Floating-Point Data Compression at 75 Gb/s on a GPU

Molly A. O'Neil
Department of Computer Science
Texas State University-San Marcos
moneil@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University-San Marcos
burtscher@txstate.edu

ABSTRACT

Numeric simulations often generate large amounts of data that need to be stored or sent to other compute nodes. This paper investigates whether GPUs are powerful enough to make real-time data compression and decompression possible in such environments, that is, whether they can operate at the 32- or 40-Gb/s throughput of emerging network cards. The fastest parallel CPU-based floating-point data compression algorithm operates below 20 Gb/s on eight Xeon cores, which is significantly slower than the network speed and thus insufficient for compression to be practical in high-end networks. As a remedy, we have created the highly parallel GFC compression algorithm for double-precision floating-point data. This algorithm is specifically designed for GPUs. It compresses at a minimum of 75 Gb/s, decompresses at 90 Gb/s and above, and can therefore improve internode communication throughput on current and upcoming networks by fully saturating the interconnection links with compressed data.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – *Parallel Processors* D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel Programming* E.4 [**Coding and Information Theory**]: Data Compaction and Compression

General Terms

Algorithms, Measurement, Performance, Design, Experimentation

Keywords

GPGPU, Lossless Data Compression, Floating-Point Data, Real-Time Compression

1. INTRODUCTION

Large-scale numeric computations, such as weather forecasting, earthquake simulations, and climate modeling, generally run on clusters of interconnected compute nodes. They generate large amounts of floating-point results that must be transferred to secondary storage or off-site for further analysis and visualization. Even larger data sets are exchanged between the compute nodes of these clusters during simulation.

Most high-end compute clusters and many smaller clusters use InfiniBand or High-Speed Ethernet links for communication. For example, the 2048-core Longhorn system [19] uses 10-Gb/s High-Speed Ethernet links and the state of the art 22,656 core Lonestar cluster [18] has a 40 Gb/s InfiniBand network. Future clusters will undoubtedly be equipped with even faster interconnects. Ethernet cards that support 40 and 100 Gb/s are already available. InfiniBand currently supports 40 Gb/s and projects speeds of 104 Gb/s for 2011 [13].

It is desirable to compress large data sets to reduce their storage requirement and to speed up their transfer. However, employing data compression in high-performance computing environments is only useful if it can match the network's data rate, i.e., if it can be done in real time. In fact, the compression and decompression throughput needs to exceed the network's bandwidth by a factor that is at least equal to the compression ratio to saturate the network with compressed data. To achieve real-time speeds, a very fast compression and decompression algorithm is required, and its implementation must be tailored to the capabilities of the hardware to reach the necessary performance.

Data compression research has resulted in many efficient lossless encoding algorithms. Dictionary-based encoders, such as the Lempel-Ziv family of algorithms, replace input strings with pointers to a dictionary data structure updated by the encoder. Variable-length entropy encoders, such as Huffman and arithmetic coding, assign prefix codes to each input symbol based on its statistical frequency of occurrence. The codes can either be determined statically up front, requiring two passes over the data, or adaptively during compression. Run-length coders store repeated values as a single instance and a count. These encoding methods, as well as transforms such as Burrows-Wheeler, form the basis of common compression utilities, including gzip and bzip2. There are also specialized compression algorithms targeted specifically at floating-point data. These floating-point compressors often employ an approach based on the suppression of leading zeros in the residuals between the input values and their predicted values. Neither existing general-purpose nor floating-point data compressors offer the throughput required to allow real-time encoding at today's high-end network speeds.

Graphics processing units (GPUs) typically have many more processing cores and wider memory busses than conventional CPUs, making them ideal for speeding up programs that exhibit a lot of parallelism, require little synchronization, and access memory in a streaming fashion, as is often the case for large-scale vector- and matrix-based codes [16]. GPUs frequently outperform CPUs on such codes by several factors [8], and their better price/performance, power/performance, and size/performance ratios have led to their use in many of the world's fastest supercomputers [23]. Given the high processing power and memory throughput of GPUs and their growing presence in computing clusters, this paper investigates whether GPUs can be used for floating-point data compression at throughputs above 40 Gb/s (after compression) to match the speeds of next-generation high-end network links.

GPGPU-4, Mar 05-05 2011, Newport Beach, CA, USA

Because their hardware is primarily designed to process blocks of pixels at high speed and with wide parallelism, GPUs differ substantially from conventional CPUs, which can make it difficult to write efficient implementations of non-graphics algorithms. Data compression and decompression are generally serial operations because the processing of every data item depends more or less on all previous data items. These data dependencies make it hard to extract parallelism. Nevertheless, we have been able to design and implement a double-precision floating-point compression algorithm, called GFC, in CUDA that compresses at or above 75 Gb/s and decompresses above 90 Gb/s on a GTX-285, i.e., meets the throughput demands of the currently fastest commodity networks.

This paper makes the following contributions. 1) It describes the GFC compression algorithm, which is specifically designed to map well to GPUs. 2) It explains several important code optimizations that are included in the GFC implementation to make it very efficient. 3) It compares GFC to five other compressors and shows that GFC compresses nearly as well as they do. 4) It demonstrates that GFC is four times faster than the fastest parallel CPU-based compressor. 5) It makes our implementation of GFC available at http://www.cs.txstate.edu/~burtscher/research/GFC/.

The rest of this paper is organized as follows. Section 2 provides an overview of the GFC compression algorithm and describes our GPU implementation thereof. Section 3 summarizes related work. Section 4 presents the evaluation methods. Section 5 discusses the results. Section 6 concludes the paper with a summary.

2. THE GFC ALGORITHM

2.1 Algorithm

The GFC algorithm is a novel lossless compression algorithm for double-precision floating-point data targeted for parallel execution on a GPU. GFC breaks the data dependencies that typically turn compression into a serial operation without (much) loss of compressibility for floating-point data, thus making the algorithm suitable for GPUs that require thousands of parallel activities to unleash their full performance.

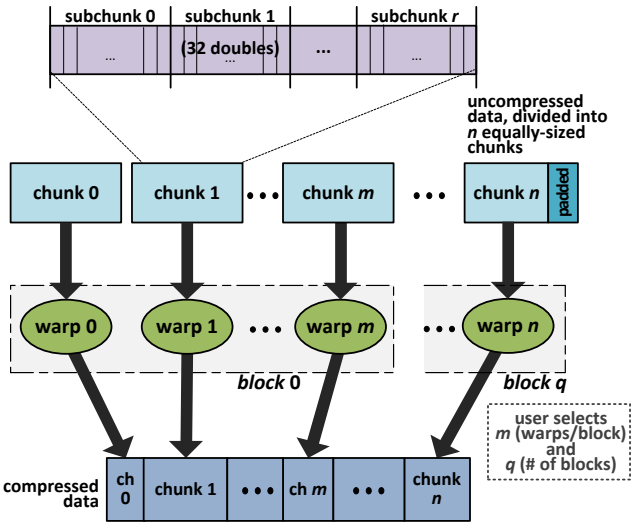

Figure 1. Overview of warp, block, and chunk assignment

GFC divides the data to be compressed into *n* chunks, as shown in Figure 1. The number of chunks is user selectable and should match the warp width of the target GPU for best performance. However, other sizes are also supported, for example, to decompress data that were compressed on a GPU with a different width. An integer multiple of 32 double-precision floating-point values is assigned to each chunk. If necessary, the last chunk is padded with extra values to reach a multiple of 32. The chunk sizes are balanced and never differ by more than 32 doubles.

The chunks are compressed and decompressed independently and can therefore be processed in parallel. Each chunk is split into subchunks of 32 doubles each. Each subchunk is compressed and decompressed using information from the previous subchunk (or all zeros if there is no previous subchunk).

For performance reasons, the 64-bit floating-point values are processed exclusively as 64-bit integers. Hence, GFC does not require any floating-point operations even though it compresses floating-point data. Like the FPC family of floating-point compressors [4], GFC generates a predicted value for each 64-bit word, which is the latest value of the given dimensionality (see below) from the previous subchunk. This predicted value is subtracted from the true value, yielding the residual. The sign bit of the residual is recorded. If the residual is less than zero, it is negated. Thus, if the prediction is accurate, the now positive residual will have many leading zeros. The residual is leading-zero-byte compressed, with the non-zero bytes plus a half-byte storing the sign and leading-zero count comprising the encoded output of the original 64-bit word.

Many scientific data sets interleave values from multiple dimensions. This increases the likelihood that, for an *n*-dimensional data set, every n^{th} value be more likely to yield a reasonable prediction for the residual calculation and leading-zero compression. For instance, the CPU-based pFPC floating-point data compressor achieves better compression ratios when the number of threads is a multiple of the dimensionality of the data set [5]. Hence, GFC takes an optional *dimensionality* parameter between 1 and 32 to improve its performance. The default dimensionality is one.

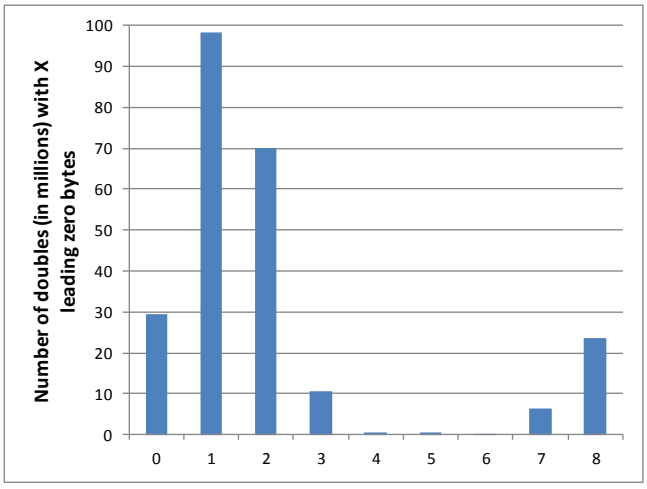

Figure 2. Leading-zero-byte distribution after application of the GFC algorithm to the studied data sets

There are nine possible leading-zero-byte counts, zero through eight. Figure 2 plots the leading-zero-byte counts for our data sets. Because a count of six almost never occurs, we chose to encode six leading zero bytes as though there were only five, which reduces the number of possibilities to eight and allows encoding

with three bits. This does not affect the correctness of the decoder; values where the residual has six leading zero bytes are merely encoded as if they had five leading zero bytes and three bytes of non-zero data. Thus, a 64-bit value is compressed into between zero and eight non-zero residual bytes, a sign bit, and a 3-bit field that specifies the number of leading-zero bytes. The sign bit and the 3-bit count field of each of the 32 values in a subchunk are emitted together, followed by the non-zero bytes of the 32 residuals.

As illustrated in Figure 3, in the best case where all values are correctly predicted, the 32 64-bit values in a subchunk are compressed down to 16 bytes, resulting in an upper bound of 16 for the compression ratio. In the worst case where all residuals have eight non-zero bytes, the "compressed" output is 16 bytes larger per subchunk than the uncompressed input, resulting in a compression ratio of 0.94, i.e., an expansion by 6%.

We evaluated several variations of this algorithm, including calculating the exclusive-or of the predicted and true values rather than the difference as well as choosing the best of 32 prediction values and using a full byte to encode the prediction selection and the leading zero count. We also investigated an algorithm that uses a fixed offset within each subchunk for the prediction value, rather than a fixed dimensionality. The approach described above yielded the best average compression ratio on our data sets.

GFC's output includes a 4-byte count specifying the number of doubles in the original data set, a 1-byte value that records the *dimensionality*, a 2-byte value specifying *n*, and one 4-byte value per chunk specifying the compressed size in bytes. This header information is emitted first and is followed by the compressed data from each chunk.

Decompression first reads the header information to determine how many chunks there are and where the data for each chunk start. Then it decompresses each chunk, one subchunk at a time, by reading the 32 sign and leading-zero-byte-count fields, computing the number of non-zero bytes, extracting the corresponding bytes, zero extending them to form the 64-bit residuals, negating them if the sign bit is set, and adding the resulting values to the appropriate values from the previous subchunk to recreate the original uncompressed values. Any padding values are removed at the end using the total number of doubles stored in the header. Note that the values used for padding are chosen so that they will be perfectly predicted and thus do not expand the compressed data unnecessarily.

2.2 Implementation

The GFC algorithm described above is designed so that it can be implemented efficiently on CUDA-capable GPUs [9], especially those with compute capability 1.3. Such GPUs consist of a number of independent streaming multiprocessors (SMs), each of which contains several tightly coupled execution cores. All SMs have access to the global memory of the GPU, which is separate from the CPU's main memory. The cores can read and write the global memory very efficiently as long as multiple cores simultaneously access adjacent memory locations. In fact, the hardware converts sets of such accesses into one or two *coalesced* memory accesses. Each SM contains a software-managed cache called shared memory that can be accessed in parallel by all cores in the SM as long as the accesses do not cause bank conflicts.

Up to 1024 program threads can be active in an SM at a time. However, sets of 32 threads are grouped into warps that must execute in lockstep, i.e., all 32 threads must execute the same instruction (on different data) in the same cycle. If this is not possible, for example, because some of the threads execute the `then` block of an `if` statement while the remaining threads do not, the hardware disables the divergent threads until they re-converge. Hence, only subsets of the threads in a warp execute concurrently whenever not all threads can execute the same instruction, resulting in loss of parallelism.

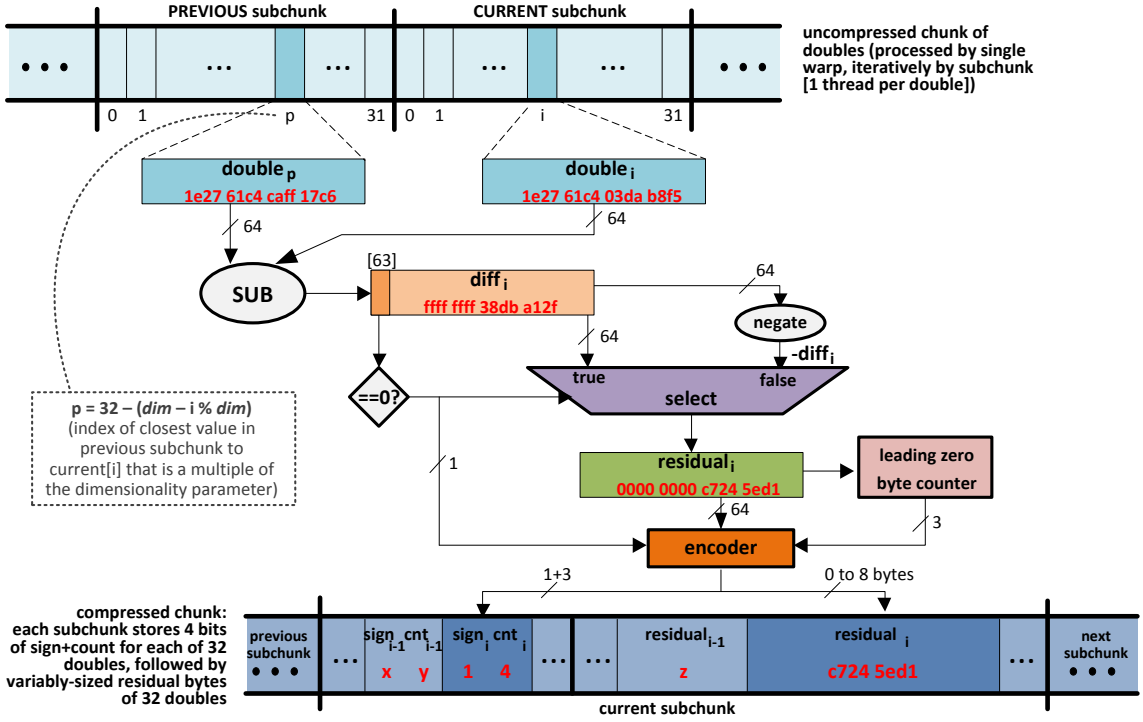

Figure 3. The GFC compression algorithm

To maximally exploit the benefits and avoid the drawbacks of GPUs, GFC includes the following code optimizations and features. To avoid thread divergence, the algorithm is implemented with just a few `if` statements. Several of them are very short and simple so that the compiler can convert them into predicated instructions that do not cause any divergence. One `if` statement in both the compressor and decompressor deliberately disables half of the threads per warp because they are not needed at that point. The remaining few `if` statements are necessary to reduce memory accesses, which is more important for performance in our memory-bound code than is minimizing thread divergence. Another optimization we included is allocating extra space in the shared memory for each warp so that the prefix sums (which are needed so that the threads can access the proper compressed bytes in parallel) can be computed without divergence.

To maximize the memory performance, our implementation accesses global memory in a coalesced fashion whenever possible. In fact, only a single access per subchunk is not coalesced in both the compressor and the decompressor. This degree of coalescing is achieved by packing and unpacking compressed bytes in shared memory. The shared memory accesses, in turn, are performed in a way that minimizes bank conflicts.

To maximally exploit parallelism, we map the chunks, which can be compressed independently, to individual warps across the SMs. In fact, the entire algorithm is implemented in a warp-based manner, meaning that warps never communicate with each other, even the warps that run on the same SM. Each warp iteratively processes the subchunks in its assigned chunk, but the 32 threads in a warp process the 32 values of a subchunk in parallel. The threads communicate via shared memory with the other threads in the same warp to accomplish this, and inter-thread communication within each warp is limited to the prefix sum computation required for access to the variably sized compressed data values. Because the warps are independent, no synchronization is needed other than the implicit global barrier at the end of the kernel that waits for all warps to finish. There is also no synchronization needed among the threads within a warp, even though they communicate, since the hardware forces them to execute in lockstep, i.e., they are always synchronized. Thus, the code includes only a few memory fences to prevent the compiler and hardware from reordering the memory accesses that communicate data between threads during the prefix sum calculation. Otherwise, it is completely synchronization and communication free.

3. RELATED WORK

Floating-point data compression algorithms are divided into lossy and lossless approaches. Audio and images can tolerate some imprecision in data reconstruction. GFC targets the many applications, e.g., scientific programs, where data loss is unacceptable.

There is little prior work on GPU-based lossless data compression. Balevic et al. [3] designed a block-parallel arithmetic coder for post-processing of scientific simulation data directly on the GPU before transfer back to the host. Their work achieved significant storage savings for the resulting compressed data on the GPU and therefore reduced the number of data transfers required. However, they demonstrated that the compression overhead outweighed the resulting timesavings in I/O transfer to the host.

More recently, Balevic [2] presented a GPU-based parallel encoding algorithm using Huffman coding that exploits atomic operations on the GPU's shared memory to enable variable-length codeword writes. This algorithm, which also relies on a parallel prefix scan to compute output positions, resulted in approximately 32 Gb/s performance on a GeForce GTX-280, up to a 35-fold improvement over their serial version running on a 2.66 GHz CPU, but slightly under the throughput of state of the art and emerging networks. Aqrawi and Elster [1] investigated both lossy and lossless compression techniques for seismic data, concluding that the GPU is inferior to the CPU for Huffman compression due to the sequential nature of the algorithm and the GPU's limited bitwise operation capabilities. Their proposed compressor utilizes the GPU for DCT filtering and the CPU for lossless run-length encoding of the transformed data

There exist several GPU implementations of the parallel prefix sum algorithm that we use in our compressor [12].

In the area of image processing, GPUs have been successfully employed for the acceleration of image transforms (e.g., DCT) [1] and texture compression algorithms such as DXT [7]. There also is substantial prior literature exploring the use of GPUs for image compression, including via lossless algorithms such as run-length encoding [15]. Existing work also explores GPU-based decompression, with the intent of reducing storage and bandwidth requirements and allowing for on-the-fly decoding and rendering on the GPU [17].

A larger body of work exists in the area of lossless floating-point data compression on the CPU. Burtscher and Ratanaworabhan presented FPC [4], a lossless compressor for double-precision floating-point data designed for high throughput. GFC derives its leading-zero-count encoding method from the FPC algorithm, although FPC uses an exclusive-or of predicted and true values rather than subtraction. Below, we compare both FPC and its parallel implementation, pFPC [5], to GFC.

4. EVALUATION METHODOLOGY

4.1 Systems and Compilers

We evaluated GFC and compared it to pFPC, a parallel CPU-based compressor for floating-point data, on one compute node of the Longhorn cluster at TACC [19]. The compute node contains two quad-core Intel Xeon E5540 processors running at 2.53 GHz and 48 GB of main memory. Each of the eight cores has a 32 kB L1 data cache and a unified 256 kB L2 cache, and the four cores on a processor share an 8 MB L3 cache. The operating system is TACC's version of x86_64 GNU/Linux 2.6.18. We used the gcc C compiler 4.1.2 with the "-O3 -pthread -std=c99" flags.

The compute node further contains four NVIDIA FX 5800 GPUs, of which we used one. The GPU has 30 streaming multiprocessors with 8 cores running at 1.3 GHz and 4 GB of global memory. Each multiprocessor has 16 kB of software-managed cache and 16,384 registers that are shared among the threads allocated to the multiprocessor. We used the CUDA compiler 3.2 with the "-O3 -arch=sm_13" flags.

4.2 Measurements

All timing measurements are performed by instrumenting the source code, i.e., by adding code to read the timer before and after the measured code section and recording the difference. For both GFC and pFPC, we measure the runtime of the compression and decompression code only, excluding the time it takes to read the input data into an array and, in the case of GFC, to transfer data to and from the GPU. Each experiment was conducted nine times in a row and the median runtime is reported. Averaged compression ratios and throughput numbers refer to the harmonic mean throughout this paper.

4.3 Data Sets

We used the 13 FPC datasets for our evaluation [4], [22]. They include program input (observational data), output (simulation results), and messages exchanged between compute nodes (MPI messages). Table 1 summarizes pertinent information about each dataset. The first two data columns list the size in megabytes and in thousands of subchunks. The middle column shows the percentage of values that are unique. The fourth column displays the first-order entropy of the doubles in bits. The last column expresses the randomness of the datasets in percent, i.e., it reflects how close the first-order entropy is to that of a truly random dataset with the same number of unique 64-bit values.

Table 1. Statistical information about the data sets

		data size (MB)	sub-chunks (1000s)	unique values (%)	1st-order entropy (bits)	random-ness (%)
MPI messages	bt	254.0	1040.6	92.9	23.7	95.1
	lu	185.1	758.3	99.2	24.5	99.8
	sp	276.7	1133.2	98.9	25.0	99.7
	sppm	266.1	1089.8	10.2	11.2	51.6
	sweep3d	119.9	491.1	89.8	23.4	98.6
simulation results	brain	135.3	554.1	94.9	24.0	99.9
	comet	102.4	419.3	88.9	22.0	93.8
	control	152.1	623.1	98.5	24.1	99.6
	plasma	33.5	137.1	0.3	13.7	99.4
observa-tional data	error	59.3	242.8	18.0	17.8	87.2
	info	18.1	73.9	23.9	18.1	94.5
	spitzer	189.0	774.1	5.7	17.4	85.0
	temp	38.1	156.0	100.0	22.3	100.0

5. RESULTS

5.1 Algorithm Comparison

Figure 4 displays the harmonic-mean compression ratio over the 13 data sets for GFC and five compression algorithms from the literature that are run in their fastest mode. The results show that GFC's mean compression ratio is in line with that of other algorithms on these hard-to-compress data sets. Previous work has demonstrated that these compression ratios suffice to substantially speed up message-passing applications if the messages can be compressed in real time [14].

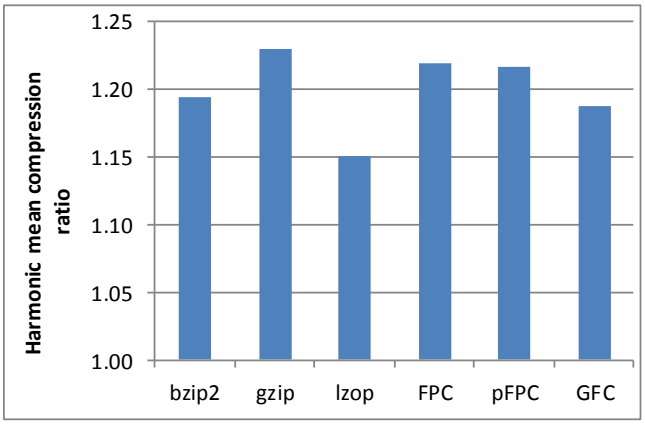

Figure 4. Harmonic-mean compression ratio of six algorithms sorted from slowest (leftmost) to fastest (rightmost)

Bzip2 [6], gzip [11], and lzop [20] are general-purpose compressors that operate at byte granularity. FPC [10], pFPC [21], and GFC are special-purpose algorithms designed to compress double-precision floating-point data. Lzop is the fastest of the studied general-purpose algorithms but achieves the lowest compression ratio. GFC and bzip2 both compress somewhat better than lzop, and gzip, FPC and pFPC achieve noticeably higher compression ratios. FPC has been shown to be one to two orders of magnitude faster than gzip and bzip2 [4]. pFPC is a parallel implementation of FPC and is the fastest CPU-based compression algorithm for floating-point data of which we are aware. Most of these compressors support slower modes in which they compress better. However, even in their fastest mode, all of them except pFPC (discussed next) are over a factor of ten slower than GFC.

Figure 5 plots the compression and decompression throughput of GFC and pFPC against their compression ratio. It shows pFPC results for progressively slower but better compressing modes from left to right. Whereas GFC compresses a little less than pFPC, Figure 5 clearly illustrates that GFC does not merely represent a continuation of pFPC's performance trend but a large improvement. In fact, a single GPU running GFC compresses our data sets 4.5 times faster and decompresses them 4.0 times faster than two Xeon CPUs (8 Nehalem cores) running pFPC.

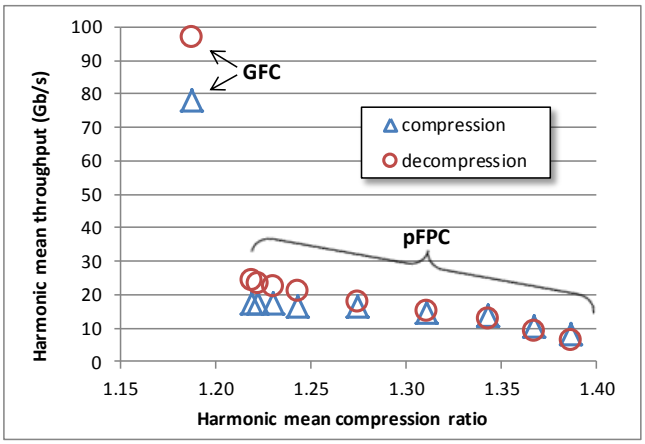

Figure 5. Throughput versus compression ratio of GFC and pFPC with different modes (fastest on left, best compressing on right)

5.2 Data Set Comparison

Figure 6 depicts the compression and decompression throughput of GFC in gigabits per second on the thirteen data sets as well as the harmonic mean. The results show that GFC's throughput is consistently high on all data sets. It compresses them at 75 Gb/s or higher and decompresses them at 90 Gb/s or higher, reaching over 87 Gb/s compression throughput and 121 Gb/s decompression throughput on *sppm*. The harmonic mean is 77.9 Gb/s for compression and 96.6 Gb/s for decompression. The results are largely independent of the data set size. However, the throughputs correlate with the compression ratio (given below). Higher compression ratios increase the throughput because less data have to be accessed.

Table 2 lists the compression ratio of GFC on each data set as well as the harmonic mean. It provides results for four different values of *n*, i.e., different numbers of chunks. Clearly, the number of chunks, and therefore the amount of parallelism, has a neg-

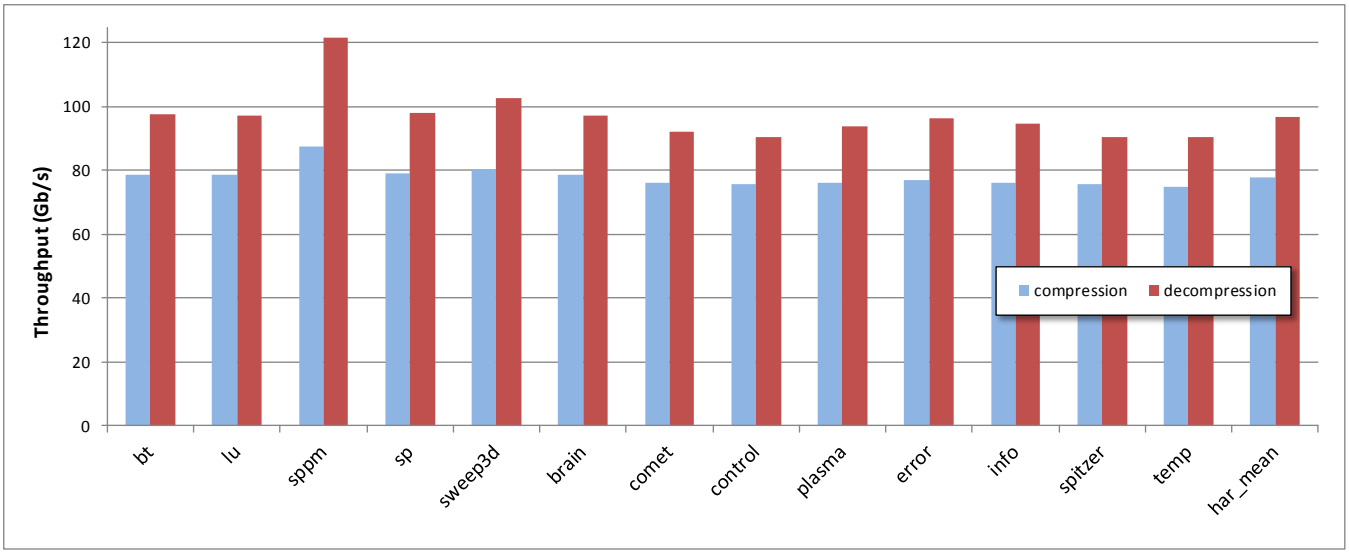

Figure 6. Compression and decompression throughput of GFC

ligible impact on the compression ratio. This means that GFC can easily provide the high levels of parallelism needed to exploit the hardware capabilities of GPUs, at least for data sets that are sufficiently large so that each chunk contains multiple subchunks.

Table 2. GFC's compression ratio as a function of the number of chunks

	1 chunk	10 chunks	100 chunks	1000 chunks
bt	1.20055	1.20055	1.20053	1.20030
lu	1.14857	1.14857	1.14854	1.14821
sppm	3.52841	3.52835	3.52764	3.52059
sp	1.20329	1.20329	1.20326	1.20298
sweep3d	1.21929	1.21929	1.21922	1.21857
brain	1.09112	1.09112	1.09109	1.09079
comet	1.11123	1.11123	1.11121	1.11106
control	1.01327	1.01327	1.01326	1.01314
plasma	1.12984	1.12982	1.12966	1.12805
error	1.23827	1.23825	1.23810	1.23656
info	1.15138	1.15135	1.15101	1.14764
spitzer	1.02263	1.02263	1.02262	1.02257
temp	1.03954	1.03953	1.03946	1.03873
har_mean	1.18803	1.18802	1.18793	1.18710

6. SUMMARY AND CONCLUSIONS

This paper describes and evaluates the GFC compression algorithm for double-precision floating-point data. This algorithm is specifically designed for use on a GPU. It compresses and decompresses at throughputs of over 75 Gb/s on a GTX-285 while still achieving a significant compression ratio on numeric data sets. GFC is over four times faster than the fastest parallel CPU-based compression algorithm, making GFC the first compressor with the potential for providing real-time compression for emerging InfiniBand and Ethernet networks that operate at 40 Gb/s and above. We expect GFC's performance to increase significantly on Fermi-based GPUs, in particular because Fermi cores support an instruction to count the leading zero bits in an integer, which is a frequent operation in the GFC code.

The system we evaluated can transfer data between the CPU and the GPU via the PCIe bus even while the GPU is compressing or decompressing. However, the maximum bandwidth of this bus is 25 Gb/s, rendering the high throughput on the GPU useless for network speeds much above 10 Gb/s. However, the same PCIe bus is used to communicate with the network interface card (NIC), which will have to be made faster to support the next generation of networks. Hence, communication with the GPU should become equally faster. In fact, NVIDIA is working on direct GPU-to-GPU data transfers, which may be extensible to support direct GPU-to-NIC transfers. Moreover, AMD's recent demonstration of the Fusion APU indicates that their CPUs and GPUs will increasingly be on the same chip, and NVIDIA's Tegra products already combine a CPU and a GPU in the same package, thus eliminating the need for explicit data transfers and allowing the full benefit of GFC to be reaped.

Our open-source CUDA implementation of GFC is freely available at http://www.cs.txstate.edu/~burtscher/research/GFC/.

7. ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported in this paper. We further thank NVIDIA Corporation for donating the GPUs that were used to develop the GFC algorithm presented in this paper.

8. REFERENCES

[1] Aqrawi, A. A. and Elster, A. C. 2010. Accelerating disk access using compression for large seismic datasets on modern GPU and CPU. Para 2010 State of the Art in Scientific and Parallel Computing, extended abstract #131.

[2] Balevic, A. 2009. Parallel variable-length encoding on GPGPUs. In *Proceedings of the 2009 International Conference on Parallel Processing*. Euro-Par'09. Springer-Verlag, Berlin, Heidelberg, 26-35.

[3] Balevic, A., Rockstroh, L., Wroblewski, M. and S. Simon. 2008. Using arithmetic coding for reduction of resulting simulation data size on massively parallel GPGPUs. In *Pro-*

ceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer-Verlag, Berlin, Heidelberg, 295-302.

[4] Burtscher, M. and Ratanaworabhan, P. 2009. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput*. 58, 1 (January 2009), 18-31.

[5] Burtscher, M. and Ratanaworabhan, P. 2009. pFPC: A parallel compressor for floating-point data. In *Proceedings of the 2009 Data Compression Conference*. DCC'09. IEEE Computer Society, Washington, DC, 43-52.

[6] Bzip2. Retrieved February 1, 2011 from http://www.bzip.org/.

[7] Castaño, I. 2009. *High Quality DXT Compression using OpenCL for CUDA*. Whitepaper. NVIDIA Corp. Retrieved February 1, 2011 from http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/OpenCL/src/oclDXTCompression/doc/opencl_dxtc.pdf.

[8] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W. and Skadron, K.. 2008. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.* 68, 10 (October 2008), 1370-1380.

[9] CUDA C Programming Guide 3.2. 2010. Retrieved February 1, 2011 from http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[10] FPC 1.1. 2009. Retrieved February 1, 2011 from http://www.csl.cornell.edu/~burtscher/research/FPC/.

[11] Gzip. Retrieved February 1, 2011 from http://www.gzip.org/.

[12] Harris, M., Sengupta, S. and Owens, J. D. 2007. Parallel prefix sum (scan) with CUDA. *NVIDIA GPU Gems 3*. Addison-Wesley Professional, chapter 39.

[13] InfiniBand Trade Association. 2010. Retrieved February 1, 2011 from http://www.infinibandta.org/content/pages.php?pg=press_room_item&rec_id=679.

[14] Ke, J., Burtscher, M. and Speight, E. 2004. Runtime compression of MPI messages to improve the performance and scalability of parallel applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing.* SC'04. IEEE Computer Society, Washington, DC, 59-65.

[15] Lietsch, S. and Marquardt, O. 2008. A CUDA-supported approach to remote rendering. In *Proceedings of the 3rd International Conference on Advances in Visual Computing.* ISVC'07. Springer –Verlag, Berlin, Heidelberg, 724-733.

[16] Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (March 2008), 39-55.

[17] Lindstrom, P. and Cohen, J. D. 2010. On-the-fly decompression and rendering of multiresolution terrain. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and* Games. I3D'10. ACM, New York, NY, 65-73.

[18] Lonestar user guide. Retrieved February 1, 2011 from http://services.tacc.utexas.edu/index.php/lonestar-user-guide.

[19] Longhorn user guide. Retrieved February 1, 2011 from http://services.tacc.utexas.edu/index.php/longhorn-user-guide.

[20] Lzop. Retrieved February 1, 2011 from http://www.lzop.org/.

[21] pFPC v1.0. 2009. Retrieved February 1, 2011 from http://users.ices.utexas.edu/~burtscher/research/pFPC/.

[22] Scientific IEEE 754 64-Bit Double-Precision Floating-Point Datasets. 2009. Retrieved February 1, 2011 from http://www.csl.cornell.edu/~burtscher/research/FPC/datasets.html.

[23] Top500 fastest supercomputers. Retrieved February 1, 2011 from http://www.top500.org/.