Parallel Variable-Length Encoding on GPGPUs

Ana Balevic

University of Stuttgart
ana.balevic@gmail.com

Abstract. Variable-Length Encoding (VLE) is a process of reducing input data size by replacing fixed-length data words with codewords of shorter length. As VLE is one of the main building blocks in systems for multimedia compression, its efficient implementation is essential. The massively parallel architecture of modern general purpose graphics processing units (GPGPUs) has been successfully used for acceleration of inherently parallel compression blocks, such as image transforms and motion estimation. On the other hand, VLE is an inherently serial process due to the requirement of writing a variable number of bits for each codeword to the compressed data stream. The introduction of the atomic operations on the latest GPGPUs enables writing to the output memory locations by many threads in parallel. We present a novel data parallel algorithm for variable length encoding using atomic operations, which archives performance speedups of up to 35-50x using a CUDA-enabled GPGPU.

1 Introduction

Variable-Length Encoding (VLE) is a general name for compression methods that take advantage of the fact that frequently occurring symbols can be represented by shorter codewords. A well known example of VLE, Huffman coding [1], constructs optimal prefix codewords on the basis of symbol probabilities, and then replaces the original symbols in the input data stream with the corresponding codewords.

The VLE algorithm is serial in nature due to data dependencies in computing the destination memory locations for the encoded data. Implementation of a variable length encoder on a parallel architecture is faced by the challenge of dealing with race conditions when writing the codewords to a compressed data stream. Since memory is accessed in fixed amounts of bits whereas codewords have arbitrary bit size, the boundaries between adjacent codewords do not coincide with the boundaries of adjacent memory locations. The race conditions would occur when adjacent codewords are written to the same memory location by different threads. This creates two major challenges for creating a parallel implementation of VLE: 1) computing destination locations for the encoded data elements with a bit-level precision in parallel and 2) managing concurrent writes of codewords to destination memory locations.

In recent years, GPUs evolved from simple graphics processing units to massively parallel architectures suitable for general purpose computation, also known

as GPGPUs. The NVIDIA GeForce GTX280 GPGPU used for this paper provides 240 processor cores and supports execution of more than 30,000 threads at once. In image and video processing, GPGPUs have been used predominantly for the acceleration of inherently data-parallel functions, such as image transforms and motion estimation algorithms [2–4]. The VLE entropy coding to our best knowledge has not been implemented on GPUs so far, due to its inherently serial nature. Some practical compression-oriented approaches on GPUs include compaction and texture compression. The compaction is a method for removing unwanted elements from the resulting data stream by using the parallel prefix sum primitive [5]. An efficient implementation of the stream reduction for traditional GPUs can be found in [6]. The texture compression is a fixed-ratio compression scheme which replaces several pixels by one value. Although it has a fast CUDA implementation [7], it is not suitable for codecs requiring a final lossless encoding pass, since it introduces a loss of fidelity.

We propose a fine-grain data parallel algorithm for lossless compression, and present its practical implementation on GPGPUs. The paper is organized as follows: Section 2 gives an overview of GPGPU architecture, in Section 3 we present a design and implementation of a novel parallel algorithm for variable-length encoding (PAVLE), and in Section 4, we present performance results and discuss effects of different optimizations.

2 GPGPU Architecture

The unified GPGPU architecture is based on a parallel array of programmable processors [8]. It is structured as a set of multiprocessors, where each multiprocessor is composed of a set of simple processing elements working in SIMD mode. In contrast to CPU architectures which rely on multilevel caches to overcome long system memory latency, GPGPUs use fine-grained multi-threading and a very large number of threads to hide the memory latency. While some threads might be waiting on data to be loaded from the memory, the fine-grain scheduling mechanism ensures that ready warps of threads (scheduling unit) are executed, thus providing effectively highly parallel computation resources.

The memory hierarchy of the GPGPU is composed of global memory (high-latency DRAM on the GPU board), shared memory and register file (low-latency on-chip memory). The logical organization is as follows: the global memory can be accessed among all the threads running on the GPU without any restrictions; the shared memory is partitioned and each block of threads can be assigned one exclusive partition of the shared memory, and the registers are private to each thread. When GPU is used as a coprocessor, the data needs to be transferred first from the main memory of host PC to the global memory. In this paper, we will assume that the input data is located in the global memory, e.g. as a result of a computation or explicit data transfer from the PC.

The recent Tesla GPGPU architectures introduce hardware support for atomic operations. The atomic operations provide a simple method for safely handling race conditions, which occur when several parallel threads try to access and mod-

ify data at the same memory location, since it is guaranteed that if an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each access to that memory location will occur and will be serialized, but the order in which they occur is not defined [9]. The CUDA 1.1+ GPU devices support the atomic operations on 32-bit and 64-bit words in the global memory, while CUDA 1.3 also introduces support for shared memory atomic operations.

3 The Parallel Variable-Length Encoding Algorithm

This section presents the parallel VLE (PAVLE) algorithm for GPGPUs with hardware support for atomic operations. The parallel variable-length encoding consists of the following parallel steps: (1) assignment of codewords to the source data, (2) calculation of the output bit positions for compressed data (codewords), and finally (3) writing (storing) codewords to the compressed data array. A high-level block-diagram of the PAVLE encoder is given in Fig. 1. Pseudocode for the

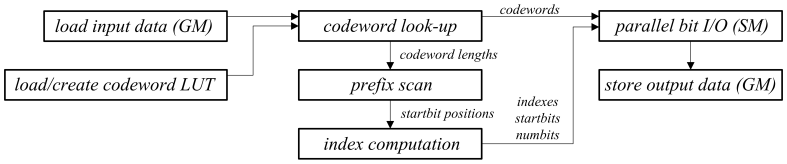


Fig. 1. Block diagram of PAVLE algorithm.

parallel VLE is given in Listing 1 with lines 2 - 5 representing the step 1, lines 6 - 8 being the step 2 and lines 9 - 28 representing the step 3. The algorithm can be simplified if one assumes a maximal codeword length, as is done in the case for the JPEG coding standard. Restricting the codeword size reduces the number of control dependencies and also reduces the amount of temporary storage required, resulting in much greater kernel efficiency.

3.1 Codeword Assignment to Source Data

In the first step, variable-length codewords are assigned to the source data. The codewords can be either computed using an algorithm such as Huffman [10], or they can be predefined, e.g. as it is frequently the case in image compression implementations. Without loss of generality, we can assume that the codewords are available and stored in a table. This structure will be denoted as the codeword look-up table (codeword LUT). Each entry in the table contains two values: the binary code for the codeword, and codeword length in bits, denoted as a *(cw, cwlen)* pair. Our implementation uses an encoding alphabet of up to 256 symbols, with each symbol representing one byte. During compression, each source data symbol (byte) is replaced with the corresponding variable-length codeword.

The PAVLE is designed in a highly parallel manner, with one thread processing one data element. The threads load source data elements and perform codeword look-up in parallel. As the current GPGPU architecture provides more efficient support for 32-bit data types, the source data is loaded as 32-bit unsigned integers to shared memory, where it is processed by blocks of threads. The 32-bit data values are split into four byte symbols, which are then assigned corresponding variable-length codewords from the codeword LUT. The codewords are locally concatenated into an aggregate codeword, and the total length of the codeword in bits is computed.

Algorithm 1 Parallel Variable Length Encoding Algorithm

```
1: k ← tid
2: for threads k = 1 to N in parallel
3:     symbol ← data[k]
4:     cw[k], cwlen[k] ← cwtable[symbol]
5: end for
6: for threads k = 1 to N in parallel
7:     bitpos[1..N] ← prefixsum(cwlen[1..N])
8: end for
9: for threads k = 1 to N in parallel
10:    kc ← bitpos[k] div ws
11:    startbit ← bitpos[k] mod ws
12:    while cwlen[k] > 0 do
13:        numbits ← cwlen[k]
14:        cwpart ← cw[k]
15:        if startbit + cwlen > wordsize then
16:            overflow ← 1
17:            numbits ← wordsize − startbit
18:            cwpart ← first numbits of cw[k]
19:        end if
20:        put_bits_atomic(out, kc, startbit, numbits, cwpart)
21:        if overflow then
22:            kc ← kc + 1
23:            startbit ← (startbit + numbits) mod wordsize
24:            remove first numbits from cw[k]
25:            cwlen[k] ← cwlen[k] − numbits
26:        end if
27:    end while
28: end for
```

3.2 Computation of the Output Positions

To store the data which does not necessarily match the size of addressable memory locations, it is necessary to compute the destination address in the memory and also the starting bit position inside the memory location. Since in the

previous parallel step the codewords were assigned to input data symbols, the dependency in computation of the codeword output locations can be resolved based on the knowledge of the codeword lengths. The output parameters for each codeword are determined by computing the number of bits that should precede each codeword in the destination memory. The bit offset of each codeword is computed as a sum of assigned codeword lengths of all symbols that precede that element in the source data array. This can be done efficiently in parallel by using a prefix sum computation.

The prefix sum is defined in terms of a binary, associative operator $+$. The prefix sum computation takes as input a sequence $x_0, x_1, ..., x_{n-1}$ and produces an output sequence $y_0, y_1, ..., y_{n-1}$ such that $y_0 = 0$ and $y_k = x_0 + x_1 + ... + x_{k-1}$. We use a data-parallel prefix sum primitive [11] to compute the sequence of output bit offsets y_k on the basis of codeword lengths x_k, that were assigned to source data symbols. A work-efficient implementation of parallel prefix sum performs $O(n)$ operations in $O(\log n)$ parallel steps, and it is the asymptotically most significant component in the algorithmic complexity of the PAVLE algorithm. Given the bit positions at which each codeword should start in the com-

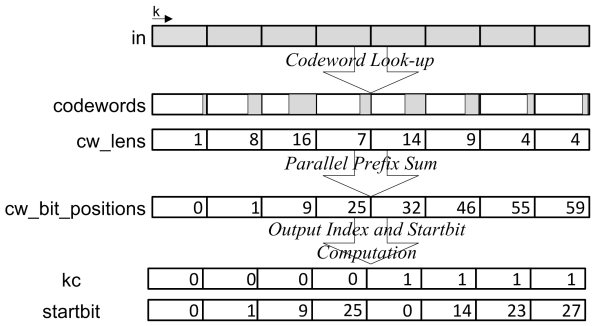


Fig. 2. An example of the variable-length encoding algorithm.

pressed data array in memory, the output parameters can be computed knowing the fixed machine word size, as given in the lines 10-11 of the pseudocode.It is assumed that the size of addressable memory locations is 32-bits, and it is denoted as *wordsize*. The variable k is used to denote the unique thread Id. It also corresponds to the index of data element processed by the thread k in the source data array. The *kc* denotes index of the destination memory word in compressed data array, and *startbit* corresponds to the starting bit position inside that destination memory word.

Fig. 2 is given as an illustration of the parallel computation of the output index and starting bit position on a block of 8 input data elements: The first two steps of the parallel encoding algorithm result in the generation of matching codewords for the input symbols, codeword lengths (as the number of bits), and output parameters for the memory writes to the output data stream. The

number of bits for each compressed data block is obtained as a byproduct of the first phase of the parallel prefix sum algorithm. Since a simple geometric decomposition is inherently applied on the GPUs as a step of the mapping process, this result can be used for concatenating the compressed data blocks into a contiguous array prior to data transfers from GPU to system memory.

3.3 Parallel Bit-Level Output

Bit-level I/O libraries designed for general-purpose CPUs process data serially, i.e., the codewords are stored one after the other into the memory. Implementation of a VLE on a parallel architecture introduces a problem of correctly dealing with race conditions that occur when different threads attempt to write their codewords to a same memory location. A recently introduced hardware support for atomic bitwise operations enables efficient execution of concurrent threads performing bit-level manipulations on the same memory locations, thus providing a mechanism for safely handling race conditions. The parallel output of codewords will produce correct results regardless of the write sequence, provided that each sequence of read-modify-write operations on a single memory location can be successfully completed without interruption, and that each output operation operation changes only the precomputed part of the destination word. The parallel bit-level output is executed in two stages: First, the contents

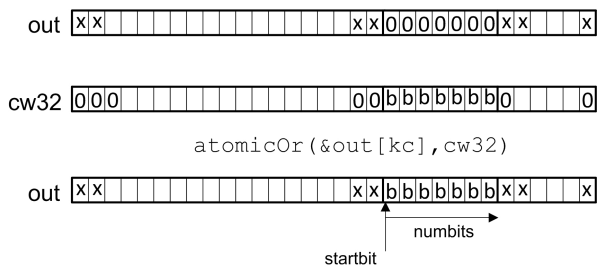


Fig. 3. Setting memory contents at index kc to the desired bit-values (codeword).

of the memory location at the destination address are prepared for the output by masking the *numbits* number of bits corresponding to the length of the codeword starting from the pre-computed bit output position. Second, the bits at these positions in the destination location are set to the value of the codeword, as illustrated in Fig. 3. If the contents of the destination memory are set in advance (all zeros), the output method can be reduced to only one atomic *or* operation. The implementation of the *put_bits_atomics* procedure for the GPGPUs supporting atomic operations (CUDA1.1+ compatible) is given in the code listing below.

A situation when a codeword crosses boundaries of a destination word in memory can occur during variable length encoding, e.g., when the *startbit* is near

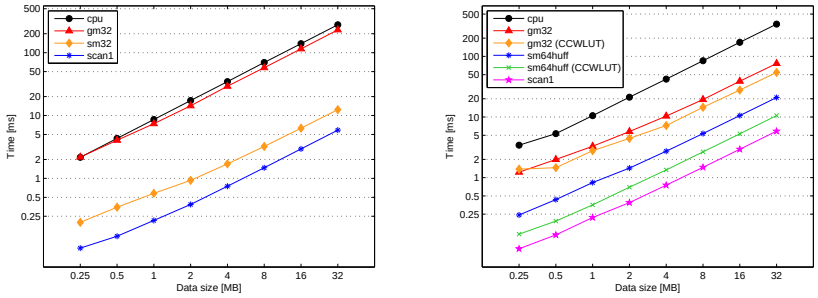
the end of the current word, and the codeword to be written requires more bits than what is available in the reminder of the current word. The crossing of the word-boundary is detected and handled by splitting the output of the codeword into two or more store operations. When the codeword cross boundaries of several machine words, some of the atomic operations can be replaced by the standard store operation. The inner parts of the codeword can be simply stored to the destination memory location(s), and only the remaining bits on both sides of the codeword need to be set using the atomic operations.

```
__device__ void put_bits_atomic(unsigned int* out, unsigned int kc,
                                unsigned int startbit, unsigned int numbits,
                                unsigned int codeword) {

    unsigned int cw32 = codeword;
    unsigned int restbits = 32-startbit-numbits;

#ifndef MEMSET0
    unsigned int mask = ((1<<numbits)-1);
    mask <<= restbits;
    atomicAnd(&out[kc], ~mask);
#endif

    if ((startbit == 0) && (restbits == 0)) out[kc] = cw32;
        else atomicOr(&out[kc], cw32 << restbits);
}
```

4 Performance Results

Performance of several kernel implementations was benchmarked on a PC with an 2.66 GHz Intel QuadCore CPU, 2 GB RAM memory, and a NVIDIA GeForce GTX280 GPU supporting atomic instructions on 32-bit words. The test data set was composed of randomly generated test data files of different sizes and different amount of information content (entropy between 0.5-8 bits/symbol). The test files were assigned variable-length codewords using the Huffman algorithm with the restriction on the maximal codeword length. The performance of a CPU encoder running on one 2.66 GHz CPU core is given as a reference. Fig. 4(a) gives a performance comparison on a data set with 2.2 bits/symbol entropy. The GPU encoder *gm32* concatenates codewords for every 4 consecutive symbols (bytes) and writes the aggregate codeword to the GPU memory using global memory atomic operations. The performance of the serial encoder and the global memory (GM) encoder *gm32* are closely matching. However, by performing the atomic operations on a temporary buffer in shared memory (SM), as in *sm32*, a speed-up of more than an order of magnitude is achieved. The performance of the scan kernel, which is the asymptotically dominant part of the parallel algorithm, is given as a reference.

The *gm32* and *sm32* kernels operate under the assumption that the size of the aggregate codeword for four consecutive symbols (bytes) will not exceed



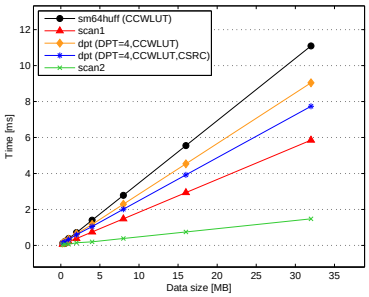
(a) CPU and GPU Kernels (b) Codeword LUT caching

Fig. 4. Kernel execution times as a function of data size.

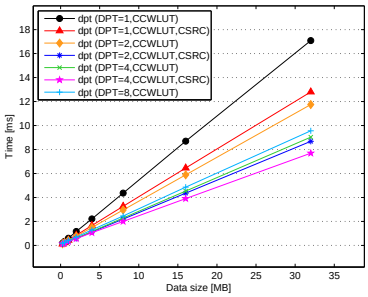
the original data length, i.e. it will always fit into one 32-bit word. When using Huffman codewords, it may happen that the aggregate codeword exceeds the original data size. We designed a second SM kernel, denoted as *sm64huff*, that has a temporary buffer for the aggregate codeword of twice the original data size (a typical buffer size in compression implementations). The performance of *sm64huff* is slightly lower than the performance of *sm32* kernel, since it must perform one additional test during the codeword output. The situation when a codeword spans more than two destination memory locations is however correctly supported. In this case, no atomic operation is needed for the part of the codeword that spans an entire memory location, and a standard store operation can be used. However, empirical evaluation showed that atomic operations on the shared memory are implemented very efficiently, and that introduction of the additional test actually hurts the performance due to the increased warp serialization.

Additional performance improvements can be achieved by caching the codeword LUT, instead of looking up the codeword for each symbol in the global memory every time a symbol occurs. Fig. 4(b) gives a comparison of kernel execution times when the codeword look-ups are performed on the shared memory. Similar results are achieved by using the texture memory, which is cached by each multiprocessor. Use of low-latency shared memory for caching the codeword LUT improved the performance of GM kernels by approximately 20%, and the performance of SM kernels by up to 55%. As the symbols that appear more frequently are replaced by the codewords of shorter length, the low entropy data (well-compressible) will result in more shorter codewords that should be stored by different threads at the same memory location. This issue could be mitigated by processing more than one 32-bit data element per thread. The average number of bits that are written by each thread in one atomic operation to the destination memory location is increased and fewer atomic operations are issued.

Additionally, increasing DPT reduces the total number of data elements that is processed by the prefix sum (*scan*), which significantly influences the run time.



(a) Comparison of standard and DPT kernels



(b) DPT Parameter effects

Fig. 5. Effects of processing more data per thread (lin scale).

Fig. 5(a) shows performance gains using the ideal DPT value; performance of *scan* using the original and reduced number of blocks are given as a reference. Additional improvements are achieved by (1) caching the codeword LUT as previously described, and (2) caching aggregate codewords for every DPT elements in a local buffer. However, further increasing DPT radically increases memory requirements, since data is compressed in a shared memory buffer prior to transfer to the global memory. Fig. 5(b) gives a comparison of run times using several different DPT values. The investigation showed that the maximal DPT is limited by the shared memory requirements, and is relatively low ($DPT_{max} = 8$ when only codeword table is cached, and $DPT_{max} = 4$ when also aggregate codewords are cached). The best results are obtained using $DPT = 4$, resulting in a 35x speed-up.

5 Conclusion

In this paper, we presented a method for parallel bit-level output of data and a novel parallel algorithm for variable-length encoding (PAVLE) for GPGPU architectures supporting atomic operations. The PAVLE algorithm was implemented on a CUDA1.3-enabled GPGPU using atomic operations on the shared memory for managing concurrent codeword writes, parallel prefix sum for computing the codewords offsets in compressed data stream and caching of the codeword look-up tables in the low-latency memory. The optimized version of PAVLE for CUDA 1.3 compatible GPGPUs achieves performance of approximately 4GB/sec using Huffman codes for encoding the data on the NVIDIA GeForce GTX280 GPGPU. We observed considerable speedups compared to the serial VLE on the state of the art PCs (up to 35x on 2.66GHz CPU, and up to 50x on a 2.40GHz CPU), thus making the PAVLE an attractive lossless compression algorithmic building block for GPGPU-based applications.

Acknowledgments The authors would like to thank Glenn R. Luecke for his support and his invaluable comments on the manuscript. Special thanks go to Tjark Bringewat for benchmarking and very constructive discussions on GPGPU kernel optimizations.

References

1. Huffman, D.: A method for the construction of Minimum-Redundancy codes. Proceedings of the IRE **40**(9) (1952) 1098–1101
2. Allusse, Y., Horain, P., Agarwal, A., Saipriyadarshan, C.: GpuCV: an opensource GPU-accelerated framework forimage processing and computer vision. 2006 IEEE International Conference on Multimedia and Expo (2008)
3. Chen, W., Hang, H.: H. 264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA). In: 2008 IEEE International Conference on Multimedia and Expo. (2008) 697–700
4. Fung, J., Mann, S.: Using graphics devices in reverse: GPU-based image processing and computer vision. In: 2008 IEEE International Conference on Multimedia and Expo. (2008) 9–12
5. Blelloch, G.E.: Prefix sums and their applications. Synthesis of Parallel Algorithms (1990) 35—60
6. Roger, D., Assarsson, U., Holzschuch, N.: Efficient stream reduction on the gpu. In Kaeli, D., Leeser, M., eds.: Workshop on General Purpose Processing on Graphics Processing Units. (Oct 2007)
7. Ignacio Castaño: High quality dxt compression using cuda. Technical report, NVIDIA (last access: May, 2008)
8. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. Micro, IEEE **28**(2) (2008) 39–55
9. NVIDIA Corporation Technical Staff: Nvidia cuda -programming guide 2.0. Technical report, NVIDIA (last access: May, 2009)
10. Atallah, M., Kosaraju, S., Larmore, L., Miller, G., Teng, S.: Constructing trees in parallel. In: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, ACM New York, NY, USA (1989) 421–431
11. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with cuda. GPU Gems **3** (2007)