View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional Universidad de Málaga

CUVLE: Variable-Length Encoding on CUDA

Antonio Fuentes-Alventosa
Department of Computer Architecture and Electronics,
University of Córdoba,
Spain
antonio.fa@gmail.com

Juan Gómez-Luna
Department of Computer Architecture and Electronics,
University of Córdoba,
Spain
el1goluj@uco.es

José Mª González-Linares
Department of Computer Architecture,
University of Málaga,
Spain
jgl@uma.es

Nicolás Guil
Department of Computer Architecture,
University of Málaga,
Spain
nguil@uma.es

***Abstract*—Data compression is the process of representing information in a compact form, in order to reduce the storage requirements and, hence, communication bandwidth. It has been one of the critical enabling technologies for the ongoing digital multimedia revolution for decades. In the variable-length encoding (VLE) compression method, most frequently occurring symbols are replaced by codes with shorter lengths. As it is a common strategy in many compression applications, efficient parallel implementations of VLE are very desirable. In this paper we present CUVLE, a GPU implementation of VLE on CUDA. Our approach is on average more than 20 and 2 times faster than the corresponding CPU serial implementation and the only known state-of-the-art GPU implementation, respectively.**

Keywords—data compression; variable-length encoding; Huffman coding; CUDA; GPU

I. INTRODUCTION

Data compression is the science of representing the information in a compact form [1]. It is one of the most important topics responsible for developments in multimedia. Our need for high definition video on the desktop or high quality music stored in a tiny device, or even transmission of multimedia data in real time would not have been met without digital compression techniques. Data compression has applications in many computer science areas, like video scene analysis, where fast algorithms for detecting scene changes and flashlight scenes directly on compressed video have been proposed [4].

Variable-length encoding (VLE) [2] is a popular compression method in which mostly used symbols are assigned with codes of shorter length, whereas rarely used symbols are assigned with codes of longer length. VLE is a common strategy in many compression applications. Thus, efficient codes for VLE are required to compute as fast as possible all these applications.

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 [9]. Hundreds of industry-leading applications are already GPU-accelerated [10]. However, data compression has been largely unaffected. To the best of our knowledge, the unique existing GPGPU implementation of VLE is PAVLE [14], presented by A. Balevic.

PAVLE 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 codes are stored in a look-up table, which is cached in the on-chip *shared* memory. As GPU architectures provide more efficient support for 32-bit data types, the source and compressed data are provided and written, respectively, in vectors of 32-bit unsigned integers. Consecutive threads load consecutive segments of elements from the source vector. Each thread uses the code table for encoding the loaded segment in its private memory and calculating the corresponding bit-length. An intra-block scan primitive is performed to calculate the bit-positions of the thread encodings on the basis of its bit-lengths. The threads of a block write concurrently its 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 source segment processed by the block. Therefore, after PAVLE execution, it is necessary to compact the content of the output vector.

In this paper we present CUVLE, a new implementation of VLE on CUDA-enabled NVIDIA GPUs. As in the case of PAVLE, the table with the codes is cached in shared memory and consecutive threads of a block process consecutive source segments. However, our approach uses the following optimization strategies:

- *Persistent blocks* [11], which equals the grid size to the maximum number of resident blocks, thereby minimizing the number of loads of the codes table in shared memory.
- *Contiguous writing of block encodings in global memory*. Our approach writes the block encodings in their correct positions in the output vector from the beginning, thereby avoiding the necessity of running any compaction algorithm.
- *Direct writing of block encodings in global memory*. As 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.

The previous optimization techniques allowed us to implement a solution for variable-length encoding on modern GPGPUs which is on average more than 2 and 20 times faster than PAVLE and the corresponding CPU serial implementation, respectively.

The rest of the paper is organized as follows. Section II gives background for data compression and CUDA. Section III presents CUVLE and describes its implementation details. Section IV shows the experimental evaluation of our algorithm and a comparison to the CPU serial implementation and PAVLE. Finally, the main conclusions are stated.

II. BACKGROUND

A. Data compression

Data compression is, in the context of computer science, the science (and art) of representing the information in a compact form [1].

The compression techniques were developed taking into account three primary problems of computers in the early days [2]. These are (a) limited memory, (b) costly storage capacity, and (c) processing capability limitations. Although developments in hardware have vastly improved computing power, our demands for processing multimedia data have also increased simultaneously. Our need for high definition video on the desktop or high quality music stored in a tiny device, or even transmission of multimedia data in real time would not have been met without digital compression techniques. With the emergency of computer networking and Internet, data compression becomes essential to reduce cost and delay of transmission.

Data compression is used in many areas of the computer science, like the field of video scene analysis. Several rapid algorithms [4] for detecting scene changes and flashlight scenes directly on compressed video have been proposed. These algorithms operate on the dc sequence which can be readily extracted from video compressed using Motion JPEG or MPEG without full-frame decompression. The dc images occupy only a small fraction of the original data size while retaining most of the essential “global” information.

Broadly, data compression can be of two types: lossless and lossy. In lossless data compression method the original data is reconstructed exactly from the compressed data after a reverse process called decompression. This means that no information is lost in the process of compression. This is opposite to lossy data compression method. Lossless compression is used when it is important that the original and the decompressed data have to remain exactly identical. This is the case, for example, of executable programs, source codes and textual documents. Image file formats like PNG use only lossless compression, while others like TIFF may use either lossless or lossy methods. Lossless data compression is used in the popular ZIP software compression tool.

Variable-length encoding (VLE) [2] is a lossless compression method in which mostly used symbols are assigned with codes of shorter length, whereas rarely used symbols are assigned with codes of longer length. As it is a common strategy in many compression applications, efficient codes for VLE are required to compute as fast as possible all these applications. Once the symbol character set and their probabilities become known, the code lengths can be decided. David Huffman created in 1951 and algorithm for systematically generating variable length codes for a given source [3]. For this task a binary tree is created using the symbols as leaves according to their probabilities and paths of those are taken as the codes. The method starts with as many trees as there are symbols. While there is more than one tree, it finds the two trees with the smallest total weight and combines them into one, setting one as the left child and the other as right. Once the tree contains all the symbols, it assigns 0’s and 1’s (left child represents ‘0’ and right child ‘1’).

B. CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA [5]. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

CUDA was developed with several design goals in mind:

- Provide a small set of extensions to standard programming languages, like C, that enable a straightforward implementation of parallel algorithms. With CUDA C/C++, programmers can focus on the task of parallelization of the algorithms rather than spending time on their implementation.
- Support heterogeneous computation where applications use both the CPU and GPU. Serial portions of applications are run on the CPU, and parallel portions are offloaded to the GPU. As such, CUDA can be incrementally applied to existing applications. The CPU and GPU are treated as separate devices that have their own memory spaces. This configuration also allows simultaneous computation on the CPU and GPU without contention for memory resources.

A CUDA program invokes parallel functions called kernels that execute across many parallel threads [6]. These threads are organized into thread blocks and grids of thread blocks. Each thread within a thread block executes and instance of the kernel. Each thread also has thread and block IDs within its thread block and grid, a program counter, registers, per-thread private local memory, inputs, and output results.

A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private local memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in global memory space after kernel-wide global synchronization.

CUDA’s hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM on Fermi / SMX on Kepler) executes one or more thread blocks; and CUDA cores and other execution units in the SMX execute thread instructions. The SMX executes threads in groups of 32 threads

called warps. While programmers can generally ignore warp execution for functional correctness and focus on programming individual scalar threads, they can greatly improve performance by having threads in a warp execute the same code paths and access memory with nearby addresses.

III. CUVLE

In this section we describe CUVLE, our algorithm for variable-length encoding on CUDA-enabled GPUs.

CUVLE uses an encoding alphabet of 256 symbols, which are unsigned bytes (0, 1 ... 255). The variable-length codes are provided in a table (*VLET*), which is implemented by two vectors of 256 elements: one that stores the values of the codes (*VLET_val*) and the other their bit-lengths (*VLET_len*). The element *i* of each vector contains the value/bit-length of the code assigned to the symbol *i*.

The source data are provided in an *input vector* of 32-bit unsigned integers. Thus, every element of the vector contains 4 symbols, since these are represented with 8 bits each. The compressed data are written in an *output vector* of 32-bit unsigned integers too.

The input vector is conceptually partitioned into segments of B × NET elements called *block-inputs*, where *B* is the number of threads of a block and *NET* is an integer value. Each block-input is processed by a block and the encoding result, which will be referred to as *block-code*, is written in the output vector. Figure 1 illustrates this inter-block mechanism.

Consecutive threads of a block process consecutive segments of NET elements called *thread-inputs*. The result of encoding a thread-input will be referred to as *thread-code*. Figure 2 shows an example of this intra-block mechanism.

Figure 3 presents the CUVLE algorithm. The first action performed by each block is caching the VLET in shared memory. Then, while there are block-inputs to be encoded, each block repeats the next steps: first, it gets the index of the first available block-input; second, each thread of the block encodes its corresponding thread-input in its private memory; third, the block calculates the bit-positions in the output vector of the thread-codes; finally, each thread writes its corresponding thread-code in the output vector.

A. VLET caching

While encoding a block-input, the VLET is used intensively for searching the values and lengths of codes in the vectors VLET_val and VLET_len, respectively. As a thread reads NET elements of a block-input and each element contains 4 symbols, each thread of a warp executes 4 × NET concurrent accesses to each one of the two VLET vectors. These are random accesses, as they depend on the source data. Therefore, the VLET caching is a very important requirement to avoid bottlenecks during the algorithm execution.

In order to perform the searches as fast as possible, CUVLE uses the fast on-chip memory on the GPU for caching the VLET. The function *cacheVLETInSharedMemory* copies (in a fully coalesced way) the vectors of global memory h_VLET_val and h_VLET_len to the vectors of shared memory s_VLET_val and s_VLET_len, respectively.

B. Persistent blocks

The straightforward way of choosing the execution configuration of the kernel would be setting as many blocks as block-inputs, so that each block processes the block-input of the same index. However, CUVLE, in order to minimize the number of VLET loads in shared memory, applies a different strategy, called *persistent blocks* [11], which equals the grid size to the maximum number of resident blocks (*MRB*).

MRB is calculated by multiplying the maximum number of resident blocks per multiprocessor (*MRBM*) by the number of multiprocessors. MRBM can be carefully calculated by taking several parameters into consideration: the maximum number of threads per multiprocessor, the shared memory needs per thread block, and the register usage per thread.

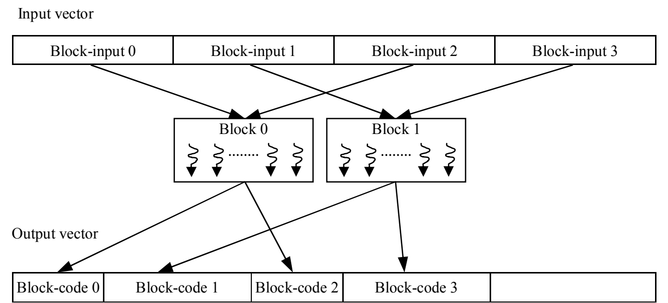

Fig. 1. Example of CUVLE basic inter-block mechanism for an input vector of 4 block-inputs and a grid of 2 blocks. The block 0 processes the block-inputs 0 and 2 and writes the corresponding block-codes 0 and 2 in the output vector. The block 1 performs the same actions with the block-inputs 1 and 3.

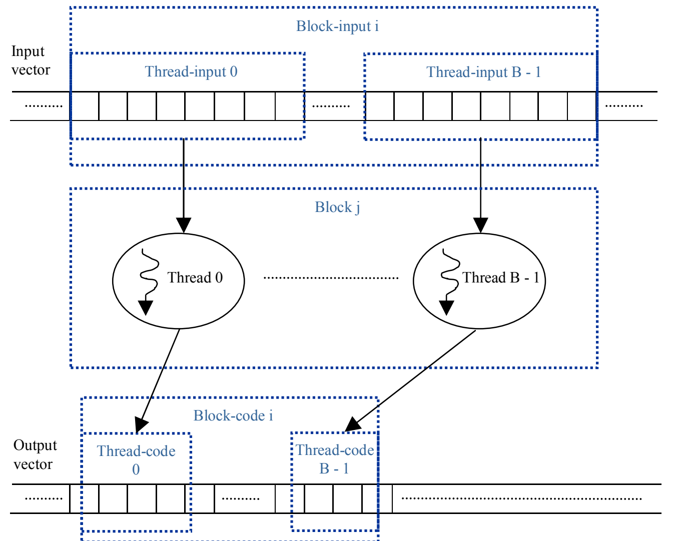

Fig. 2. Example of CUVLE basic intra-block mechanism for NET = 8. The block *j* processes the block-input *i* and writes the corresponding block-code *i* in the output vector. Each thread *k* of the block *j* encodes the thread-input *k* and writes the thread-code *k* in the output vector.

```
// Declaration of arrays in shared memory for VLET caching
__shared__ uint s_VLET_val[ALPHABET_SIZE];
__shared__ uint s_VLET_len[ALPHABET_SIZE];

// Declaration of array in shared memory for storing the
// bit-lengths of the thread-codes belonging to a block-code
__shared__ uint s_lens[BLOCK_SIZE];

// VLET caching
cacheVLETInSharedMemory(s_VLET_val, s_VLET_len,
                        h_VLET_val, h_VLET_len);

// Get the index of the first block-input to be encoded
uint blockinput_idx = blockIdx.x;

// While there are block-inputs to be encoded...
while (blockinput_idx < number_of_blockinputs){
   // Encode the thread-input assigned to the thread in the
   // current block-input
   uint threadcode[4 × NET];
   uint threadcode_len;
   encodeThreadInput(threadcode, threadcode_len,
                     h_input, blockinput_idx,
                     s_VLET_val, s_VLET_len);

   // Calculate the bit-positions of the thread-codes belonging
   // to the current block-code
   s_lens[threadIdx.x] = threadcode_len;
   uint threadcode_bitpos = calculateBitPosOfThreadCodes(s_lens);

   // Write the thread-code to the output vector
   writeThreadCode(h_output, threadcode_bitpos,
                   threadcode, threadcode_len);

   // Get the index of the next block-input to be encoded
   blockinput_idx += gridDim.x;
}
```

Fig. 3. CUVLE algorithm.

C. Thread-input reading and encoding

A thread calls the function *encodeThreadInput* to encode a thread-input in its private memory. For each of the NET elements of a thread-input, a thread loads it from global memory and for each of its 4 bytes, from the most significant to the least: first, searches in the VLET the value and the bit-length of the associated code; second, updates the array *threadcode* and the variable *threadcode_len* with the read values. The codes are concatenated in *threadcode* and the bit-lengths accumulated in *threadcode_len*.

The size of *threadcode* is 4 × NET because a thread-code is composed by the concatenation of 4 × NET codes and the maximal bit-length of a code assumed by CUVLE is 32 bits (the size of an unsigned integer).

If NET > 1, the accesses of a warp for reading a thread-input are not coalesced, as CUDA literature [7, 8] recommends for maximizing the performance. However, they satisfy the principle of spatial locality because consecutive threads of a warp read consecutive thread-inputs. Therefore, the transparent cache hierarchy of Fermi and Kepler ensures a good performance while reading the input vector. In Fermi architecture, CUVLE must utilize both L1 and L2 caches to achieve the best performance. L1 should be used because the accesses fulfill the principle of spatial locality and its line size (128 bytes) is larger than the one of L2 (32 bytes). In Kepler architecture, accesses to global memory are cached only in L2 (L1 is reserved for local memory accesses) [6] but this memory is improved, as it offers up to 2x of the bandwidth per clock available in Fermi.

D. Calculation of the bit-positions of thread-codes in the output vector

The function *calculateBitPosOfThreadCodes* calculates the output bit-positions of the thread-codes that belong to a block-code. It operates across all the threads of a block combining the efficient *intra-block scan* algorithm of S. Sengupta et al. [12] with the *adjacent block synchronization* mechanism proposed by S.Yan et al. [13].

The function utilizes an intermediate array *I* in global memory, with as many elements as number of block-inputs (or, equivalently, block-codes). This vector is used for storing the bit-positions of the block-codes in the output vector with the exception of the first block-code, whose bit-position is zero.

The bit-position of a thread-code in the output vector (*pos-tc-out*) is calculated using the following expression:

$$pos\text{-}tc\text{-}out = pos\text{-}tc\text{-}bc + pos\text{-}bc\text{-}out$$

where *pos-tc-bc* and *pos-bc-out* are the bit-positions of the thread-code in its block-code and of the block-code in the output vector, respectively.

- The values of the parameter *pos-tc-bc* for the thread-codes of a block-code are calculated applying the scan operation on the basis of their bit-lengths.
- Given a block-input j, if it is the first ($j = 0$) *pos-bc-out* is zero. Otherwise, its value is read from $I[j - 1]$.

Let us see how the values of *I* are written. At the beginning, all its elements are initialized to zero. When the bit-length of a block-code j is calculated (through the scan operation):

- If $j = 0$, the bit-length is stored in $I[0]$.
- If $j > 0$, a particular thread of the block:
  1. Reads continuously the element $I[j - 1]$ until its value is nonzero. The readings are carried out using atomic operations to avoid getting old cached values.
  2. Adds the value read in the previous step to the bit-length of the block-code.
  3. Stores the value calculated in the previous step in $I[j]$.

E. Writing the thread-codes in the output vector

A thread calls the function *writeThreadCode* to write the corresponding thread-code in the output vector at the bit-position calculated in the previous step. The number of elements of the output vector to update depends on the bit-length and the bit-position of the thread-code. As the second to penultimate elements are exclusively written by the thread, the standard store operation is used for it. However, as the first and ultimate elements are generally edited by more than one thread, each one fits safely its specific bits using atomic OR operations. Note that all the elements of the output vector must be initialized to zero.

As a thread-code is composed by a sequence of 4 × NET codes and the minimum bit-length of a code is 1 bit, if NET ≥ 8, the minimum bit-length of a thread-code is ≥ 32 bits, the size of the elements of the output vector. This fact has the following implications:

- The first element of the output vector updated by each thread is different.
- Each element of the output vector is updated at most by two threads of the grid (the last bits of the first thread-code and the first bits of the second thread-code).

Assuming NET ≥ 8, let us analyze when position conflicts (i.e., two threads colliding while accessing the same memory location) may appear between two consecutive threads *t1* and *t2*, whose thread-codes are *tc1* and *tc2*, respectively, and the need for atomic operations:

- *The threads belong to the same warp*. As a warp is a SIMD unit, it executes in lock-step, so it is not possible a collision because *t1* and *t2* do not write the second element of *tc1* and *tc2*, respectively, until they have written the first element, which means that when *t1* is going to write the last element of *tc1*, *t2* has already written the first element of *tc2* previously.
- *The threads belong to different warps*. In this case, there may be collisions, as warps execute independently and in a non-predictable order. However, the probability of collision is low, given that warp schedulers typically alternate warps in a round-robin fashion.

Taking into account our previous analysis let us see (assuming NET ≥ 8) how a thread can write its thread-code using only two atomic operations per warp and minimizing the number of OR operations (atomic or not):

- *First element of the output vector to update*. If the thread is the first of a warp, it writes the corresponding bits performing an atomic OR operation with the value stored in the target position, preventing, in this way, collisions with other thread that updates the same element. Hereafter, we will call the second thread the *colliding thread*. Otherwise, it directly stores the corresponding value in the target position because the colliding thread (the immediately preceding) belongs to the same warp and so, as we have just seen, they cannot collide. For example, in Figure 2, thread 0 fits the bits of the first output element using an atomic OR operation. In contrast, threads 1 to 31 directly write the bits of their first target positions.
- *Second to penultimate elements*. Their contents are directly stored, as they are exclusively updated by the thread. In Figure 2, threads 0 and B - 1 directly write the values of the second to fourth and of the second to third target elements, respectively.
- *Last element*. If the thread is the last of a warp, it updates the target position performing an atomic OR operation for preventing collisions. Otherwise, it executes a non-atomic OR operation because the colliding thread (the immediately subsequent) belongs to the same warp and they cannot collide. In Figure 2, thread B - 1 fits the bits of the last output element using an atomic OR operation. In contrast, the 31 immediately preceding threads execute a non-atomic OR operation to fit the bits of their last target positions.

In order to eliminate the warp divergences caused by the evaluation of conditions in the updating of the first and last target positions, CUVLE simplifies the writing of a thread-code as follows:

- *First and last target positions*: performing an atomic OR operation.
- *Second to penultimate target positions*: directly storing their values.

IV. Experimental Evaluation

We evaluated CUVLE and compared it to PAVLE and a CPU serial implementation, which will be referred to as CPU-VLE in this section. The PAVLE and CPU-VLE implementations were obtained from the source code provided by A. Balevic [15]. CPU-VLE, as our approach, assumes a maximal code bit-length of 32 bits.

Our test machine had a 2.67Ghz Intel Core i7 920 CPU and 12 GB of RAM. The GPUs that we utilized were a GeForce GT 640 2GB GDDR5 (Kepler architecture with compute capability 3.5) and a GeForce GTX 550 Ti (Fermi architecture with compute capability 2.1).

We used randomly-generated test files with values of entropy between 0 and 8 bits per symbol and sizes between 0.25 and 256 MB. The variable-length codes were generated using the Huffman method for the construction of minimum redundancy codes [3].

Tables I and II show the minimum, maximum and average values of CUVLE speedup with respect to the other algorithms using the Kepler and Fermi GPUs, respectively. In the case of PAVLE, we distinguish between the kernel execution time and the total encoding time. The latter takes into account the extra processing of compaction required on the output vector after PAVLE kernel completion. As it can be seen, the speedups were very similar in both architectures.

As expected, the best values of performance were obtained for NET >= 8, condition that guarantees the next points:

- There are not intra-warp position conflicts.
- The maximum degree of inter-warp position conflict is 2.
- The thread-codes can be written minimizing the number of OR operations.

More precisely, the best results were obtained using NET = 8 or NET = 16 for files with small sizes (less than or equal to 1MB) and NET = 16 for the rest. Therefore, the value NET = 16 always guarantees an efficient encoding.

Experimental evaluation showed that CUVLE is more than two times faster than PAVLE. The main reasons are the next:

TABLE I. CUVLE SPEEDUP USING THE KEPLER CARD.

Algorithm	Minimum	Maximum	Average
CPU-VLE	22.3	29.2	27.1
PAVLE (kernel)	1.6	2.2	2.0
PAVLE (total)	2.5	4.2	2.7

TABLE II. CUVLE SPEEDUP USING THE FERMI CARD.

Algorithm	Minimum	Maximum	Average
CPU-VLE	20.3	23.6	21.8
PAVLE (kernel)	1.7	1.9	1.8
PAVLE (total)	2.0	3.9	2.5

a) *Contiguous writing of block-codes in global memory*. PAVLE writes the block-codes in the output vector at the same positions of their corresponding block-inputs in the input vector. Consequently, the block-codes are not stored contiguously and, after completion of PAVLE, it is necessary the execution of a second algorithm to compact the content of the output vector using the bit-lengths of the block-codes, which are previously stored in global memory by PAVLE. CUVLE writes the block-codes in their correct positions in the output vector from the beginning, which avoids the necessity of running any compaction algorithm and, consequently, entails a great saving of time.

b) *Direct writing of block-codes in global memory*. PAVLE, first, writes a block-code to a temporary buffer in shared memory and, then, copies the content of the buffer to the output vector. As our approach writes directly a block-code in global memory:

- Saves the time to make additional operations of writing and reading in a buffer in shared memory, avoiding the corresponding data-dependent appearance of bank conflicts.
- Saves the required space for the temporary buffer, increasing the *occupancy*, which is the ratio between the number of active warps within a streaming multiprocessor and the maximum number of active warps.
- Benefits on the high performance of global atomic operations on modern GPGPUs architectures [6].

c) *Persistent blocks*. As CUVLE applies the persistent blocks strategy, the grid has many fewer blocks than that of PAVLE. So, the number of VLET copies from global to shared memory is much lower.

V. CONCLUSIONS

This work has presented CUVLE, a highly optimized approach to VLE on GPU. Our algorithm uses several optimization techniques that outperform the throughput of PAVLE, the unique parallel previous GPGPU implementation, to the best of our knowledge. CUVLE applies the persistent blocks strategy to carry out a very small number of VLET copies in shared memory. Our approach writes the codes in their correct positions from the beginning, which avoids the necessity of running any compaction algorithm. Moreover, the codes are written directly in global memory, as the performance of global atomic operations is high on modern GPGPUs architectures. Thus, the occupancy is increased by saving the reserved space for a write buffer in shared memory.

The experimental evaluation showed that CUVLE is a good and suitable alternative for variable-length encoding on modern GPGPUs, as it is more than 20 and 2 times faster than the corresponding CPU serial implementation and PAVLE, respectively.

ACKNOWLEDGMENT

We thank NVIDIA for a hardware donation to the University of Córdoba under CUDA Teaching Center 2014-2015 Awards, and the Junta de Andalucía of Spain for financial support (TIC-1692).

REFERENCES

[1] Ida Mengyi Pu, “Fundamental Data Compression”, Butterworth-Heinemann, 2005, pp. 1-6.

[2] Banerji, “Multimedia Technologies”, Tata McGraw-Hill Education, 2010, pp. 59-73.

[3] David A. Huffman. "A method for the construction of minimum-redundancy codes". Proceedings of the Institute of Radio Engineers, 40(9): 1098–1101, September 1952.

[4] Boon-Lock Yeo, "Bede Liu: Rapid scene analysis on compressed video", 1995, IEEE Transactions on Circuits and Systems for Video Technology, 5(6), pp. 533-544.

[5] NVIDIA: CUDA Getting Started Guide for Linux (2013). http://docs.nvidia.com/cuda/pdf/CUDA_Getting_Started_Linux.pdf

[6] NVIDIA: Kepler compute architecture. White paper (2012). http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[7] NVIDIA: CUDA C Programming Guide 5.5 (2013). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[8] NVIDIA: CUDA C Best Practices Guide 5.5 (2013). http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[9] NVIDIA: CUDA Zone (2014). https://developer.nvidia.com/cuda-zone

[10] NVIDIA: GPU-Accelerated Applications (2014). http://www.nvidia.com/content/tesla/pdf/gpu-apps-catalog-mar14-digital-fnl-hr.pdf

[11] Martín, P.J., Ayuso, L.F., Torres, R., Gavilanes, A.,"Algorithmic strategies for optimizing the parallel reduction primitive in CUDA", 2012, Proceedings of the 2012 International Conference on High Performance Computing and Simulation, HPCS 2012, art. no. 6266966, pp. 511-519.

[12] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for GPUs", NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003, no. 1, pp. 1-17, 2008.

[13] Yan, S., Long, G., Zhang, Y.,"StreamScan: Fast scan algorithms for GPUs without global barrier synchronization",2013, ACM SIGPLAN Notices, 48 (8), pp. 229-238.

[14] Balevic, A.,"Parallel variable-length encoding on GPGPUs", 2010, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6043, pp. 26-35.

[15] Balevic, A., Subversion Repository ana-b-pavle. https://xp-dev.com/svn/ana-b-pavle/