CAVLCU: an efficient GPU-based implementation of CAVLC

Antonio Fuentes-Alventosa1 · Juan Gómez-Luna2 · José María González-Linares3 · Nicolás Guíl3 · R. Medina-Carnicer1

Accepted: 27 October 2021 / Published online: 29 November 2021
© The Author(s) 2021

Abstract
CAVLC (Context-Adaptive Variable Length Coding) is a high-performance entropy method for video and image compression. It is the most commonly used entropy method in the video standard H.264. In recent years, several hardware accelerators for CAVLC have been designed. In contrast, high-performance software implementations of CAVLC (e.g., GPU-based) are scarce. A high-performance GPU-based implementation of CAVLC is desirable in several scenarios. On the one hand, it can be exploited as the entropy component in GPU-based H.264 encoders, which are a very suitable solution when GPU built-in H.264 hardware encoders lack certain necessary functionality, such as data encryption and information hiding. On the other hand, a GPU-based implementation of CAVLC can be reused in a wide variety of GPU-based compression systems for encoding images and videos in formats other than H.264, such as medical images. This is not possible with hardware implementations of CAVLC, as they are non-separable components of hardware H.264 encoders. In this paper, we present CAVLCU, an efficient implementation of CAVLC on GPU, which is based on four key ideas. First, we use only one kernel to avoid the long latency global memory accesses required to transmit intermediate results among different kernels, and the costly launches and terminations of additional kernels. Second, we apply an efficient synchronization mechanism for thread-blocks (In this paper, to prevent confusion, a block of pixels of a frame will be referred to as simply block and a GPU thread block as thread-block.) that process adjacent frame regions (in horizontal and vertical dimensions) to share results in global memory space. Third, we exploit fully the available global memory bandwidth by using vectorized loads to move directly the quantized transform coefficients to registers. Fourth, we use register tiling to implement the zigzag sorting, thus obtaining high instruction-level parallelism. An exhaustive experimental evaluation showed that our approach is between 2.5× and 5.4× faster than the only state-of-the-art GPU-based implementation of CAVLC.

Keywords CAVLC · GPU · CUDA · H.264 · Parallel implementations · Data compression · Variable-length encoding
1 Introduction

In the current digital era, the massive use of multimedia data, such as images and videos, together with the necessity to overcome the restrictions of storage space and communication bandwidth, have given an essential role to data compression.

Generally speaking, data compression can be lossless or lossy, depending on whether the original content is preserved or not [33]. Lossless compression is used when it is necessary that the original and uncompressed data remain exactly the same, such as executable programs and textual documents. Lossy compression discards some information to increment the amount of data reduction. Image file formats like PNG use only lossless compression, while others like TIFF may use either lossless or lossy methods [2]. Entropy coding [33] is a type of lossless compression in which mostly used patterns are assigned with codes of shorter length, whereas rarely used patterns are assigned with codes of longer length.

CAVLC (Context-Adaptive Variable Length Coding) is a high-performance entropy technique for video and image compression [14, 32]. In this method, different sets of variable-length codes are chosen depending on already encoded syntax elements. It is the most commonly used entropy technique in the video standard H.264.

In the last two decades, many designs for CAVLC have been proposed. The majority of these solutions are based on hardware, such as FPGA [4, 6, 12, 28] and ASIC approaches [1, 3, 13, 21]. In contrast, parallel software implementations of CAVLC [5, 31, 38, 39, 43] are currently very scarce.

One of the most successful trends in high-performance computing is general-purpose computation on graphics processing units (GPGPU), thanks to programming environments such as CUDA [26] and OpenCL [15]. Efficient implementations of CAVLC on GPU are currently very useful for the following reasons. First, they can be exploited as the entropy component in GPU-based H.264 encoders, which are a very suitable solution when it is necessary to implement functionality not provided by GPU built-in H.264 hardware encoders (e.g., NVENC in NVIDIA graphics cards [27]). In that case, many adaptations of CAVLC proposed in different fields, like data encryption [19, 40–42] and information hiding [16, 17, 45, 46], can be applied. Second, implementations of CAVLC on GPU can be reused and easily adapted in the development of a great variety of GPGPU compression systems for encoding both images and videos in formats other than H.264, like medical images [20, 30, 37]. This is not possible with hardware implementations of CAVLC, as they are non-separable components of hardware H.264 encoders.

In this paper, we present CAVLCU, an optimized implementation of CAVLC on GPU developed in CUDA. As our approach is built using only one CUDA kernel, it avoids the long latency global memory accesses required to transmit intermediate results among different kernels, and the costly launches and terminations of additional kernels. In our algorithm, thread-blocks that process adjacent frame regions (in horizontal and vertical dimensions) share results in global memory space using an efficient synchronization mechanism. Additionally, CAVLCU simplifies the zigzag sorting of the blocks, as each thread, after reading its block...
through a vectorized load, sorts it efficiently in the register space through few high throughput operations with high degree of instruction-level parallelism.

Therefore, our main contributions in this work are the following. First, a highly optimized GPU-based approach to CAVLC implemented in CUDA. Second, comparison of our implementation with the only existing state-of-the-art GPGPU implementation [38, 39]. An exhaustive experimental evaluation showed that our solution is between 2.5× and 5.4× faster than the state-of-the-art implementation [38, 39].

The rest of the paper is organized as follows. Sections 2 and 3 give background for CAVLC and the state-of-the-art GPU-based implementation of CAVLC [38, 39], respectively. Section 4 presents CAVLCU. Section 5 shows the experimental evaluation of our algorithm and a comparison to the state-of-the-art solution [38, 39]. Section 6 presents applications of CAVLC. Finally, the main conclusions are stated in Sect. 7.

2 Context-adaptive variable length coding (CAVLC)

CAVLC (Context-Adaptive Variable Length Coding) [14, 32] is a high efficient entropy method for encoding the quantized transform coefficients in video and image compression. In this technique, different sets of variable-length codes are chosen depending on already encoded syntax elements. Since the variable-length codes are designed to match the corresponding conditioned statistics, the entropy coding performance is improved by 5-10% in comparison to prior standards designs (like MPEG, H.261/3) using a single variable-length code.

CAVLC is one of the two entropy methods in H.264 [14], the most widely used video coding standard [29]. The alternative is CABAC (Context-Adaptive Binary Arithmetic Coding) [32], a method of arithmetic coding in which the probability models are updated based on previous coding statistics. Compared to CABAC, CAVLC has lower compression efficiency, but higher coding speed and lower complexity. Thus, it is widely used in low-delay, ‘conversational’ applications such as video conferencing, with relatively low computational requirements. Moreover, CAVLC is supported in all H.264 profiles, unlike CABAC which is not supported in baseline and extended profiles.

Next, Subsection 2.1 gives a detailed description of CAVLC algorithm, and Subsection 2.2 presents an example to clarify its operation.

2.1 CAVLC algorithm

CAVLC operates on blocks of 4×4 and 2×2 coefficients. It follows the steps presented in Algorithm 1 for encoding a block [32]. First, as shown in Fig. 1 for a 4×4 block, the coefficients are scanned in zigzag order. The resulting array will be referred to as zigzag array in the rest of the paper. Then, CAVLC constructs the output bitstream by concatenating a series of binary variable length codes (VLCs) assigned to the following data elements (symbols) of the zigzag array: CoeffToken,


Fig. 1 Example of zigzag scan for a 4x4 block

trailing ones, levels, TotalZeros and runs. In the next subsections, we define the referenced symbols and describe how they are encoded.



Algorithm 1: CAVLC block encoding. The operator & represents binary concatenation

```
zigzag array ← scan the coefficients of the block in zigzag order
encoding ← VLC of symbol CoeffToken of zigzag array
&
   Bit signs of trailing ones of zigzag array
&
   VLCS of levels of zigzag array
&
   VLC of symbol TotalZeros of zigzag array
&
   VLCS of runs of zigzag array
```


2.1.1 CoeffToken

The magnitude of nonzero coefficients tends to be larger at the start of the zigzag array, near the first coefficient, and smaller towards the higher frequencies. In addition, the absolute value of the last nonzero coefficients often equals to 1. The last up to three -1 or +1 coefficients are referred to as trailing ones (T1s), while the remaining nonzero coefficients as levels. The symbol CoeffToken (coefficient token) represents both the total number of nonzero coefficients (TotalCoeff) and the number of trailing ones (NumT1s).

The VLC assigned to CoeffToken is obtained from a lookup table that, in the case of a 4 x 4 block, is chosen from three VLC tables and one 6-bit fixed length code table, whose contents are specified in Table 9-5 of the H.264 standard [14]. An extract of this latter is shown in Table 1. As it can be seen, the choice of the
lookup table is done in function of a parameter $nC$, which is calculated from the number of coefficients in the blocks to the left and above of the current block (parameters $nA$ and $nB$, respectively). This implies that the lookup table selection is context adaptive. Figure 2 illustrates the relationship between a block and its neighbours. The parameter $nC$ is calculated as shown in Table 2, where $\gg$ indicates binary right shift. The availability of each neighbouring block is determined by its existence and its membership in the same slice of the current block.

**Table 1** Extract of the table 9-5 of the H.264 standard

| NumT1s | TotalCoeff | $0 \leq nC < 2$ | $2 \leq nC < 4$ | $4 \leq nC < 8$ | $8 \leq nC$ |
|--------|------------|-----------------|-----------------|-----------------|-----------------|
| 0      | 0          | 1               | 11              | 1111            | 0000 11         |
| 0      | 1          | 0001 01         | 0010 11         | 0011 11         | 0000 00         |
| 2      | 5          | 0000 0010 1     | 0000 101        | 0100 1          | 0100 10         |
| 3      | 5          | 0000 100        | 0011 0          | 1010            | 0100 11         |
| 2      | 16         | 0000 0000 0000 0101 | 0000 0000 0001 01 | 0000 0000 11 | 1111 10 |
| 3      | 16         | 0000 0000 0000 1000 | 0000 0000 0001 00 | 0000 0000 10 | 1111 11 |

**Table 2** Calculation of parameter $nC$. The operator $\gg$ represents binary right shift

| Condition | $nC$         |
|-----------|--------------|
| Left and top blocks are available | $(nA + nB + 1) \gg 1$ |
| Only the left block is available | $nA$ |
| Only the top block is available | $nB$ |
| Neither neighbouring block is available | 0 |
2.1.2 Trailing ones

The T1s are encoded in reverse order with their sign bits (‘0’ for positive and ’1’ for negative).

2.1.3 Levels

The levels are encoded in reverse order with VLCs composed of a prefix and a possible suffix. The prefix is made up of a string of zero or more bits ‘0’ followed by a stop bit ’1’. The length of the suffix (SuffixLength) is between 0 and 6 in normal cases. If SuffixLength > 0, the last bit of the suffix stores the sign of the level.

Table 3 shows an extract of the seven VLC tables used for levels encoding in the H.264 baseline profile [11], each one corresponding to a different value of

| Level | VLC | Length |
|-------|-----|--------|
| +1    | 1   | 1      |
| −1    | 01  | 2      |
| +2    | 001 | 3      |
| −2    | 0001| 4      |
| ...   | ... | ...    |
| +7    | 00000000000001 | 13 |
| −7    | 00000000000001 | 14 |
| ± 8 to ± 15 | 0000000000000001xxxx | 19 |
| ≥ ±16 | 000000000000000001xxxxxxxxxxxx | 28 |

| Level | VLC | Length |
|-------|-----|--------|
| ± 1   | 1x  | 2      |
| ± 2   | 01x | 3      |
| ...   | ... | ...    |
| ± 5   | 00001x | 6   |
| ± 15  | 00000000000001x | 16  |
| ≥ ±16 | 0000000000000001xxxxxxxxxxxx | 28  |

| Level | VLC | Length |
|-------|-----|--------|
| ± 1 to ± 32 | 1xxxxxx | 7    |
| ± 33 to ± 64 | 01xxxxxx | 8    |
| ...   | ... | ...    |
| ± 449 to ± 480 | 000000000000001xxxxxx | 21  |
| ≥ ±481 | 000000000000000001xxxxxxxxxxxx | 28  |
SuffixLength. Table Lev-VLCT[0] is selected for SuffixLength = 0, Table Lev-VLCT[1] for SuffixLength = 1, and so on. Lev-VLCT[0] has its own structure while the remaining VLC tables share a common structure. In all cases, when the magnitude of the level is too large, its value is stored entirely in the suffix, whose length is set to 12. As the last bit represents the sign, the maximum magnitude that CAVLC can encode is $2^{11} = 2048$ in the baseline profile [11].

Algorithm 2 [14, 32] shows how the levels are encoded. The selection of each VLC table is context adaptive, as it depends on the magnitude of the previous coded level.

Algorithm 2: CAVLC levels encoding

```plaintext
if TotalCoeff > 16 and NumTIs < 3 then
    SuffixLength ← 1
else
    SuffixLength ← 0
end if

for each level in reverse order do
    effective_level ← level
    if level is the first to encode and NumTIs < 3 then
        subtract 1 to magnitude of effective_level
    end if
    encode effective_level using Lev-VLCT[SuffixLength]
    if SuffixLength = 0 then
        SuffixLength ← 1
    end if
    if SuffixLength < 6 then
        T ← 3 × $2^{\text{SuffixLength} - 1}$
        if magnitude(level) > T then
            SuffixLength ← SuffixLength + 1
        end if
    end if
end for
```

2.1.4 TotalZeros

The symbol TotalZeros is the sum of all zeros preceding the last nonzero coefficient in the zigzag array. The VLC assigned to TotalZeros is obtained from a lookup table that, in the case of a $4 \times 4$ block, is selected from 15 VLC tables, whose contents are specified in Tables 9-7 and 9-8 of the H.264 standard [14]. An extract of these tables is shown in Tables 4 and 5. As it can be seen, the choice of the lookup table is done in function of the symbol TotalCoeff. If TotalCoeff is 0 or 16, TotalZeros is not encoded because it is known that all coefficients are zero or nonzero, respectively.
Table 4  Extract of the table 9-7 of the H.264 standard

| TotalZeros | TotalCoeff |
|------------|------------|
| 0          | 1 111 0101 0001 1 0101 0000 01 0000 01 |
| 1          | 011 110 111 111 0100 0000 1 0000 1 |
| 2          | 010 101 110 0101 0011 111 101 |
| 14         | 0000 0001 0 0000 00 – – – – – – |
| 15         | 0000 0000 1 – – – – – – |

Table 5  Extract of the table 9-8 of the H.264 standard

| TotalZeros | TotalCoeff |
|------------|------------|
| 0          | 0000 01 0000 01 0000 1 0000 0000 00 00 0 |
| 1          | 0001 0000 00 0000 0 0001 0001 001 01 1 |
| 2          | 0000 1 0001 001 001 01 1 1 – |
| 7          | 001 0000 1 – – – – – – |
| 8          | 0000 00 – – – – – – |

2.1.5 Runs

The parameter run of a nonzero coefficient is defined as the sum of all consecutive zeros that precede it. The runs are encoded in reverse order using one of the 7 VLC tables specified in Table 9-10 of the H.264 standard [14], whose content is presented in Table 6. The selection of each VLC is done in function of the symbol run and a second parameter, called ZerosLeft, which is the number of zeros that remain to be encoded. ZerosLeft is initialized to TotalZeros and decreases as more runs are encoded. The runs encoding is finished in the following two cases: (1) All zeros have already been encoded. (2) The current nonzero coefficient is the last in the reverse order, which implies that the maximum value to be encoded is 14.
2.2 CAVLC example

In this subsection, we present an example of CAVLC encoding, corresponding to the 4x4 block of Fig. 1. As shown, the zigzag array is \{5, 1, 0, -1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0\}.

We assume that the two neighbouring blocks are available, and that the values of the parameters \(nA\) and \(nB\) are 4 and 6, respectively. Hence, \(nC = (4 + 6 + 1) >> 1 = 5\). As TotalCoeff = 5 and NumT1s = 3, the VLC assigned to CoeffToken is 1010 (see Table 1). Note that the last four nonzero coefficient have magnitude 1, but only the last three ones are taken into account.

The values of the T1s in reverse order are +1, +1, and −1. Therefore, the VLC assigned is 001.

The steps for levels encoding (see Algorithm 2 and Table 3) are the next:

1. TotalCoeff = 5 and NumT1s = 3, hence the condition \(\text{TotalCoeff} > 10\) and \(\text{NumT1s} < 3\) is not fulfilled and SuffixLength is initialized to 0.
2. The first level in the reverse order is +1. As the condition \(\text{NumT1s} < 3\) is not satisfied, the absolute value of the level is not decremented.
3. SuffixLength is 0; hence, on the one hand, Lev-VLCT[0] is selected and the VLC assigned to level +1 is 1. On the other hand, SuffixLength is assigned the value 1.
4. As SuffixLength is less than 6, on the one hand, the threshold \(T = 3 \times 2^{\text{SuffixLength}-1} = 3 \times 2^{1-1} = 3\). On the other hand, since magnitude(level) = 1 and \(T = 3\), the condition magnitude(level) > \(T\) is not fulfilled and SuffixLength is not incremented.
5. The last level to encode is +5. As SuffixLength is 1, Lev-VLCT[1] is selected and the VLC assigned to level is 000010.

TotalZeros is 2 and TotalCoeff is 5. Therefore, the value assigned to TotalZeros is 0011 (see Table 4).

The runs of coefficients 5, 1, −1, 1 and 1 are 0, 0, 1, 0 and 1, respectively. Their encoding is done as follows (see Table 6):

1. Initially, the value of ZerosLeft equals to TotalZeros, i.e., 2.
2. The first run in the reverse order is 1 and ZerosLeft is 2; therefore, the VLC assigned is 01 and the value of ZerosLeft changes to 1.
3. The second run is 0 and ZerosLeft is 1; hence, the VLC assigned is 1 and the value of ZerosLeft does not change.
4. The third run is 1 and ZerosLeft is 1, therefore the VLC assigned is 0 and the value of ZerosLeft changes to 0.
5. As all zeros have been reached, the runs encoding is finished.

Finally, taking into account the different encodings seen in this subsection, the resulting bitstream of our example is the following: 1010-001-1-000010-0011-01-1-0

3 Solution of Su et al.

The only state-of-the-art GPU-based implementation of CAVLC is the solution presented by Su et al. [38, 39], which was developed in CUDA. It satisfies the real-time processing for HDTV 720p and its throughput is 11.17 to 6.29 times higher than that of the published software encoders on DSP and multi-core platforms.

By profiling the instructions of CAVLC, Su et al. found the main factors that limit the potential of parallelism [38, 39], which are the context-based data dependence, the memory accessing dependence and the control dependence. The context-based data dependence is due to the self-adaptive feature of CAVLC. Since the value of the parameter nC depends on nA and nB, it is not possible to calculate the parameter nC of a block until the symbols TotalCoeff of the neighbouring left and top blocks have been calculated. The memory accessing dependence is caused by the inherently serial nature of variable length encoding. To determine the position of each VLC in the output bitstream it is necessary to know the lengths of the VLCs that precede it. Control dependence is caused by the existence of different processing paths in two layers: the frame layer and the block layer. In the first layer, the branches are due to the different frame types and the different components of a frame (luma DC, luma AC, chroma DC and chroma AC). In the second layer, the different processing paths are caused by the irregular characteristic of symbol data, such as whether sign trail is 1 or -1 and whether levels are zero or not.
In order to eliminate or weaken the dependencies described above, Su et al. divided the execution of CAVLC into four paths according to the four components of a frame, and the CAVLC pipeline of each path was divided into three stages: two scans, coding and lag packing.
3.1 Two scans

Two scans are employed to calculate the CAVLC symbols: a forward scan and a backward scan.

The forward scan aims at the quantized coefficients and the results include the symbols TotalCoeff and the zigzag arrays, as it is shown in Fig. 3. In this stage, each thread is assigned to deal with a block. In order to satisfy the requirement of coalesced access to global memory [22, 25], the shared memory is used as a buffer.

The backward scan is executed on the zigzag arrays generated in the first scanning and the results consist of the values of nC and the remaining CAVLC symbols (NumT1s, T1s, levels, TotalZeros and runs). In order to make better use of the local data, a frame is divided into several regions of 4×2 macroblocks. One thread-block calculates the values of nC of blocks in the same region, as it is shown in Fig. 4. The program first loads all data needed to the shared memory, then each thread visits nA and nB, where one symbol TotalCoeff can be used as either nA or nB.

3.2 Coding

For the sake of minimizing the performance loss of the target parallel CAVLC encoder due to control dependence, Su et al. proposed a component-based coding mechanism. In this method, the program codes the symbols frame by frame in order of luma DC, luma AC, chroma DC, chroma AC instead of processing the four components macroblock by macroblock. The coding method is very similar for the different types of blocks; the main difference is the use of specific lookup tables for each component. In addition, the lookup tables are firstly loaded to the shared memory to speed up the lookup operation. The configuration is similar to that of calculating the value of nC and the results are the encodings (bitstreams and bit-lengths) of each block. A memory unit of 26 short words is used to store the bitstream of a block. Figure 5 shows the organization of a thread-block for encoding the symbols.

![Diagram of thread-block organization](image-url)
In the vector of coded words, the grey areas represent the bitstreams of the encodings, while the white regions are the unused spaces.

### 3.3 Lag packing

Once all the blocks are encoded, parallel writing is executed. According to the lengths of the bitstreams, the output positions are obtained and a parallel packing is performed. Thus, it can not only eliminate the constraint of accessing dependence, but it also improves the performance of writing.

### 4 Efficient GPU-based implementation of CAVLC (CAVLCU)

In this section, we present CAVLCU, our parallel implementation of CAVLC on CUDA. It is also compared with Su et al. proposal so that the achieved performance improvement can be clearly established. Our solution is built using only one CUDA kernel that has been specifically designed for encoding the luma AC blocks of a frame. The method for the remaining types of blocks (luma DC, chroma DC and chroma AC) is essentially the same, with very few variations.

The inputs of CAVLCU are the following:

- **The coefficients of the frame.** They are provided in a vector of 16-bit integers \(d_{coeffs}\), whose layout is shown in Fig. 6 for SQCIF format (128×96). As it can be seen, the array is divided in as many subvectors as macroblocks (MBs) in the frame (48 in the case of SQCIF). Each MB subvector is divided into 16 subvectors, each one corresponding to a different block of the MB.
CAVLCU: an efficient GPU-based implementation of CAVLC

### From top to bottom)

At the same time, each MB subvector is divided into 16 subvectors, each one corresponding to a different block of the MB; the i-th subvector stores the 16 coefficients of the i-th 4×4 block of the MB. Both blocks and coefficients are provided in the raster scan order as well.

- **The prediction modes of the MBs.** They are supplied in a vector of 8-bit integers \(d_{MB\text{-}pred\text{-}modes}\), where the i-th element is assigned to the i-th MB of the frame.

- **The slice IDs of the MBs.** They are provided in a vector of 16-bit integers \(d_{MB\text{-}slices}\), where the i-th element is assigned to the i-th MB of the frame.

Similarly, the outputs of CAVLCU are the following:

**Fig. 7** Layout of CAVLCU output vector of CAVLC encodings (\(d\_enc\_words\)) for SQCIF format (128×96). The array is divided in as many subvectors of size 16 as blocks in the frame (768 in the case of SQCIF). The i-th subvector is used for storing the encoding of the i-th block of the frame. The grey areas correspond to the CAVLC encodings, whose lengths are variable.

**Fig. 8** Partition of a QCIF frame (176×144) in regions of size 4.
The encodings of the blocks. They are written in a vector of 32-bit integers, \(d_{\text{enc\_words}}\), where the \(i\)-th subvector of size 16 (\(BLK\_ENC\_SIZE\)) is used for storing the encoding of the \(i\)-th block of the frame, as Fig. 7 illustrates for SQCIF format.

The binary lengths of the encodings. They are stored in a vector of 16-bit integers, \(d_{\text{enc\_lens}}\), where the \(i\)-th element is assigned to the \(i\)-th block of the frame.

As illustrated for a QCIF frame (176x144) in Fig. 8, CAVLCU divides a frame into equally-sized groups of consecutive MBs in the raster scan order, which will be referred to as regions. The execution configuration of the kernel uses a one-dimensional grid with as many thread-blocks as regions in the frame (\(NUM\_REG\)): the \(i\)-th thread-block of the grid processes the \(i\)-th region of the frame. The dimensions of the thread-blocks are \(4\times4\times\text{REG\_SIZE}\), where \(\text{REG\_SIZE}\) is the number of MBs of each region. As it is shown in Fig. 9 for the region 7 of Fig. 8, the \(i\)-th MB of the

---

| MB 0      | MB 1      |
|-----------|-----------|
| Blk 0     | Blk 0     |
| Th(0,0,0) | Th(0,0,0) |
| Blk 1     | Blk 1     |
| Th(1,0,0) | Th(1,0,0) |
| Blk 2     | Blk 2     |
| Th(2,0,0) | Th(2,0,0) |
| Blk 3     | Blk 3     |
| Th(3,0,0) | Th(3,0,0) |
| Blk 4     | Blk 4     |
| Th(0,1,0) | Th(0,1,0) |
| Blk 5     | Blk 5     |
| Th(1,0,1) | Th(1,0,1) |
| Blk 6     | Blk 6     |
| Th(2,0,1) | Th(2,0,1) |
| Blk 7     | Blk 7     |
| Th(3,0,1) | Th(3,0,1) |

| MB 2      | MB 3      |
|-----------|-----------|
| Blk 0     | Blk 0     |
| Th(0,0,0) | Th(0,0,0) |
| Blk 1     | Blk 1     |
| Th(1,0,0) | Th(1,0,0) |
| Blk 2     | Blk 2     |
| Th(2,0,0) | Th(2,0,0) |
| Blk 3     | Blk 3     |
| Th(3,0,0) | Th(3,0,0) |
| Blk 4     | Blk 4     |
| Th(0,0,1) | Th(0,0,1) |
| Blk 5     | Blk 5     |
| Th(1,0,1) | Th(1,0,1) |
| Blk 6     | Blk 6     |
| Th(2,0,1) | Th(2,0,1) |
| Blk 7     | Blk 7     |
| Th(3,0,1) | Th(3,0,1) |

**Fig. 9** Mapping of region data to elements of a thread-block in the case of region 7 of Fig. 8. The first MB of the region is assigned to the first half-warp of the thread-block, the second MB to the second half-warp, and so on. For each MB, the first block is assigned to the first thread of the corresponding half-warp, the second block to the second thread, and so on.

- **The encodings of the blocks.** They are written in a vector of 32-bit integers, \(d_{\text{enc\_words}}\), where the \(i\)-th subvector of size 16 (\(BLK\_ENC\_SIZE\)) is used for storing the encoding of the \(i\)-th block of the frame, as Fig. 7 illustrates for SQCIF format.
- **The binary lengths of the encodings.** They are stored in a vector of 16-bit integers, \(d_{\text{enc\_lens}}\), where the \(i\)-th element is assigned to the \(i\)-th block of the frame.

As illustrated for a QCIF frame (176x144) in Fig. 8, CAVLCU divides a frame into equally-sized groups of consecutive MBs in the raster scan order, which will be referred to as regions. The execution configuration of the kernel uses a one-dimensional grid with as many thread-blocks as regions in the frame (\(NUM\_REG\)): the \(i\)-th thread-block of the grid processes the \(i\)-th region of the frame. The dimensions of the thread-blocks are \(4\times4\times\text{REG\_SIZE}\), where \(\text{REG\_SIZE}\) is the number of MBs of each region. As it is shown in Fig. 9 for the region 7 of Fig. 8, the \(i\)-th MB of the
region is assigned to the i-th half-warp of the thread-block, and the i-th block of a MB is encoded by the i-th thread of the corresponding half-warp.

Algorithm 3 shows the pseudocode of CAVLCU kernel. The parameters NUM_MB, NUM_BLK and NUM_COEFF represent, respectively, the number of MBs, 4×4 blocks and coefficients of the frame; on the other hand, NUM_WORD_ENC is the number of 32-bit words used for storing the CAVLC encodings of the frame, whose value is the product of NUM_BLK by BLK_ENC_SIZE. Each thread performs the next steps. First, it calculates the indexes of the block to be encoded and the MB to which the block belongs. Second, it reads the coefficients of the block, and the prediction mode and slice ID of the MB. Third, it sorts the block in zigzag order to get the zigzag array. Fourth, it calculates a set of symbols from the zigzag array. Fifth, it calculates the parameter nC. Sixth, it uses the symbols and the parameter nC to encode the block.

**Algorithm 3: CAVLCU algorithm**

```c
kernel CAVLCU // outputs
d_enc_words[NUM_WORD_ENC],
d_enc_lens[NUM_BLK],
// inputs
d_coeffs[NUM_COEFF],
d_MB_pred_modes[NUM_MB],
d_MB_slices[NUM_MB]

// Calculate the block and MB indexes
blk_idx ← calculate_thread_ID()
MB_idx ← blk_idx / 16

// Read the coefficients of the block
longlong4 block_vec ← ((longlong4 *)d_coeffs)[blk_idx]

// Read the prediction mode and the slice ID of the MB
MB_pred_mode ← d_MB_pred_modes[MB_idx]
MB_slice ← d_MB_slices[MB_idx]

// Sort the block in zigzag order
short zz_array[16]
sort_in_zigzag_order(zz_array, block_vec)

// Calculate a set of symbols from the zigzag array
calculate_symbols // outputs
TotalCoeff, NumTis, Tis, ZigzagLevelsMask, ZigzagArrayMask,
// inputs
zz_array, MB_pred_mode

nC ← manage_nC(TotalCoeff, MB_slice, blk_idx, MB_idx)

encode_block // outputs
d_enc_words, d_enc_lens,
// inputs
blk_idx,
TotalCoeff, NumTis, Tis,
ZigzagLevelsMask, ZigzagArrayMask,
nC

end kernel
```
4.1 Calculation of block and MB indexes

As the i-th block of the frame is processed by the i-th thread of the grid, the index of the block equals to the thread ID in the grid, whose value is the following:

\[
\text{blockIdx.x} \times (\text{blockDim.x} \times \text{blockDim.y} \times \text{blockDim.z}) + \\
+ \text{threadIdx.z} \times \text{blockDim.x} \times \text{blockDim.y} + \\
+ \text{threadIdx.y} \times \text{blockDim.x} + \text{threadIdx.x}
\]

Since blockDim.x = blockDim.y = 4 and blockDim.z = REG_SIZE, the index of the block is calculated using the following expression:

\[
16 \times \text{REG\_SIZE} \times \text{blockIdx.x} + 16 \times \text{threadIdx.z} + \\
+ 4 \times \text{threadIdx.y} + \text{threadIdx.x}
\]

As a MB is composed of 16 blocks, the index of the MB is obtained by dividing the block index by 16.

4.2 Coefficients reading

Each thread reads the 16 coefficients of its block through one vectorized access using the built-in vector type \text{longlong4} [25], whose definition is shown in Algorithm 4. Since the sizes of types long long int and short are 8 and 2 bytes, respectively, each member of the variable block_vec contains 4 coefficients of the current block; as shown in Table 7, the member \text{x} contains the first 4 coefficients, the member \text{y} the next 4, and so on. Vectorized loads are an important CUDA optimization because they increase bandwidth and reduce both instruction count and latency [18].

In contrast, the solution of Su et al. uses the shared memory as a buffer to fulfill the requirement of coalesced global memory accesses recommended in CUDA literature [22, 25]. Since the maximum amount of shared memory per multiprocessor is 48 KB for GPUs with compute capability less than 3.7 [24] and the size of a block is 32 bytes, the occupancy is penalized in these architectures. For example, if the number of threads per thread-block is 128, the theoretical occupancy is reduced to 75% (3.x) or 67% (2.x) [24].

| Table 7 Mapping of block coefficients to longlong4 members |
|----------------------------------------------------------|
| Bits | Member 0–15 | 16–31 | 32–47 | 48–63 |
|------|-------------|-------|-------|-------|
| \text{x} | Coeff. 0 | Coeff. 1 | Coeff. 2 | Coeff. 3 |
| \text{y} | Coeff. 4 | Coeff. 5 | Coeff. 6 | Coeff. 7 |
| \text{z} | Coeff. 8 | Coeff. 9 | Coeff. 10 | Coeff. 11 |
| \text{w} | Coeff. 12 | Coeff. 13 | Coeff. 14 | Coeff. 15 |
4.3 Zigzag sorting

The coefficients of a block are extracted from the variable block_vec and are written in the private array zz_array in zigzag order. This operation is based on the mapping shown in Table 7. The performance of this operation is high for the following reasons. First, there are no dependencies between the different coefficient extractions; hence, the degree of instruction-level parallelism is high. Second, each coefficient extraction is performed with few operations of high throughput (two binary shifts and a cast). Third, zz_array is placed in register space [22] because (1) it is small; (2) it is indexed with constant quantities, and (3) the kernel does not use more registers than available.

The solution of Su et al., after loading the blocks in shared memory, write them back to global memory in zigzag order; the coefficients will be read later again for calculating the CAVLC symbols. Conversely, CAVLCU executes the zigzag sorting in a much more efficient way, as it only consists of few high throughput operations with high degree of ILP reading and writing in the register space and saving costly memory global accesses.

4.4 Calculation of the symbols

Algorithm 5 shows the pseudocode for calculating the following symbols, which will be used for encoding the current block:

- The CAVLC symbols TotalCoeff, NumT1s and T1s.
- A 16-bit binary mask (ZigzagArrayMask) which represents the structure of the zigzag array and hence implicitly the CAVLC symbols TotalZeros and runs. If the i-th coefficient of the zigzag array is non-zero, the i-th most significant bit of the mask is 1; otherwise, this bit is 0. In the example of Fig. 1, the value of the mask is 1101101000000000.
- A second 16-bit binary mask (ZigzagLevelsMask) which represents the structure of the zigzag array excluding the trailing ones. In the example of Fig. 1, the value of the mask is 1100000000000000.
Algorithm 5: Calculation of symbols

```c
#define MASK_OF_COEFF(coeff_idx) (1 << (15 - coeff_idx))

function calculate_symbols() // outputs
    TotalCoeff, NumTIs, TIs,
    ZigzagLevelsMask, ZigzagArrayMask,
    // inputs
    zz_array[16], MB_pred_mode

    // Initialize symbols
    NumTIs ← 0
    TIs ← 0
    ZigzagLevelsMask ← 0
    ZigzagArrayMask ← 0

    // Process AC coefficients
    for coeff_idx = 15 to 1 step -1 do
        if zz_array(coeff_idx) <> 0 then
            update_symbols() // outputs
                NumTIs, TIs,
                ZigzagLevelsMask, ZigzagArrayMask,
                // inputs
                zz_array(coeff_idx),
                MASK_OF_COEFF(coeff_idx)
        end if
    end for

    if MB_pred_mode = INTRA_16x16 then
        ZigzagArrayMask ← (ZigzagArrayMask << 1)
    else
        // Process DC coefficient
        if zz_array[0] <> 0 then
            update_symbols() // outputs
                NumTIs, TIs,
                ZigzagLevelsMask, ZigzagArrayMask,
                // inputs
                zz_array[0],
                MASK_OF_COEFF(0)
        end if
    end if

    TotalCoeff ← __popc(ZigzagBlock_Mask)
end function
```
Each thread performs the next steps. First, it initializes all the symbols to 0. Second, for each nonzero AC coefficient stored in zz_array (i.e., all but the first), from the last to the first, it updates all the symbols except TotalCoeff performing the steps presented in Algorithm 6. Third, if the prediction mode of the current MB is not Intra 16\times16, it processes the DC coefficient (i.e., the first) in the same way as in step 2. Otherwise, it ignores the DC coefficient and left-shift the symbol ZigzagArrayMask one bit, as only the subblock formed by the AC coefficients must be considered. Fourth, it calculates TotalCoeff from ZigzagArrayMask using the CUDA function \texttt{__popc} [23], which counts the number of bits that are set to 1 in a 32 bit integer. The throughput of \texttt{__popc} is high as it compiles to a single instruction [25].

In the solution of Su et al., each thread iterates two times over the coefficients of a block for calculating its CAVLC symbols: TotalCoeff in the first iteration and the remaining ones (NumT1s, T1s, levels, TotalZeros, runs) in the second. All the symbols are written in global memory and later read for transferring them between the corresponding kernels. CAVLCU optimizes significantly this process for the following reasons. First, it iterates only one time over the coefficients of a block for calculating the necessary symbols. Second, the number of symbols processed in the loop is reduced to only 4 integers: NumT1s, T1s, ZigzagLevelsMask and ZigzagArrayMask. Third, as shown in Algorithm 6, the update of the symbols in each loop iteration is performed very efficiently, as it only requires two OR operations for the symbols ZigzagLevelsMask and ZigzagArrayMask, an addition for NumT1s and a binary left shift, an OR operation and a comparison for T1s. Fourth, our algorithm saves read/write global memory operations performed by the solution of Su et al. as transferring symbols between kernels is not required.

**Algorithm 6: Symbols updating from a non-zero coefficient**

```plaintext
function update_symbols(/!
\texttt{outputs}
NumT1s, T1s,
ZigzagLevelsMask, ZigzagArrayMask,
\texttt{/!
\texttt{inputs}
coeff, mask_of_coeff
}
if abs(coeff) = 1 and NumT1s < 3 and ZigzagLevelsMask = 0 then
  // The coefficient is a trailing one
  T1s ← (T1s << 1) | (coeff = 1 ? 0 : 1)
  NumT1s ← NumT1s + 1
else
  // The coefficient is a level
  ZigzagLevelsMask ← ZigzagLevelsMask | mask_of_coeff
end if
ZigzagArrayMask ← ZigzagArrayMask | mask_of_coeff
end function
```
4.5 Calculation of parameter $n_C$

According to the method described in Subsection 2.1.1, each thread calculates the parameter $n_C$ of its block from the information associated to the left and top neighbouring blocks ($info_A$ and $info_B$, respectively). Each block information is composed of the symbol TotalCoeff and the slice ID. The symbols TotalCoeff of the left and top blocks are the parameters $n_A$ and $n_B$, respectively. The slice ID of the left and top blocks will be denoted as $SliceID_A$ and $SliceID_B$, respectively.

Each thread $(x, y, z)$ gets $info_A$ as follows. If the current block is not in the first column of its MB, $n_A$ is read from the left thread $(x - 1, y, z)$ using the CUDA function $\_\_shfl\_up$ [25], as shown in Fig. 10. As both left and current block are in the same MB, $SliceID_A$ is the slice ID of the current MB. If the current block is in the first column of the first MB of a region, $info_A$ is read from an intermediate array in global memory ($d\_info_A$) of dimensions $\text{NUM\_REG} \times 4$. As illustrated in Fig. 11, each thread $(0, y, 0)$ of a thread-block $i$ reads $info_A$ from the element $d\_info_A[i - 1][y]$, which is written by the thread $(3, y, \text{REG\_SIZE} - 1)$ of the thread-block $i - 1$. If the current block is in the first column of the second or posterior MB of a region, $info_A$ is read from an intermediate array in shared memory ($s\_info_A$) of dimensions $\text{REG\_SIZE} \times 4$. As illustrated in Fig. 12, each thread $(0, y, z)$ with $z > 0$ of a thread-block reads $info_A$ from the element $s\_info_A[z - 1][y]$, which is written by the thread $(3, y, z - 1)$.

![Fig. 10](image-url) Reading of parameter $n_A$ using the CUDA function $\_\_shfl\_up$

![Fig. 11](image-url) Transmission of parameter $n_A$ through global memory
In a similar way, each thread \((x, y, z)\) gets \(\text{info}_B\) as follows. If the current block is not in the first row of its MB, \(n_B\) is read from the top thread \((x, y - 1, z)\) using the CUDA function \(\_\_\text{shfl\_up}\) [25], as shown in Fig. 13. As both top and current block are in the same MB, \(\text{SliceID}_B\) is the slice ID of the current MB. If the current block is in the first row of its MB, \(\text{info}_B\) is read from an intermediate array in global memory \((d\_\text{info}_B)\) of dimensions \(\text{NUM\_MB\_VER} \times \text{NUM\_MB\_HOR} \times 4\), where \(\text{NUM\_MB\_VER}\) and \(\text{NUM\_MB\_HOR}\) are the number of MBs in the vertical and horizontal dimensions of the frame, respectively. As illustrated in Fig. 14, each thread \((x, 0, z)\) of a thread-block reads \(\text{info}_B\) from the element \(d\_\text{info}_B[r - 1][c][x]\), where \(r\) and \(c\) are, respectively, the row and the column of the current MB. Each element \(d\_\text{info}_B[r][c][x]\) is written by the thread \((x, 3, z)\) of the half-warp that processes the MB in row \(r\) and column \(c\).

Algorithm 7 shows the pseudocode for managing the parameter \(n_C\). Each thread performs the next steps. First, it represents the necessary information of the current block (the symbol TotalCoeff and the slice ID of its MB) in a compact way using a 32-bit integer \((\text{info})\), where the 5 least significant bits store TotalCoeff, the sixth
least significant bit is set to 1, which ensures info is nonzero, and the 16 most significant bits store the slice ID. Second, it calculates the indexes of the row and the column of the MB in the frame. Third, if proceeds, writes info in the intermediate arrays as described above. Fourth, it synchronizes with other threads of the block for ensuring the array s_info_A contains the correct values. Fifth, it gets nA and nB as explained above. If a neighbouring block is unavailable, the corresponding reading function (read_nA or read_nB) returns −1. Sixth, it calculates nC from nA and nB using the method shown in Table 2.
As in our previous works [7, 8], the thread-block synchronization mechanism proposed by Yan et al. [47] is used for synchronizing the reads with the writes in global memory. In this case, it is applied on both horizontal (d_info_A) and vertical (d_info_B) dimensions and the reads are performed using atomic operations. The elements of d_info_A and d_info_B are initialized to 0 statically. Since all the values written are nonzero (due to the fact that the sixth least significant bit is set to 1), the read of each element is performed executing the CUDA atomic function \texttt{atomicExch} repeatedly until a nonzero value is returned. Additionally, the use of this function restores the stored value to 0, which allows subsequent uses of the intermediate arrays in global memory, and avoids getting old cached values.

As static initialization of variables in shared memory is illegal in CUDA, a different synchronization mechanism is used in the accesses to s_info_A. In this case, the CUDA intrinsic function \texttt{__syncthreads()} guarantees that each element is not read until its value has been written. On the other hand, the use of the keyword \texttt{volatile} in the declaration of the array s_info_A ensures any reference to this variable compiles to an actual memory read or write instruction [25].

CAVLCU reduces significantly the number of global memory accesses with respect to the solution of Su et al. for the following reasons. First, in our solution, each thread-block, on the one hand, only writes in global memory the symbols TotalCoeff of the last column and the last row of its region and, consequently, on the other hand, only reads from global memory the parameters nA and nB of the first column and the first row, respectively. In contrast, the solution of Su et al. writes all the symbols TotalCoeff of the frame in global memory and each thread-block not only reads from global memory the parameters nA and nB of the first column and

\begin{algorithm}
\caption{Management of parameter nC}
\begin{algorithmic}
\Function{manage_nC}{TotalCoeff, MB_slice, blk_idx, MB_idx}
\State info $\leftarrow$ (MB_slice $\ll$ 16) + 64 + TotalCoeff
\State MB_row $\leftarrow$ MB_idx / \text{NUM}_\text{MB}_\text{HOR}
\State MB_col $\leftarrow$ MB_idx $\%$ \text{NUM}_\text{MB}_\text{HOR}
\State \text{write_info}_B$\text{of_right_MB}$(info, MB\_row, MB\_col)
\State \text{write_info}_A$\text{of_right_MB}$(info, MB\_col)
\State _\text{syncthreads}()
\State nB $\leftarrow$ \text{read_nB}(TotalCoeff, MB\_slice, MB\_row, MB\_col)
\State nA $\leftarrow$ \text{read_nA}(MB\_col, TotalCoeff, MB\_slice)
\If{nA $= -1$ and nB $= -1$}
\State nC $\leftarrow$ nB
\ElsIf{nB $= -1$}
\State nC $\leftarrow$ nA
\ElsIf{nA $= -1$}
\State nC $\leftarrow$ nB
\Else
\State nC $\leftarrow$ (nA + nB + 1) $\gg$ 1
\EndIf
\State return nC
\EndFunction
\end{algorithmic}
\end{algorithm}

\[\text{ Springer}\]
the first row of its region but also all the symbols TotalCoeff of the region. Second, in the approach of Su et al., once the parameters nC are calculated, they are written in global memory to be read in the coding stage. Therefore, CAVLCU saves two operations in global memory for writing and reading all the parameters nC of a frame.

4.6 Block encoding

The first action of this stage is to call the CUDA warp synchronization function \texttt{syncwarp()} [25] to force reconvergence. This prevents the independent thread scheduling of modern architectures (Volta and later) from increasing the number of global memory writes.

Each thread \(i\) of the grid writes the encoding of its block in the subvector \(i\) of BLK_ENC_SIZE elements \((d_{\text{blk\_enc}})\) of \(d_{\text{enc\_words}}\) (see Fig. 7), and the bit-length of the encoding in the element \(i\) of \(d_{\text{enc\_lens}}\).

The block encoding is constructed in the way specified in Sect. 2. As the VLCs assigned to the CAVLC symbols are obtained, their bits are concatenated in a 32-bit variable \((\text{word\_val})\) and their lengths added in a second 32-bit variable \((\text{word\_len})\) 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 \(d_{\text{blk\_enc}}\), and the value and length of the remaining encoding are stored in \(\text{word\_val}\) and \(\text{word\_len}\), respectively. The process continues until all the VLCs are written. The bit-length of the encoding is written in the element \(i\) of \(d_{\text{enc\_lens}}\). Its value is calculated using the following expression, where word_idx is the index of the last accessed position of \(d_{\text{blk\_enc}}\):

\[
\text{word\_idx} \times 32 + \text{word\_len}
\]

The lookup tables of the symbols CoeffToken, TotalZeros and runs are stored in arrays in global memory, which are initialized statically. The base type is the CUDA intrinsic vector type \texttt{uchar2} [25], whose definition is presented in Algorithm 4. The members x and y represent, respectively, the bit-length and the value of a variable-length code. The CUDA function \texttt{__ldg} [25] is used for caching the reads in the read-only data cache [22]. In contrast, the solution of Su et al. uses the shared memory for caching the lookup tables. Both memory systems have a small latency but the use of the read-only data cache saves a synchronization barrier and additional instructions for caching the lookup tables programatically.

The VLCs of the levels are calculated using the method for encoding levels without lookup tables presented by Hoffman et al. [11]. The values of the levels are extracted from the variable \(\text{block\_vec}\) using the positions stored in the symbol ZigzagLevelsMask.

The symbols TotalZeros and runs are obtained from the positions in reverse order of the nonzero coefficients in the zigzag array. As illustrated in Table 8 for the example of Fig. 1, the coefficients positions are 0 for the last coefficient, 1 for the penultimate coefficient, and so on. In the case of TotalZeros the following expression is used:
CAVLCU: an efficient GPU-based implementation of CAVLC

TotalCoeff - last_coeff_pos where last_coeff_pos is the position of the last nonzero coefficient. In our example, TotalCoeff is 5 and last_coeff_pos is 9; therefore, TotalZeros is 16 - 5 - 9 = 2. The symbol run associated to each nonzero coefficient is calculated using the next expression:

prev_coeff_pos - coeff_pos - 1

where coeff_pos and prev_coeff_pos are the positions of the current and immediately previous nonzero coefficients, respectively. Table 9 shows the calculation of the runs for our example.

The positions of the nonzero coefficients are obtained from ZigzagArrayMask and ZigzagLevelsMask in reverse order (i.e., from the last nonzero coefficient to the first) calling the CUDA function __ffs [23], which finds the position of the least significant bit set to 1 in a 32 bit integer. After each function call, the last bit of the corresponding mask is set to 0 using the next expression, where ~ and << are the bitwise operators AND, NOT and left shift, respectively,

ZigzagArrayMask & ~((1 << (coeff_pos - 1))

5 Experimental evaluation

We have evaluated CAVLCU and compared it to the only existing state-of-the-art GPGPU implementation of CAVLC, which is the solution proposed by Su et al. [38, 39]. It will be referred to as CAVLC_SU in this section. We implemented CAVLC_SU from scratch following the description of the algorithm given by their authors [38, 39] and their support through private communication with Huayou Su.

We used two GPUs to test the algorithms, a GeForce GTX 970 (Maxwell architecture with compute capability 5.2) and a GeForce RTX 2080 (Turing architecture with compute capability 7.5).

In order to compare CAVLCU with CAVLC_SU, we measured the execution time, the number of global transactions and the number of executed instructions for the first 50 frames of the video sequences City (QCIF), Mother and Daughter (CIF), and Ducks take off (720p) [44]. Each test was performed with a GOP length of 10 and for 11 values of the quantization parameter (QP) between 0 and 50. The number of threads per thread-block was 128 in all cases; hence, the value of the parameter REG_SIZE of CAVLCU was 8.

Figures 15, 16 and 17 present the execution times in milliseconds and Table 10 the minimum, maximum and average values of CAVLCU speedup with respect to CAVLC_SU. As it can be seen, the results on both Maxwell and Turing architectures showed that our algorithm clearly outperforms the solution of Su et al. [38,
Table 9 Calculation of symbols Runs of zigzag array of Fig. 1. The parameters coeff_pos and prev_coeff_pos represent the positions of the current and immediately previous nonzero coefficients, respectively.

| Coefficient | coeff_pos | prev_coeff_pos | Run          |
|-------------|-----------|----------------|--------------|
| 1           | 9         | 11             | 11−9−1 = 1   |
| 1           | 11        | 12             | 12−11−1 = 0  |
| -1          | 12        | 14             | 14−12−1 = 1  |

Table 10 Execution time speedup of CAVLCU with respect to CAVLC_SU

| Video clip          | Maxwell GPU | Turing GPU |
|---------------------|-------------|------------|
|                     | Minimum     | Maximum    | Average    | Minimum     | Maximum    | Average    |
| City                | 2.7         | 3.6        | 3.3        | 4.2         | 6.2        | 5.2        |
| Mother and Daughter | 2.5         | 3.5        | 3.3        | 3.9         | 5.4        | 4.8        |
| Ducks take off      | 3.3         | 5.4        | 4.1        | 3.0         | 6.7        | 5.1        |

Fig. 15 Execution Time (ms) versus Quality Parameter (QP) for video clip “City” (QCIF)

39], since CAVLCU is between 2.5 and 5.4 faster than CAVLC_SU on the first architecture and between 3.0 and 6.7 on the second.

As the main improvement of our implementation is given by the reduction of global memory access, Table 11 compares the number of these memory operations for CAVLCU and CAVLC_SU. It shows that our implementation reduces in a 75.70% the number of global memory transactions in Maxwell architecture and a 65.86% in Turing architecture. Thus, since CAVLCU is built using only one kernel, it saves many of the CAVLC_SU global memory accesses required

Springer
CAVLCU: an efficient GPU-based implementation of CAVLC

for communicating intermediate results among kernels for the following reasons. First, the forward scan of CAVLC_SU reads all the coefficients of the frame from global memory and write them back ordered in zigzag; later, the coefficients are read again by the backward scan. CAVLCU reads the coefficients only once (in an efficient way through vectorized accesses) and does not need to write them back. Second, the forward scan of CAVLC_SU writes the symbols TotalCoeff in global memory for its posterior reading by the backward scan; in a similar way, the backward scan writes the remaining symbols in global memory for its posterior reading in the coding stage. In contrast, CAVLCU holds all the symbols in the register space and does not need to process them in global memory. Third,
Table 11  Number of global transactions and executed instructions

| Video clip          | Solution | Maxwell GPU | Turing GPU |
|---------------------|----------|-------------|------------|
|                     |          | Global Load Transac. | Global Store Transac. | Executed Instructions | Global Load Transac. | Global Store Transac. | Executed Instructions |
| City                | CAVLC_SU | 22,479,009  | 5,108,650  | 84,451,824 | 7,001,064  | 5,108,650  | 84,782,589 |
|                     | CAVLCU   | 5,208,773   | 1,489,900  | 36,641,924 | 2,731,105  | 1,489,900  | 38,980,702 |
|                     | Improvement | 4.32        | 3.43       | 2.30        | 2.56        | 3.43       | 2.17        |
| Mother and Daughter | CAVLC_SU | 71,176,225  | 15,051,190 | 235,433,090 | 21,833,529  | 15,051,190 | 235,679,200 |
|                     | CAVLCU   | 16,723,445  | 4,683,296  | 96,217,749  | 8,558,844  | 4,683,296  | 102,257,746 |
|                     | Improvement | 4.26        | 3.21       | 2.45        | 2.55        | 3.21       | 2.30        |
| Ducks take off      | CAVLC_SU | 719,933,101 | 181,420,862 | 2,754,450,830 | 224,018,729 | 181,420,862 | 2,767,884,867 |
|                     | CAVLCU   | 163,912,274 | 54,654,524  | 1,226,364,124 | 83,021,412  | 54,654,524  | 1,301,713,005 |
|                     | Improvement | 4.39        | 3.32       | 2.25        | 2.70        | 3.32       | 2.13        |
the backward scan of CAVLC_SU writes all the parameters nC in global memory for its posterior reading in the coding stage. In contrast, CAVLCU uses the thread-block synchronization mechanism of Yan et al. [47] for transmitting only the parameters nC that are strictly necessary in an efficient way.

As seen in Table 11, CAVLCU improves the number of executed instructions by a factor higher than 2, due to the greater simplicity of our algorithm, which uses vectorized loads for reading the blocks, saves intermediate results transmission among kernels and, unlike CAVLC_SU, does not cache the lookup tables in shared memory.

6 CAVLC applications

Over the years, many adaptations of CAVLC have been proposed in different fields, like data encryption [19, 40–42] and information hiding [16, 17, 45, 46].

Mian et al. [19] proposed a technique which consists in encrypting codeword indexes and looking up codeword tables to determine the new codewords according to the encrypted indexes. They embedded encryption into the process of encoding TotalCoeffs, 4×4 block TotalZeros, chroma DC 2×2 block TotalZeros and runs. Experiments showed that the algorithm is able to provide compromise between security and complexity, and has little effect on compression performance.

Wang et al. [42] demonstrated that two fast selective encryption methods for CAVLC and CABAC [34–36] are not as efficient as only encrypting the sign bits of nonzero coefficients. As a much stronger scrambling effect can be achieved encrypting the sign bits of intra prediction modes and motion vectors, they proposed a tunable encryption method based on these three ways of encryption. Experiments showed that this method has null or very little impact on compression performance. It can run in real-time and its computational cost is minimal. It is secure against the replacement attack when all three control factors are set to one.

Tabash and Izharuddin [40] presented a technique based on Baker’s map, a two-dimensional chaotic map, which is used to design a pseudorandom number generator (PRNG). The proposed PRNG is used to encrypt the sign of transformed coefficients, the codewords of runs and the pattern of trailing ones. Experiments showed good encryption results, where the visual information was successfully encrypted. The proposed method is secure against common attacks and has low computational requirements.

Kim et al. [16] proposed a fragile watermarking scheme where the hidden information is embedded in the first sign bit of the CAVLC trailing ones encodings. The bitrate of the watermarked video remains the same and the PSNR is higher than 43 dB.

Liao et al. [17] presented an information hiding algorithm which follows the next steps: (1) Generate random sequences based on chaotic maps to select the block positions for embedding the data. (2) Assign the i-th hidden bit to the parity of the number of trailing ones of the i-th block, which implies, when appropriate, setting the last trailing one to 0 or adding a one-value coefficient after the last nonzero coefficient. This method has low computationally complexity and, hence, can be
real-time realized. Experiments showed that the degradation of video quality is negligible and the same overall size of the video bit-stream is maintained.

Xu et al. [45] presented a scheme for data hiding directly in the encrypted H.264/AVC video bitstreams. The codewords of three sensitive parts (intraprediction modes, motion vector differences and levels) are encrypted with stream ciphers. Then, additional data may be embedded in the encrypted domain (specifically in levels codewords suffixes whose length is greater than one) without knowing the original video content. Data extraction can be done either in the encrypted domain or in the decrypted domain. In addition, experimental results showed that the file size is preserved and that the degradation in video quality caused by data hiding is quite small. In [46], Xu et al. proposed an improved version of their scheme that can achieve higher embedding capacity. Specifically, when the level suffix length is equal to 1, data embedding is performed by paired code-word substitution; when the level suffix length is greater than 2, the multiple-based notational system is adopted.

In addition to its use in video coding, CAVLC has many interesting applications and great possibilities in other areas of video and image compression, like medical image compression [20, 30, 37].

Sridhar et al. [37] proposed an advanced medical image compression technique based on integer DCT (Digital Cosine Transform), SPIHT (Set Partitioning In Hierarchical Trees) and CAVLC. Simulations on different medical images (including CT skull, angiogram and MR images) showed better results compared to JPEG and JPEG2000 schemes.

Mohanty et al. [20] presented a framework to stream histopathology image of a patient over a lossy network. Firstly, the image is divided into a number of fixed size tiles to provide access to regions of interest to the remote pathologist. Secondly, each tile is compressed using a proposed variant of WebP. Finally, a proposed greedy scheme packs macroblocks in such a way that that the number of undecodable received macroblocks is minimized. Although JPEG and JPEG2000 have been used to compress histopathology images, the authors selected WebP because the size of a file compressed by the former methods is 25%-34% more than that of the same file compressed by the last method [9, 10]. Nevertheless, they observed that the FCFS (First Come First Serve) inter-macroblock dependency introduced by WebP is not suitable to stream histopathology images because it cannot prioritize the decoding of an important macroblock. Hence, they modified WebP by using CAVLC in place of CABAC encoder.

Priya et al. [30] proposed a region-based compression method for compressing medical images in DICOM (Digital Imaging and Communications in Medicine) format. Their method consists of the following steps: (1) Using fuzzy C-means clustering, the image is segmented in regions of interest (ROI) and non-regions of interest (NROI). (2) The NROI and ROI areas are compressed using, respectively, CAVLC and a lossless compression method based on DWT (Discrete Wavelet Transform) and SPIHT. (3) The outputs of CAVLC and the lossless compression method are merged to get the compressed image. Experiments results showed that the presented method outperforms, in terms of PSNR (peak signal-to-noise ratio), SSIM (structural similarity index measure) and CR (compression ratio), the conventional
methods EZW (Embedded Zerotrees of Wavelet), STW (Spatial-orientation Tree Wavelet) and SPIHT.

7 Conclusions

This work has presented CAVLCU, a highly optimized GPU-based approach to CAVLC implemented in CUDA, which improves the only state-of-the-art implementation on GPU.

Thus, our algorithm outperforms the throughput of previous implementation by applying several optimization strategies. On the one hand, CAVLCU is built using only one kernel to avoid the long latency global memory accesses required to transmit intermediate results among different kernels, and the costly launches and terminations of additional kernels. On the other hand, our algorithm applies thread-block synchronization mechanism to manage efficiently the data dependence between thread-blocks in the calculation of the parameters nC. Moreover, CAVLCU optimizes the zigzag sorting of the blocks, as, after their reading through vectorized loads, sort them efficiently in the register space through few high throughput operations with high degree of instruction-level parallelism.

Experimental evaluation showed that CAVLCU is between 2.5 and 5.4 faster than the unique state-of-the-art GPU-based implementation.

We believe that our work is very useful for the following reasons. First, our algorithm is a significantly improved alternative to the only existing GPU-based solution. Second, our method can be exploited as the CAVLC component in GPU-based H.264 encoders, which are a very suitable solution when GPU built-in H.264 hardware encoders lack certain necessary functionality, such as data encryption and information hiding. Third, as CAVLC is a high-performance entropy compression method, apart from its wide use in the video standard H.264, it can be applied in many other compression systems. Hence, taking into account the massive use of multimedia data compression in the current digital era, our solution can be exploited in the development of many GPU-based applications for encoding both images and videos in formats other than H.264, like medical images. This is not possible with hardware implementations of CAVLC, as they are non-separable components of hardware H.264 encoders.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.
References

1. Babionitakis K, Doumenis G, Georgakarakos G, Lentaris G, Nakos K, Reisis D, Sifnaios I, Vlassopoulos N (2008) A real-time H. 264/AVC VLSI encoder architecture. J Real-Time Image Process 3(1-2):43–59
2. Banerji A, Ghosh AM (2010) Multimedia Technologies. Tata McGraw Hill, New Delhi
3. Chang C W, Lin W H, Yu H C, Fan CP (2014) A high throughput CAVLC architecture design with two-path parallel coefficients procedure for digital cinema 4K resolution H. 264/AVC encoding. In: Circuits and Systems (ISCAS), 2014 IEEE International Symposium on (pp. 2616-2619). IEEE
4. Chu X, Wu S, Chang F, He W (2012) Efficient implementation of the CAVLC entropy encoder based on FPGA [J]. J Xidian Univ 3:017
5. Damak T, Werda I, Samet A, Masmoudi N (2008) DSP CAVLC implementation and optimization for H. 264/AVC baseline encoder. In: Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on (pp. 45-48). IEEE
6. El-Ghobashy WA, Ebian M, Mowafi O, Zekry AA (2015) An Efficient Implementation Method of H. 264 CAVLC video coding using FPGA. In: Computer Engineering Conference (ICENCO), 2015 11th International (pp. 212-216). IEEE
7. 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 (pp. 1-6). IEEE
8. Gómez-Luna J, Chang LW, Sung IJ, Hwu WM, Guil N (2015) In-Place Data Sliding Algorithms for Many-Core Architectures. In: Parallel Processing (ICPP), 2015 44th International Conference on (pp. 210-219). IEEE
9. Google Inc. WebP compression study. Draft 0.1 (May 2011). https://developers.google.com/speed/webp/docs/webp\_study
10. Google Inc. Comparative study of WebP, JPEG, and JPEG2000 (August 2012). https://developers.google.com/speed/webp/docs/c\_study
11. Hoffman MP, Balster EJ, Scarpono F, Hill K (2011) An Efficient Software Implementation of the CAVLC Encoder for H.264/AVC. In: Proceedings of the 2011 IEEE National Aerospace and Electronics Conference (NAECON), Dayton, OH., pp. 333-337
12. Hoffman MP, Balster EJ, Turri WF (2016) High-throughput CAVLC architecture for real-time H. 264 coding using reconfigurable devices. J Real-Time Image Process 11(1):75–82
13. Hsia SC, Liao WH (2010) Forward computations for context-adaptive variable-length coding design. IEEE Trans Circ Syst II Exp Briefs 57(8):637–641
14. ITU-T Recommendation H.264 (2019) Advanced video coding for generic audiovisual services
15. Khronos group: OpenCL (2020). https://www.khronos.org/opencl/
16. Kim SM, Kim SB, Hong Y, Won CS (2007) Data Hiding on H. 264/AVC Compressed Video. In: International Conference Image Analysis and Recognition (pp. 698-707). Springer, Berlin, Heidelberg
17. Liao K, Lian S, Guo Z, Wang J (2012) Efficient information hiding in H 264/AVC video coding. Telecommun Syst 49(2):261–269
18. Luitjens J (2013) CUDA Pro Tip: Increase Performance with Vectorized Memory Access, Dec. https://devblogs.nvidia.com/cuda/pro-tip-increase-performance-with-vectorized-memory-access/
19. Mian C, Jia J, Lei Y (2007) An H. 264 Video Encryption Algorithm Based on Entropy Coding. In: Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IH-MSP 2007) (Vol. 2, pp. 41-44). IEEE
20. Mohanty M, Ooi W T (2012) Histopathology Image Streaming. In: Pacific-Rim Conference on Multimedia (pp. 534-545). Springer, Berlin, Heidelberg
21. Mukherjee R, Banerjee A, Maulik A, Chakrabarty I, Dutta PK, Ray AK (2017) An Efficient VLSI Design of CAVLC Encoder. In: Region 10 Conference, TENCON 2017-2017 IEEE (pp. 805-810). IEEE
22. NVIDIA: CUDA C Best Practices Guide 11.0 (2020). https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
23. NVIDIA: CUDA Math API (2020) https://docs.nvidia.com/cuda/cuda-math-api/index.html
24. NVIDIA: CUDA Occupancy Calculator (2020) https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA\_Occupancy\_Calculator.xls
25. NVIDIA: CUDA C Programming Guide 11.0 (2020) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
26. NVIDIA: CUDA Zone (2020) https://developer.nvidia.com/category/zone/cuda-zone
27. NVIDIA: NVENC Video Encoder API Programming Guide (2020) https://docs.nvidia.com/video-technologies/video-codec-sdk/nvenc-video-encoder-api-prog-guide/index.html
28. Orlanidic M, Svarstad K (2017) An efficient hardware architecture of CAVLC encoder based on stream processing. Microelectron J 67:43–49
29. Ozer J (2016) Encoding for Multiple Screen Delivery. Udemy
30. Priya C, Ramya C (2018) Medical image compression based on fuzzy segmentation. Int J Pure Appl Math 118(20):603–610
31. Ren J, He Y, Wu W, Wen M, Wu N, Zhang C (2009) Software parallel CAVLC encoder based on stream processing. In: Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on (pp. 126-133). IEEE
32. Richardson, Iain EG (2010) The H.264 Advanced Video Compression Standard, Wiley: Hoboken
33. Salomon D, Motta G (2010) Handbook of data compression. Springer, New York
34. Shahid Z, Chaumont M, Puech W (2009) Fast Protection of H. 264/AVC by Selective Encryption of CAVLC and CABAC. In: 2009 IEEE International Conference on Multimedia and Expo (pp. 1038-1041). IEEE
35. Shahid Z, Chaumont M, Puech W (2009) Fast protection of H. 264/AVC by selective encryption. In: Proceedings Of The Singaporean-French Ipal Symposium 2009: SinFra’09 (pp. 11-21)
36. Shahid Z, Chaumont M, Puech W (2011) Fast protection of H 264/AVC by selective encryption of CAVLC and CABAC for I and P frames. IEEE Trans Circ Syst Video Technol 21(5):565–576
37. Sridhar KV, Prasad KK (2008) Medical Image Compression Using Advanced Coding Technique. In: 2008 9th International Conference on Signal Processing (pp. 2142-2145). IEEE
38. Su H, Chen M, Wu N, Ren J, Zhang C (2014) Efficient parallel video processing techniques on GPU: from framework to implementation. The Sci World J
39. Su H, Zhang C, Chai J, Wen M, Wu N, Ren J, A High-Efficient Software Parallel CAVLC Encoder Based on GPU. In: 2011 34th International Conference on Telecommunications and Signal Processing (TSP), Budapest, 2011, pp. 534-540
40. Tabash FK, Izharuddin M (2017) Efficient Encryption Technique for H. 264/AVC Based on CAVLC and Baker’s Map. In: 2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI) (pp. 2759-2764). IEEE
41. Tabash FK, Izharuddin M, Tabash MI (2019) Encryption techniques for H. 264/AVC videos: a literature review. J Inform Secur Appl 45:20-34
42. Wang Y, O’Neill M, Kurugollu F (2013) A tunable encryption scheme and analysis of fast selective encryption for CAVLC and CABAC in H. 264/AVC. IEEE Trans Circ Syst Video Technol 23(9):1476–1490
43. Xiao Z, Baas B (2008) A High-Performance Parallel CAVLC Encoder on a Fine-Grained Many-Core System. In: Computer Design. ICCD 2008. In: IEEE International Conference on (pp. 248-254). IEEE
44. Xiph.org Video Test Media [derf’s collection] (2020). https://media.xiph.org/video/derf/
45. Xu D, Wang R, Shi YQ (2014) Data hiding in encrypted H. 264/AVC video streams by codeword substitution. IEEE Trans Inform Foren Secur 9(4):596–606
46. Xu D, Wang R, Shi YQ (2016) An improved scheme for data hiding in encrypted H. 264/AVC videos. J Vis Commun Image Rep 36:229–242
47. Yan S, Long G, Zhang Y (2013) StreamScan: fast scan algorithms for GPUs without global barrier synchronization. ACM Sigplan Notices 48(8):229–238

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

Antonio Fuentes-Alventosa¹ · Juan Gómez-Luna² · José Maria González-Linares³ · Nicolás Guil³ · R. Medina-Carnicer¹

Antonio Fuentes-Alventosa
antonio.fa@gmail.com

Juan Gómez-Luna
juang@ethz.ch

José Maria González-Linares
jgl@uma.es

Nicolás Guil
nguil@uma.es

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
³ Department of Computer Architecture, University of Málaga, Málaga, Spain