The Journal of Supercomputing (2023) 79:8447–8474
https://doi.org/10.1007/s11227-022-04994-3

GVLE: a highly optimized GPU-based implementation of variable-length encoding

Antonio Fuentes-Alventosa[1] · Juan Gómez-Luna[2] · R. Medina-Carnicer[1]

Accepted: 3 December 2022 / Published online: 18 December 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract
Nowadays, the massive use of multimedia data gives to data compression a fundamental role in reducing the storage requirements and communication bandwidth. Variable-length encoding (VLE) is a relevant data compression method that reduces input data size by assigning shorter codewords to mostly used symbols, and longer codewords to rarely utilized symbols. As it is a common strategy in many compression algorithms, such as the popular Huffman coding, speeding VLE up is essential to accelerate them. For this reason, during the last decade and a half, efficient VLE implementations have been presented in the area of General Purpose Graphics Processing Units (GPGPU). The main performance issues of the state-of-the-art GPU-based implementations of VLE are the following. First, the way in which the codeword look-up table is stored in shared memory is not optimized to reduce the bank conflicts. Second, input/output data are read/written through inefficient strided global memory accesses. Third, the way in which the thread-codes are built is not optimized to reduce the number of executed instructions. Our goal in this work is to significantly speed up the state-of-the-art implementations of VLE by solving their performance issues. To this end, we propose GVLE, a highly optimized implementation of VLE on GPU, which uses the following optimization strategies. First, the caching of the codeword look-up table is done in a way that minimizes the bank conflicts. Second, input data are read by using vectorized loads to exploit fully the available global memory bandwidth. Third, each thread encoding is performed efficiently in the register space with high instruction-level parallelism and lower number of executed instructions. Fourth, a novel inter-block scan method, which outperforms those of state-of-the-art solutions, is used to calculate the bit-positions of the thread-blocks encodings in the output bit-stream. Our proposed mechanism is based on a regular segmented scan performed efficiently on sequences of bit-lengths of 32 consecutive thread-blocks encodings by using global atomic additions. Fifth, output data are written efficiently by executing coalesced global memory stores. An exhaustive experimental evaluation shows that our solution is on average 2.6× faster than the best state-of-the-art implementation. Additionally, it shows that the scan

---

Extended author information available on the last page of the article

algorithm is on average 1.62× faster if it utilizes our inter-block scan method instead of that of the best state-of-the-art VLE solution. Hence, our inter-block scan method offers promising possibilities to accelerate algorithms that require it, such as the scan itself or the stream compaction.

Keywords Data compression · Variable-length encoding · Huffman coding · GPU · CUDA

1 Introduction

In the current digital era, huge amounts of multimedia data, such as images and videos, are generated continuously [1]. For example, at the time of writing this paper, 720,000 hours of video are uploaded to YouTube by day [2]. Since the rate of growth of data is much higher than the rate of growth of technologies (e.g., DVDs, Blu-ray, ADSL, optical fibers, etc.), data compression has nowadays an essential role in reducing the cost of data storage and transmission [1].

Variable-length encoding (VLE) is a popular data compression method in which most frequently occurring symbols are replaced by codewords of shorter length, whereas rarely used symbols are substituted by codewords of longer length [3]. Since VLE is a common strategy in many compression algorithms [3, 4], such as the widely used Huffman coding [5, 6], acceleration of VLE is key to speed them up. In order to achieve this goal, during the last decade and a half, efficient implementations of VLE have been proposed in the area of General Purpose Graphics Processing Units (GPGPU) [7–14], which is, nowadays, mainstream high-performance computing [15–17].

The first GPGPU VLE solution is the algorithm PAVLE, proposed by Balevic [7]. This method uses an encoding alphabet of up to 256 symbols, with each symbol representing one byte. Without loss of generality, it assumes that the values and bit-lengths of the codewords are stored in a look-up table, which is cached in the on-chip shared memory. This table will be referred to as *VLET* in the rest of the paper. As GPU architectures provide more efficient support for 32-bit data types, the source and compressed data are provided and written, respectively, in two vectors of 32-bit unsigned integers. Consecutive threads load consecutive segments of elements from the source vector. Each thread uses the VLET for encoding the loaded segment in its private memory and calculating the corresponding bit-length. An intra-block scan primitive [18] is performed to calculate the bit-positions of the thread encodings in the corresponding thread-block encoding on the basis of their bit-lengths. The threads of a thread-block write concurrently their encodings in a buffer in shared memory using atomic operations to deal with the race conditions that occur when parts of adjacent encodings are written to the same memory location. Once the writing is finished, the content of the buffer is copied to the output vector at the same position of the corresponding source segment in the input vector. After the encoding is finished, a second kernel is launched to compact the output vector. Experimental evaluation showed speedups with respect to the serial implementation on a

2.66 GHz Intel QuadCore CPU of up to 35x. Fuentes-Alventosa et al. [8] presented CUVLE, a new implementation of VLE on CUDA. As in the case of PAVLE, the VLET is cached in shared memory, and consecutive threads process consecutive source segments. However, their approach uses the following optimization strategies. First, persistent blocks [19], which equals the grid size to the maximum number of resident thread-blocks, thereby minimizing the number of VLET loads in shared memory. Second, contiguous writing of thread-block encodings in global memory, which avoids the necessity of running any compaction extra kernel. The bit-positions of the thread encodings in the output vector are calculated by combining the efficient intra-block scan algorithm of Sengupta et al. [20] with the adjacent thread-block synchronization mechanism proposed by Yan et al. [21]. Third, direct writing of thread-block encodings in global memory. Since CUVLE does not use an intermediate buffer in shared memory, it saves the time to make additional operations, avoids the appearance of bank conflicts and saves the reserved space for the buffer. Experimental evaluation showed that CUVLE is on average more than 20 and 2 times faster than the corresponding CPU serial implementation and PAVLE, respectively. The test machine had a 2.67Ghz Intel Core i7 920 CPU and 12 GB of RAM, and the GPUs utilized were a GeForce GT 640 2GB GDDR5 and a GeForce GTX 550 Ti. Rahmani et al. [9] proposed a CUDA-based Huffman coder that does not have any constraint on the maximum code bit-length by generating an intermediate byte stream where each byte represents a single bit of the compressed output stream. After the Huffman tree generation is done serially on the CPU, the encoding is performed in parallel on the GPU following the next three steps (each one implemented with a different kernel). First, the code offsets for each input symbol in the intermediate stream are calculated using the scan method presented in [18]. Second, the intermediate stream is generated by the i-th thread of the second kernel writing the code of the i-th input symbol to its corresponding memory slots in the intermediate stream. Third, the output stream is obtained by each thread of the third kernel reading 8 consecutive bytes from the intermediate stream, and generating a single byte of the output stream. As the encoding is implemented with three kernels, this solution has two main overheads: the extra long latency global memory accesses required to transmit intermediate results between kernels, and the costly launches and terminations of the kernels. Experimental evaluation on the NVIDIA GTX 480 GPU showed speedups with respect to the CPU serial implementation of up to 22x on an Intel Core 2 Quad CPU running at 2.40 GHz. The work of Yamamoto et al. [10] focused on GPU acceleration of Huffman encoding and decoding and was developed in CUDA. As in the case of CUVLE, the VLE stage is implemented with only one kernel. The authors exposed that their kernel is similar to CUVLE, but it is much more faster because, instead of using Yan et al.'s mechanism [21], it utilizes a novel adjacent thread-block synchronization method, which is much more efficient. The reason is that, in the Yan et al.'s algorithm [21], each thread-block looks back the result written in global memory by only one thread-block, while, in the Yamamoto et al.'s approach [10], each thread-block looks back 32 previous results simultaneously. Experimental evaluation for ten files on NVIDIA Tesla V100 GPU showed that Yamamoto et al.'s VLE implementation is between 2.87 and 7.70 times

faster than CUVLE. For this reason, the best state-of-the-art implementation of VLE on GPU is the solution of Yamamoto et al.

The main performance issues of the state-of-the-art GPU-based implementations of VLE [8, 10] are the following. First, the way in which the VLET is cached is not optimized to reduce the shared memory bank conflicts. Second, each thread reads/writes the elements of its input/output segment one by one, which results in inefficient strided global memory accesses. Third, the way in which the thread-codes are built is not optimized to reduce the number of executed instructions. In order to solve these issues, we propose GVLE, a highly optimized implementation of VLE on GPU, which significantly speeds up the solution of Yamamoto et al. As in previous approaches [7, 8, 10], the VLET is cached in shared memory, and consecutive threads process consecutive segments of the input vector. However, GVLE uses the following optimization strategies. First, the VLET storage in shared memory is done in a way that minimizes the bank conflicts. Second, the input segments are read by using vectorized loads to exploit fully the available global memory bandwidth [22]. Third, each thread, after reading its assigned segment, encodes it efficiently in the register space with high instruction-level parallelism and lower number of executed instructions. Fourth, a novel inter-block scan method, which outperforms those of Yan et al. [21] and Yamamoto et al. [10], is used to calculate the bit-positions of the thread-blocks encodings in the output vector. Our proposed mechanism is based on a regular segmented scan performed efficiently on sequences of bit-lengths of 32 consecutive thread-blocks encodings by using global atomic additions [23]. Fifth, the thread-block encodings are written efficiently to the output vector by executing coalesced global memory stores [24].

Our main contributions in this work are the following:

- A highly optimized GPU-based approach to VLE, called GVLE,[1] that significantly improves the state-of-the-art implementations [8, 10].
- A novel inter-block scan method for calculating the bit-positions of thread-blocks encodings that outperforms those used in [8] and [10].
- A comparison of our solution with the best state-of-the-art implementation [10]. An exhaustive experimental evaluation shows that our proposal is on average 2.6× faster than the method presented in [10].
- A comparison of our inter-block scan method with that of [10]. The experimental results show that the speedup of the scan operation using our inter-block scan algorithm is on average 1.62× with respect to using the method of [10].

The rest of the paper is organized as follows. Section 2 gives background for CUDA, VLE and the Yamamoto et al.'s implementation of VLE [10]. Section 3 presents GVLE and compares our method with the one proposed in [10], so that the achieved performance improvement can be clearly established. Section 4 shows the experimental evaluation of our algorithm and a comparison to the method of Yamamoto

[1] The source code is available at https://github.com/z12fuala/GVLE.

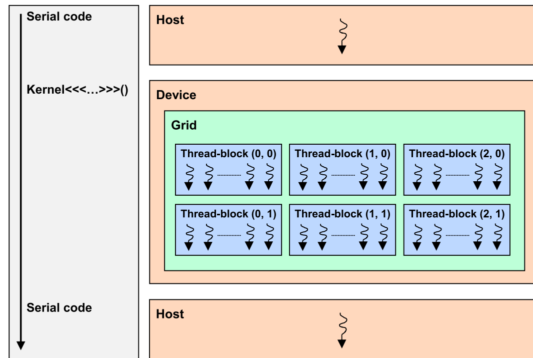

Fig. 1 Execution of a CUDA program

et al. [10], CUVLE [8] and the serial implementation of VLE. Section 5 reviews related work. Finally, the main conclusions are stated in Section 6.

2 Background

This section is structured in the following way. Section 2.1 gives a brief overview of CUDA, and cites several relevant documents that can provide further background to readers. Section 2.2 defines VLE, and highlights its important role in data compression. Section 2.3 gives a detailed description and a critical analysis of the best state-of-the-art implementation of VLE on GPU ([10]).

2.1 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing framework developed by NVIDIA for GPGPU computing [16]. Since its release in 2007, thousands of applications have been developed on CUDA [15, 16], so it is one of the main responsible technologies for the GPGPU computing revolution.

CUDA greatly facilitates to developers the implementation of parallel algorithms by providing a small set of extensions to popular languages such as C, C++, Fortran, Python and MATLAB [16]. In this work, we have utilized CUDA C++.

As shown in Fig. 1 [23], in a CUDA program, the sequential parts runs on the CPU (usually referred to as the *host*), while the compute intensive parts are executed by thousands of threads on the GPU (commonly named as the *device*). The functions executed on the GPU are called *kernels*, which are defined by the programmer using the *__global__* declaration specifier. The number of threads that execute a kernel is specified using the <<<...>>> *execution configuration* syntax. They are organized into one-, two- or three-dimensional blocks of threads, which are called *thread-blocks*. A kernel is executed by a set of identical thread-blocks, called *grid*, which can also have up to three dimensions.

The architecture of a CUDA GPU is composed of a set of *streaming multiprocessors* (*SMs*) [23]. When a kernel is launched, the thread-blocks of the grid are

distributed to the available multiprocessors with execution capacity. The threads of each thread-block run concurrently on a single multiprocessor, and each multiprocessor can concurrently execute many thread-blocks. As the execution of the thread-blocks finishes, new ones are launched in the available multiprocessors. The SMs execute the threads in groups of 32 called *warps*. The threads of a warp start at the same program address, but each one has its own instruction address counter and register state, and, hence, they are free to branch and execute independently. Although CUDA developers can ignore this behavior for the correctness of their applications, they can greatly improve their performance by minimizing the warp divergence.

CUDA threads can access three types of memory spaces during their execution [23]:

- Each thread has private memory consisting of *registers* and *local memory*. Its lifetime is that of the thread.
- Each thread-block has *shared memory* visible to all threads in it. Its lifetime is that of the thread-block. The *__shared__* qualifier is used for the declaration of variables in shared memory.
- All threads of a grid have access to a read/write *global memory*, and two other read memories: the *constant memory*, used to store non-modifiable values, and the *texture memory*, optimized for accesses with 2D spatial locality. The contents of these memories are persistent between the different kernel calls of the same application.

Global memory is the most abundant of these memory spaces [24]. On the other hand, global, local, and texture memory have the highest access latency, followed by constant memory, shared memory, and registers [24]. A very important optimization technique is the coalescing of global memory accesses [23, 24]. When a warp performs an operation on global memory, the memory accesses of its threads are coalesced into one or more memory transactions according to the size of the accessed words and the distribution of the memory addresses. The more scattered the accesses are, the more transactions are necessary, and, hence, the more reduced the throughput is.

2.2 Variable-length encoding (VLE)

Input data to a compression algorithm can be modeled as a sequence of elements, called *symbols*, belonging to an alphabet [4]. A symbol can be an ASCII character, a byte, an audio sample, etc. Given an alphabet $S = \{s_0, s_1, ..., s_{n-1}\}$, its digital representation is called the *code* $C = \{c_0, c_1, ..., c_{n-1}\}$, and the representation c_i of each symbol is called the *codeword* for symbol s_i. Codes are classified into *fixed length codes* (*FLC)* and *variable-length codes* (*VLC*), depending on the length of their codewords is fixed or variable, respectively. The process of assigning codewords to symbols of input data is called *encoding*, and the reverse process is called *decoding*.

Variable-length encoding (*VLE*) [3] is a compression method in which input data size is reduced by using a VLC that assigns shorter codewords to mostly used

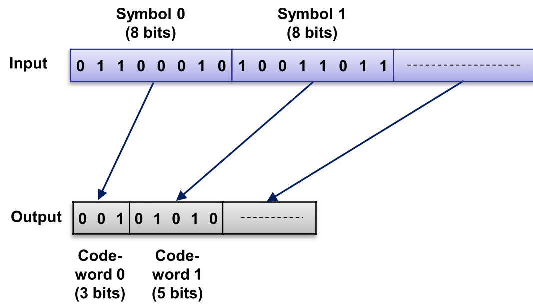

Fig. 2 Variable-Length Encoding (VLE). Input data size is reduced by assigning shorter codewords to mostly used symbols, and longer codewords to rarely utilized symbols

symbols, and longer codewords to rarely utilized symbols. Figure 2 illustrates VLE. For example, consider the alphabet $S = \{A, B, C, D, E\}$ and the 9-symbols input data string *BAAAAAAAC* [4]. On the one hand, if the encoding is performed using a 3-bit FLC, the bit-length of the result is $9 \times 3 = 27$. On the other hand, if the encoding is performed using the VLC $C = \{0, 100, 101, 110, 111\}$, the bit-length of the result is $1 \times 3 + 7 \times 1 + 1 \times 3 = 13$, which is less than half that obtained with the FLC.

VLE is one of the main building blocks in many compression algorithms [3, 4], such as the popular Huffman coding [5, 6], which is the most relevant entropy coding method at present [25]. Huffman coding is a component of the Deflate algorithm, which is used in the file compression programs ZIP, 7ZIP, GZIP, and PKZIP, and in the image compression format PNG, for example [26]. Additionally, Huffman coding is the most used entropy coding algorithm in multimedia encoding standards such as JPEG, MPEG, H.264 and VC-1 [27], and is a critical step in an increasing number of high-performance computing applications [12, 28, 29]. Since VLE is an essential step in so many important present and future compression algorithms, its acceleration is fundamental to speed them up.

2.3 Solution of Yamamoto et al

The best state-of-the-art implementation of VLE on GPU is the solution presented by Yamamoto et al. [10], which was developed in CUDA. It is composed of only one kernel, which will be referred to as *YAVLE* in the rest of the paper. In this section, we give a detailed description of YAVLE based on the paper of Yamamoto et al. [10], and the source code of their solution published on GitHub at github.com/daisuke-takafuji/Huffman_coding_Gap_arrays.

YAVLE operates on 8-bit symbols and, therefore, it utilizes an alphabet of up to 256 symbols. The VLET is provided in a vector (*d_VLET*) of 256 elements, whose base type is a structure (*Codeword*) with two 32-bit unsigned int members that represent the value and the bit-length of a codeword. Yamamoto et al. assume that the maximum bit-length of codewords is 16 because Huffman coding with this limited maximum codeword can be generated efficiently [10, 30, 31]. In fact, actually, the maximum codeword length is limited in the most implementations of Huffman coding [10].

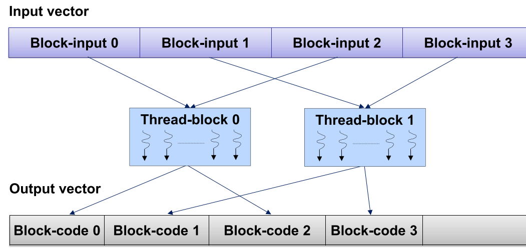

Fig. 3 Inter-block mechanism for an input vector of 4 block-inputs and a grid of 2 thread-blocks. The thread-block 0 processes the block-inputs 0 and 2, and writes the corresponding block-codes 0 and 2 in the output vector. The thread-block 1 performs the same actions with the block-inputs 1 and 3

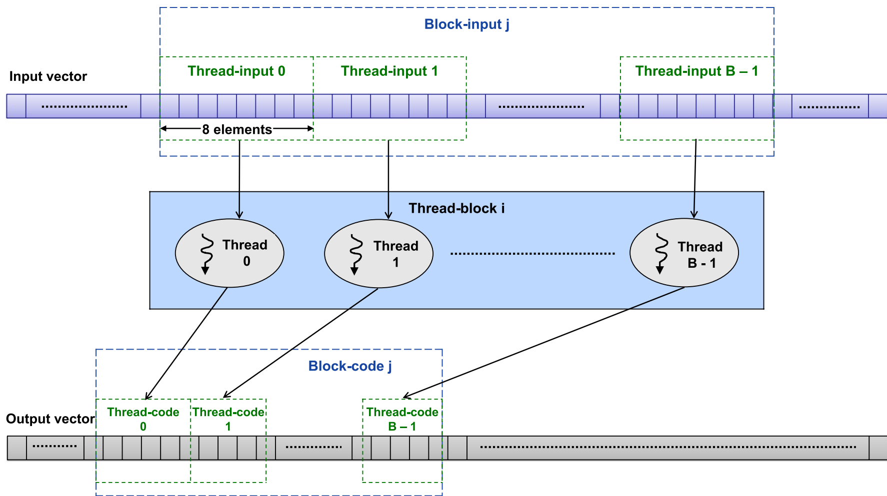

Fig. 4 Intra-block mechanism. The thread-block i (of size B) processes the block-input j and writes the corresponding block-code j in the output vector. Each thread k of the thread-block i encodes the thread-input k and writes the thread-code k in the output vector

The source data are supplied in a vector of 32-bit unsigned integers (*d_input*). Hence, each element contains 4 symbols. The compressed data are written in an output vector (*d_output*) of 32-bit unsigned integers too. Let B be the thread-block size. The vector *d_input* is partitioned into segments of size $B \times 8$, which will be named as *block-inputs*. Each block-input is processed by a thread-block and the encoding result, which will be referred to as *block-code*, is written in *d_output*. Figure 3 illustrates this inter-block mechanism. Consecutive threads of each thread-block process consecutive segments of eight elements (i.e., 32 symbols) of the corresponding block-input, which will be named as *thread-inputs*. The encoding of a thread-input will be referred to as *thread-code*. Figure 4 clarifies this intra-block mechanism.

Algorithm 1 provides a high-level description of YAVLE. Each thread-block caches the VLET in shared memory (Step 1) and encodes a different subset of block-inputs (Steps 2 to 7). The number of thread-blocks of the grid is set to

the maximum number of resident thread-blocks. Hence, the number of VLET loads in shared memory is minimized. The indexes of the block-inputs are obtained from a zero-initialized global counter (Steps 2 and 7). The function *get_global_counter_value* uses the CUDA function *atomicAdd* [23] to return the successive values of the global counter, that is, 0, 1, 2, and so on. This mechanism ensures that, when a thread-block starts the processing of a block-input i, the management of block-inputs $0, 1, ..., i - 1$ have already begun. The processing of each block-input consists of Steps 3 to 6.

Algorithm 1: YAVLE algorithm

```
1  kernel YAVLE(// output
2               uint *d_output,
                // input
3               uint *d_input,
4               Codeword d_VLET[256],
5               ull *d_scan
6               )
     // Step 1: cache VLET in shared memory
7    Codeword s_VLET[256] ← cache_VLET(d_VLET)

     // Step 2: get index of first block-input to encode
8    uint blockinput_idx ← get_global_counter_value()

     // While there are block-inputs to encode...
9    while (blockinput_idx < num_blockinputs)
       // Step 3: calculate index of thread-input
10      uint thinput_idx ← blockinput_idx × blockDim.x +
11                         threadIdx.x

       // Step 4: calculate bit-length of thread-code
12     uint thcode_len ← YAVLE_calc_thcode_len(d_input,
13                                             thinput_idx,
14                                             s_VLET)

       // Step 5: calculate bit-position of thread-code
       // in output vector
15     ull thcode_pos ← YAVLE_calc_thcode_pos(thcode_len,
16                                             d_scan)

       // Step 6: write thread-code to output vector
17     YAVLE_write_thcode(d_output, thcode_pos,
18                        d_input, thinput_idx, s_VLET)

       // Step 7: get index of next block-input to encode
19     blockinput_idx ← get_global_counter_value()
20   end while
21 end kernel
```

In Sects. 2.3.1 to 2.3.4, we describe Steps 4 to 6 of Algorithm 1, respectively.

2.3.1 Calculation of bit-lengths of thread-codes

In Step 4 of Algorithm 1, each thread iterates over the 32 symbols of its thread-input to calculate the bit-length *thcode_len* (line 12) of the corresponding thread-code. For each symbol, if the element to which it belongs has not been loaded from *d_input* yet, that element is read. Then, the symbol is extracted from its element, and its codeword is obtained from the VLET. The bit-length of the thread-code is computed by accumulating the bit-lengths of its codewords.

2.3.2 Calculation of bit-positions of thread-codes in output vector

In Step 5 of Algorithm 1, the bit-position *thcode_pos* (line 15) of each thread-code in the output vector is computed by adding the bit-position of the thread-code in its block-code (*pos_of_thcode_in_bcode*) to the bit-position of the block-code in the output vector (*blockcode_pos*).

On the one hand, the intra-block scan method of Sengupta et al. [20] is performed on the parameters *thcode_len* of the current block-code to compute the corresponding bit-positions *pos_of_thcode_in_bcode* and the bit-length of the block-code (*blockcode_len*).

On the other hand, a novel inter-block scan algorithm [10], which is described in the next section, is executed on the parameters *blockcode_len* of the block-codes to compute their bit-positions *blockcode_pos*.

2.3.2.1 Yamamoto et al.'s inter-block scan method The inter-block scan method of Yamamoto et al. uses an auxiliary global vector *d_scan* (line 5) of 64-bit unsigned integers, whose initial values are zero. The number of elements of *d_scan* equals to the number of block-codes, and each element i is assigned to the block-code i. Each value written in *d_scan* has two mutually exclusive flags, named as A and P, which are located in bits 56 and 63, respectively. Given an element *d_scan*[i], if the flag A is set, then it stores the bit-length of block-code i; otherwise, if the flag P is set, it holds the sum of bit-lengths of block-codes 0 to i, which is the bit-position of the block-code $i + 1$ in the output vector.

Given a block-code i, the first warp of its assigned thread-block computes the parameter *blockcode_pos* by following the steps presented in Algorithm 2. If the block-code is the first, *blockcode_pos* is clearly zero (line 3). Otherwise, the warp iterates on the necessary 32-elements segments of *d_scan* before to *d_scan*[i] to compute *blockcode_pos* (lines 8 to 20). This parameter is obtained by the sum of the elements *d_scan*[k], *d_scan*[$k + 1$], ... *d_scan*[$i - 1$], where k is the index of the last element previous to *d_scan*[i] with the flag P activated.

Algorithm 2: Yamamoto et al.'s inter-block scan method to compute the parameter *blockcode_pos* of each block-code *i*

```
1  if i = 0 then // Block-code i is the first

      // Write the bit-position of block-code 1,
      // which equals to the bit-length of block-code 0,
      // in d_scan
2     d_scan[i] ← SET_FLAG_P(blockcode_len)

      // Since the block-code is the first, its bit-position
      // is clearly 0
3     blockcode_pos ← 0

4  else // Block-code i is not the first

      // Write the bit-length of block-code i
      // in d_scan
5     d_scan[i] ← SET_FLAG_A(blockcode_len)

      // Initialize the bit-position of block-code i
6     blockcode_pos ← 0

      // Compute the indexes of the elements of the first
      // 32-elements segment before d_scan[i] (i.e., the
      // segment composed of d_scan[i-32], d_scan[i-31],
      // ... d_scan[i-1])
7     int j ← i - 1 - warp_lane

8     while true
          // Read repeteadly the current 32-elements
          // segment until all their elements are non-zero
9         ull aux ← d_scan[j]
10        while aux = 0 and j > -1
11            aux ← d_scan[j]
12        end while

          // Perform a warp reduction to compute the sum of
          // the elements k, k + 1, ... 31 of the current
          // segment, where k is 0, if no element has the
          // flag P activated, or the index of the last
          // element with the flag P activated, otherwise
13        ull prev_32_blockcodes_sum ← warp_reduction(aux)

          // Update the bit-position of block-code i
14        blockcode_pos ← blockcode_pos +
15                        DEL_FLAG(prev_32_blockcodes_sum)

          // Exit the loop if the bit-position of the
          // block-code i has already been calculated
16        if FLAG_P_IS_SET(prev_32_blockcodes_sum) then
17            exit while
18        end if

          // Compute the indexes of the elements of the next
          // 32-elements segment before d_scan[i]
19        j ←  j - 32
20    end while

      // Write the bit-position of the next block-code in
      // d_scan
21    ull next_blockcode_pos ← blockcode_pos + blockcode_len
22    d_scan[i] ← SET_FLAG_P(next_blockcode_pos)
23 end if
```

The main difference between CUVLE [8] and YAVLE is in the method used to calculate the parameters *blockcode_pos*. Yamamoto et al.'s inter-block scan is much more efficient than Yan et al.'s technique [21] employed by CUVLE because, to obtain the bit-position of a block-code i, the first processes rapidly 32-element segments previous to $d_scan[i]$, whose values are read simultaneously by the first warp of the thread-block, while the second iterates only over one previous element ($d_scan[i-1]$). This optimization is the unique reason of the significant speedup of YAVLE with respect to CUVLE [10].

2.3.3 Writing of thread-codes to output vector

In Step 6 of Algorithm 1, each thread iterates over the 32 symbols of its thread-input in the same way as it does to calculate the bit-length of the thread-code (Step 4). Let *d_thcode* be a pointer to the first element of *d_output* that will be occupied by the thread-code. As the codewords assigned to the symbols are obtained from the VLET, their bits are concatenated in a 32-bit variable (*word_val*) and their bit-lengths added in a second 32-bit variable (*word_len*) while the bit-length of the resulting encoding is less than or equal to 32. When the last condition is not satisfied, the first 32 bits of the resulting encoding are written in the corresponding element of *d_thcode*, and the value and bit-length of the remaining encoding are stored in *word_val* and *word_len*, respectively. The process continues until all the codewords are written. To avoid race conditions with the previous and next thread-codes, the first and last writes are performed by using atomic OR operations [23].

3 Highly optimized GPU-based implementation of VLE (GVLE)

In this section, we present *GVLE*, our GPU-based implementation of VLE, which has been developed using the popular NVIDIA CUDA framework [16]. It is also compared with Yamamoto et al.'s proposal so that the achieved performance improvement can be clearly established.

As previous solutions [8, 10], GVLE is composed of only one kernel, whose execution configuration sets the number of thread-blocks to the maximum number of resident thread-blocks. The inputs and outputs of GVLE are the same as those of YAVLE, except that, in the case of GVLE, the VLET is provided in two separate vectors, one of 256 16-bit unsigned integers (*d_VLET_val*) and the other of 256 8-bit unsigned integers (*d_VLET_len*), which store the values and the bit-lengths of the codewords, respectively. As in the case of YAVLE, it is assumed that the maximum bit-length of codewords is 16.

Algorithm 3 presents the pseudo code of GVLE. As in previous approaches [8, 10], each thread-block caches the VLET in shared memory (Step 1) and encodes a different subset of block-inputs (Steps 2 to 9). The technique used to get the indexes of the block-inputs (Steps 2 and 9) is the same as that of YAVLE (Sect. 2.3). Let us define a *warp-code* as the encoding of the 32 thread-inputs processed by a warp, that is to say, the concatenation of the thread-codes computed by a warp. The processing of each block-input consists of Steps 3 to 8.

In Sect. 3.1, we describe Step 1 of Algorithm 3, and, in Sects. 3.2 to 3.6, Steps 4 to 8, respectively.

Algorithm 3: GVLE algorithm

```
1  kernel GVLE(// output
2              uint *d_output,
               // input
3              uint *d_input,
4              ushort d_VLET_val[256],
5              uchar d_VLET_len[256],
6              ull *d_scan
7              )
     // Step 1: cache VLET in shared memory
8    ushort s_VLET_val[256] ← d_VLET_val
9    uchar s_VLET_len[256] ← d_VLET_len

     // Step 2: get index of first block-input to encode
10   uint blockinput_idx ← get_global_counter_value()

     // While there are block-inputs to encode...
11   while (blockinput_idx < num_blockinputs)
       // Step 3: calculate index of thread-input
12     uint thinput_idx ← blockinput_idx × blockDim.x +
13                        threadIdx.x

       // Step 4: read thread-input
14     uchar32 thinput ← ((uchar32 *)d_input)[thinput_idx]

       // Step 5: calculate thread-code
15     uint seg_val[16], seg_len[16], thcode_len
16     GVLE_calc_thcode(// output
17                      seg_val, seg_len, thcode_len,
                        // input
18                      thinput, s_VLET_val, s_VLET_len)

       // Step 6: calculate parameters of warp-code
19     uint pos_of_thcode_in_wcode, wcode_len, wcode_pos
20     GVLE_calc_wcode_param(
             // output
21           pos_of_thcode_in_wcode, wcode_len, wcode_pos,
             // input
22           thcode_len, d_scan)

       // Step 7: build warp-code in shared memory
23     uint s_warpcode[num_warps][513]
24     GVLE_build_warpcode(// output
25                        s_warpcode[warp_idx],
                          // input
26                        seg_val, seg_len, thcode_len,
27                        pos_of_thcode_in_wcode, wcode_pos)

       // Step 8: write warp-code to output vector
28     GVLE_write_warpcode(// output
29                         d_output,
                           // input
30                         wcode_pos, s_warpcode[warp_idx],
31                         wcode_len)

       // Step 9: get index of next block-input to encode
32     blockinput_idx ← get_global_counter_value()
33   end while
34 end kernel
```

3.1 VLET caching

In Step 1 of Algorithm 3, since the VLET is used intensively for searching the code-words, each thread-block caches it in the fast on-chip shared memory. The global memory vectors *d_VLET_val* (line 8) and *d_VLET_len* (line 9) are copied to the identical shared memory vectors *s_VLET_val* and *s_VLET_len*, respectively, in a fully coalesced way.

The warp accesses to the VLET are random because they depend on the source data. For this reason, in order to minimize the bank conflicts caused by irregular warp accesses [23, 24], the VLET is implemented with two separate vectors (lines 8 and 9) whose base types have the minimum size necessary to store codewords of up to 16 bits (16-bits for *s_VLET_val* and 8-bits for *s_VLET_len*). In contrast, as YAVLE caches the VLET in a single vector whose base type (*Codeword*) has a size of 64-bits, the number of bank conflicts is much higher. The reason is that, in the case of YAVLE, each codeword is stored in two consecutive 32-bits elements of shared memory, while, in the case of GVLE, two codewords' values are cached in one 32-bit element, and four codewords' bit-lengths are kept in one 32-bit element.

On the other hand, although GVLE has to access two vectors (instead of one, as YAVLE) to get the value and the bit-length of one codeword, these readings are fast because they are executed in parallel at the instruction level.

3.2 Reading of thread-inputs

In Step 4 of Algorithm 3, each thread reads the 32 symbols of its thread-input through one vectorized access using the custom vector type *uchar32*, which is composed of 32 8-bit unsigned integers, and stores the thread-input in the variable *thinput* (line 14). Vectorized loads are an important CUDA optimization because they increase bandwidth and reduce both instruction count and latency [22].

In contrast, YAVLE reads the elements of its thread-input one by one, which results in inefficient strided global memory accesses [23, 24]. In addition, YAVLE, instead of loading its thread-input from global memory once, reads it twice: the first time to calculate the bit-length of the thread-code, and the second time to compute the bit-stream of the thread-code on the fly during its writing in the output vector.

3.3 Calculation of thread-codes

In Step 5 of Algorithm 3, each thread searches in *s_VLET_val* and *s_VLET_len* the codewords assigned to the 32 symbols stored in *thinput* to compute the corresponding thread-code (lines 16 to 18). Since a thread-code is the concatenation of 32 consecutive codewords and the bit-length of each codeword is no more than 16-bits, a thread-code is made up of 16 binary segments (each segment i corresponding to the concatenation of the codewords $2 \times i$ and $2 \times i + 1$), whose bit-lengths are not greater than 32-bits. Taking this into account, the values and the bit-lengths of the segments are calculated and cached in the private arrays *seg_val* and *seg_len* (line

15), of 16 32-bit unsigned integers each, respectively. Additionally, the bit-length *thcode_len* of each thread-code (line 15) is obtained by adding the bit-lengths of its segments.

The calculation of thread-codes is efficient for the following reasons. First, there are no dependencies between the different computations of segments, hence the degree of instruction-level parallelism is high. Second, each segment calculation is performed with few operations of high throughput (one binary shift and two sums). Third, there is no warp divergence in the computation of segments. Fourth, the arrays *seg_val* and *seg_len* are placed in the register space [24] because (1) they are small, (2) they are indexed with constant quantities, and (3) the kernel does not use more registers than available.

Since YAVLE computes the bit-stream of each thread-code by concatenating its 32 codewords on the fly during its writing to the output vector, the number of executed instructions is higher than that of GVLE.

3.4 Calculation of parameters of warp-codes

In Step 6 of Algorithm 3, each thread-block calculates the following parameters, which are necessary for the posterior processing of the warp-codes of the current block-code (line 19):

- Bit-position of each thread-code in its warp-code (*pos_of_thcode_in_wcode*).
- Bit-length of each warp-code (*wcode_len*).
- Bit-position of each warp-code in the output vector (*wcode_pos*).

On the one hand, the intra-block scan method of Sengupta et al. [20] is executed on the bit-lengths of the thread-codes of the current block-code to calculate the parameters *pos_of_thcode_in_wcode*, *wcode_len*, the bit-position of each warp-code in the block-code (*pos_of_wcode_in_bcode*), and the bit-length of the block-code (*blockcode_len*).

On the other hand, the bit-position of each block-code in the output vector (*blockcode_pos*) is obtained by carrying out a scan operation on the bit-lengths of the block-codes using a novel inter-block scan algorithm, which is proposed in the next section.

Once a warp gets the parameters *blockcode_pos* and *pos_of_wcode_in_bcode*, it computes *wcode_pos* by adding them.

3.4.1 Our inter-block scan method

In our algorithm, the global vector *d_scan* (line 6), whose elements are initially zero, is used to perform a regular segmented inclusive scan on segments of bit-lengths of 32 consecutive block-codes. Each segment i is composed of the bit-lengths of block-codes $32 \times i$ to $32 \times i + 31$, and its prefix sum is written in the corresponding 32-elements sub-vector i of *d_scan*. The scan of each segment is performed by a set of 32 thread-blocks, which will be referred to as *sub-grid*.

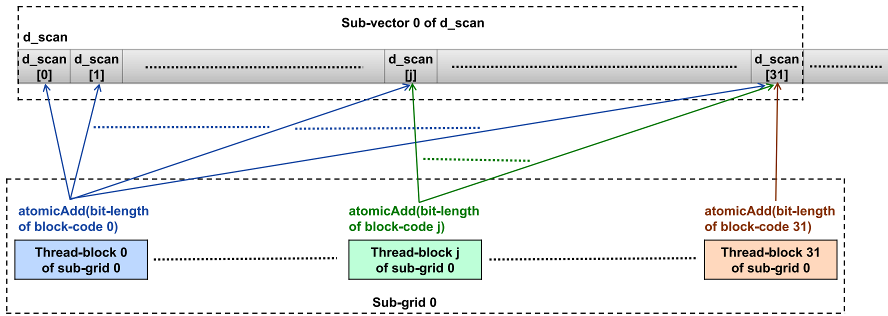

Fig. 5 GVLE inter-block scan on the first segment

Each segment i is processed by a sub-grid i, which is composed of the thread-blocks that manage the block-codes $32 \times i$ to $32 \times i + 31$. The scan of each segment i is calculated directly in the *d_scan* sub-vector i by the first warp of each thread-block j of the sub-grid i performing an atomic addition [23] of the bit-length of its block-code to the elements j to 31 of the sub-vector i. Note that the number of atomic additions carried out by each sub-grid on each element j of the corresponding sub-vector is $j + 1$. Figure 5 illustrates this mechanism for the first segment.

In order to detect that the segmented prefix sum has already been calculated for a particular *d_scan* sub-vector element, bits 57 to 62 of each element are used to store the number of atomic additions performed on it. This sums counter is implemented by performing the atomic additions with the bit 57 of the bit-lengths of the block-codes set to 1. The bits 0 to 56 of each element j of each *d_scan* sub-vector i are used to store the corresponding segmented scan value, i.e., the sum of bit-lengths of block-codes $32 \times i$ to $32 \times i + j$. In the case of the thirty-second element of each sub-vector i except the first, a second value is assigned to its bits 0 to 56 in a posterior stage of our algorithm, which is the not-segmented scan value, i.e., the sum of bit-lengths of block-codes 0 to $32 \times i + 31$. To distinguish between these two mutually exclusive values, in the second case, a flag will be activated in the bit 63, which will be referred to as *flag P*. Table 1 shows an example of GVLE inter-block scan, which presents an extract of the first 64 values written in *d_scan* (i.e., the corresponding to the first two segments). Note that *d_scan*[63] is the only element that has the flag P activated (the bit 63 is 1). The reason is that it stores the sum (1, 206, 728) of all the bit-lengths of segments 0 and 1. The remaining elements hold the segmented scan value (bits 0 to 56), and the number of atomic additions performed on them (bits 57 to 62).

Let us define a *sub-code* as the bit-stream composed of the block-codes managed by a sub-grid. Given a thread-block j of a sub-grid i, the first warp of the thread-block follows the next steps to calculate the parameter *blockcode_pos* of the corresponding block-code:

1. It performs an atomic addition of the bit-length of the block-code (with the bit 57 set to 1) to the elements j to 31 of the sub-vector i.

Table 1 Example of GVLE inter-block scan on the two first segments

i	Bit-length of block-code i	Value stored in bits 57 to 63 of d_scan[i]	Value stored in bits 0 to 56 of d_scan[i]
0	19,036	1	19,036
1	18,641	2	37,677
2	18,210	3	55,887
.	.	.	.
29	18,976	30	565,395
30	19,209	31	584,604
31	19,026	32	603,630
32	18,775	1	18,775
33	19,223	2	37,998
34	19,331	3	57,329
.	.	.	.
61	18,632	30	565,170
62	18,865	31	584,035
63	19,063	64	1,206,728

2. It gets the bit-position of the block-code in its sub-code, which will be referred to as *pos_of_bc_in_sc*, in the following way. If $j = 0$, clearly *pos_of_bc_in_sc* is 0. Otherwise, it reads repeatedly the element $j - 1$ of the sub-vector i until its sums counter is j. The parameter *pos_of_bc_in_sc* is obtained by resetting the bits 57 to 62.
3. It gets the bit-position of the sub-code in the output vector, which will be referred to as *pos_of_sc_in_out*, in the following way. If $i = 0$, clearly *pos_of_sc_in_out* is 0. Otherwise, it reads repeatedly the element 31 of the sub-vector $i - 1$ until its sums counter is 32 or the flag P is activated (i.e., the value stored in bits 57 to 63 is 32 or 64). If the flag P of the read value is not activated, it repeats the same procedure on sub-vectors $i - 2$, $i - 3$, ... until the read value has the flag P activated or there are no more sub-vectors to process. The parameter *pos_of_sc_in_out* is obtained by accumulating the read values, with the bits 57 to 63 set to 0, as they are read.
4. The parameter *blockcode_pos* is obtained by adding *pos_of_bc_in_sc* to *pos_of_sc_in_out*.
5. If $j = 31$ (i.e., the thread-block is the last of the sub-grid):

   (a) It computes the bit-length of the sub-code i (*sc_len*) by adding *pos_of_bc_in_sc* to *blockcode_len*.
   (b) It computes the sum of *pos_of_sc_in_out* to *sc_len*, and stores it in the element 31 of the sub-vector i with the flag P activated. Note that the written value is the bit-position of the sub-code $i + 1$ in the output vector.

As will be shown in Sect. 4, our inter-block scan method outperforms that of Yamamoto et al. The reasons are the following:

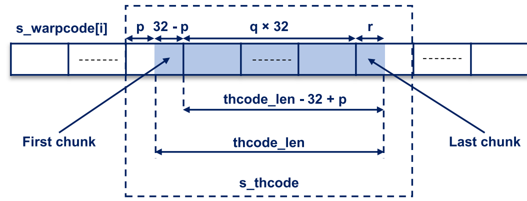

Fig. 6 Writing of a thread-code in the shared memory buffer s_warp_code[i]

1. Since the scan on different segments are executed independently by the corresponding sub-grids, the degree of parallelism is higher in our algorithm.
2. In YAVLE, each thread-block assigns the bit-length of its block-code to a single element of *d_scan*, while, in GVLE, each thread-block uses the bit-length of its block-code to update $32 - j$ elements of *d_scan*, where j is the index of the thread-block within its sub-grid. As the number of elements updated by thread-blocks 0, 1, ..., 31 of a sub-grid are 32, 31, ... 1, respectively, the average number of elements updated per thread-block is 16.
3. In GVLE, the scan of each segment is performed directly on its sub-vector by using atomic operations. In contrast, in YAVLE, after writing the bit-lengths on *d_scan*, it is necessary to read them in groups of 32 elements to perform the scan operation.

3.5 Building of warp-codes in shared memory

In Step 7 of Algorithm 3, each warp i of each thread-block builds its warp-code in the shared memory buffer *s_warp_code*[i] (line 23) of 513 32-bit unsigned integers. The warp-code is written right-shifted the same number of bits that it will be in the target sub-vector of *d_output*. The size of each buffer is 513 for the following reasons. On the one hand, since the bit-length of each codeword is no more than 16-bits, the number of codewords of each thread-code is 32, and the warp size is 32, the maximum number of bits of a warp-code is $16 \times 32 \times 32 = 16,384$ bits, which can be stored in $16,384/32 = 512$ unsigned integers. On the other hand, as each warp-code is written right-shifted, an extra unsigned integer is necessary, so the size of each warp buffer is $512 + 1 = 513$.

The warp-code is built in the buffer by each thread of the warp writing its thread-code, which was cached previously in the private arrays *seg_val* and *seg_len* ((lines 15 to 18)), in the bit-position of the tread-code within its warp-code. Given a thread-code, let *s_thcode* be the buffer sub-vector in which it is written, p the bit-position of the thread-code in *s_thcode*, and *thcode_len* the bit-length of the thread-code. We call q and r the quotient and the remainder of the division of ($thcode_len - 32 + p$) by 32, respectively. As shown in Fig. 6, the first $32 - p$ bits of the thread-code (which will be referred to as *first chunk*) are written right-aligned in *s_thcode*[0], the following q 32-bits sequences in *s_thcode*[1], ... *s_thcode*[q], and, if $r > 0$, the last r bits (which will be denoted by *last chunk*) in *s_thcode*[$q + 1$]. The writing of the warp-code is carried out by following the next steps:

1. Each thread writes all the bits of its thread-code, except the last chunk, in the elements *s_thcode*[0], ... *s_thcode*[*q*].
2. All threads of the warp synchronize by executing the CUDA function *__syncwarp* [23].
3. Each thread, if its thread-code has a last chunk, writes it in the first r bits of *s_thcode*[$q + 1$].

Note that the warp synchronization ensures that no race conditions exist in the writing of those elements of the buffer in which the last chunk of a thread-code and the first chunk of the next thread-code are stored.

3.6 Writing of warp-codes to output vector

In Step 8 of Algorithm 3, each warp, after writing its warp-code in the shared memory buffer, iterates over 32-elements segments of the buffer to copy them to the target *d_output* sub-vector in a coalesced way [24] (lines 28 to 31). The first and last elements of the target *d_output* sub-vector are written using atomic OR operations ([23]) to preserve the last chunk of the previous warp-code and the first chunk of the next warp-code, respectively, if they have already been written.

In YAVLE, each thread writes the elements of its thread-code directly to global memory one by one, which results in inefficient strided global memory accesses.

4 Experimental evaluation

To evaluate GVLE and compare it to YAVLE, we have used the Standard Canterbury Corpus, consisting of 11 files (alice29.txt, asyoulik.txt, cp.html, fields.c, grammar.lsp, kennedy.xls, lcet10.txt, plrabn12.txt, ptt5, sum, xargs.1) and the Large Canterbury Corpus, consiting of 3 files (bible.txt, e.coli, world192.txt), which are available at http://www.data-compression.info/Corpora/CanterburyCorpus/. Furthermore, to fully utilize the resources of the target GPU, we have increased each of the 14 files by replicating its original content the minimum number of times necessary to make the final size greater than or equal to 100 megabytes.

The method used to compute the VLETs is the Huffman coding, and we have obtained the implementation of YAVLE from the source code of Yamamoto et al.'s solution, published on GitHub at github.com/daisuke-takafuji/Huffman_coding_Gap_arrays.

In order to measure precisely the execution times of the kernels, we run one warm-up iteration and then fifty iterations to report their statistical values.

Our test machine has a 3.50Ghz Intel Core i7-7800X CPU, 32 GB of RAM, and a GeForce RTX 2080 GPU (Turing architecture with compute capability 7.5).The CUDA toolkit and the GPU driver versions are 11.1 and 512.15, respectively. We have used the default optimization flag (− O3) [32].

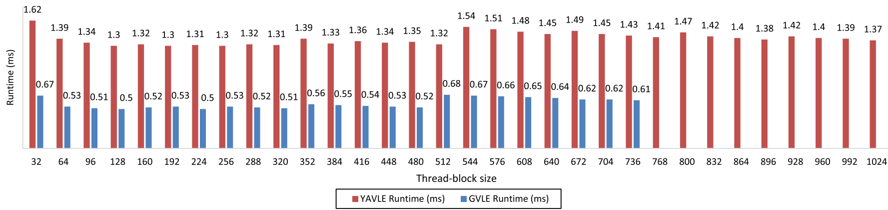

Fig. 7 YAVLE and GVLE runtimes for thread-block sizes between 32 and 1024 (in steps of 32 threads)

Table 2 Number of shared memory load bank conflicts and runtimes of kernels YAVLE and EXP_VLET, and corresponding improvements in EXP_VLET over YAVLE

Parameter	YAVLE	EXP_VLET	Improvement
Shared load bank conflicts	13,839,817	3,262,067	4.24×
Runtime (ms)	1.30	1.07	1.21×

4.1 Sensitivity analysis of the thread-block size

To analyze the effect of the thread-block size on the performance of YAVLE and GVLE, we have measured their average runtimes for all possible values of the thread-block size that are multiples of 32. Figure 7 shows the obtained results. Note that, in the case of GVLE, the maximum thread-block size is 736, due to the shared memory buffer used to build the warp-codes.

As it can be seen, the effect of the thread-block size on the performance of YAVLE and GVLE is low. Since 128 is an optimal thread-block size for both GVLE and YAVLE, we have used this value in the remaining experiments.

4.2 Comparison of GVLE with YAVLE

In order to determine the contribution of each of our optimizations in the performance improvement in GVLE with respect to YAVLE, we have developed a set of kernels that starting from YAVLE, gradually implement the different optimization techniques of GVLE. In the following sections, we present the obtained results.

4.2.1 VLET implementation

We have built the kernel *EXP_VLET* from YAVLE by substituting YAVLE's implementation of VLET (i.e., one vector of 256 elements of type *Codeword*) for that of GVLE (i.e., a vector of 256 16-bit unsigned integers to store the values of the codewords, and a second vector of 256 8-bit unsigned integers to hold the bit-lengths of the codewords).

Table 3 Number of global load/reduction/store transactions, number of executed instructions and runtimes of kernels EXP_VLET and EXP_GM, and corresponding improvements in EXP_GM over EXP_VLET

Parameter	EXP_VLET	EXP_GM	Improvement
Global load transactions	60,039,145	7,164,487	8.38×
Global reduction transactions	4,307,360	205,459	20.96×
Global store transactions	7,156,553	2,231,864	3.21×
Executed instructions	1,153,782.69	882,368.02	1.31×
Runtime (ms)	1.07	0.70	1.53×

Table 4 Number of executed instructions and runtimes of kernels EXP_GM and EXP_REG, and corresponding improvements in EXP_REG over EXP_GM

Parameter	EXP_GM	EXP_REG	Improvement
Executed instructions	882,368.02	542,291.61	1.63×
Runtime (ms)	0.70	0.57	1.23×

As shown in Table 2, EXP_VLET is 1.21× faster than YAVLE due to the improvement in the shared memory load bank conflicts (4.24×).

4.2.2 Global memory reading and writing

We have obtained the kernel *EXP_GM* from EXP_VLET by replacing its global memory reading and writing methods for those of GVLE. On the one hand, instead of reading the thread-inputs element by element through strided global memory accesses, they are read through vectorized accesses using the custom vector type *uchar32*. On the other hand, instead of writing the thread-codes directly to global memory element by element through strided global memory accesses, each warp, after writing its warp-code in its shared memory buffer, iterates over 32-elements segments of the buffer to copy them to global memory in a coalesced way.

As shown in Table 3, EXP_GM is 1.53× faster than EXP_VLET due to the improvement in the global load transactions (8.38×), the global reduction transactions (20.96×), the global store transactions (3.21×) and the executed instructions (1.31×).

4.2.3 Thread-codes building

We have developed the kernel *EXP_REG* from EXP_GM by performing the following two changes. First, the calculation of the bit-length of the thread-code (Step 4 of Algorithm 1) is replaced by the complete calculation of the thread-code (Step 5 of Algorithm 3). Second, each warp-code is built in shared memory by concatenating the 16-binary segments of each thread-code (Step 7 of Algorithm 3), instead of by linking the 32 codewords of each thread-code (Step 6 of Algorithm 1).

Table 5 Number of global atomic/load/store transactions, number of executed instructions and runtimes of kernels EXP_REG and GVLE, and corresponding improvements in GVLE over EXP_REG

Parameter	EXP_REG	GVLE	Improvement
Global atomic transactions	26,051	141,610	0.18×
Global load transactions	7,259,723	6,721,148	1.08×
Global store transactions	2,231,864	2,181,301	1.02×
Executed instructions	542,291.61	517,852.75	1.05×
Runtime (ms)	0.57	0.50	1.14×

Table 6 Number of shared memory load bank conflicts, number of global load/reduction/store transactions, number of executed instructions and runtimes of kernels YAVLE and GVLE, and corresponding improvements in GVLE over YAVLE

Parameter	YAVLE	GVLE	Improvement
Shared load bank conflicts	13,839,817	3,170,669	4.36×
Global load transactions	60,261,119	6,721,148	8.97×
Global reduction transactions	4,307,361	205,459	20.96×
Global store transactions	7,156,553	2,181,301	3.28×
Executed instructions	1,319,851.51	517,852.75	2.55×
Runtime (ms)	1.30	0.50	2.57×

As shown in Table 4, EXP_REG is 1.23× faster than EXP_GM due to the improvement in the executed instructions (1.63×).

4.2.4 Inter-block scan method

The unique difference between the kernels EXP_REG and GVLE is that the former uses the Yamamoto et al.'s inter-block scan method (Sect. 2.3.3), while the latter uses our inter-block scan algorithm (Sect. 3.4.1).

As shown in Table 5, although the number of global atomic transactions of EXP_REG is 0.18× that of GVLE, GVLE is 1.14× faster than EXP_REG due to the improvement in the global load transactions (1.08×), the global store transactions (1.02×) and the executed instructions (1.05×).

4.2.5 Global contribution of our optimization strategies

Table 6 compares YAVLE and GVLE by presenting the values of the performance parameters referenced in previous sections. As it can be seen, GVLE is 2.57× faster than YAVLE due to the improvement in the shared memory load bank conflicts (4.36×), the global load transactions (8.97×), the global reduction transactions (20.96×), the global store transactions (3.28×), and the executed instructions (2.55×). Finally, Fig. 8 presents the runtimes of YAVLE and GVLE, and Table 7 the corresponding statistics. As it can be seen, the acceleration of GVLE is significant, since its value is between 1.97× and 3.11×.

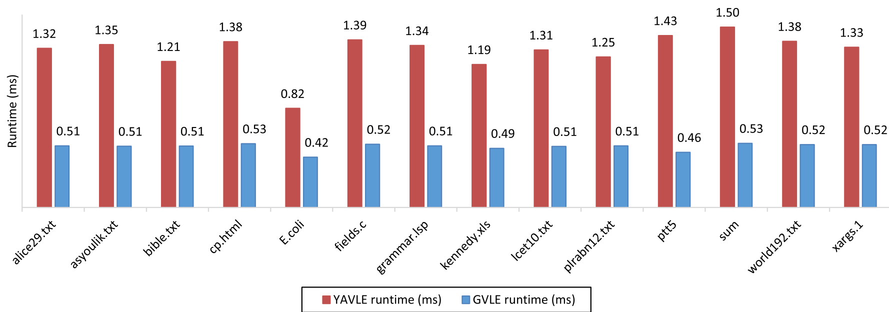

Fig. 8 YAVLE and GVLE runtimes for each test file

Table 7 Statistics of GVLE, YAVLE, CUVLE and CPU_VLE Runtimes, and of GVLE Speedups

	Average	Minimum	Maximum
GVLE runtime (ms)	0.50	0.42	0.53
YAVLE runtime (ms)	1.30	0.82	1.50
Speedup of GVLE	2.57×	1.97×	3.11×
CUVLE runtime (ms)	6.85	6.59	7.36
Speedup of GVLE	13.63×	12.93×	16.67×
CPU_VLE runtime (ms)	191.75	112.07	225.93
Speedup of GVLE	377.15×	273.15×	425.90×

4.3 Comparison between GVLE, CUVLE and the serial implementation of VLE

Table 7 compares the statistics of GVLE, CUVLE and the implementation of VLE on CPU (*CPU_VLE*). As it can be seen, GVLE is on average 13.63× faster than CUVLE, our previous implementation of VLE, which represents a significant advance in our research on GPU-based acceleration of VLE. On the other hand, the speedup of VLE with respect to the serial implementation of VLE is considerable, since it is on average 377.15×.

4.4 Comparison between inter-block scan methods

In order to compare the performance of our inter-block scan method with those of Yamamoto et al. [10] and Yan et al. [21], we have developed three kernels that perform the scan operation using the method of Sengupta et al. [20] for the intra-block scan, and one of the methods under study for the inter-block scan. We call the kernels that use our inter-block scan method, that of Yamamoto et al, and that of Yan et al., *GVLE_scan*, *YAVLE_scan* and *CUVLE_scan*, respectively. The input and output vectors are the same as those used in our previous experiments, with the

Table 8 Statistics of kernels GVLE_scan, YAVLE_scan and CUVLE_scan Runtimes, and of GVLE_scan Speedups

	Average	Minimum	Maximum
GVLE_scan runtime (ms)	1.22	1.20	1.27
YAVLE_scan runtime (ms)	1.97	1.95	2.05
Speedup of GVLE_scan	1.62×	1.56×	1.65×
CUVLE_scan runtime (ms)	47.13	46.68	48.02
Speedup of GVLE_scan	38.32×	36.87×	39.49×

particularity that each thread, instead of reading eight consecutive elements of the input vector, reads only one.

Table 8 compares the statistics of the kernels. As it can be seen, GVLE_scan clearly outperforms YAVLE_scan, since the speedup is between 1.56× and 1.65×. Moreover, the speedup of GVLE_scan_with respect to CUVLE_scan is very high, as it is between 36.87× and 39.49×. It can be seen that although our inter-block scan algorithm is the least influential optimization in the acceleration of YAVLE, it provides a significant speedup in the case of the scan operation. Therefore, it can be used to accelerate significantly algorithms that require performing an inter-block scan, such as the scan operation itself or the stream compaction [33].

5 Related work

In this section, we review some GPU-based solutions in which VLE is partially implemented, since it only operates on small data chunks, and does not concatenate the resulting encodings. Each data chunk is mapped to a thread-block [11, 12], a warp [13] or even a thread [13, 14]. In the corresponding compression algorithms, the concatenation is not necessary [11, 12] or is implemented with a separate component [13].

Tian et al. [11] proposed cuSZ, a CUDA-based implementation of SZ [34], which is one of the best error-bounded lossy compressors for scientific data. cuSZ splits the whole dataset into multiple chunks, and compresses them independently, which favors coarse grained decompression. A dual-quantization scheme is applied to completely remove the strong data dependency in SZ's prediction-quantization step. The quantization codes generated by the dual-quantization procedure are compressed by a customized Huffman coding, which follows the next four steps. First, calculate the statistical frequency for each quantization bin (as a symbol) using the method proposed by Gómez-Luna et al. [35]. Second, build the Huffman tree based on the frequencies and construct a base codebook. Third, transform the base codebook to the canonical Huffman codebook [36]. Fourth, encode in parallel according to the codebook, and concatenate Huffman codes into a bitstream (called deflating). Experimental evaluation on a NVIDIA V100 GPU showed that cuSZ improves SZ's compression throughput by up to 13.1x over the production version running on two 20-core Intel Xeon Gold 6148 CPUs.

In a later work, Tian et al. [12] presented an efficient CUDA-based Huffman encoder that outperforms the scheme presented in [11]. The main novelties of this work are the following. First, the development of an efficient parallel codebook construction on GPUs that scales effectively with the number of input symbols. Second, a novel reduction-based encoding scheme that can efficiently merge the codes on GPU. Experimental evaluation showed that their solution can improve the encoding throughput by up to 5.0x and 6.8x on NVIDIA RTX 5000 and V100 GPUs, respectively, over their previous proposal [11], and by up to 3.3 over the multithread encoder on two 28-core Xeon Platinum 8280 CPUs.

Zhu et al. [13] presented an efficient parallel entropy coding method (EPEnt), which was implemented in CUDA, to accelerate the entropy coding stage of JPEG image compression algorithm. EPEnt has three phases: coding, shifting and stuffing. In the coding phase, the 8×8 blocks of quantized transformed coefficients are encoded in parallel, via run-length encoding and Huffman coding, to form their corresponding bitstreams. In the shifting phase, the bitstreams are shifted to ensure that the bitstreams of adjacent coefficient blocks can be correctly concatenated. Finally, in the stuffing phase, the output stream is produced by concatenating the shifted bitstreams. Experimental evaluation on a NVIDIA GTX 1050Ti GPU showed that compared with sequential implementation on a 2.4 GHz i7-4700HQ CPU, the maximum speedup ratio of entropy coding can reach 39 times without affecting compressed images quality.

Fuentes-Alventosa et al. [14] proposed CAVLCU, an efficient implementation of CAVLC on CUDA, which was based on four key ideas. First, CAVLCU is composed of only one kernel to avoid the long latency global memory accesses required to transmit intermediate results between different kernels, and the costly launches and terminations of additional kernels. Second, the efficient Yan et al.'s synchronization mechanism [21] is used for thread-blocks that process adjacent frame regions (in horizontal and vertical dimensions) to share results in global memory space. Third, the available global memory bandwidth is exploited fully by using vectorized loads to move directly the quantized transform coefficients to registers. Fourth, register tiling is used to implement the zigzag sorting, thus obtaining high instruction-level parallelism. Experimental evaluation on NVIDIA GPUs GeForce GTX 970 and GeForce RTX 2080 showed that CAVLCU is between 2.5x and 5.4x faster than the best previous GPU-based implementation of CAVLC [37, 38].

6 Conclusions

This work has presented GVLE, a highly optimized GPU-Based implementation of variable-length encoding. Our solution overcomes the main performance issues of the state-of-the-art GPU-based implementations of VLE by using the next optimization strategies. First, the caching of the codeword look-up table is done in a way that minimizes the bank conflicts. Second, input data is read by using vectorized loads to exploit fully the available global memory bandwidth. Third, each thread encoding is performed efficiently in the register space with high instruction-level parallelism and lower number of executed instructions.

Fourth, a novel inter-block scan method, which outperforms those of state-of-the-art solutions, is used to calculate the bit-positions of the thread-blocks encodings in the output bit-stream. Our proposed mechanism is based on a regular segmented scan performed efficiently on sequences of bit-lengths of 32 consecutive thread-blocks encodings by using global atomic additions. Fifth, output data are written efficiently by executing coalesced global memory stores.

An exhaustive experimental evaluation shows that our solution is between 1.97× and 3.11× faster than the best state-of-the-art implementation due to the improvement in the shared memory load bank conflicts (4.36×), the global load transactions (8.97×), the global reduction transactions (20.96×), the global store transactions (3.28×) and the executed instructions (2.55×). Moreover, experimental results show that the speedup of the scan operation using our inter-block scan algorithm is on average 1.62× and 38.32× with respect to using the methods of Yamamoto et al. and Yan et al., respectively. Therefore, our method can be used to accelerate significantly algorithms that require performing an inter-block scan, such as the scan operation itself or the stream compaction.

Acknowledgements Not applicable.

Author Contributions All authors contributed to the study conception. Design, data collection and analysis were performed by AF-A. The first draft of the manuscript was written by AF-A and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding None.

Data availability Not applicable.

Declarations

Conflict of interest No, I declare that the authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethical approval and consent to participate The corresponding author has read the Springer journal policies on author responsibilities and submits this manuscript in accordance with those policies.

Consent for publication I have read and understood the publishing policy, and submit this manuscript in accordance with this policy.

References

1. Jayasankar U, Thirumal V, Ponnurangam D (2021) A survey on data compression techniques: from the perspective of data quality, coding schemes, data type and applications. J King Saud Univ-Comput Inf Sci 33(2):119–140
2. Wise J "How many videos are uploaded to youtube a day in 2022?", June 2022. https://earthweb.com/how-many-videos-are-uploaded-to-youtube-a-day/
3. Banerji A, Ghosh AM (2010) Multimedia technologies. Tata McGraw Hill, New Delhi
4. Pu IM (2005) Fundamental data compression. Butterworth-Heinemann
5. Huffman DA (1952) A method for the construction of minimum-redundancy codes. Proc IRE 40(9):1098–1101

6. Moffat A (2019) Huffman coding. ACM Comput Surv (CSUR) 52(4):1–35
7. Balevic A (2009) Parallel variable-length encoding on GPGPUs. In European Conference on Parallel Processing, Springer, Berlin, Heidelberg. pp 26–35
8. Fuentes-Alventosa A, Gómez-Luna J, González-Linares JM, Guil N (2014) CUVLE: variable-length encoding on CUDA. In Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on IEEE. pp 1–6
9. Rahmani H, Topal C, Akinlar C (2014) A parallel Huffman coder on the CUDA architecture. In 2014 IEEE Visual Communications and Image Processing Conference, IEEE. pp 311–314
10. Yamamoto N, Nakano K, Ito Y, Takafuji D, Kasagi A, Tabaru T (2020) Huffman coding with gap arrays for GPU acceleration. In 49th International Conference on Parallel Processing-ICPP. pp 1–11
11. Tian J, Di S, Zhao K, Rivera C, Fulp MH, Underwood R, Cappello F (2020) Cusz: an efficient gpu-based error-bounded lossy compression framework for scientific data. arXiv preprint arXiv:2007.09625
12. Tian J, Rivera C, Di S, Chen J, Liang X, Tao D, Cappello F (2021) Revisiting huffman coding: toward extreme performance on modern gpu architectures. In 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE. pp 881–891
13. Zhu F, Yan H (2022) An efficient parallel entropy coding method for JPEG compression based on GPU. J Supercomput 78(2):2681–2708
14. Fuentes-Alventosa A, Gómez-Luna J, González-Linares JM, Guil N, Medina-Carnicer R (2022) CAVLCU: an efficient GPU-based implementation of CAVLC. J Supercomput 78(6):7556–7590
15. NVIDIA: GPU-Accelerated Applications (2020) https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/gpu-applications-catalog.pdf
16. NVIDIA: CUDA Zone (2022) https://developer.nvidia.com/category/zone/cuda-zone
17. Khronos group: OpenCL (2022) https://www.khronos.org/opencl/
18. Harris M, Sengupta S, Owens JD (2007) Parallel prefix sum (scan) with CUDA. GPU Gems 3(39):851–876
19. Martín PJ, Ayuso LF, Torres R, Gavilanes A (2012) Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In 2012 International Conference on High Performance Computing & Simulation (HPCS), IEEE. pp 511–519
20. Sengupta S, Harris M, Garland M (2008) Efficient parallel scan algorithms for GPUs. NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003, 1(1), 1–17
21. Yan S, Long G, Zhang Y (2013) StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming pp 229–238
22. Luitjens J “CUDA Pro Tip: increase Performance with Vectorized Memory Access”, Dec. 2013. https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/
23. NVIDIA: CUDA C Programming Guide (2022) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
24. NVIDIA: CUDA C Best Practices Guide (2022) https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
25. Manz O (2021) Well Packed-Not a Bit Too Much. Springer Fachmedien Wiesbaden
26. Gyasi-Agyei A (2019) Telecommunications engineering: principles and practice. World Scientific, Singapore
27. Unger H, Kyamaky K, Kacprzyk J. (Eds.). (2011). Autonomous Systems: Developments and Trends (Vol. 391). Springer
28. Lal S, Lucas J, Juurlink B (2017) E^2MC: entropy encoding based memory compression for GPUs. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE. pp 1119–1128
29. Choukse E, Sullivan MB, O’Connor M, Erez M, Pool J, Nellans D, Keckler SW (2020) Buddy compression: enabling larger memory for deep learning and HPC workloads on GPUs. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), IEEE. pp 926–939
30. Larmore LL, Hirschberg DS (1990) A fast algorithm for optimal length-limited Huffman codes. J ACM (JACM) 37(3):464–473

31. Katajainen J, Moffat A, Turpin A (1995) A fast and space-economical algorithm for length-limited coding. In International Symposium on Algorithms and Computation. Springer, Berlin, Heidelberg. pp 12–21
32. NVIDIA CUDA Compiler Driver NVCC (2022) https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html
33. Luna JGG, Chang LW, Sung IJ, Hwu WM, Guil N (2015) In-place data sliding algorithms for many-core architectures. In: 2015 44th International Conference on Parallel Processing, IEEE. pp 210–219
34. Di S, Cappello F (2016) Fast error-bounded lossy HPC data compression with SZ. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE. pp 730–739
35. Gómez-Luna J, González-Linares JM, Benavides JI, Guil N (2013) An optimized approach to histogram computation on GPU. Mach Vis Appl 24(5):899–908
36. Barnett ML (2003) U.S. Patent No. 6,657,569. Washington, DC: U.S. Patent and Trademark Office
37. Su H, Zhang C, Chai J, Wen M, Wu N, Ren J (2011) A high-efficient software parallel CAVCL encoder based on GPU, 2011 34th International Conference on Telecommunications and Signal Processing (TSP), Budapest, pp 534–540
38. Su H, Wen M, Wu N, Ren J, Zhang C (2014) Efficient parallel video processing techniques on GPU: from framework to implementation. Sci World J, 2014

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Antonio Fuentes-Alventosa[1] · Juan Gómez-Luna[2] · R. Medina-Carnicer[1]

✉ Antonio Fuentes-Alventosa
antonio.fa@gmail.com

Juan Gómez-Luna
juang@ethz.ch

R. Medina-Carnicer
rmedina@uco.es

1 Department of Computer Sciences and Numerical Analysis, University of Córdoba, Córdoba, Spain

2 Department of Information Technology and Electrical Engineering, ETH Zürich, Zürich, Switzerland