Fine-Grain Parallelization of Entropy Coding on GPGPUs

Ana Balevic*

*E-mail: ana.balevic@gmail.com

ABSTRACT

Massively parallel GPUs are increasingly used for acceleration of data-parallel functions, such as image transforms and motion estimation. However, the entropy coding stage is typically executed on the CPU due to inherent data dependencies in lossless compression algorithms. We propose a simple way of efficiently dealing with these dependencies, and present two novel parallel compression algorithms specifically designed for efficient execution on many-core architectures. By fine-grain parallelization of lossless compression algorithms not only do we significantly speed-up the entropy coding, but we also enable completion of all encoding phases on the GPU.

KEYWORDS: Many-core Architectures; CUDA GPUs; Parallel Compression; Huffman; VLE; RLE.

1 Introduction

A major trend in computer architecture design is a shift towards integration of increasing number of processing cores on chip multiprocessors (CMPs). The performance boosts from increasing the CPU clock frequency have diminished due to increased power dissipation, memory wall and saturation of gains by speculative instruction execution. An extreme example of emerging many-core architectures are GPUs, which evolved into powerful massively parallel processing architectures for general purpose computation featuring large number of simple in-order processing elements (240 cores, 2009), working in the SIMD-like manner. As Moore's law continues to be reliable predictor of increases in the aggregate computing power, the trend towards parallelization leads into an era of highly parallel architectures, demanding for efficient parallel software solutions capable of utilizing abundant computation resources.

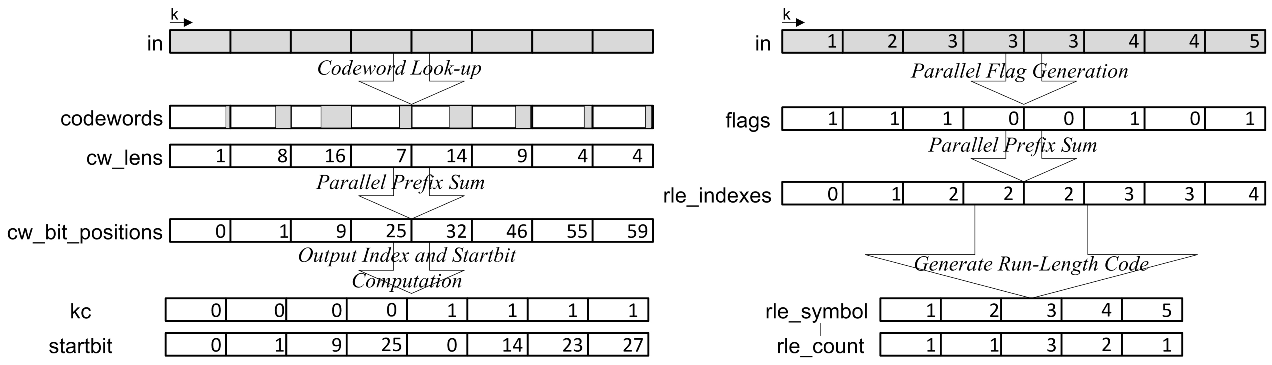
Traditional design of image and video compression systems is targeted towards specific highly-optimized solutions implemented directly in hardware, or software designs for general-purpose architectures optimized for a single processor memory hierarchy [AHL02, MP08]. The parallel codec solutions for multi-core processors are typically based on a simple distribution of work among different processors, which then process the assigned data chunks using standard serial compression algorithms. The massively parallel GPUs are increasingly used for acceleration of inherently data-parallel functions, such as image transforms and motion estimation algorithms. However, the execution of the last coding block, i.e. the entropy coding, is pushed back on the CPU due to inherent data dependencies in the algorithms used at this stage.

In the next section, we propose a simple method for efficiently dealing with these dependencies, and present parallel compression algorithms for fixed-length and variable-length encoding designed for efficient execution on many-core architectures. These algorithms can be easily combined with the already available data-parallel transforms, to enable completion of the final coding stage of GPU-accelerated codecs on the card.

2 Parallelization of Lossless Compression Methods

The parallel encoding algorithms can be organized into the following phases: (I) Assignment of codes to the input data, (II) Computation of destination memory locations for the codes, and (III) Concurrent output of the codes to the compressed data array. The input data is stored in the global memory of a GPU (DRAM). The data is structured for processing into blocks of typically 1-4KB in size, which are processed by many threads in parallel.

The GPU architecture is organized as a set of streaming multiprocessors; each streaming multiprocessor (SM) contains a set of simple processing elements. In the GPU computational model (CUDA), lightweight threads are structured into a grid of thread blocks, and each data block is assigned to one block of threads (TB). The low-latency shared memory is used for caching and communication between threads of a TB. For maximal efficiency, it is recommended to have a very large number of threads as it is the main mechanism for memory latency hiding [NVI08]. Thus, it can be assumed that the effective number of threads processing a data block in parallel is on the same order as the number of elements, so we design fine-grain parallel algorithms for maximal efficiency.



(a) Variable-Length Encoding (b) Run-Length Encoding

Figure 1: Example Workflows of Parallel Compression Algorithms.

Some of the most commonly used algorithms at the entropy coding stage are Variable-Length Encoding (VLE) and Run-Length Encoding (RLE). The VLE, e.g. Huffman Coding, takes advantage of the fact that the frequently occurring symbols can be represented in a fewer bits. Although the variable-length codes can be efficiently computed and assigned to the input data in parallel, a difficulty for fine-grain parallelization arises from the data dependencies in computing the destination memory locations for the encoded data. In the run length encoding, there is a dependency in the computation of codes (run-length pairs) and indexes of their output locations, as the value and the index of each code depend on the values of previously encountered data elements in the input data array.

In compression algorithms, as codes and their parameters can be in most cases computed in advance, some of the dependencies in compression algorithms can be resolved by using a parallel programming primitive, the prefix sum (scan) [Ble90]. In VLE, the scan is applied on the lengths of the codes assigned to the input data. The scan on code lengths results in an array containing the offsets (in bits) from the start of a memory array; the address of the destination memory location and the target bit-position inside the memory word can then be easily determined using the size of a machine addressable word as a parameter. In the RLE, there is a need for a slightly complex approach - first, it is necessary to identify which elements of the input data array form a symbol run. This can be determined by generating the flags that determine if an element in the array equals to the element before it. This information is used to compute the set of output locations in the compressed data array, and then to determine the values of the codes. An illustration is given in Fig. 1.

2.1 Parallel Variable-Length Encoding (PAVLE)

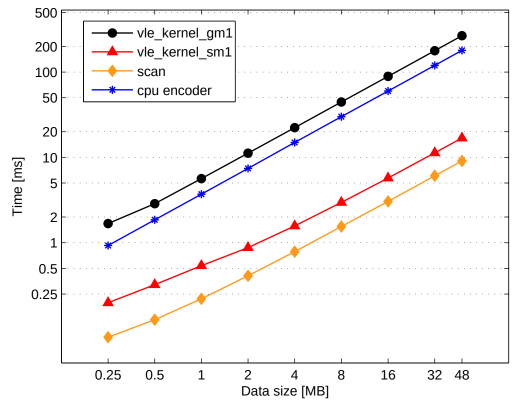
The fine-grain parallelization of a VLE introduces a problem of dealing with data races that occur when adjacent codes are written to the same memory location by different processors. The parallel output of codes will produce correct results regardless of the write sequence, provided that each sequence of read-modify-write operations on a single memory location can be successfully completed without interruption, and that each output operation changes a disjunctive partition of the destination word. By using a recently introduced hardware support for atomic bitwise operations to construct the algorithm for parallel bit I/O, we enabled the efficient execution of concurrent threads performing bit-level manipulations on the same memory locations without race conditions and managed to successfully complete the VLE encoding process in parallel.

2.2 Parallel Run-Length Encoding (PARLE)

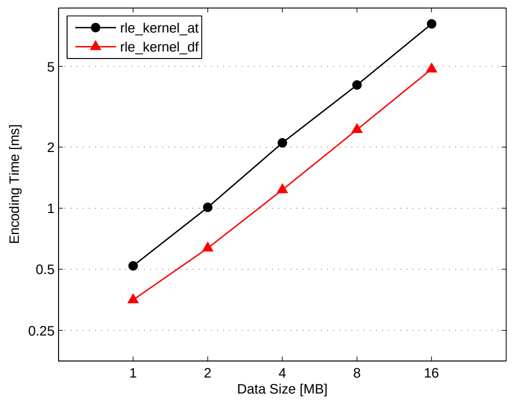
In the RLE, we need to compute indexes of the elements that should be stored and their codes (length of the symbol run). The first approach for computing the codes is based on the use of another parallel primitive, the reduction, for summing up the number of times a symbol appeared in its run. Further analysis of dependencies in the original algorithm resulted in a method which does not depend on the architectural support for the atomic operations. Instead of accumulating number of occurrences for each symbol in parallel, the indexes of last elements in each run are determined on the basis of the flags. These values are then used to compute the total number of elements that appear in between these locations: the resulting count corresponds to the number of times an element appeared in its run.

3 Performance Results on GPGPU Tesla Architecture

Both parallel compression algorithms have a work complexity of $O(n)$ and a parallel run time $O(\log n)$, dominated by the complexity of prefix sum. Performance 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 performance results are given in Fig. 2; the performance of the scan kernel is given as a reference. A significant im-



(a) PAVLE (global vs. shared memory)



(b) PARLE (reduction vs. differences)

Figure 2: Performance of Parallel Compression on nVidia GeForce GTX280 GPU.

provement in PAVLE performance is achieved by using the shared memory atomic operations. With additional algorithmic and code optimizations, we managed to further accelerate PAVLE by approximately 20%, and PARLE by 50%.

4 Conclusions and Future Work

We presented two novel methods for parallel lossless compression designed for efficient execution on many-core architectures supporting fine-grain parallelism. The parallelization of the Variable-Length and Run-Length encoding resulted in processing rates up to 3.5GB/s and 7.5GB/s on a modern GPU. These two parallel compression algorithms can be easily combined with the already available GPU-accelerated image transforms and motion estimation algorithms, to enable execution of all codec components directly on GPUs. With this approach we speed-up the entropy coding by 5-20x, and reduce the data transfer time from the GPU to system memory. Our current work comprises performance modeling of parallel compression algorithms, and on-chip memory compression for many-core architectures.

References

[AHL02] I. Ahmad, Y. He, and M. L. Liou. Video compression with parallel processing. *Parallel Computing*, 28(7-8):1039–1078, 2002.

[Ble90] Guy E Blelloch. Prefix sums and their applications. *Synthesis of Parallel Algorithms*, pages 35—60, 1990.

[MP08] A. Merigot and A. Petrosino. Parallel processing for image and video processing: Issues and challenges. *Parallel Computing*, 34(12):694–699, 2008.

[NVI08] NVIDIA Corporation Technical Staff. Nvidia cuda -programming guide 2.0, last access: Dec, 2008.