Indexing Millions of Packets per Second using GPUs

Francesco Fusco
ETH Zurich

Michail Vlachos
IBM Research - Zurich

Xenofontas Dimitropoulos
ETH Zurich

Luca Deri
ntop.org

Abstract

Network traffic loggers are devices that record a recent window of the entire traffic in one or more network links. The traffic is stored in packet repositories that enable retrospective analyses, e.g., for forensic investigation. Traffic loggers deployed over very high-speed networks must process and store millions of packets per second using commodity hardware. To enable interactive explorations of such large repositories, data indexing mechanisms are required. Indexing packets at wire rates (10 Gbps and above) on commodity hardware imposes unparalleled requirements for high speed index creation. Such indexing speeds are presently untenable with modern indexing technologies and current processor architectures. In this work, we propose to intelligently offload indexing to commodity Graphical Processing Units (GPUs). We introduce algorithms for building compressed bitmap indexes in real time on GPUs and show that we can achieve indexing speeds of up to 185 millions records per second, which corresponds to multi-10-Gbps line rates and an improvement by one order of magnitude compared to the state-of-the-art.

1 Introduction

Traditional network monitoring and network security applications, such as Intrusion Detection Systems (IDS), analyze the traffic flowing through an observation point using a stream-processing approach. Therefore, network traffic is analyzed on-the-fly without need to store it on disk. However, several applications have emerged that require the storage of 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 network connectivity problems.

Network recording devices (traffic loggers) process and store a recent window (e.g., the last week) of *raw* network traffic data. When deployed on high-speed links they must be able to store millions of packets per second and several Terabytes of data per day, without losing a single packet. In addition to storing incoming network traffic, packet loggers must enable efficient search operations over the collected data. Implementing searches as linear scans over a storage subsystem that is constantly taxed by writing incoming new data, is not feasible. Therefore high-speed indexing technologies capable to index packets in real-time are required.

Compressed bitmap indexes have been recognized as a very effective indexing technology for network traffic data [3, 8, 9]. They are more compact in size than competitive approaches, such as tree-based indexes, and provide significant speedup over complex multi-attribute queries. In our previous work, we have shown that by introducing bitmap indexing support into the de-facto packet processing library, *libpcap*, packet searches can be accelerated by up to 3 orders of magnitude [4].

Storing and indexing packets from 10 Gbps links using commodity hardware is a major challenge as the maximum packet rates observed on a single 10 Gbps link is 14.88 million packets per second. Recent research has shown that it is nowadays possible to process packets at 10 Gbps using commodity hardware [7]. However, the maximum packet rate observed on a 10 Gbps link is an *order of magnitude higher* than the maximum indexing throughput reported by previous research [4, 5].

In this paper, we propose the adoption of GPUs as indexing coprocessors to enable high-speed packet indexing in the context of network traffic recording using commodity hardware. Our first contribution is that we introduce novel algorithms to build two well-known compressed bitmap indexes, namely WAH [10] and PLWAH [2], *entirely* and at *high-speed* on GPUs, thereby releasing precious CPU and memory resources for fetching, processing and storing packets on disk. Second, we show that using a GPU we can accelerate the indexing throughput of a CPU by **one order of magnitude**. In

our experiments we realize impressive indexing rates of up to 185 million records per second. Third, we compare WAH and PLWAH and show that the throughput cost of additional operations for building a more complex encoding (PLWAH) in a GPU is greatly overshadowed by the savings of the more compact compression. Finally, we evaluate the indexing throughput using high-entropy data, which can be observed during Distributed Denial of Service (DDoS) attacks and other anomalous conditions. We show that the GPU throughput is less affected by the cardinality of the data, and, therefore, is more suitable for a packet logging environment, where high throughput has to be achieved not just in the average case, but also, and more importantly under adverse conditions. Our work is opening the path to real-time indexing at 10Gbps and above with commodity hardware.

2 Background and Motivation

A bitmap index is an indexing data structure for numerical records. It provides expedient access to the row positions matching a given value of an attribute. More importantly, it can efficiently answer queries that involve boolean operations over multiple attributes, e.g. "`SELECT * WHERE` $DstPort = 80$ `AND` $Proto = 17$". When indexing an attribute that can assume n distinct values, the corresponding bitmap index 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).

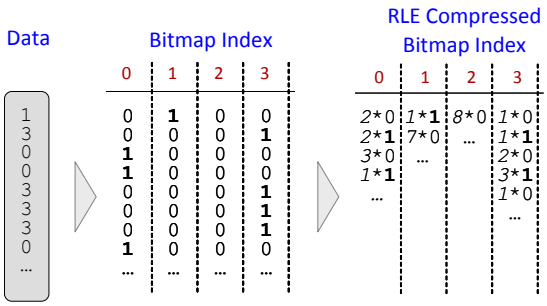

Figure 1: A bitmap index of cardinality 4.

Bitmap indexes can become very large when the number of columns or rows grow. For this reason, compressed bitmap indexes have been introduced to reduce the index size while preserving, or even accelerating, the index lookup time. Columns are compressed independently using light-weight compression techniques, like Run Length Encoding (RLE), that typically enable to perform boolean operations over multiple columns in the compressed domain. Several variants of compressed bitmap index encodings 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 [4, 5]. Both encodings use compression symbols aligned to the word. In this paper we focus on the 32-bit aligned version of these encodings, as 64-bit versions are known to produce twice as big indexes [2] and we show that WAH- and PLWAH-based indexes can be entirely built at very high-speed using modern GPUs.

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 [2].

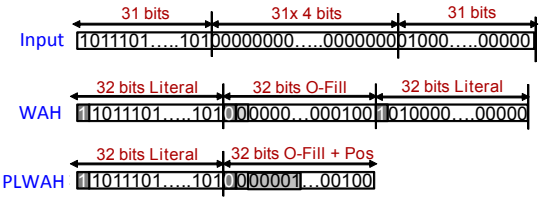

Figure 2: An uncompressed bitmap index (top) and the corresponding WAH (middle) and PLWAH (bottom) compressed bitmaps.

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 [6]. 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, 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 it is not necessary to modify the entire set of n columns but only $d \leq 31$ columns, where d is the number of distinct values in the chunk.

The indexing throughput provided by the algorithm is largely influenced by the distribution of the input values. The distribution dictates memory locality and therefore the overall performance. Intuitively, the algorithm exhibits poor locality when indexing data with many distinct values. For this reason, uniformly distributed values represent a worst case scenario. Poor memory locality prevents the parallelism offered by modern processors to be fully exploited. In our experiments (and especially in the case of high-entropy data), the rate of instructions per cycle (IPC) observed during indexing on a CPU was low due to cache misses, meaning that the CPU is underutilized. The problem is even worse in multi-threaded implementations, as the amount of cache *available* per thread decreases with the number of threads.

In a packet logging context, where the same memory and cache hierarchies are constantly stressed for processing packets, poor memory locality my deteriorate the system performance to the extent that packets are lost. In this paper, **we propose the adoption of GPUs to entirely offload the host** from the data-intensive task of building bitmap indexes, thereby saving precious computational and memory resources for packet logging.

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 contrast to CPUs, which rely on large caches to optimize 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. Our approach is substantially different from previous research on GPU-accelerated bitmap indexes [1], where a GPU has been simply used to compress and decompress a *single pre-built* bitmap column, but not on building high cardinality indexes in real-time.

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*: $keys[i]$ stores a value (e.g., port 80) and $offsets[i]$ stores the offset to the corresponding bitmap column. The length of a bitmap column, expressed in number of words, is calculated as the difference between $offsets[i+1]$ and $offsets[i]$. The length of the *keys* and *offsets* arrays correspond to 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 *keys* array is sorted.

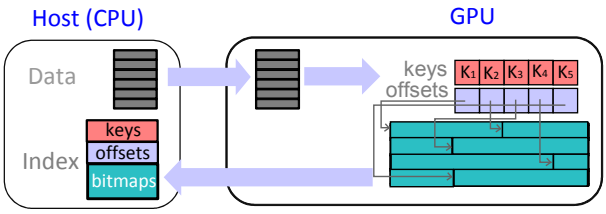

Figure 3: Workflow for indexing data on GPUs

Overview: Exploiting the massive parallelism offered by GPUs is not trivial; it requires a complete algorithm redesign. 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 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.

Algorithm 1 GPUIndexCreate(values, n)

Input: an array *values* storing n input values
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)

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 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** $i = 0 \rightarrow n-1$ **do in parallel**
2: setBit($lit[i]$, $rid[i]$ mod 31)
3: setBit($lit[i]$, 31) *//mark header as literal*
4: $chIds[i] \leftarrow rid[i]$ mod 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.

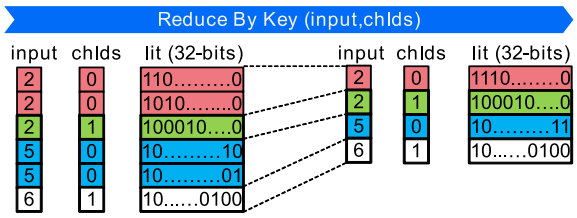

Figure 4: Literals are created by *reducing* partial literals.

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** $i = 0 \rightarrow n-1$ **do in parallel**
2: $key[i] = (chIds[i], input[i])$
3: **end for**
4: *// Merge the partial literals*
5: $k \leftarrow$ reduceByKey(key, lit, OP::bitwiseOR)
6: **return** k

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 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: $heads \leftarrow$ createHeads(*keys*)
2: **for** $i = 1 \rightarrow n-1$ **do in parallel**
3: **if** $heads[i] \neq 0$ **then**
4: $chIds[i] \leftarrow chIds[i] - chIds[i-1] - 1$
5: **else**
6: **if** $chIds[i] \neq 0$ **then**
7: $chIds[i] \leftarrow chIds[i] - 1$
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** $i = 0 \rightarrow n-1$ **do in parallel**
2: $outIndex[2*i] = chIds[i]$
3: **end for**
4: **for** $i = 0 \rightarrow n-1$ **do in parallel**
5: $outIndex[2*i+1] = literals[i]$
6: **end for**
7: *// Remove the zeros from the output*
8: $idxLen \leftarrow$ 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 → n − 1 do in parallel
3:     tmpArray[i] ← (1 + (chId[i] == 0 ? 0 : 1)
4: end for
5: // Compute the length of each bitmap index column
6: keycnt ← reduceByKey(input, tmpArray, OP::Sum)
7: // Transform the lengths in offsets
8: offsets ← 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 → n − 1 do in parallel
2:     if chIds[i] ≠ 0 then
3:         popcnt = populationCount(lit[i])
4:         freeBits = leadingZero(chIds[i])
5:         if popcnt == 1 AND freeBits ≥ 7 then
6:             encodePosition(chIds[i], leadingZero(lit[i]))
7:             lit[i] ← 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. In our implementation we do not include `1-Fill` words in our dictionary because this pattern, as we have shown in our previous work, is extremely uncommon in network traffic data [5]. 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 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 64-byte packets).

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 portability across NVIDIA GPUs. Thrust provides efficient data-parallel implementations of sorting and other primitives, including *scan*, *reduce*, *scatter*, and *gather*. To evaluate the performance of our solution we used a 3.4Ghz Intel i7-2600K processor with 8 Mb of cache and an NVIDIA GTX-670 GPU fitted in a PCI-e Gen 2.0 slot.

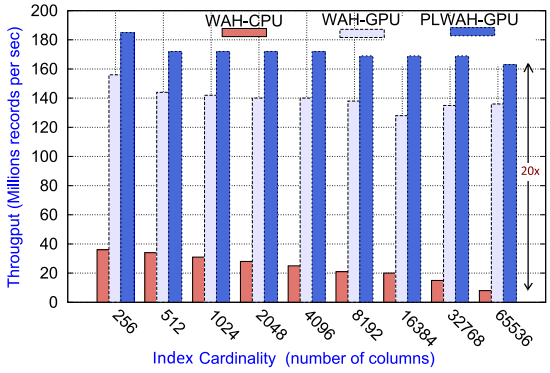

Figure 5: Indexing throughput vs cardinality for WAH on a CPU, WAH on a GPU, and PLWAH on a GPU.

Our main design goal is to enable high-speed packet indexing in the context of a network traffic recorder.

Therefore, indexing should be able to operate flawlessly under very diverse traffic distributions, like high-entropy distributions that result from DDoS 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 [6], which is the only previous work on index creation.

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. Assuming a worst case scenario of 64-byte packets, this means that on a GPU we can, for example, index more than 12 attributes per packet at a sustained packet rate of 10 Gbps or fewer attributes per packet at multi-10-Gbps rates.

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.

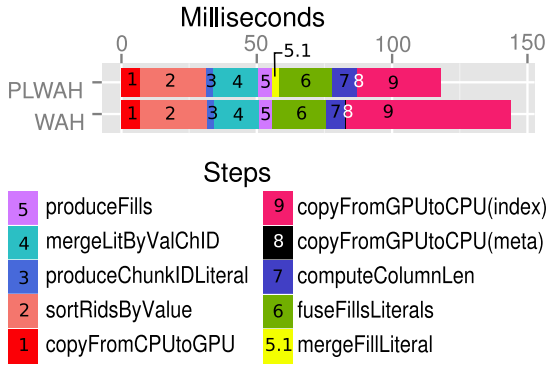

Figure 6: Time spent in different steps of our algorithm for building the WAH and PLWAH indexes on a GPU.

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 the system has to process packets at wire-rate without experiencing any packet loss. We have shown that GPUs can provide indexing throughput that are one order of magnitude higher than those achieved by CPUs. Therefore, we believe that this work is opening the path to wire-rate multi-10 Gbps packet indexing using commodity hardware.

References

[1] Andrzejewski, W., and Wrembel, R. GPU-WAH: applying GPUs to compressing bitmap indexes with word aligned hybrid. In *Proc. of the 21st Int. Conf. on Database and expert systems applications: Part II* (2010), DEXA'10, pp. 315–329.

[2] Deliège, F., and Pedersen, T. B. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proc. of the 13th Int. Conf. on Extending Database Technology* (2010), EDBT '10, pp. 228–239.

[3] Deri, L., Lorenzetti, V., and Mortimer, S. Collection and exploration of large data monitoring sets using bitmap databases. In *Proc. of the 2nd Int. Workshop on Traffic Monitoring and Analysis* (2010), pp. 73–86.

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

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

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

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

[8] Stockinger, K., , et al. Network traffic analysis with query driven visualization sc 2005 hpc analytics results. In *Proc. of the ACM/IEEE Conf. on Supercomputing* (2005), pp. 72–.

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

[10] Wu, K., Otoo, E. J., and Shoshani, A. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst. 31* (March 2006), 1–38.