Indexing million of packets per second using GPUs

Francesco Fusco
ETH Zurich
Gloriastrasse 35
Zurich, Switzerland
fusco@tik.ee.ethz.ch

Michail Vlachos
IBM Research - Zurich
Säumerstrasse 4
Rüshlikon, Switzerland
mvl@zurich.ibm.com

Xenofontas Dimitropoulos
ETH Zurich
Gloriastrasse 35
Zurich, Switzerland
fontas@tik.ee.ethz.ch

Luca Deri
ntop.org
Pisa, Italy
deri@ntop.org

ABSTRACT

Network traffic recorders are devices that record massive volumes of network traffic for security applications, like retrospective forensic investigations. When deployed over very high-speed networks, traffic recorders must process and store millions of packets per second. To enable interactive explorations of such large traffic archives, packet indexing mechanisms are required. Indexing packets at wire rates (10 Gbps and above) on commodity hardware imposes unparalleled requirements for high throughput index creation. Such indexing throughputs are presently untenable with modern indexing technologies and current processor architectures. In this work, we propose to intelligently offload indexing to commodity General Processing Units (GPUs). We introduce algorithms for building compressed bitmap indexes in real time on GPUs and show that we can achieve indexing throughputs of up to 185 millions records per second, which is an improvement by one order of magnitude compared to the state-of-the-art. This shows that indexing network traffic at multi-10-Gbps rates is well within reach.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; C.2.3 [**Network Operations**]: Network monitoring

General Terms

Algorithms, Performance

Keywords

Indexing, packet traces, GPU

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IMC'13, October 23–25, 2013, Barcelona, Spain.
Copyright 2013 ACM 978-1-4503-1953-9/13/10 ...$15.00.
http://dx.doi.org/10.1145/2504730.2504756 .

1. INTRODUCTION

The volume of data that crosses the Internet has increased rapidly in the last years and it is expected to keep increasing fast in the future. Analysis from Cisco predicts that the volume of Internet traffic will quadruple between 2011 and 2016 reaching 1.3 Zettabytes per year in 2016 [1]. This deluge of data imposes very demanding scalability requirements on network monitoring systems. In particular, a major challenge is building network traffic monitoring and archival systems that support efficient search operations over large traffic datasets. Efficiently searching traffic data is very important because it drastically increases the utility of traffic archival systems. Otherwise, traffic archives are like a world wide web without search engines.

Specifically, this is a major challenge for several network monitoring and security devices (typically called network traffic recorders or loggers in the industry) that need to efficiently process and store a recent window (e.g., the last week) of *raw* network traffic, so as to enable post mortem analyses, for example: to show the evidence of a crime, to resolve disputes of network-related performance issues (e.g., in trading environments), or to troubleshoot connectivity problems. When deployed on high-speed links, traffic recorders must be able to store millions of packets per second and several Terabytes of data per day, without losing a single packet. Recent research has shown that it is nowadays possible to process [10] and even dump on disk [4] packets at 10 Gbps using commodity hardware. In this context, implementing searches as linear scans over a storage subsystem that is constantly taxed by writing incoming new data, is not feasible. To enable efficient search operations, real-time packet indexing technologies for high-speed links are required.

Recent work has identified compressed bitmap indexes as a very effective indexing technology for network traffic data [5, 12, 13]. Bitmap indexes are more compact in size than competitive approaches, such as tree-based indexes, and provide significant speedup over complex multi-attribute queries. In addition, our previous work [6] has shown that by introducing bitmap indexing support into the de-facto packet processing library, *libpcap* [2], packet searches can be accelerated by up to 3 orders of magnitude.

The goal of this work is to enable the creation of bitmap indexes for packet data in *real-time* in the context of packet

recording systems, by using solely commodity off-the-shelf hardware and without using distributed architectures. In fact, creating indexes *off-line* using distributed architectures is not desirable due to increased costs, higher complexity, and physical deployment constraints (i.e., rack space). Creating bitmap indexes in real-time in very high-speed links (10 Gbps and beyond) represents a major challenge: wire-rate packet indexing poses unprecedented requirements for indexing throughput. The packet indexing throughput reported by the most related previous work is *one order of magnitude* lower than the incoming packet rate that can be observed on 10 Gbps links [6, 7]. The main obstacle to providing the required throughput is poor memory locality, which, in a packet recording context, where the same memory and cache hierarchies are constantly stressed for processing packets, can severely deteriorate system performance.

To meet our goal, we propose the adoption of GPUs as indexing coprocessors to i) expand the traffic recording system with a high-bandwidth memory subsystem dedicated to indexing, ii) entirely offload the host from the bitmap index creation, thereby saving precious computational and bandwidth resources for high-speed packet recording and, iii) achieve indexing throughputs required to index packet header data in real-time from high-speed links.

In this paper we make the following contributions:

- We introduce algorithms to build two well-known compressed bitmap indexes, namely WAH [15] and PLWAH [3], *entirely* and at *high-speed* on GPUs, thereby releasing precious resources to the host for fetching, processing and storing packets on disk.

- We build the GPU-based indexing component of a traffic recording system and show that GPUs can achieve impressive indexing throughputs of up to 185 million of records per second, which is one order of magnitude higher than the state-of-the-art.

- We compare two state-of-the-art compressed bitmap indexes and show that in the GPU space, the more complex encoding (PLWAH) results in both smaller indexes and higher throughput.

- We evaluate the indexing throughput using high-entropy data, which reflect adverse conditions resulting from attacks. We learn that the throughput on a GPU is less affected by the cardinality of the data than on a CPU. GPUs are therefore more suitable when high throughput has to be sustained under adverse conditions.

Our findings show that by using GPUs real-time packet indexing at multi-10-Gbps links is well within reach. Future work needs to assess further opportunities and challenges that will arise in the integration of our GPU-based indexing component in a complete system. GPUs have already been proposed to accelerate different networking workloads [8, 14, 11]. However, to the best of our knowledge, our work is the first to exploit GPUs for indexing in the context of high-performance packet recording systems. It makes a step forward on exploiting the parallelism of current and upcoming computing architectures for high-speed network monitoring. Our contributions are not only useful in the context of indexing network monitoring data, but also for any application that requires high-throughput creation of bitmap indexes.

2. BACKGROUND

A bitmap index is an indexing data structure for numerical records that enables expedient access to the row positions matching a given value of an attribute. More importantly, bitmap indexes corresponding to distinct attributes can be used to efficiently answer queries involving boolean operations over multiple attributes, e.g. "`SELECT * WHERE` $DstPort = 80$ `AND` $Proto = 17$". The bitmap index corresponding to an attribute that can assume n distinct values is a binary matrix with n columns and as many rows as the number of indexed records (as shown in the example of Figure 1).

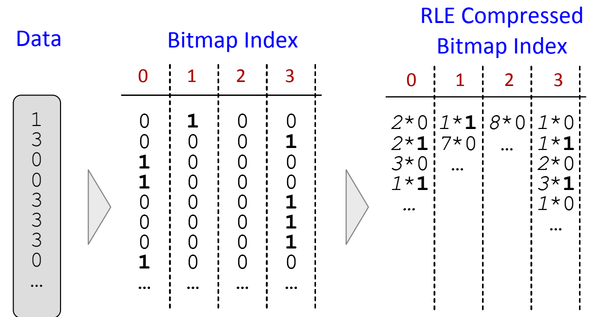

Figure 1: A bitmap index of cardinality 4.

Bitmap indexes can become very large when the number of columns (or rows) grow. Compressed bitmap indexes have been introduced to reduce the index size while preserving, or even accelerating, the index lookup time. Columns are independently compressed using light-weight compression techniques, like Run Length Encoding (RLE), that typically enable boolean operations to be performed over multiple columns in the compressed domain. Several encodings for compressing bitmap indexes have been proposed. WAH and its extension PLWAH are among the best-known compressed bitmap indexes as they stand out for their lookup performance. A thorough evaluation of the response times and compression ratios achieved by WAH and PLWAH in the context of indexing network traffic data can be found in previous work [6, 7]. Both encodings use word-aligned compression symbols.

Word Aligned Hybrid (WAH) uses a dictionary of two compression symbols: a literal `L` stores a chunk of 31 heterogeneous bits, and a fill `F` encodes a sequence of homogeneous bits. A sequence of 31 or more consecutive `0`'s is compressed using a 0-Fill symbol (`0F`), and similarly a sequence of 31 or more `1`'s is converted into a 1-Fill symbol (`1F`). The symbol type (`1F` and `0F`) is given in a 2-bit header, and a number k is recorded in the remaining 30-bits. k indicates how many $(31 \times k)$ consecutive bits of uncompressed 0's or 1's are encoded. Figure 2 provides an example of an uncompressed bitmap and its WAH-compressed counterpart.

Position Lists Word Aligned Hybrid (PLWAH) is an extension of WAH that aims at compressing sparse bitmaps better. PLWAH has the same literal `L` symbol with WAH, but introduces a different format for the 0-Fill and 1-Fill symbols. In particular, the 30-bit payload of a fill-word is used to store both the fill length k and a list of positions. The list stores as 5-bit numbers the positions of each 1 in the next literal. In this way a sparse literal can be encoded

within the fill word and then suppressed to save space as shown in the bottom of Figure 2. Therefore, the maximum length that can be encoded in each fill is $31 \times (2^{31-(5i)} - 1)$, where i is the number of positions. In practice, a single 5-bit position makes PLWAH indexes half the size of WAH indexes [3].

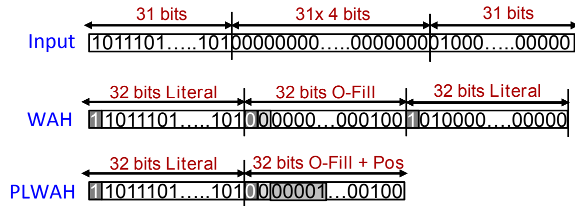

Figure 2: An uncompressed bitmap index (top) and the corresponding WAH (middle) and PLWAH (bottom) compressed bitmaps.

Given a list of input values of an attribute, the compressed bitmap can be built *incrementally*, i.e., without needing to create first the uncompressed bitmap, using the algorithm described by Lemire [9]. Input values are processed in *chunks* of 31 consecutive values. For each chunk, up to 31 literals (each corresponding to a distinct value) are created. The new literals are then appended at the end of the corresponding bitmap columns. Before appending a literal to its corresponding column, the current *chunk identifier* is compared to the identifier of the last literal appended to the column. If the difference *delta* between the current and the previous chunk identifier is greater than 1, then a 0-Fill word whose length is exactly $delta - 1$ needs to be inserted before the new literal. In this way, when an index is updated with a new chunk, only $d \leq 31$ out of the n columns have to be updated, where d is the number of *distinct* values present in the chunk.

The main obstacle to providing high-speed indexing throughput is poor memory locality, which in a packet recording context can seriously deteriorate the system performance. Intuitively, the described algorithm exhibits poor memory locality when indexing data with many distinct values. This is confirmed by our profiling measurements that show that, especially in the case of high-entropy data, cache misses are taxing the system and causing the CPU resources to be underutilized. In modern multi-core architectures, poor memory locality represents a serious performance limitation as it prevents the parallelism to be fully exploited. In fact, in multi-threaded implementations, the amount of cache *available* per thread decreases with the number of threads, making memory locality a serious problem.

3. BITMAP INDEXING ON GPUS

Modern GPUs are advanced data-parallel architectures providing hundreds of cores and an aggregated memory bandwidth that is several times higher than the bandwidth available to modern processors. In the context of packet recording, completely offloading the index creation to a GPU, removes pressure from the host's memory hierarchies, which are constantly taxed for fetching and processing packets. In contrast to CPUs, which rely on large caches to hide memory latencies, GPUs are optimized for throughput and exhibit massive data-parallelism: hundreds of hardware threads execute, in parallel, the same computation over distinct data portions. The challenge is therefore turning complex computations into sequences of simple, but highly-parallel, computing steps. In this section, we describe the steps of highly parallel computations that enable us to build compressed bitmap indexes at very high speeds directly on GPUs.

3.1 Data Flow

A given batch of numerical values of a packet header attribute (e.g., port numbers) is copied from the main memory to the GPU (see Figure 3), which computes and returns a serialized compressed bitmap index to the host memory together with metadata required to access individual index columns. The metadata consists of two parallel arrays called *keys* and *offsets*. The *keys* array is sorted and stores m distinct keys, where m is the number of distinct values present in the input data, which is smaller than the attribute cardinality (e.g., less than 65,536 for port numbers). The *offsets* array stores, for each key K_i, the offset within the index where the corresponding bitmap column can be loaded[1].

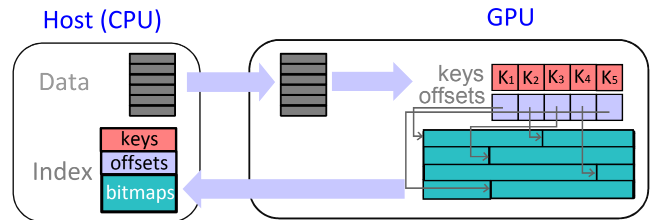

Figure 3: Data flow for indexing data on GPUs

3.2 Index creation

Exploiting the massive parallelism offered by GPUs is not trivial; it requires a complete algorithm redesign. The core idea of our approach is to exploit the high integer sorting performance provided by modern GPUs to be able to build all the bitmap index columns in parallel.

In particular, our algorithm starts with a step that associates each input value with a *row identifier (rid)*, which encodes the position of the value in the input *batch*. The input values and the corresponding rids are stored in two arrays, which are then sorted by the input values. Sorting can be performed very efficiently on GPUs. In this way, the array of rids is logically segmented into regions, one for each *distinct* input value, containing monotonically increasing identifiers. The rids array is used to produce the literal (`L`) and 0-Fill (`OF`) compression symbols. The parallelism of the GPU is further exploited by processing each individual rid on a different thread. Using a number of highly parallel refinement steps that eliminate redundant information by means of *data reductions*, the rids are turned into symbols of the compressed bitmap index.

WAH indexing. Next, we explain the steps required to build a WAH bitmap index on a GPU. Algorithm 1 shows the pseudocode of our indexing algorithm. The algorithm first copies the n input values to the array *input* in the GPU memory. Next, in line 2, it builds a second array *rids* with

[1]The length of the bitmap column i, expressed in number of words, can be computed as $offsets[i+1] - offsets[i]$.

Algorithm 1 GPUIndexCreate(values, n)

```
Input: an array values storing n input values
Output: the corresponding compressed bitmap index
1: input ← copyFromCPUToGPU(values,n)
2: sortRidsByValue(rid, input);
3: (chIds, lit) = produceChunkIDLiteral(rids)
4: k = mergeLitByValChID(input, chIds, lit)
5: produceFills(input, chIds, k)
6: idxLen = fuseFillsLiterals(chIds, lit, index, k)
7: keyCount = computeColumnLen(chIds, input, k)
8: copyFromGPUToCPU(keyCount keys and offsets)
9: copyFromGPUToCPU(index, idxLen)
```

the raw identifiers where each value is encountered. *rids* is initialized to increasing row positions 0 to $n-1$. Subsequently, the input data and the row positions are reordered in such a way that: i) the input values are sorted in ascending order, and, ii) *rids* contains the positions corresponding to the values.

Afterwards (line 3), the algorithm builds two new arrays of the same length as the input data, *chIds* and *lit*, that contain the chunk identifiers and the *partial* literals corresponding to the row identifiers. A *partial* literal is a 32-bit word with a 1-bit header and a 31-bit payload. The payload has *one* bit set to 1 as shown in Algorithm 2.

Algorithm 2 produceChunkIDLiteral(*rid*, *n*)

```
Input: an array rid of n row identifiers
Output: two parallel arrays of chunk identifiers and partial literals
1: for i = 0 → n − 1 do in parallel
2:     setBit(lit[i], rid[i] mod 31)
3:     setBit(lit[i], 31) //mark header as literal
4:     chIds[i] ← rid[i] / 31
5: end for
```

Next, in line 4, literals are created by merging the partial literals corresponding to the same input value and the chunk identifier. This operation, described in Algorithm 3, is a *segmented* parallel reduction. In fact, the *input* and *chIds* arrays divide *lit* into logical segments corresponding to each distinct tuple $(input[i], chIds[i])$. The logical segments carry the partial literals that need to be encoded in a single complete literal. Within each segment the values are reduced using a bitwise OR (see Figure 4). The result of this is that the three arrays (*chIds*, *input*, *lit*) are compacted and their length reduces from n to k, where k is the number of distinct $(input[i], chIds[i])$ pairs.

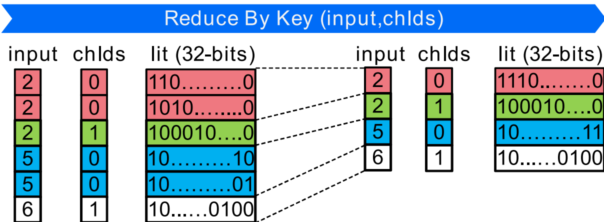

Figure 4: Literals are created by *reducing* partial literals.

The *chIds* array stores the chunk identifier corresponding to a given *literal*. The next step (line 5) is to turn the $chIds[i]$ word into the `0-Fill` symbol that precedes the literal $lit[i]$. This can be obtained by taking the difference between consecutive positions of *chIds*. If two adjacent literals belonging to the same input value are consecutive (i.e., $chIds[i] = 1 + chIds[i-1]$), the length of the 0-Fill between the two will be zero (i.e., there will not be a 0-Fill between them in the final index). It is worth to remark that this operation can be performed, in parallel, for all *chIds*, which correspond to multiple keys. A corner case is represented by the first *chId* of each key. In order to distinguish this case, a parallel array called *heads* that marks the beginning of each key has to be created. In particular, $heads[i]$ is greater than 0 if $chIds[i]$ is the first *chdId* of a key and 0 otherwise. The array *heads* can be efficiently computed in parallel by performing an *adjacent difference* over the elements of the *input* array. These operations are described in Algorithm 4.

Algorithm 3 mergeLitByValChID(*input*,*lit*,*chIds*,*n*)

```
Input: three arrays input, lit and chIds of size n
Output: lit stores complete literal symbols instead of partial literals
1: for i = 0 → n − 1 do in parallel
2:     key[i] = (chIds[i], input[i])
3: end for
4: // Merge the partial literals
5: k ← reduceByKey(key, lit, OP::bitwiseOR)
6: return k
```

Algorithm 4 produceFills(*keys*, *chIds*, *n*)

```
Input: the array chIds of chunk indentifier
       the array keys of values
Output: the chIds array stores 0-Fill symbols
1: heads ← createHeads(keys)
2: for i = 1 → n − 1 do in parallel
3:     if heads[i] == 0 then
4:         chIds[i] ← chIds[i] − chIds[i − 1] − 1
5:     else
6:         if chIds[i] ≠ 0 then
7:             chIds[i] ← chIds[i] − 1
8:         end if
9:     end if
10: end for
```

Finally (line 6 of Algorithm 1), the final compressed bitmap index is created as a concatenation of bitmap index columns. 0-Fill words and literal words are contained in the two parallel arrays *chIds* and *lit*. The output index is created by interleaving the *chIds* and *lit* arrays. This is accomplished by a *scatter* operation within each array in even and odd positions. There is still one step remaining: removing from the index the zero-values that are present whenever two consecutive literals are encountered. This operation is referred to as *stream compaction*. The pseudocode describing the process is shown in Algorithm 5.

Algorithm 5 fuseFillsLiterals(*chIds*, *lit*, *outindex*, *n*)

```
Input: the array chIds of chunk identifiers
       the array lit of literals
Output: the array outidx stores the index
1: for i = 0 → n − 1 do in parallel
2:     outIndex[2 ∗ i] = chIds[i]
3: end for
4: for i = 0 → n − 1 do in parallel
5:     outIndex[2 ∗ i + 1] = literals[i]
6: end for
7: // Remove the zeros from the output
8: idxLen ← streamCompaction(outindex, 0)
9: return idxLen
```

Once the index is created we prepare the metadata, that is, the two distinct arrays *keys* and *offsets*. Recall that the input data have been sorted and compacted via the `mergeLitByValChID` step. We use a segmented reduction to

find for a given key the length of the corresponding bitmap column. The length is computed by counting literals and fills, which are present in the GPU memory. The only consideration to be taken into account is that the zero elements in *chIds* have been removed from the final index and therefore do not have to be included in the bitmap column length. For that, we compute in-memory a temporary array *tmpArray* such that $tmpArray[i]$ is 1 when $chIds[i] == 0$ and 2 otherwise. Finally, since we would like to compute the offsets instead of the lengths for each key, we perform an *inclusive scan*. This process is described in Algorithm 6. At this stage the *index*, the *keys* and the *offsets* are ready to be copied from the GPU to the host memory.

Algorithm 6 computeColumnLen(*chIds*, *input*, *n*)

Input: the two paralell arrays *input* and *chIds* of size n
Output: the array *lengths* stores the offsets
1: *// Prepare an array for the lengths*
2: **for** $i = 0 \to n-1$ **do in parallel**
3: $tmpArray[i] \leftarrow (1 + (chId[i] == 0\ ?\ 0 : 1)$
4: **end for**
5: *// Compute the length of each bitmap index column*
6: $keycnt \leftarrow$ reduceByKey(*input*, *tmpArray*, OP::Sum)
7: *// Transform the lengths in offsets*
8: $offsets \leftarrow$ inclusiveScan(*tmpArray*)
9: **return** *keycnt*

PLWAH indexing. PLWAH is a variant of WAH that uses an additional compression step, called `mergeFillLiteral` (Algorithm 7), which merges sparse literals with the previous 0-Fill word. This additional step is executed just after step 5 of Algorithm 1 and leverages the `popc` and `clz` instructions offered by NVIDIA GPUs to count the number of bits set to one and the leading zeros, respectively. Additionally, PLWAH requires the line 3 of Algorithm 6 to be slightly modified in order to consider the case of sparse literals that have been merged into their corresponding 0-Fill.

Algorithm 7 mergeFillLiteral(*chIds*, *lit*, *n*)

Input: the two parallel arrays *chIds* and *lit* of length n
Output: literals with a single bit set are merged with the previous 0-Fill
1: **for** $i = 0 \to n-1$ **do in parallel**
2: **if** $chIds[i] \neq 0$ **then**
3: popcnt = populationCount($lit[i]$)
4: freeBits = leadingZero($chIds[i]$)
5: **if** $popcnt == 1$ AND $freeBits \geq 7$ **then**
6: encodePosition($chIds[i]$, leadingZero($lit[i]$))
7: $lit[i] \leftarrow 0$
8: **end if**
9: **end if**
10: **end for**

Limitations and considerations. WAH and PLWAH are double-sided, meaning that they can compress both sequences of 0's and sequences of 1's by using `0-Fill` and `1-Fill` words, respectively. Our algorithms do not include `1-Fill` words because this pattern, as we have shown in our previous work [7], is extremely uncommon in network traffic data. This makes the implementation simpler and more efficient.

The number of values that our GPU-implementation can index in a single batch is limited by two parameters: the maximum fill length that can be encoded by a single `0-Fill` symbol and the memory of the GPU. Our algorithm cannot create two consecutive `0-Fill` words, and, therefore, the maximum number of input values must be smaller than $31 \times ((2^{31} - 1) - 1)$ (more than 60 billions of numbers) for WAH. In a packet logging context, this number is not a practical limitation as packet traces are customarily stored in batches of Millions of packets, which usually corresponds to multi-gigabyte packet traces. A more important consideration is the available memory on the GPU. Our algorithm requires four arrays to be resident in the GPU memory for storing: the input values, the literals, the chunk identifiers, and the temporary buffers for sorting. In practice, a modern GPU with 4Gb of RAM is more than capable of indexing up to 50 Million records, which translates to traces of several Gigabytes (more than 3Gb considering the smallest packet sizes).

4. EVALUATION

We implemented our algorithms using *Thrust*, which is a C++ library provided by the NVIDIA SDK designed to enhance code productivity and more importantly performance portability across NVIDIA GPUs. To evaluate the performance of our solution, we have used similarly priced CPU and GPU: a 3.4Ghz Intel i7-2600K processor with 8 Mb of cache and a NVIDIA GTX-670 GPU fitted in a PCI-e Gen 2.0 slot.

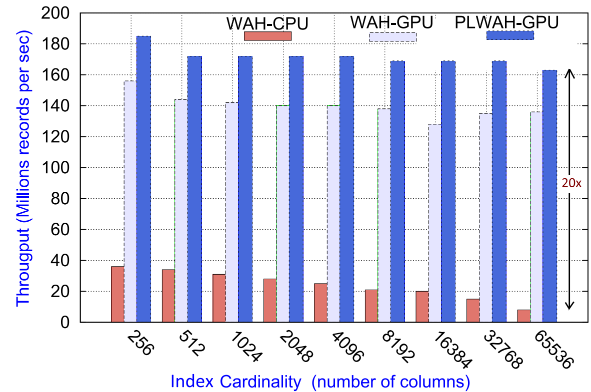

Figure 5: Indexing throughput vs cardinality for WAH on a CPU, WAH on a GPU, and PLWAH on a GPU.

Our main design goal is to enable high-speed packet indexing in the context of network traffic recording. Therefore, indexing should be able to operate flawlessly under very diverse traffic distributions, like high-entropy distributions that result from attacks. With this in mind, we evaluate how the cardinality of the index affects the indexing speed under uniformly distributed data, which is the most pessimistic scenario. We use indexes of increasing column cardinality (from 256 up to 65,536 columns) and we compare the performance obtained when indexing on a GPU using our algorithms and on a CPU using the online indexing algorithm for WAH of Lemire [9], which is the state-of-the-art. On the CPU we use a single thread implementation because it serves as a reference point that enables to estimate an upper bound on the performance that multi-core CPUs can realize. However, we highlight that the goal of this work is to show that it is feasible to build indexes entirely in a GPU without CPU intervention as in a packet recording sys-

tem CPU computational and bandwidth resources are very scarce.

In Figure 5 we show the indexing throughput of WAH on a CPU and of WAH and PLWAH on a GPU. We first find that **using a GPU we achieve up to a 20-fold speedup over indexing on a CPU**. In addition, for a cardinality of 256, we reach with PLWAH a maximum speed of 185 Million records per second. With 64-byte packets, this suggests that on a GPU we can, for example, index more than 12 attributes per packet at a sustained packet rate of 10 Gbps.

Moreover, we observe that while on the CPU the throughput decreases rapidly with the cardinality of the index, **on a GPU the throughput exhibits a much better scaling behavior**. In particular, on the CPU the throughput incurs a 4.5-fold decrease when the cardinality increases from 256 to 65,536. In sharp contrast, the throughput on the GPU decreases only by a factor of 1.13 as we can effectively exploit the available parallelism. The reason for this small decrease is that the complexity of the sorting algorithm of *Thrust* depends on the actual length (in bits) of the values to be sorted.

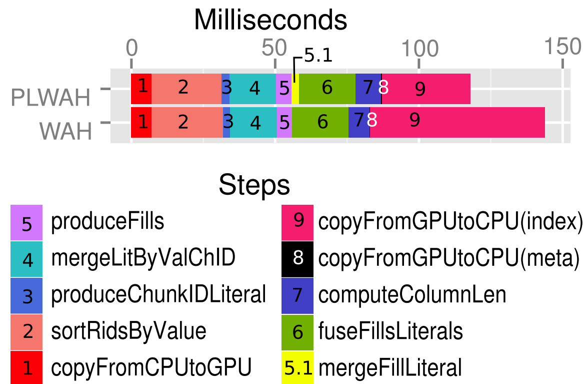

Figure 6: Time spent in different steps of our algorithm for building the WAH and PLWAH indexes on a GPU.

Furthermore, PLWAH, despite being a more complex encoding than WAH, can provide substantially higher throughputs. To better understand this point, in Figure 6 we illustrate the time spent in each step of PLWAH and WAH when indexing 20 Million random 16 bit numbers (65,536 cardinality). Recall that, compared to WAH, PLWAH uses an additional step that merges sparse literals with the previous `0-Fill` word. This step, which is indicated as 5.1 in Figure 6, is extremely fast and allows the time required to copy the index from the GPU to the host memory to be drastically reduced due to the smaller index size. From this measurement we learn that **the cost of the additional operations for building a more complex encoding in a GPU is greatly overshadowed by the savings of the more compact compression**.

5. CONCLUSION

Indexing high-speed streams of network measurement data in real-time poses significant performance challenges, especially in the context of network traffic recording, where traffic recording systems have to process packets at wire-rate without experiencing any packet loss under every traffic mix. In this paper we have shown that GPUs can provide indexing throughputs that are one order of magnitude higher than the state-of-the-art. Therefore, this work opens the path to wire-rate multi-10-Gbps packet indexing using commodity hardware.

6. REFERENCES

[1] Cisco Visual Networking Index Forecast (2011 - 2016). `http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html`.

[2] TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/.

[3] F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proc. of the 13th Int. Conf. on Extending Database Technology*, EDBT '10, 2010.

[4] L. Deri, A. Cardigliano, and F. Fusco. 10 gbit line rate packet-to-disk using n2disk. In *Proc. of the 5th Int. Workshop on Traffic Monitoring and Analysis*, 2013.

[5] L. Deri, V. Lorenzetti, and S. Mortimer. Collection and exploration of large data monitoring sets using bitmap databases. In *Proc. of the 2nd Int. Workshop on Traffic Monitoring and Analysis*, 2010.

[6] F. Fusco, X. Dimitropoulos, M. Vlachos, and L. Deri. pcapIndex: an index for network packet traces with legacy compatibility. *SIGCOMM Computer Communication Review*, 42(1):47–53, Jan. 2012.

[7] F. Fusco, M. Vlachos, and M. Stoecklin. Real-time creation of bitmap indexes on streaming network data. *The VLDB Journal*, 21:287–307, 2012.

[8] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proc. of the ACM SIGCOMM 2010 conference*, pages 195–206, 2010.

[9] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *CoRR*, abs/0901.3751, 2009.

[10] L. Rizzo. Netmap: a novel framework for fast packet I/O. In *Proc. of the 2012 USENIX Annual Technical Conf.*, 2012.

[11] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proc. of the Int. Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[12] K. Stockinger et al. Network traffic analysis with query driven visualization sc 2005 hpc analytics results. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, 2005.

[13] T. Taylor, S. E. Coull, F. Monrose, and J. McHugh. Toward efficient querying of compressed network payloads. In *Proc. of the 2012 Usenix Annual Technical Conf.*, 2012.

[14] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, 2008.

[15] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions of Database Systems*, 31:1–38, March 2006.