Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 9, Number 1, pages 91–93, January 2020

A New Huffman Code for Accelerating GPU Decompression (preliminary version)*

Naoya Yamamoto, Koji Nakano, Yasuaki Ito, Daisuke Takafuji
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

Akihiko Kasagi, Tsuguchika Tabaru
Fujitsu Laboratories Ltd.
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa, 211-8588, Japan

***Abstract*—A Huffman code is a variable-length entropy code used in data compression. The compression operation of a Huffman code can be done very efficiently on the GPU. However, the decompression is very hard to parallelize, because the compressed data have no separator to identify each variable-length codeword. We present a Huffman Code with Gap Array (GHCA), which accelerates the GPU decompression. The experimental results using Geforce RTX2080Ti GPU shows that, the GHCA may have at most 1.5% size overhead, but the decompression is 1.536-4713 times faster than the previously published GPU decoding for Huffman codes.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

A Huffman code [1] is a prefix-free entropy code that encodes a sequence $X = x_0x_1 \cdots x_{n-1}$ of source symbols into a sequence $h(X) = h(x_0)h(x_1) \cdots h(x_{n-1})$ of codewords such that the number of bits of $h(X)$ is minimized, where $h(x)$ denotes the codeword of symbol x. Figure 1 shows an example of a Huffman code. For example, a sequence $X = DDABAB$ of symbols is *encoded* into $Y = 11011000010001$. Also, Y is *decoded* into X. A Huffman code is represented by a binary tree such edges to the left and right children from an internal node have labels 0 and 1, respectively. Each leaf corresponds to a symbol and the codeword can be obtained by picking labels in the path from the root to it. Thus, a Huffman code is *a prefix code* such that no codeword is a prefix of the other codeword, and each codeword can be uniquely identified by reading a codeword sequence from the beginning. A Huffman code is one of the most frequently used compression scheme used in many compressed file formats such as gzip, zip, png, and jpeg. So, it is important to accelerate the encoding and decoding tasks of Huffman codes.

Parallel encoding of Huffman codes are very easy [2]. By computing the prefix sums of the number of bits of codewords corresponding to symbols in an input sequence, we can determine the bit positions of codewords in the encoded sequence. After that, the codeword of each symbol is written in the corresponding position in parallel. Since the prefix-sums can be computed very efficiently in the GPU [3]–[5], Huffman code encoding can be done very efficiently. On the other hand, parallel decoding of Huffman code is very hard because a codeword sequence has no separator to identify each codeword. Each codeword can be identified only by reading codeword from the beginning. Quite recently, Weissenberger and Schmidt [6], [7] showed an interesting GPU implementation of decoding of Huffman codes called *GPUHD*. Their idea is to use *self-synchronization* property [8] of Huffman codes. More specifically, suppose that we start decoding from some intermediate bit position of a codeword sequence. It may be possible that the bit position is not the beginning of a codeword, and the decoded symbol is not correct. However, we may have a correct decoded symbol after several wrong decoded symbols are output. Once we have a correct decoded symbol, the decoded results after it are correct. Using self-synchronization of Huffman codes, decoding of Huffman codes can be done in parallel [6]. More specifically, a codeword sequence is partitioned into equal-sized segments, and a thread is assigned to each segment. A segment starts to decode segments until it finds self-synchronization. In this GPU implementation, each segment is decoded at least twice. Also, in the worst case, the first thread cannot find self-synchronization, and it must work for decoding all segments.

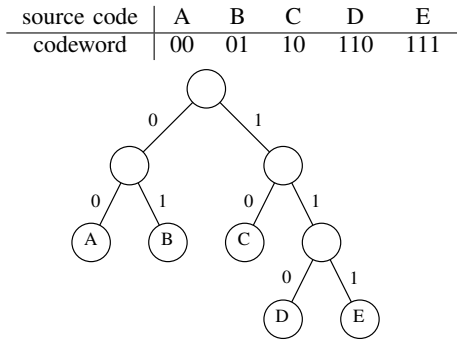

Fig. 1. An example of a Huffman code and the corresponding binary tree

*This article is a preliminary work for presenting at a non peer-reviewed workshop, the 10th International Workshop on Networking, Computing, Systems, and Software (NCSS), held in Nagasaki, Japan, November 2019. The full version will be submitted to a peer reviewed conference and/or journal.

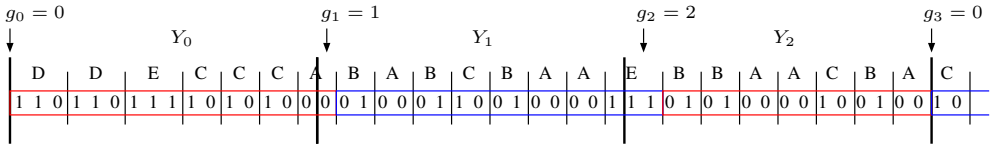

Fig. 2. Gaps of segments with $b = 16$ bits each

The main contribution of this article presents a new Huffman code for accelerating GPU decompression. Our idea is to add a gap array to a codeword sequence during the process of decoding. Suppose that a codeword sequence to be decoded is partitioned into equal-sized segments. A codeword may be separated into two adjacent segment. We call such codeword *an incomplete codeword*. *A gap* of a segment is the number of bits of the first incomplete word. A gap array is an array storing the gaps of all segments. Figure 2 illustrates the gaps of segments. The gap array can be computed very easily in parallel during the encoding process. In most Huffman code implementations, the number of codewords is limited to 16, we can assume that a gap is an 4-bit integer. Thus, the size of an gap array is very small if we use enough large segments.

II. Huffman code with gap array (HCGA)

The main purpose of this section is to present a Huffman code with a gap array. As we will show later, decoding of Huffman code is hard to parallelize. By means of self-synchronization of Huffman codes, decoding can be parallelized. However, for malicious codewords, it redundant decoding operations are performed and the decoding time is much larger than the sequential decoding by a single CPU. In this section, we will how a gap array which is attached to a sequence of encoded codewords of a Huffman code.

Let $X = x_0x_1\cdots x_{n-1}$ be a sequence of n symbols and $Y = y_0y_1\cdots y_{m-1}$ be a sequence of m bits obtained by encoding X by an Huffman code h. In other words, $Y = h(x_0)h(x_1)\cdots h(x_{n-1})$ holds. Suppose that a sequence of codewords Y is partitioned into equal-sized segments of b bits each. We introduce a new data structure that we call *gap array* G. The gap array G is attached to the encoded codewords Y to accelerate decoding of Y into X. The gap array is an array of gaps of segments. A segment may have an incomplete codeword, because there is a crossing codeword between it and the previous segment. *The gap* of a segment is the number of bits in the incomplete codeword. If it has no incomplete codeword then the gap is zero. Let g_i denote the gap of segment Y_i. Figure 2 illustrates gaps of segments of 16 bits each. We assume that a crossing codeword of neighboring segments belongs to the latter one. Thus, a segment has complete codewords and the following crossing codeword (if exists). For example, the symbols of codewords in Y_1 in the figure are BABCBAAE.

A. Decoding of a Huffman code with a gap array

We use the Single Kernel Soft Synchronization (SKSS) technique [9], [10] to accelerate decoding on the GPU. We

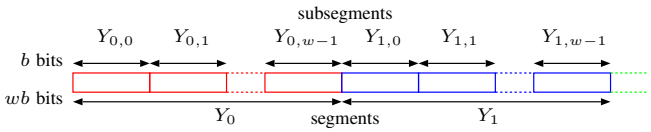

Fig. 3. Segments and subsegments for decoding

partition Y into $\frac{m}{wb}$ segments $Y_0, Y_1, \ldots, Y_{L-1}$ of $L = wb$ bits each as illustrated in Figure 3, where w (≥ 32) and b be parameters to be determined later. Each segment Y_i is further partitioned into w subsegments $Y_{i,0}, Y_{i,1}, \ldots, Y_{i,w-1}$ of b bits each, We use a CUDA block with w threads is assigned to a segment to decode it. Also, each thread of the CUDA block is assigned to a subsegment and works for decoding it. Let $c(i)$ and $c(i,j)$ denote the number of codewords in segment Y_i and subsegment $Y_{i,j}$, respectively. Further, let $p(i) = c(0)+c(1)+\cdots+c(i)$ and $p(i,j) = c(i,0)+c(i,1)+\cdots+c(i,j)$ be the prefix-sums. Clearly, codewords in segment Y_i are decoded in $x[p(i-1)], x[p(i-1)+1], \ldots, x[p(i-1)+c(i)-1]$ and those in segment $S_{i,j}$ must be coded in $x[p(i-1)+p(i,j-1)], x[p(i-1)+p(i,j-1)+1], \ldots, x[p(i-1)+p(i,j-1)+c(i,j)-1]$. Thus, if all values of c and p are computed, we can determine positions of x for segment/subsegments where decoded symbols are written.

The decoding can be done in only one kernel call, which invokes multiple CUDA blocks. We use zero-initialized global counter k in the global memory to assign serial numbers to invoked CUDA block. More specifically, the first thread of a CUDA block performs atomicAdd(&k,1), which adds 1 in k as an atomic operation and return the value of k before addition. The first threads of CUDA blocks receive 0, 1, 2, ..., and we call a CUDA block with the first thread receiving i *CUDA block* i. Hence, it is guaranteed that, when CUDA block i starts running, CUDA blocks 0, 1, ..., $i-1$ have been invoked. Note that it is not guaranteed that CUDA blocks are invoked in the order of blockID given by kernel. In our GPU Huffman decoding, it must be guaranteed to avoid deadlocks.

Each CUDA block i works for decoding segment Y_i in 5 steps. In Steps 1 to 4, each CUDA block i computes the values of $c(i)$, $p(i)$, $p(i-1)$, $c(i,j)$ and $p(i,j)$, In particular, it writes the values of $c(i)$ and $p(i)$ in the global memory, because the other CUDA blocks may read them. In Step 5, each CUDA block i decodes codewords in section Y_i and write the decoded symbols in $x[p(i-1) : p(i-1)+c(i)-1]$. The details of operations performed by each CUDA block i in Steps 1 to 5 are spelled out as follows:

TABLE I
THE PERFORMANCE OF ORIGINAL HUFFMAN CODE BY GPUHD AND HCGA BY OUR IMPLEMENTATION

File	type	description	Size Mbytes	Compression ratio GPUHD	HCGA	overhead	Decompression time GPUHD	HCGA	Speed-up
bible	text	Collection of sacred texts or scriptures	4.047	54.82	55.67	0.86%	0.348ms	0.080ms	4.357
enwiki	xml	Wikipedia dump file	1095	68.30	69.37	1.07%	46.530ms	10.779ms	4.317
mozilla	exe	Tarred executables of Mozilla	51.22	78.05	79.27	1.22%	3.356ms	0.689ms	4.871
mr	image	Medical magnetic resonance image	9.97	46.37	47.10	0.72%	0.592ms	0.336ms	1.764
nci	database	Chemical database of structures	33.55	30.47	30.95	0.48%	1.487ms	0.968ms	1.536
prime	text	50th Mersenne number	23.71	44.12	44.81	0.69%	1.619ms	0.357ms	4.528
sao	bin	The SAO star catalog	7.251	94.37	95.85	1.47%	0.577ms	0.107ms	5.389
webster	html	The 1913 Webster Unabridged Dictionary	41.45	62.54	63.52	0.98%	2.093ms	0.467ms	4.482
linux	src	Linux kernel 5.2.4	871.3	70.23	71.32	1.10%	41.922ms	9.164ms	4.575
malicious	text	Never self-synchronizes until the end	1073	25.00	25.39	0.39%	92797ms	19.689ms	4713

Step 1 Each thread j ($0 \leq j \leq wb-1$) executes the sequential decoding algorithm to decode all complete codewords and the following codeword of $S_{i,j}$ to compute $c(i,j)$. Since the gap value of $S_{i,j}$ is available this can be done by simply executing the sequential decoding algorithm.

Step 2 CUDA block i computes $c(i) = c(i,0)+c(i,1)+\cdots+c(i,w-1)$. This can be done by the prefix-scan algorithm in a CUDA block [3] and thread 0 writes $c(i)$ in the global memory.

Step 3 The first warp with 32 threads looks back previous 32 CUDA blocks i' ($i-32 \leq i' \leq i-1$) if they have written in $c(i')$ and/or $p(i')$ in the global memory. More specifically, they computes the maximum i' such that

- $p(i')$ has been written by CUDA block i', and
- $c(i'+1), c(i'+2), \ldots, c(i-1)$ have been written by CUDA blocks $i'+1, i'+2, \ldots, i-1$, respectively.

They repeat the same procedure until they find i' satisfying these conditions.

Step 4 CUDA block i computes $p(i-1) = p(i')+c(i'+1)+c(i'+2)+\cdots+c(i-1)$ and $p(i) = p(i-1)+c(i)$, and and writes $p(i)$ in the global memory.

Step 5 Each thread j of each CUDA block i executes the sequential decoding algorithm for codewords in subsegment $Y_{i,j}$ and decoded symbols are written in $x[p(i,j-1) : p(i,j-1)+c(i,j-1)-1]$.

III. EXPERIMENTAL RESULTS

This section shows experimental results of Huffman decoding using Geforce RTX2080Ti GPU. We have evaluated the performance of our HCGA and GPUHD [6], which decodes original Huffman code. We have used various 10 data in Table I. The size of a subsegment is $b = 256$ bits. The file malicious is an intentionally generated compressed data that never self-synchronize until the end. From the table, we can see that the size overhead is less than 1.5%. Also, the running time is of HCGA 1.536-5.389 times faster than GPUHD for the first 9 data. The running time of GPUHD is quite large because the first thread must work for decoding all codewords. On the other hand HGA is quite fast and the speedup is 4713.

IV. CONCLUSION

We have presented a Huffman Code with Gap Array (GHCA), which accelerates the GPU decompression. The experimental results using Geforce RTX2080Ti GPU shows that, the GHCA may have at most 1.5% size overhead, but the decompression is 1.536-4713 times faster than the previously published GPU decoding for Huffman codes.

REFERENCES

[1] D. A. Huffman, "A method for the construction of minimum-redundancy codes," in *Proc. of the IRE*, vol. 40, no. 9, Sep. 1952, pp. 1098 – 1101.

[2] H. Rahmani, C. Topal, and C. Akinlar, "A parallel Huffman coder on the CUDA architecture," in *Proc. of IEEE Visual Communications and Image Processing Conference*, Dec. 2014.

[3] M. Harris, S. Sengupta, and J. D. Owens, "Chapter 39. parallel prefix sum (scan) with CUDA," in *GPU Gems 3*. Addison-Wesley, 2007.

[4] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled look-back," NVIDIA, Tech. Rep. NVR-2016-002, March 2016.

[5] D. Merrill, "CUB : A library of warp-wide, block-wide, and device-wide GPU parallel primitives," https://nvlabs.github.io/cub/, 2017.

[6] A. Weissenberger and B. Schmidt, "Massively parallel Huffman decoding on GPUs," in *Proc. of International Conference on Parallel Processing*, Aug. 2018.

[7] A. Weissenberger, "CUHD - a massively parallel Huffman decoder," https://github.com/weissenberger/gpuhd.

[8] T. Ferguson and J. Rabinowitz, "Self-synchronizing Huffman codes (corresp.)," *IEEE Trans. on Information Theory*, vol. 30, no. 4, pp. 687 – 693, Jul. 1984.

[9] S. Funasaka, K. Nakano, and Y. Ito, "Single kernel soft synchronization technique for task arrays on CUDA-enabled GPUs, with applications," in *Proc. International Symposium on Networking and Computing*, Nov. 2017, pp. pp.11–20.

[10] Y. Emoto, S. Funasaka, H. Tokura, T. Honda, K. Nakano, and Y. Ito, "An optimal parallel algorithm for computing the summed area table on the GPU," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, Feb. 2018, pp. 763–772.