GPU-Accelerated Decoding of Integer Lists

Antonio Mallia Michał Siedlaczek Torsten Suel Mohamed Zahran
New York University
{antonio.mallia,michal.siedlaczek,torsten.suel,mohamed.zahran}@nyu.edu

ABSTRACT

An inverted index is the basic data structure used in most current large-scale information retrieval systems. It can be modeled as a collection of sorted sequences of integers. Many compression techniques for inverted indexes have been studied in the past, with some of them reaching tremendous decompression speeds through the use of SIMD instructions available on modern CPUs. While there has been some work on query processing algorithms for Graphics Processing Units (GPUs), little of it has focused on how to efficiently access compressed index structures, and we see some potential for significant improvements in decompression speed.

In this paper, we describe and implement two encoding schemes for index decompression on GPU architectures. Their format and decoding algorithm is adapted from existing CPU-based compression methods to exploit the execution model and memory hierarchy offered by GPUs. We show that our solutions, GPU-BP and GPU-VByte, achieve significant speedups over their already carefully optimized CPU counterparts.

KEYWORDS

Compression; Inverted Indexes; Graphics Processing Units (GPUs);

ACM Reference Format:
Antonio Mallia Michał Siedlaczek Torsten Suel Mohamed Zahran. 2019. GPU-Accelerated Decoding of Integer Lists. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM'19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3357384.3358067

1 INTRODUCTION

An inverted index is the key data structure used in most current large-scale text search systems. It is composed of *posting lists*, one for each distinct term in a collection. A posting list is a sequence of the IDs of the documents containing the corresponding term, usually along with respective in-document frequencies or other information needed for ranking. Given the extremely large collections indexed by current search engines, even a single node of a large search cluster typically contains many billions of integers. Thus, both space efficiency and index access speed are crucial to maintain acceptable query response times [9]. This motivates the use of specialized index compression techniques that reduce space while also supporting extremely fast decompression. Such techniques have been extensively studied, and recently much attention has been directed towards improving decoding throughput by taking advantage of SIMD instructions available on modern CPU architectures [6, 7, 11, 13–15].

While CPU manufacturers are constantly widening vector register sizes, their degree of parallelism comes nowhere close to the one offered by current GPUs. There is some amount of recent work on techniques for processing search queries on GPUs [3–5, 8, 16], but most of it does not focus on how the index is compressed. Sometimes the inverted index is used in uncompressed form, while most other papers focus on query processing and spend little effort on speeding up decompression. Although graphics card memory is increasing in size, such memory is still quite expensive, and thus index compression is crucial when dealing with large corpora. Thus we believe that there is still a potential for significant improvements in decompression speed for GPU-based inverted index structures.

Our contributions. We list here our main contributions.

(1) We explore efficient ways of decoding the posting lists of an inverted index on GPUs. Any improvements for index decoding speed is likely to translate into faster query processing. Furthermore, an efficient and well-assessed compression solution is necessary to further improve GPU-accelerated query processing algorithms.
(2) We design and implement two encoding schemes that are able to perform index decompression on GPU architectures. Their format and decoding algorithm is adapted from existing CPU-based compression methods to exploit the programming model offered by GPUs.
(3) We conduct an extensive experimental analysis to demonstrate the effectiveness of our approach. Compared to existing techniques implemented for standard CPUs, our GPU counterparts are: (1) one to almost two orders of magnitude faster, and (2) only marginally larger in size.

2 BACKGROUND AND RELATED WORK

We start by providing background on GPUs and index compression.

Graphics Processing Units. Modern GPUs, massively parallel architectures consisting of thousands of cores and high memory bandwidth, can achieve superior performance on many demanding applications [10]. However, to fully exploit the functionality of GPUs, data and algorithms need to be carefully adapted to its programming model. In the context of GPUs, threads are grouped into blocks of 32, known as *warps*. Every thread in a warp executes the same instruction according to the SIMD paradigm. For this reason, it is important to avoid branch divergence within warps.

GPUs come with their own, hierarchically structured, device memory. The CPU transfers data to and from the GPU's *global memory*. Although this comes with a high overhead, recent GPUs feature up to 32 GB of memory capacity, enough to store an inverted index for tens of millions of documents. Despite global memory's

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11.
https://doi.org/10.1145/3357384.3358067

high bandwidth, directly accessing it by GPU threads is inefficient due to long latency response times. Thus, specific access patterns utilizing L2 and L1 caches are required for efficient algorithms.

Delta encoding. Typically, document IDs in each posting list are sorted and then represented by the differences between consecutive numbers, called *delta gaps*. As a result, the values we encode are smaller, which can significantly improve compression; this is especially true for dense lists. Once decoded, delta gaps are converted to IDs by computing a prefix sum.

Binary packing. *Binary Packing* [1] groups numbers into fixed-sized blocks. For each block, its selector b is the smallest number of bits required to binary-encode the largest element of the group. The selector is binary-encoded in one byte, followed by the values belonging to the group, each encoded in b bits. For better performance, we can store four selectors at a time in one 32-bit word, followed by their respective groups. In the experiments described in Section 4, we use blocks of 32 elements, and we refer to this approach as BP32. Lemire and Boytsov [6] proposed a Binary Packing method that exploits SIMD instructions. This method, called SIMD-BP, packs 128 consecutive integers into as few 128-bit words as possible. Selectors are stored in groups of 16, to fully utilize 128-bit SIMD operations.

Variable Byte. The encodings in the *Variable Byte* family are known for their high decoding speed. Arguably the simplest and best known is VByte, which uses 7 bits per byte to store the binary representation of a number and one remaining bit to indicate whether the same binary code continues in the next byte. These continuation bits, when put together, form unary-encoded byte-lengths of encoded numbers. To improve decoding speed, Dean [2] proposed VarintGB, which groups these lengths together and encodes them in binary instead: one byte is used to store four 2-bit sizes of the next four integers, followed by their binary representations.

StreamVByte [7] combines the benefits of VarintGB and SIMD instructions. Like VarintGB, it stores four integers per block with a 1-byte binary descriptor. However, descriptors are stored sequentially in a separate bit stream, which improves access speed.

3 PROPOSED SOLUTIONS

Our GPU-based methods are based on Binary Packing and Variable Byte encodings that achieve very fast decoding speeds on CPUs and are particularly suitable for GPU implementation.

3.1 GPU Binary Packing

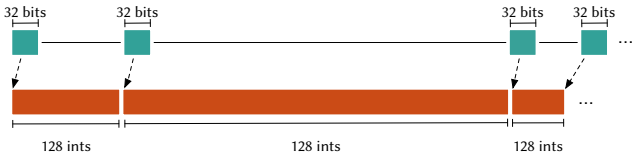

Figure 1: Representation of the GPU-BP128 **schema**

Format. Instead of block descriptors, as in BP32, we store an array of 32-bit integers that point to block endpoints. For convenience, we insert an additional endpoint element that points to the beginning, as shown in Figure 1. Since each block contains the same number of elements, the bit-length of each element in a block can be computed by dividing the length of that block by the number of elements, while the block length is given by the difference between two consecutive endpoints.

Algorithm 1: Decoding algorithm of GPU-BP

```
1 Function Decode(out, endpoints, n)
      In  : The endpoints array endpoints, number of elements
            to decompress n
      Out : Decompressed array out
2     for each block j do
3         for each thread i do
4             begin ← endpoints[j]
5             end ← endpoints[j + 1]
6             b ← Bits(begin, end)
7             offset ← i × b
8             p ← j × BLOCK_SIZE + i
9             out[p] ← Extract(begin, offset, b)
```

Decoding. In our algorithm, each thread block is responsible for decoding one compressed block, while each thread decodes one element of the block. The block boundaries and the element offsets can be quickly determined by accessing the *endpoints* array and performing several arithmetic operations as shown in Algorithm 1. In Section 4, we report experiments for blocks of 128 and 256 elements, referred to as GPU-BP128 and GPU-BP256, respectively.

3.2 GPU VByte

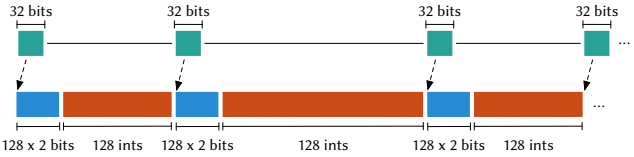

Figure 2: Representation of the GPU-VByte128 **schema**

Format. As with VarintGB, integers are grouped into fixed-sized blocks, and each block is preceded by a fixed number of 2-bit selectors. Similarly to GPU Binary Packing, we also store an array of endpoints of the blocks, as illustrated in Figure 2. In our experiments, we use blocks of 128 and 1024 elements, referring to these methods as GPU-VByte128 and GPU-VByte1024.

Algorithm 2: Decoding algorithm of GPU-VByte

```
1 Function Decode(out, endpoints, n)
      In  : The endpoints array endpoints, number of elements
            to decompress n
      Out : Decompressed array out
2     for each block j do
3         for each thread i do
4             begin ← endpoints[j]
5             b[i] ← 8 × (1+Extract(begin, i × 2, 2))
6             offsets[i] ← InclusiveSum(b)
7             data ← begin + 2 × BLOCK_SIZE
8             p ← j × BLOCK_SIZE + i
9             out[p] ← Extract(data, offset[i], b[i])
```

Decoding. As with GPU-BP, each thread decodes a single element in a block. Since elements are encoded with different numbers of bits, each thread must read its own size and store it to an array shared by the thread block. Furthermore, to retrieve the offsets to

each binary representation, a cumulative sum over the array must be computed. The pseudocode is provided in Algorithm 2.

4 EXPERIMENTAL RESULTS

Testing details. All algorithms are implemented in C++11 and compiled with GCC 4.8.5 with the highest optimization settings. The tests are performed on a machine with an Intel Xeon E5-2690 v4, with 28 cores, clocked at 3.50 GHz with 256GB RAM, running Linux 3.10.0. Only a single CPU core was used in each run. The server also has an NVIDIA Tesla V100 GPU with 16GB memory. The GPU programming platform is CUDA Toolkit 9. We also ran our GPU codes on an NVIDIA GeForce GTX 1080, which is a lower-end GPU. Algorithms running on the latter are marked with an asterisk (*). Before timing, we ensure that the required integer lists are fully loaded in memory. It is reasonable to believe that in a real-world scenario the inverted index is transferred to the GPU only once, and thus does not contribute to the query processing time. We use the FastPFor[1] library for the CPU-based implementations of BP32, VarintGB, SIMD-BP and StreamVByte. Reported times do not include the cost of summing up the d-gaps to retrieve the document IDs. Posting lists were not partitioned into blocks: although blocks are important for document-at-a-time processing on CPUs, it is questionable if they are useful on GPUs. Encoding blocks of posting lists only for the CPU compression algorithms would have been an unfair comparison, as it would have slowed these down even further due to the additional indirection. Our source code is publicly available [2] for readers interested in replicating the experiments.

Datasets. We first ran our experiments on synthetic datasets previously used by Anh and Moffat [1]. They adopted two different distribution models to generate lists of distinct integers: the Uniform model, which produces lists of integers according to a uniform distribution, and the Clustered model, where values are clustered such that sub-segments of integers contain similar values. We generated datasets of random integers in the range $[0, 2^{29})$ with both the Uniform and the Clustered model. Sparse lists contain 2^{16} integers, and dense lists contain 2^{25} integers.

We also ran experiments on standard datasets, in particular:.

- Gov2: the TREC 2004 Terabyte Track test collection consisting of 25 million .gov sites crawled in early 2004.
- ClueWeb09: the ClueWeb 2009 TREC Category B collection consisting of 50 million English web pages crawled between January and February 2009.

Documents were parsed using Apache Tika[3]. The words were lowercased and stemmed using the Porter2 stemmer; no stopwords were removed. Document IDs were assigned according to a lexicographic order of URLs [12]. To evaluate how decoding speed would improve realistic query processing settings, we randomly selected 1000 queries from the TREC 2005 and TREC 2006 Terabyte Track Efficiency Task (500 from each query set) and measured decoding times for all the posting lists in these queries.

Compressed size. Tables 1 and 2 summarize the compression (in bits per integer) achieved under several setups. For all datasets, our solutions show a fairly small difference in size with respect to their CPU counterparts, with the exception of BP32, which results in a more compact representation on GPUs due to a smaller block size.

Method	Uniform		Clustered	
	Sparse	Dense	Sparse	Dense
BP32	15.71	6.67	13.91	5.40
SIMD-BP	16.04	6.99	14.26	5.60
GPU-BP128	16.22	7.18	14.45	5.79
GPU-BP256	16.20	7.18	14.58	5.86
VarintGB	17.76	10.00	16.27	10.02
StreamVByte	17.76	10.00	16.27	10.02
GPU-VByte128	18.16	10.50	16.68	10.52
GPU-VByte1024	17.78	10.06	16.27	10.08

Table 1: Compression (in bpi) achieved by the encoders on Uniform and Clustered datasets for sparse and dense lists.

Method	Gov2		ClueWeb09	
	docids	freqs	docids	freqs
BP32	5.06	3.44	7.07	3.62
SIMD-BP	6.56	4.56	8.92	5.03
GPU-BP128	6.66	4.27	8.80	4.63
GPU-BP256	7.15	4.50	9.34	5.01
VarintGB	10.86	10.37	11.13	10.34
StreamVByte	10.86	10.37	11.13	10.34
GPU-VByte128	11.67	10.92	11.93	10.92
GPU-VByte1024	11.39	10.34	11.69	10.36

Table 2: Compression (in bpi) achieved by the encoders on Gov2 and ClueWeb09 datasets.

Decompression speed. Figure 3 plots the achieved decoding speed in millions of integers per second, while varying the lengths of the lists. Although the SIMD CPU version of the decoding algorithms is always faster than the serial CPU one, it is interesting to notice how the gap decreases as the lists become longer, turning into an almost marginal difference. This behaviour is caused by the capacity of the CPU cache, which results in limited SIMD advantages. Furthermore, VarintGB, in the case of longer and thus denser lists, encodes most of the values using 8 bits, which becomes a naive case to be optimized by the branch predictor of CPU and, thus, faster to execute.

On the other hand, our GPU solutions are positively affected by increasing lists lengths, due to the increase in available data parallelism. BP32 exhibits an almost flat trend, while VarintGB, for very dense lists, reaches the same performance of StreamVByte. This can be explained by the fact that, from a certain point, most elements can be represented with a single byte, making the task easier for the CPU and less appealing for vectorization. GPU-BP and GPU-VByte attain similar decoding speeds, with the former being slightly faster due to its simpler implementation. The overall fastest speed is obtained by GPU-BP256, reaching over 100 billion integers per second decoding speed for long list, way beyond anything previously reported. We also see a significant gap between the top-of-the-line Tesla V100 GPU and the lower-end GTX 1080, though even the latter is still much faster than the CPU methods.

Table 4 summarizes results obtained on Gov2 and ClueWeb09 by decoding the posting lists of terms appearing in actual queries. These posting lists turn out to be quite long, which indicates our proposed solutions could be very competitive in real scenarios.

Speedup factors relative to the decompression speed shown in Figure 3 of GPU algorithms against their CPU/SIMD counterparts

[1] https://github.com/lemire/FastPFor
[2] https://github.com/amallia/gpu-integers-compression
[3] http://tika.apache.org

	Method	List length									
		2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
CPU	GPU-BP128*	3.11	7.26	11.10	9.81	10.85	11.94	12.34	14.21	17.40	19.40
	GPU-BP128	2.94	7.09	12.27	19.57	26.80	33.58	37.02	48.20	54.83	63.29
	GPU-BP256	3.31	7.82	13.45	23.81	37.39	37.69	43.61	56.52	67.28	78.02
	GPU-VByte128*	7.31	11.69	16.90	16.25	18.40	22.68	25.85	11.29	11.07	11.13
	GPU-VByte128	6.26	11.62	21.19	34.33	49.79	60.75	74.76	36.03	35.96	36.19
	GPU-VByte1024	6.47	11.94	20.48	31.71	44.29	52.10	61.71	29.85	29.43	29.46
SIMD	GPU-BP128*	1.66	3.06	4.83	4.36	4.81	5.26	5.56	10.96	14.24	14.70
	GPU-BP128	1.58	2.99	5.35	8.70	11.89	14.81	16.68	37.18	44.89	47.95
	GPU-BP256	1.77	3.30	5.86	10.58	16.59	16.63	19.65	43.60	55.09	59.11
	GPU-VByte128*	1.78	2.91	4.17	3.97	4.37	4.75	4.97	9.16	11.29	11.39
	GPU-VByte128	1.53	2.89	5.24	8.41	11.82	12.74	14.39	29.26	36.66	37.05
	GPU-VByte1024	1.58	2.97	5.06	7.76	10.52	10.93	11.88	24.24	30.00	30.15

Table 3: Speedup factor achieved by GPU-BP and GPU-VByte against their CPU and SIMD counterparts.

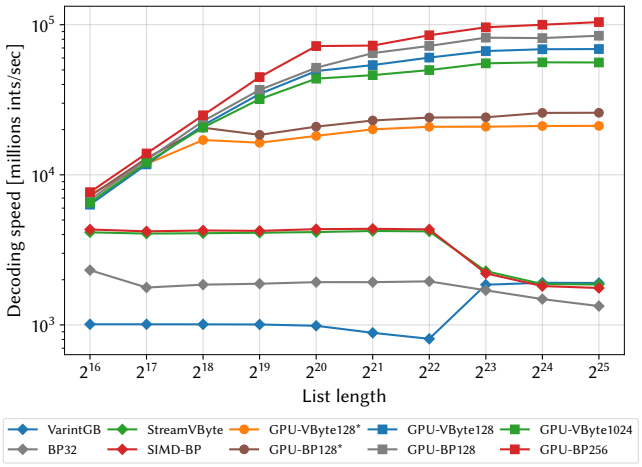

Figure 3: Decompression speed (in millions of integers per second) for lists of different lengths on Uniform dataset. (Results for Clustered dataset are nearly identical.)

on synthetic datasets are reported in Table 3. At its peak, GPU-BP is able to decode more than 100 billion integers per second, reaching an impressive speedup factor of 78x versus BP32. Speedup factors for Gov2 and ClueWeb09, reaching up to 58x, are included in Table 4.

Method	Gov2		ClueWeb09	
	docids	freqs	docids	freqs
BP32	1 665	1 522	1 476	1 396
SIMD-BP	2 257 (x1.36)	2 151 (x1.41)	2 029 (x1.38)	1 989 (x1.43)
GPU-BP128*	22 773 (x13.68)	22 339 (x14.68)	23 622 (x16.00)	23 277 (x16.67)
GPU-BP128	56 017 (x33.64)	54 790 (x36.00)	65 605 (x44.45)	64 586 (x46.27)
GPU-BP256	70 552 (x42.37)	67 802 (x44.55)	84 293 (x57.11)	81 527 (x58.40)
VarintGB	2 073	2 039	1 517	1 544
StreamVByte	2 204 (x1.06)	2 215 (x1.09)	1 545 (x1.02)	1 544 (x1.00)
GPU-VByte128*	17 807 (x8.59)	17 887 (x8.77)	18 724 (x12.34)	18 702 (x12.11)
GPU-VByte128	49 555 (x23.91)	48 518 (x23.80)	57 496 (x37.90)	56 311 (x36.47)
GPU-VByte1024	41 598 (x20.07)	40 717 (x19.97)	46 385 (x30.58)	46 199 (x29.92)

Table 4: Decoding speed (in millions of integers per second) achieved by the encoders on Gov2 and ClueWeb09 datasets.

5 CONCLUSIONS

In this paper, we described GPU-BP and GPU-VByte, two encoding techniques based on Binary Packing and Variable Byte that are adapted to the high degree of parallelism available in GPU architectures. Our experimental evaluation showed that our methods consistently outperform their CPU/SIMD analogs. Our future work will focus on using the new encoding methods in conjunction with fast query processing algorithm properly adapted to GPUs.

Acknowledgements. This research was supported by NSF Grant IIS-1718680 and a grant from Amazon.

REFERENCES

[1] Vo Ngoc Anh and Alistair Moffat. 2010. Index Compression Using 64-bit Words. *Softw. Pract. Exper.* 40 (2010), 131–147.
[2] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*. 1–1.
[3] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. 2008. Using Graphics Processors for High-performance IR Query Processing. In *WWW*. 1213–1214.
[4] Roussian R. A. Gaioso, Veronica Gil Costa, Hélio Guardia, and Hermes Senger. 2018. A Parallel Implementation of WAND on GPUs. In *PDP*. 10–17.
[5] Haibing Huang, Mingming Ren, Yue Zhao, Rebecca J. Stones, Rui Zhang, Gang Wang, and Xiaoguang Liu. 2017. GPU-Accelerated Block-Max Query Processing. In *ICA3PP*. 225–238.
[6] D. Lemire and L. Boytsov. 2015. Decoding Billions of Integers Per Second Through Vectorization. *Softw. Pract. Exper.* 45 (2015), 1–29.
[7] Daniel Lemire, Nathan Kurz, and Christoph Rupp. 2018. Stream VByte: Faster byte-oriented integer compression. *Inf. Process. Lett.* 130 (2018), 1–6.
[8] Yang Liu, Jianguo Wang, and Steven Swanson. 2018. Griffin: Uniting CPU and GPU in Information Retrieval Systems for Intra-query Parallelism. In *PPoPP*. 327–337.
[9] Antonio Mallia, Michał Siedlaczek, and Torsten Suel. 2019. An experimental study of index compression and DAAT query processing methods. In *ECIR*. 353–368.
[10] John Nickolls and William J. Dally. 2010. The GPU Computing Era. *IEEE Micro* 30 (2010), 56–69.
[11] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. 2015. Vectorized VByte Decoding. *CoRR* abs/1503.07387 (2015).
[12] Fabrizio Silvestri. 2007. Sorting out the Document Identifier Assignment Problem. In *ECIR*. 101–112.
[13] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. 2011. SIMD-based Decoding of Posting Lists. In *CIKM*. 317–326.
[14] Andrew Trotman and Kat Lilly. 2018. Elias Revisited: Group Elias SIMD Coding. In *ADCS*. 4:1–4:8.
[15] Andrew Trotman and Jimmy Lin. 2016. In Vacuo and In Situ Evaluation of SIMD Codecs. In *ADCS*. 1–8.
[16] Di Wu, Fan Zhang, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. 2010. Efficient lists intersection by CPU-GPU cooperative computing. In *IPDPS*. 1–8.