Using Arithmetic Coding for Reduction of Resulting Simulation Data Size on Massively Parallel GPGPUs

Ana Balevic, Lars Rockstroh, Marek Wroblewski, and Sven Simon

Institute for Parallel and Distributed Systems, Universitaetsstr. 38, 70596 Stuttgart, Germany
ana.balevic@ipvs.uni-stuttgart.de

Abstract. The popularity of parallel platforms, such as general purpose graphics processing units (GPGPUs) for large-scale simulations is rapidly increasing, however the I/O bandwidth and storage capacity of these massively-parallel cards remain the major bottle necks. We propose a novel approach for post-processing of simulation data directly on GPGPUs by efficient data size reduction immediately after simulation that can considerably reduce the influence of these bottlenecks on the overall simulation performance, and present current performance results.

Keywords: HPC, GPGPU, I/O Bandwidth, Data Compression, Arithmetic Coding.

1 Introduction

In general, simulations require not only high amount of computing power but also the transfer and storage of large amounts of data. Due to high computational requirements, large-scale simulations are typically conducted on parallel systems comprising of multiple computing nodes. With recent advances in development of massively parallel graphics cards suitable for general purpose computations (GPGPUs) and their general affordability, with prices as low as $400 per unit (2008) featuring up to 128 streaming processors, the computers used in simulations are increasingly equipped with one or more GPGPUs integrated as arithmetic co-processors that enable hundreds of GFLOPs of raw processing power in addition to the CPU.

One of the major bottlenecks of such parallel computing systems, besides the storage of large amount of data, is the I/O bandwidth required for run-time communication and synchronization of numerous processing elements, as well as the transfer of the resulting data from the arithmetic co-processors to the central processing unit. As the time spent in data transfers between computational nodes can significantly reduce observed speedups and thus severely influence the performance benefit of using a parallel system for computations, there is a demand for novel approaches to the storage and transfer of data.

The study of data compression algorithms in the computer science has resulted in efficient coding algorithms such as Huffman coding, Arithmetic/Range coding, Lempel-Ziv family of dictionary compression methods and various transforms such as Burrows-Wheeler Transform (BWT), that are now a part of widely used compression utilities, such as Zip, RAR, etc as well as image and video codecs. In this paper we explore use of entropy coding algorithms on high performance computing systems containing massively parallel GPGPUs, such as NVidia GeForce 8800 GT, for efficient stream reduction by processing of the resulting simulation data in between the simulation steps, and prior to the transfer and storage on the host computer.

The paper is structured as follows. In Section 2, the current approaches to data size reduction on GPUs are reviewed. An overview of fundamental compression methods is given in Sections 3 and 4. Section 5 presents design of a block-parallel entropy coding algorithm. Sections 6 and 7 give current performance results in compression of floating-point data from a light scattering simulation on a GPGPU, and an overview of the future research.

2 Related Work

The popularity of parallel platforms, such as GPGPUs for large-scale simulations is rapidly increasing, however the I/O bandwidth and storage capacity of GPGPUs remain a bottle neck. In simulations of large systems, a variety of approaches has been used to reduce run-time size of simulation data set. Some common approaches include different methods for the storage of sparse matrices, use of reduced precision for calculations, etc. Fast lossless compression approaches to floating-point data, including the overviews of older approaches can be found in [1, 2]. As GPGPUs impose numerous constraints on the data types that could be efficiently used for storage of the simulation data, it is worth exploring which other approaches to the data size reduction are available and could be efficiently used on GPGPUs. The most notable approaches for data reduction that used on GPUs are stream reduction and texture compression of computer graphics:

Texture compression is driven by the need for reducing the amount of physical memory required for the storage of texture images that enhance gaming experience. A distinctive characteristic of the texture compression is that it provides a fixed ratio compression coupled with single-memory data access, which makes it ideally suited for computer graphics. Texture compression is a lossy data compression scheme, with common implementations being S3TC family of algorithms (DXT1-DXT5), and DXTC. For gaming purposes, the loss of fidelity is acceptable and can even account for a perceptually better experience as the decrease in data size enables the storage of higher-resolution textures in the memory of a graphics card.

The second notable approach on GPUs is stream reduction, which is the process of removing elements that are not necessary from the output stream. Stream reduction is frequently used in multi-pass GPU algorithms, where the stream output of the first pass is used as the input for the next pass. An efficient

implementation of the stream reduction on GPUs is given in [3], and achieves a linear performance by using divide-and-conquer approach that is well applicable to GPGPU block-oriented architectures.

3 Data Compression

Data compression deals with the data size reduction by removing redundancy from the input data. The theoretical bound of the compression, i.e. the maximum theoretical compression ratio, is given by Claude Shannon's Source Coding Theorem, which establishes that the average number of bits required to represent an uncertain event is given by its entropy (H). Data compression methods are classified according to the information preservation to lossless and lossy. Lossless compression algorithms are used in areas where absolutely accurate reconstruction of the original data is necessary, such as in compression of text, medical, scientific data, etc. In the applications targeted toward human end-users, lossy compression is applied to audio, video and image data in order to provide perceptively (near) lossless or acceptably distorted representation of data by using perceptual models of the human audio-visual system. We consider two fundamental lossless algorithms for the compression of the simulation data, which could be easily combined with intermediate lossy steps, e.g. quantization, if further increase of the compression ratio at the expense of accuracy is desired:

Huffman Coding: As a statistical lossless data compression algorithm, Huffamn coding gives a reduction in the average code length used to represent the symbols of an alphabet by assigning shorter codewords to more frequent symbols and vice versa. The Huffman code is an optimal prefix code in the case where exact symbols probabilities are known in advance and are integral powers of 1/2 [4]. In real-world scenarios, the exact distribution of symbol probabilities is rarely known in advance, so this means either acceptance of lower compression rates or use of adaptive Huffman algorithms that provide one-pass encoding and adaptation to changing statistics of the input data. The major disadvantage of the adaptive Huffman coding is relatively high cost of tree maintenance operations, especially in GPU environments, where non-aligned memory access are penalized in terms of performance. When the symbol probabilities are highly skewed, which is often in the case of the simulation data, Huffman coding does not provide good compression rates as the generated codewords, being external nodes of a binary tree, are always represented by an integral number of bits.

Arithmetic Coding: Arithmetic coding treats the whole input data stream as a single unit that can be represented by one *real* number in the interval $[0, 1)$. As the input data stream becomes longer, the interval required to represent it becomes smaller and smaller, and the number of bits needed to specify the final interval increases. Successive symbols in the input data stream reduce this interval in accordance with their probabilities. The more likely symbols reduce the range by less, and thus add fewer bits to the coded data stream.

By allowing fractional bit codeword length, arithmetic coding attains the theoretical entropy bound to compression efficiency, and thus provides better

compression ratios than Huffman coding on input data with highly skewed symbol probabilities. The arithmetic coding gives greater compression, is faster for adaptive models, and clearly separates the model from the channel encoding [5]. As simulation data is usually biased to certain values (or could be transformed into a set of biased data e.g. by some sort of predictive coding), we chose to further experiment with arithmetic coding for simulation data compression on GPGPUs.

4 Fundamental Principles of Arithmetic Coding

The central concept behind arithmetic coding with integer arithmetic is that given a large-enough range of integers, and frequency estimates for the input stream symbols, the initial range can be divided into sub-ranges whose sizes are proportional to the probability of the symbol they represent[4, 5]. Symbols are encoded by reducing the current range of the coder to the sub-range that corresponds to the symbol to be encoded. Finally, after all the symbols of the input data stream have been encoded, transmitting the information on the final sub-range is enough for completely accurate reconstruction of the input data stream at the decoder. The fundamental sub-range computation equations are given recursively as:

$$low^n = low^{n-1} + (high^{n-1} - low^{n-1})P_l(x_n) \tag{1}$$

$$high^n = low^{n-1} + (high^{n-1} - low^{n-1})P_h(x_n) \tag{2}$$

where P_l and P_h are the lower and higher cumulative probabilities of a given symbol (or cumulative frequencies) respectively, and low and high represent the sub-range boundaries after encoding of the n-th symbol from the input data stream. As an illustration of the arithmetic coding concepts, a basic encoding to a real number, for the input sequence 'bac', with the given symbol distribution is depicted in Fig. 1. The decoding algorithm works in an analogous way, and must be synchronized with the encoder. The practical integer-implementation of the arithmetic coder function according to the same principle as illustrated in Fig. 1, but uses frequencies of occurrence instead of symbol probabilities and a range of $[0, N)$, where typically N is an integer value $N >> 1$.

To avoid arithmetic overflows on 32-bit architectures, a maximally 31-bit integer range can be used to represent the full range of the coder. To avoid underflows, which would happen if the current sub-range would become too small to distinctively encode the symbol, i.e. when the upper and lower boundaries of the range converge, several methods have been proposed for range renormalization[4–6].

For the range renormalization and generation of the compressed data bit stream, we use a method of dividing the initial range into quarters described in detail in [6] that works as follows: After the coder detects that the current sub-range falls into a certain quarter, it is ensured that the leading bits of the numbers representing the sub-range boundaries are set, and cannot be changed

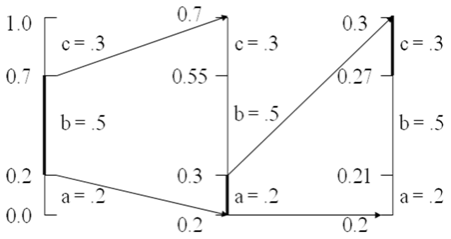

Fig. 1. Example of arithmetic encoding of the input sequence 'bac'. Symbols of the alphabet $A = a, b, c$ have probabilities of occurrence $P = .2, .5, .3$ Final range is $[0.27, 0.3)$. The sequence 'bac' can be thus coded with 0.27.

by subsequent symbols. A series of scaling operations is conducted, and the set bits are output one after the other, thus generating the compressed data output stream. These operations result in the renormalization of the coder range back to the full supported range, thus eliminating possibility of incorrect en/decoding due to the range underflow.

5 Block-Parallel GPGPU Arithmetic Encoder

Simulations run on general purpose graphics hardware often produce large amount of data that after a number of iterations hardly fits into the memory of a graphics card, thus imposing a need for a memory transfer so that free space is made available for subsequent iterations. As the frequent data transfers from the memory of a GPGPU to the host PC reduce the overall performance of the simulation, it is our goal to lessen the frequency of these data transfers. We propose processing of simulation data directly on the GPGPUs after each simulation step to reduce the resulting data size, and thus resources required for the storage and transfer of results.

First, the simulation data is partitioned into the data blocks as in [7, 3], which are then processed by a number of replicated coders running in parallel, as depicted in Fig 2. Each block of simulation data is processed by an instance of the encoder running in a separate thread. In the CUDA computational model threads are executed in the thread blocks, each of which is scheduled and run on a single multi-processor. Our block-parallel encoder implementation can be executed by multiple blocks containing multitude of threads, where each thread executes the CUDA-specific code that implements the arithmetic encoding process described in Sect.4 (Fig.2,Step1). The data block size, as well as the number of blocks and threads, is configurable as the compression kernel execution parameter. Based on different block sizes, different compression ratios are obtained - typically resulting in higher compression ratio for larger data block sizes.

After the complete input stream is encoded, the coded data blocks are prepared for the storage at the adequate global memory locations, prior to the transfer to the host computer. The first preparation step for storage in the parallel

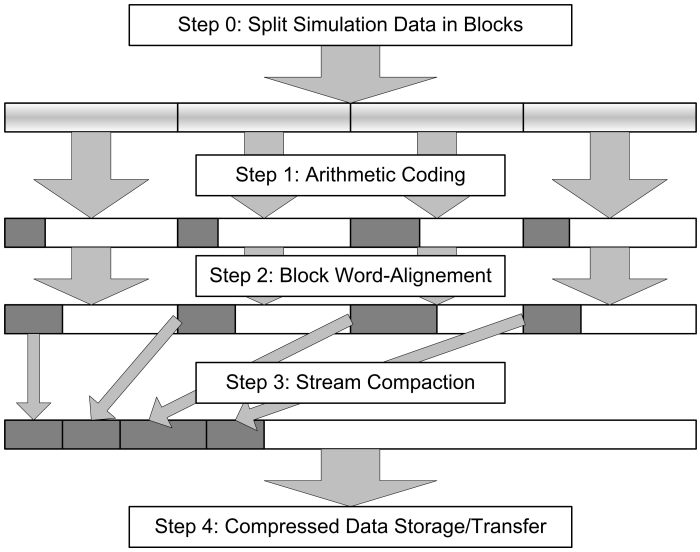

Fig. 2. Block-Parallel GPGPU Simulation Data Compression Process

implementation of encoder on GPGPU is alignment of the coded data bitstream to the byte or word boundary (Fig.2,Step2). The padding to the boundary increases the compressed data length. The decrease of the performance ratio due to this operation is dependent on the initial data block length and its entropy - the larger the resulting coded block is, the smaller difference those couple of padded bits make.

To obtain highly biased data model, the floating-point numbers from the simulation are processed on the byte level. Each of 256 possible byte values is assigned a corresponding symbol. The model constructed in this manner exploits the statistical properties of data much better than if we would assign each different floating-point value a single symbol, typically resulting in probabilities highly biased to some symbol e.g. 0x00. Another advantage of this modeling approach is that it can be without any modification applied to other data types, without a loss of generality. After byte-level arithmetic encoding, the coded data is aligned to the word boundary, e.g. 8-bit before transferring the results into the global memory of device.

The compacted output stream is obtained by the concatenation of the codewords at the block-level by stream compaction (Fig.2,Step3) that produces a single continuous array containing the coded data. The concatenation process is executed fully in parallel on the GPGPU, by creating an array of the coded data block lengths from the resulting data of encoding process. After generation of the codewords and alignment to desired word boundary length (i.e. 8-bits or 32 bits), the information on the coded block lengths is used to generate the array of the pointers to the starting positions of the coded data coming from parallel coders by using parallel prefix sum primitives.

For correct functioning of the method, the stream compaction is not necessary, as the data from each coded block can be transferred separately to the host computer followed by storage into a continuous array. However, it is worth examining, as the burst mode for data transfer generally achieves better performance than the iterative data transfer.

6 Performance Results and Discussion

The block-parallel implementation of integer-based arithmetic coder for GPGPU was tested on data from the simulation of light scattering [8]. As the test data for the compression were taken the results of finite-difference time-domain simulation iterations on the grid of 512x512 cells. The distinctive characteristics of the test data set were low values with highly biased symbol probability distribution, resulting in very low entropy when using the model described in Sect. 5. The output of parallel arithmetic encoder running on the GPGPU is decompressed by sequential decoder runing on the host PC, and the results are verified by byte comparison functions, as well as the external file comparison tool WinDiff.

Test Data Set		Block Size [B]	Parallel Coders	CR	GPU Encode T [ms]	Direct Transf. T[ms]	Comp. Transf. [ms]	Total Time [ms]	
								Iterat.	Burst
1	Size: 1MB	512	2048	438	5.33	1.69	0.038	44.15	5.89
	H=0.00289908 [b/B]	1024	1024	765	5.25	1.61	0.037	25.20	5.80
	CPU Encode T=0.5s	4096	256	1727	8.98	1.59	0.06	14.54	9.46
2	Size: 4 MB	1024	4096	979	11.39	4.44	0.04	87.86	11.97
	H=0.00037884 [b/B]	4096	1024	1288	13.50	4.49	0.059	33.52	14.08
	CPU Encode T=1.6s	8192	512	1528	15.60	4.40	0.23	26.14	16.67
3	Size: 16 MB	1024	16384	242	51.22	16.13	0.23	354.4	52.05
	H=0.01047371 [b/B]	4096	4096	293	59.28	15.59	0.37	136	60.19
	CPU Encode T=6.5s	8192	2048	304	76.76	16.90	0.56	115	77.86

Table 1. Performance results on test configuration: AMD Athlon 2.41GHz, 2GB RAM, nVidia GeForce 8800GT 128SPs, 768MB RAM. CUDA 1.0. Total time corresponds to the time required for compression and transfer of data including the overheads, such as. alignment, stream compaction and block sizes array transfer.

The performance results in Table 1. show that the parallel implementation of arithmetic encoder achieves compression ratios (CR) competitive with a sequential coder, but in a considerably shorter time, with the compression ratio approaching the lower entropy bound as the data block size increases. The transfer times for the compressed data (Col. 7) are significantly lower than those for the direct transfer of data (Col. 6) without any compression; however as the compression process inevitably introduces an overhead, the gains achieved so far are mostly in the required space on the GPGPU for the storage of the temporary results, with more work on the speed optimization of the codec required for making it a competitive method for reduction of the I/O bandwidth requirements. The storage savings are a significant achievement, as the frequency with which the simulation data needs to be transferred considerably influences overall simulation speed-up. If the storage of simulation results requires less space,

there is a more room for the new data, resulting in a lower number of required memory transfers from the GPGPU to the host computer, and thus a better overall simulation performance.

7 Conclusions and Future Work

The implementation of the block-parallel arithmetic encoder proved that use of statistical coding methods for the compression of simulation data directly on GPGPUs has a potential for the efficient reduction of simulation data size. The compression ratios of the parallel coder approach entropy as the theoretical boundary of compression ration with the increasing block sizes. Furthermore, the parallel implementation exhibits a significant speed-up over the sequential data compression algorithm, thus showing high potential to reduce influence of the limited resources for storage and transfer on the simulation performance on parallel systems. Our ongoing work focuses on optimization of computational performance of entropy coders. Further work will examine strategies for pre-processing of simulation data that could account for high compression efficiency coupled with high processing speed.

References

1. Isenburg, M.: Fast and efficient compression of floating-point data. IEEE Transactions on Visualization and Computer Graphics **12**(5) (2006) 1245–1250 Member-Peter Lindstrom.
2. Ratanaworabhan, P., Ke, J., Burtscher, M.: Fast lossless compression of scientific floating-point data. In: DCC '06: Proceedings of the Data Compression Conference, Washington, DC, USA, IEEE Computer Society (2006) 133–142
3. 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)
4. Sayood, K., ed. In: Lossless Compression Handbook. Academic Press (2003)
5. Howard, P.G., Vitter, J.S.: Arithmetic coding for data compression. Technical Report Technical report DUKE–TR–1994–09 (1994)
6. Bodden, E.: Arithmetic coding revealed - a guided tour from theory to praxis. Technical Report 2007-5, Sable (2007)
7. Boliek, M.P., Allen, J.D., Schwartz, E.L., Gormish, M.J.: Very high speed entropy coding. (1994) 625–629
8. Balevic, A., Rockstroh, L., Tausendfreund, A., Patzelt, S., Goch, G., Simon, S.: Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose gpus. In: Proceedings of 2008 IEEE 11th International Conference on Computational Science and Engineering. (2008)