per block.

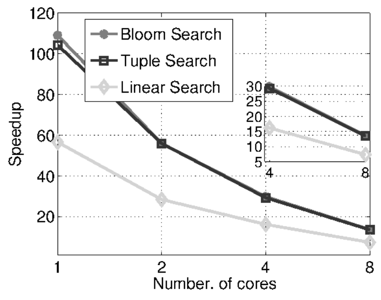
Data structures dimensioning: Table 2 shows how shared memory is partitioned in tuple and Bloom search. In both cases, we use three subtables per cuckoo hash table, which provides a low worst-case lookup time and a theoretical load factor of 90%. Each slot in the table stores a 288-bit OpenFlow tuple and the priority used for matching. The number of slots per subtable changes according to the available shared memory and the classification algorithm.

In Bloom search, we use $k = 2$ hash functions per Bloom filter in order to keep the per-tuple computation low, and consider three values for the filter size m, namely 1/3rd, 1/6th, and 1/12th of the available shared memory per block. We choose these ratios to ensure that m is a power of two, which accelerates the modulo operations in the hash computations (cf. Section 4.2). Table 2 shows that as m decreases, the maximum number of rules per class increases, since more shared memory space is available for the hash table. However, smaller filters also result in slightly higher false positive rates. These rates are relatively low and remain constant as B varies, since the ratio between the number of rules and the filter size is practically unchanged.

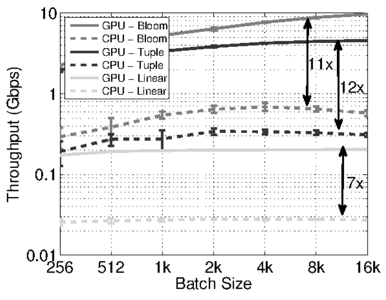
Figure 5(c) (bottom plot) shows the impact of different filter sizes (1, 2, and 4 KB) on Bloom search throughput; we use the same rule set of the top plot and set $B = 4$. Clearly, a similar throughput is measured for the different filter sizes, e.g., less than a 5% throughput decrease as m reduces from 4 to 1 KB. This indicates that the few extra table lookups due to the slightly higher false positive rate do not critically impact the classification throughput. A similar behavior was also observed for $B = 2$ and $B = 8$ (not showed).

In summary, the choice of the Bloom filter size m and the number B of blocks per SM largely depends on the rule set. We thus use the information derived from real rule sets (cf. Figure 3) to set m and B for the performance evaluation. Since the number of classes ranges between 80 and 245, we set $B = 4$ and, since the maximum number of rules per class ranges between 100 and 400 rules, we set $m = 2$ KB. The chosen configuration is shown in Table 2 in boldface.

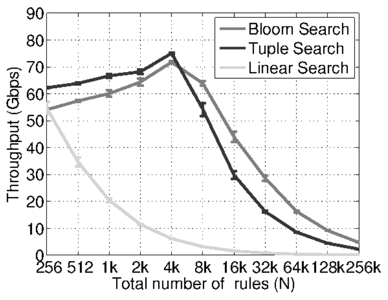
With $B = 4$ and $m = 2$ KB, each class can store up to 282 and 339 rules for Bloom and tuple search, respectively. However, we set the maximum number of rules per class n to 256 for both algorithms in order to (1) not overload the table, leaving the load factor at 90%, and (2) simplify the evaluation in Section 7.3, since the performance is the same for different numbers of rules per class. Classes with $n > 256$ are logically organized in multiple classes with the same mask, e.g., a class with 1K rules is split into four classes, each with 256 rules.



(a) $N = 128\text{K}, C = 512$.



(b) $N = 128\text{K}, C = 512$.



(c) $N = 256C$.

Figure 6: (a) The speedup of GPU over CPU as a function of the number of cores, (b) throughput of both GPU and CPU as a function of the batch size, and (c) throughput of GPU as a function of the total number N of rules.

GPU metrics: We measure our implementation of linear, tuple, and Bloom search using the GPU performance metrics discussed in Section 3. Each algorithm was implemented to reuse as many registers as possible; in total, the per-thread register count is 19 for both Bloom and tuple search, and 20 for linear search. The number of blocks and threads per blocks are chosen as discussed above, which coupled with the low register count and the proper shared memory partition guarantees 100% occupancy. Load efficiency is also 100%, as expected (cf. Algorithms 2 and 3).

7.2 GPU versus CPU

In this section, we compare the performance of packet classification in GPU and CPU. The two Intel Nehalem quad-core Xeon X5550 2.66 GHz used in our comparison is about $400 and is equivalent to the price of our NVIDIA GTX 580 GPU. For the comparison, we port the CUDA code for each algorithm to OpenCL, the open standard for parallel programming of heterogeneous systems [12]. OpenCL is a library that allows us to run packet classification in CPU in a parallel fashion. Next, we call *speedup* the ratio between the throughput measured in GPU and the throughput measured in CPU for a given experiment.

For these experiments, we assume a synthetic rule set composed by $N = 128\text{K}$ OpenFlow rules equally distributed among $C = 512$ classes. These parameters are chosen to represent a challenging rule set composed of a large number of rules and classes. We first assume a batch of 8K tuples is offloaded to GPU and later we vary the batch size from 256 up to 16K tuples.

Figure 6(a) plots the speedup of each packet classification algorithm as a function of the number of used CPU cores. In general, the speedup decreases as the number of cores increases; however, even when the maximum of 8 CPU cores are used, the GPU guarantees 7x speedup (linear search), 11x speedup (tuple search), and 12x speed (Bloom search). Figure 6(a) also shows an overall higher speedup for both tuple and bloom search compared to linear search; this happens because linear search in CPU takes better advantage of caching due to its serial nature.

Figure 6(b) plots the throughput achieved by each packet classification algorithm in both CPU (8 cores) and GPU as a function of the batch size. We see that the speedup is largely independent of the batch size. This occurs because our implementations are designed to maximize CPU/GPU utilization even if the number of input packets is low. That is, we still use a large number of threads by having each thread match the same packet on a different fraction of the rule set. Nevertheless, increasing the batch size is beneficial since the same thread can work on multiple packets absorbing the cost of loading rules to shared memory (GPU) or cache (CPU). Figure 6(b) shows another important result: tuple and Bloom search in CPU outperform linear search in GPU, which highlights the algorithmic benefits of these packet classification strategies. At its peak, Bloom search achieves about 0.7 Gbps in CPU, which is remarkably fast considering the rule set contains 128K rules.

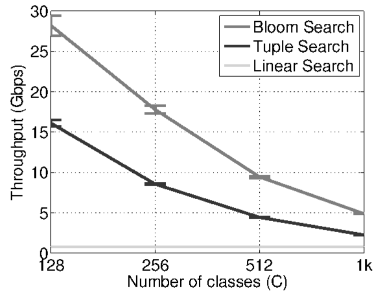
7.3 Throughput Analysis

Rule set size: In this section, we first evaluate the throughput achieved by linear, tuple, and Bloom search as a function of the total number of rules in a set. We use synthetic rules sets composed by OpenFlow rules [5, 17], *i.e.,* 288-bit tuples with 12 fields from L2 to L4. We generate several synthetic traces where C grows exponentially from 1 to 1K classes, with each class containing the maximum number of rules (*i.e.,* $n = 256$ rules). In this case, the total number N of rules grows exponentially from 256 to 256K.

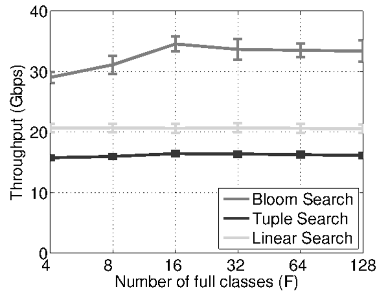
Figure 6(c) plots the throughput measured for each algorithm, as the total number of rules (and classes) increases. When $N = 256$ rules, the three algorithms have similar performance: tuple search achieves the highest throughput of 62 Gbps, whereas Bloom and linear search achieve about 55 Gbps. As N increases, the throughput of linear search drops and diverges from both Bloom and tuple search. However, our implementation of linear search still guarantees more than 20 Gbps with a set of 1K rules. When the number of classes is low ($C \leq 16$), tuple search achieves a throughput 10% higher than Bloom search. This indicates that the additional complexity required by Bloom search to load and check Bloom filters is not amortized by the gain of avoiding multiple hash table lookups, as required by tuple search. For $C > 16$, Bloom search outperforms tuple search guaranteeing about a 50% throughput increase, e.g., 45 Gbps versus 30 Gbps with 16K rules ($C = 64$). The figure also shows that Bloom search supports a 10-Gbps link with minimum packet size assuming 128K rules and 512 different classes.

Figure 6(c) also shows a counterintuitive result. As N grows from 256 to 4K rules (1 to 16 classes), the throughput measured for both Bloom and tuple search increases. This occurs because in both implementation a block is responsible for at least a class. Hence, when the number of classes is lower than the number of SMs in the GPU (16 SMs in GTX580), some blocks are idle and waste resources, as previously discussed. It follows that increasing the number of rules, and thus classes, generates a higher throughput.

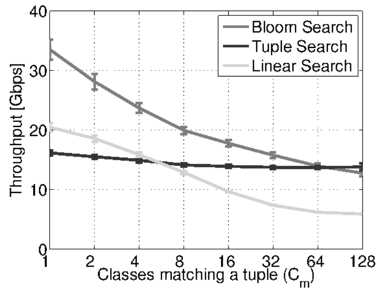
Number of classes: We now evaluate the throughput of each algorithm as the number of classes C increases, but the total number of rules N remains fixed. Figure 7(a) plots the throughput achieved by linear, tuple, and Bloom search as a function of C when $N = 32\text{K}$ rules. Since each class contains 256 rules at maximum, at least $N/256$ classes are required, *i.e.,* 128 classes.



(a) $N = 32\text{K}$.



(b) $N = 1\text{K}, C = 128$.



(c) $N = 1\text{K}, C = F = 128$.

Figure 7: Throughput as a function of (a) the number of classes C, (b) the number of full classes F, and (c) the number of classes matching a tuple C_m.

Overall, Bloom search is the fastest packet classification algorithm, reaching more than 10 Gbps when rules are organized in less than 512 classes. In comparison, tuple search achieves about half the throughput of Bloom search, e.g., 5 Gbps versus 10 Gbps when $C = 512$ classes. Linear search is the slowest packet classification algorithm, reaching only 1 Gbps regardless of the value of C, since it does not take advantage of the presence of classes.

Rule distribution: We now evaluate the throughput of each algorithm assuming that not all classes have the same number of rules. First, we introduce a parameter that expresses how rules are distributed among classes. By definition, a class exists only if it contains at least one rule. We define F as the number of *full* classes, *i.e.,* the number of classes that contain $[N - (C - F)]/F$ rules, whereas the remaining $(C - F)$ classes contain a single rule. This allows us to express the rule distribution as skewed, *i.e.,* $F = 1$ and thus a single class contains most of rules whereas the remaining classes contain a single rule, to uniform, *i.e.,* $F = C$ and thus each class contains the same number of rules.

We set $N = 1\text{K}$ and $C = 128$ to generate several synthetic traces where the distribution of rules among classes varies from skewed ($F = 4$) to uniform ($F = 128$). Figure 7(b) shows the throughput of linear, tuple, and Bloom search as F grows. Overall, the figure shows that both linear and tuple search do not depend on F, whereas a lower throughput is measured for Bloom search in presence of a skewed distribution ($F \leq 16$). For linear search, this is intuitive, since there is no notion of classes. For tuple search, this confirms that the number of classes, and not the number of rules per class, is key to predicting performance. For Bloom search, we notice a 20% throughput increase as F grows from 4 to 16 classes. This is due to the presence of idle blocks for small values of F (cf. Section 7.1). Despite each block works on two classes (*i.e.,* 128 classes divided by 64 blocks), the load distribution is uneven. In fact, the probability of finding a match in one of the classes with a single entry is very low, which means that blocks responsible for such classes only perform Bloom filter lookups, whereas F blocks perform both Bloom filter and hash table lookups.

Overlapping rules: We now evaluate the throughput of each algorithm in the presence of overlapping rules, where an incoming tuple matches several rules at once. By definition, overlapping rules must belong to different classes; we thus define C_m as the number of classes that match an incoming tuple. For this experiment, we set $N = 1\text{K}$ and $C = F = 128$, *i.e.,* uniform rule distribution and 8 rules per class, and generate several synthetic traces where C_m grows exponentially from 1 to 128 classes.

Figure 7(c) shows the throughput of each algorithm in this scenario. The throughput of linear search reduces from 20 to only 6 Gbps as C_m grows from 1 to 128 classes. This throughput decrease is due to the higher contention caused by atomic operations (ATOMICMAX, line 20 in Algorithm 1). A similar effect is noticeable for tuple search, though less evident due to the higher complexity of the tuple search kernel compared to linear search. The throughput of Bloom search largely decreases as C_m increases. The main reason of such reduction is the increase in the number of hash table accesses to solve conflicts. In the worst case, when $C_m = 128$ classes, tuple search becomes faster than Bloom search, which is intuitive since, when all classes match each incoming tuple, the lookup in the Bloom filter does not provide any benefits, but instead only introduces additional work.

5-tuple rules: We have assumed so far 288-bit OpenFlow rules composed of 12 fields. We now experiment with sets with 128-bit rules composed of 5 tuples, as commonly used by firewalls. We set $N = 1\text{K}$ and $C = F = 128$, *i.e.,* uniform rules distribution and 8 rules per class, and $C_m = 1$, *i.e.,* no overlapping rules, to generate synthetic traces where rules are composed by 5 and 12 fields.

Figure 8 compares the throughput of our algorithms when the tuple size is 5 and 12 fields. The results for 12 fields are the same as Figure 7(c) for the case of $C_m = 1$. Overall, reducing the tuple size guarantees a higher throughput, e.g., the throughput achieved by Bloom search grows from 35 to 50 Gbps. Tuple search shows

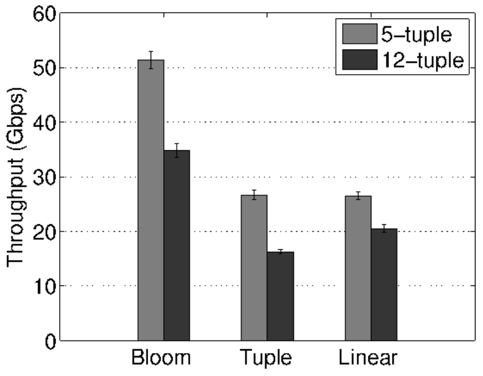


Figure 8: Throughput as a function of the number of fields in a tuple f ; N=1K, C=F=128, C_m=1.

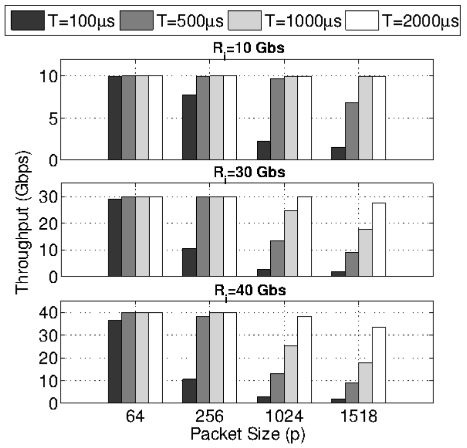


Figure 9: Bloom search throughput as a function of packet size p and maximum latency T.

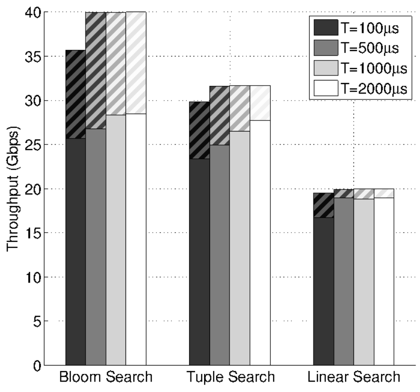


Figure 10: Forwarding throughput as a function of T.

the highest percentage increase, a 63% increase from 16 to 26 Gbps versus 43% and 25% for Bloom and linear search, respectively.

7.4 Delay Assessment

Packet processing in GPU requires large batches to maximize throughput (cf. Figure 5). However, large batches may sometimes come at the expense of additional delay. As an example, a batch of a throusand 64-byte packets requires only 67 μs to arrive in a 10 GbE link. For 1518-byte packets, however, the same thousand packets take 1.2 ms.

So far, a fixed batch size of $P = 8$K packets was assumed in order to maximize GPU performance. In a real switch with delay constraints, however, dynamic batching is required to offload packets to the GPU as either the batch is complete or a timeout occurs. In the previous example, in order to guarantee a maximum delay of 67 μs per packet when the source is transmitting at 5 Gbps, the batch size must drop to a maximum of 500 packets. We implement a mechanism in GSwitch (cf. Section 5) to allow an operator to manually specify the maximum per-packet buffering delay. The switch then configures a timeout interval to guarantee that this delay is not exceeded.

We now assess the impact of imposing a maximum tolerable packet delay on the throughput of GSwitch, considering different packet sizes and input rates. We initially focus on Bloom search, our fastest packet classification algorithm, assuming no packet I/O. The analysis is then generalized to other classification algorithms, and packet I/O is also turned on to analyze its effect on the forwarding throughput. For both experiments, the realistic scenario with the ACL trace is used.

Figure 9 shows the throughput of Bloom search as a function of the packet size, as the maximum delay T grows from 100 μs up to 2 ms. For a maximum packet size p, we consider the worst-case scenario, where traffic is composed only by packets of that size. Each subplot refers to an input rate R_i of 10, 30, and 40 Gbps, respectively. Each combination of $\langle$input rate, packet size, delay$\rangle$ imposes a maximum batch size supported by GSwitch. Figure 9 shows that, for minimum-sized 64-byte packets, the GPU forwards traffic at almost 40 Gbps with a delay as low as 100 μs. As the packet size increases to 256 bytes, GSwitch incurs a maximum delay of 500 μs to accumulate enough packets in a batch and sustain the maximum throughput. The delay must increase to 1 ms for 1 KB packets, but even 2 ms is not enough for 1518-byte packets. In this case, however, since the packet size is large, the maximum rate is only 812 Kpps per interface, which could be easily handled by the CPU. In fact, for the combination $\langle$40 Gbps, 1518 bytes, 2 ms$\rangle$ the batch size is equal to 6K packets; for this batch size, Bloom search in CPU reaches 0.7 Gbps or about 1 Mpps with 64-byte packets, cf. Figure 6(b).

We now enable packet I/O at GSwitch and show the performance of all packet classification algorithms in Figure 10. We assume 64-byte packets in order to stress the network I/O and packet classification modules. Hardware flow control is active at the NICs to slow down the traffic generator when GSwitch cannot forward packets fast enough. This occurs due to a limitation either in the packet I/O or in the packet classification algorithm. In order to isolate these effects, we plot as stacked bars the additional throughput achieved by GSwitch assuming no real packet I/O, but with batches of incoming packets being generated directly at the switch.

Overall, Figure 10 shows that, with packet I/O enabled, GSwitch achieves a maximum forwarding throughput of 27 Gbps using both tuple and Bloom search. With locally generated traffic, however, the GPU supports the maximum aggregate traffic of 40 Gbps using Bloom search when the maximum delay is larger than 100 μs. Tuple and linear search, on the other hand, are still limited by the packet classification at the GPU, and are not able to achieve 40 Gbps, even if not limited by packet I/O.

8. CONCLUSION

In this work, we investigate the benefits of using GPUs for packet classification in software-based switches such as Open vSwitch. We first propose efficient GPU-accelerated implementations for linear search, a baseline packet classification algorithm, for tuple search, the algorithm currently implemented by Open vSwitch, and for Bloom search, a proposed extension of tuple search that uses Bloom filters to prefilter table lookups. We then build *GSwitch*, a GPU-accelerated software switch with high-speed packet I/O. We show by experimental evaluation that, under a realistic scenario, GSwitch is capable of attaining attaining a much higher throughput than current software-based packet I/O engines.

Acknowledgments

We would like to thank our shepherd, Robert Ricci, and the anonymous reviewers for their constructive comments.

9. REFERENCES

[1] Open vswitch. `http://openvswitch.org/`.

[2] BLOOM, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 7*, 13 (July 1970), 442–426.

[3] CASCARANO, N., ROLANDO, P., RISSO, F., AND SISTO, R. iNFAnt: NFA pattern matching on GPGPU devices. *SIGCOMM Comput. Commun. Rev.* (2010), 20–26.

[4] CISCO. Nexus1000v. `http://cisco.com/`.

[5] CONSORTIUM, O. S. Openflow switch specification. Tech. Rep. Version 1.1.0, Feb. 2011.

[6] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proc. ACM SIGCOMM* (Toronto, Canada, Aug. 2011).

[7] DERI, L. Improving Passive Packet Capture: Beyond Device Polling. In *Proc. SANE* (Amsterdam, The Netherlands, Sept. 2004).

[8] DHARMAPURIKAR, S., KRISHNAMURTHY, P., AND TAYLOR, D. E. Longest Prefix Matching Using Bloom Filters. In *Proc. ACM SIGCOMM* (Karlsruhe, Germany, Aug. 2003).

[9] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: a network programming language. *SIGPLAN Not. 46* (2011), 279–291.

[10] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: A GPU-Accelerated Software Router. In *Proc. ACM SIGCOMM* (New Dehli, India, Aug. 2010).

[11] KANG, K., AND YANGDONG, D. Scalable packet classification via GPU metaprogramming. In *DATE* (Grenoble, France, Mar. 2011).

[12] KHRONOS OPENCL WORKING GROUP. The opencl specification. `http://khronos.org/`.

[13] LIN, C.-H., TSAI, S.-Y., LIU, C.-H., CHANG, S.-C., AND SHYU, J.-M. Accelerating String Matching Using Multi-threaded Algorithm on GPU. In *Proc. GLOBECOM* (Miami, FL, USA, Dec. 2010).

[14] MU, S., ZHANG, X., ZHANG, N., LU, J., DENG, Y. S., AND ZHANG, S. IP routing processing with graphic processors. In *Proc. DATE* (Dresden, Germany, Mar. 2010).

[15] NVIDIA. Profiler User's Guide – CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/profiler-users-guide/index.html/`.

[16] NVIDIA. GTX 580. `http://geforce.com/hardware/desktop-gpus/geforce-gtx-580/`.

[17] OPENFLOW. Switching reference system. `http://www.openflow.org/wp/downloads/`.

[18] OVERMARS, M. H., AND VAN DER STAPPEN, A. F. Range searching and point location among fat objects. *Journal of Algorithms 21* (1996), 629–656.

[19] PAGH, R., AND RODLER, F. F. Cuckoo Hashing. *Journal of Algorithms 51*, 2 (May 2004), 122–144.

[20] RIZZO, L. Netmap: a novel framework for fast packet I/O. In *Proc. USENIX* (Boston, MA, June 2012).

[21] SANDERS, J., AND KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1 ed. Addison-Wesley Professional, July 2010.

[22] SONG, H., HAO, F., KODIALAM, M., AND LAKSHMAN, T. IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards. In *Proc. IEEE INFOCOM* (Rio de Janeiro, Brazil, Apr. 2009).

[23] SRINIVASAN, V., SURI, S., AND VARGHESE, G. Packet Classification Using Tuple Space Search. In *Proc. ACM SIGCOMM* (Cambridge, USA, Aug. 1999).

[24] TAYLOR, D. E., AND TURNER, J. S. ClassBench: A Packet Classification Benchmark. In *Proc. IEEE INFOCOM* (Hong Kong, China, Mar. 2004).

[25] WANG, Y., ZU, Y., ZHANG, T., PENG, K., DONG, Q., LIU, B., MENG, W., DAI, H., TIAN, X., XU, Z., WU, H., AND YANG, D. Wire speed name lookup: a GPU-based approach. In *Proc. NSDI* (Lombard, IL, Apr. 2013).

[26] ZHAO, J., ZHANG, X., WANG, X., AND XUE, X. Achieving O(1) IP lookup on GPU-based software routers. In *Proc. ACM SIGCOMM* (New Delhi, India, Aug. 2010).