International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-4 Number-1 Issue-14 March-2014

Reducing the Disk IO Bandwidth Bottleneck through Fast Floating Point Compression using Accelerators

Ajith Padyana[1], Devi Sudheer[2], Pallav Kumar Baruah[3], Ashok Srinivasan[4]

Abstract

Compute-intensive tasks in high-end high performance computing (HPC) systems often generate large amounts of data, especially floating-point data that need to be transmitted over the network. Although computation speeds are very high, the overall performance of these applications is affected by the data transfer overhead. Moreover, as data sets are growing in size rapidly, bandwidth limitations pose a serious bottleneck in several scientific applications. Fast floating point compression can ameliorate the bandwidth limitations. If data is compressed well, then the amount of data transfer is reduced. This reduction in data transfer time comes at the expense of the increased computation required by compression and decompression. It is important for compression and decompression rates to be greater than the network bandwidth; otherwise, it will be faster to transmit uncompressed data directly [1]. Accelerators such as Graphics Processing Units (GPU) provide much computational power. In this paper, we show that the computational power of GPUs and CellBE processor can be harnessed to provide sufficiently fast compression and decompression for this approach to be effective for data produced by many practical applications. In particularly, we use Holt`s Exponential smoothing algorithm from time series analysis, and encode the difference between its predictions and the actual data. This yields a lossless compression scheme. We show that it can be implemented efficiently on GPUs and CellBE to provide an effective compression scheme for the purpose of saving on data transfer overheads.

The primary contribution of this work lies in demonstrating the potential of floating point compression in reducing the I/O bandwidth bottleneck on modern hardware for important classes of scientific applications.

Manuscript received January 16, 2014.

Ajith Padyana, Department of Mathematics and Computer Scuence, Sathya Sai Institute of Higher Learning, Muddenahalli Campus, India.

Devi Sudheer, IBM IRL, Delhi, India.

Pallav Kumar Baruah, Department of Mathematics And Computer Scuence, Sathya Sai Institute of Higher Learning, Prashanthi Nilayam Campus, Puttaparthi, Ananthpur, India.

Ashok Srinivasan, Department of Computer Science, Florida State University, Florida, USA.

Keywords

Throughput, Compression Ratio, Holt's Exponential Smoothening, CellBE, GPU.

1. Introduction

There has been tremendous improvement in computing power of high end computing system in recent years. But I/O has not been keeping pace with computing power. The gap between computing power and I/O has been growing over the years and it is further increasing due to availability of various accelerators which enhance computations. For example, world's second fastest GPU enabled Titan Supercomputer has a peak performance of 20 petaflops/s (Pflop/s) and a maximum possible I/O bandwidth of 240 GB/s[3]. The ratio of I/O performance to compute performance is even worse in the top supercomputers. The current trend of increase in core-counts are causing applications that were previously not I/O bound to become I/O bound, and the situations is expected to become worse in the near future. If we can use computational power for reducing the data size, then this will reduce the data transfer overhead, leading to increased overall application performance.

In addressing this issue, we keep in mind the following typical computing paradigm. Typical scientific simulations involve a large number of iterations. In each iteration, the state of a physical system, which consists of a large set of floating point numbers, is updated. Every few iterations, the state of the system is output to disk. This data is later analyzed or visualized. For example, until recently NCCS at ORNL housed Jaguar, which was one of the fastest supercomputer in the world, and the Lens GPU cluster is available for data analysis and visualization. Data may also be output for check

pointing purposes, which is increasingly becoming important, in order to provide fault tolerance.
When I/O time becomes a bottleneck, applications either have to run more slowly or to perform I/O less frequently. We propose a technique to compress the data produced by applications to address this I/O bottleneck issue. We use the abundant computing power available to reduce I/O usage, which is a scarce resource. For example, if we obtain a 50% compression ratio, then we are effectively doubling the bandwidth available to applications. However, it is important for the overhead associated with compression and decompression to be small. If we wish to store the data produced for analysis later, then it should be faster to compress the data and transfer it, rather than to transfer the uncompressed data directly. Similarly, it should be faster for the data analysis or visualization application to read the compressed data and uncompressed it, rather than for it to read the uncompressed data directly.

Compression has been proposed in the past as a solution to bandwidth bottlenecks. However, the overhead associated with it has prevented it from being used for floating point data in practical situations. We propose a high throughput compression algorithm and apply it on a set of applications. The compression it achieves is less than that of popular compression algorithms. However, its low computational overhead makes it effective in dealing with various types of I/O bottlenecks for large classes of applications.

Our compression algorithm predicts future data in a data stream based on prior data, using techniques from time series analysis. It then computes some measure of the difference between the predicted value and the actual data. If the prediction is accurate, then we obtain several zeros in the most significant bits. These results can then be encoded efficiently. We give further details below.

Table 1: Data Sets used for evaluation of algorithms

No	Name	Applications
1	Bloweybq	Sparse Matrix
2	Msg_sppm	3-D Hydro Dynamics
3	Andrews	Eigen Value Problem
4	Lp_ken_18	Linear Programming
5	rail4284	Linear Programming
6	para-4	Semiconductor Device
7	2D_27628_bjtcai	Semiconductor Device
8	Ohne2	Semiconductor Device
9	c-58	Non Linear Optimizations
10	Memplus	Memory Circuit
11	apache2	Finite Difference
12	Num_comet	Astro physics simulations
13	a0nsdsil	Lin.Complimentary Problem
14	Msg_bt	CFD
15	Ted_AB_unscaled	FEM: Thermoelasticity
16	msg sweep 3D	Neutron Transport
17	num control	Min. in data assimilation
18	Num_brain	Simulation of Brain impact
19	Msg_sp	CFD
20	Climate	Climate Modelling

Let the data set to be stored at some stage in the computation be given by $S = (x_1, x_2, ..., x_n)$. For example, this may represent the weather conditions at all grid points, at a certain point in time, in a climate modeling application. We use time series analysis techniques to predict x_i after seeing the first *i-1* components of the vector. Even though the dimensions are usually spatially, rather than temporally, related, time series analysis can be used. Our ability to predict depends on short range correlations that are often present in scientific applications. For example, the weather conditions at a certain grid point in a weather forecasting applications may be somewhat close to that at nearby points. We hope to be able to predict the sign, exponent and the most significant bits of the mantissa, while the least significant bits are hard to predict. If p_i is the predicted value, then we compute an exclusive-OR of the bit-patterns of the floating point numbers representing x_i and p_i and encode the result *losslessly*.

We target the implementation details of our technique to GPUs, which are used as accelerators on the Forge system, for instance. With the abundance of computing power provided by accelerators, such as GPUs and Cell Processors, we hope that our work will further stimulate research in high throughput floating point compression techniques to ameliorate bandwidth bottlenecks. The outline of the rest of the paper is as follows. We next describe our compression algorithm in next section. We then describe our data sets and experimental setup different sections respectively. We present our optimizations and empirical evaluation results in next section. We summarize related work in next section and present our conclusions.

2. Time Series Based Floating point Compression Algorithm

We use the Holt exponential smoothing algorithm for time-series prediction. This is an extension of exponential smoothing that includes a term for linear trends. It is also known as double exponential smoothing. Assume that at time t we have observed y_t and estimated the level (a smoothed average) L_t and the slope b_t in the series. Then a k-step ahead forecast is $F_{t+k} = L_t + b_{t+k}$. Holt`s method allows the estimates of level and slope to be adjusted with each new observation. The updating equations for L_t and b_t from L_{t-1}, $b_{t-1}$and y_t are:

$$Lt = \alpha yt + (1 - \alpha)(Lt - 1 + bt - 1) \quad (1)$$

$$bt = \beta(Lt - Lt - 1) + (1 - \beta)bt - 1 \quad (2)$$

for given α and β in [0,1].

To start the process, both L_1 and b_1 must be specified. Possible starting values are $L_1 = y_1$ and $b_1 = y_2 - y_1$. Thus, no forecasts can be made until y_1 and y_2 have been observed. By convention, we let $F_1 = y_1$.

We next describe the application of Holt`s linear exponential smoothening to the compression of the floating point numbers. Algorithm 1 describes the steps for fast floating point compression using Holt`s exponential smoothening. Decompression can be performed in a similar manner.This algorithm compresses linear sequences of single precision floating point numbers. It predicts the next floating point number, which is taken to be zero initially. As part of prediction, it uses series of previous values to predict the current value as given in equation 1 and 2. Once the prediction is done, the original number is XOR-ed with the predicted value. The XOR operation turns identical bits into zeros. Hence, if the prediction is accurate, the XOR would result in many leading zero bits. The compression algorithm then determines the number of leading zero bytes and encodes the count in a two-bit value. The resulting two-bit code and the non-zero remainder bytes are written to the compressed stream. The latter are emitted without any form of encoding. In figure 1 we have shown the format for storing compressed data.

Algorithm 1 holt‘s Linear exponential for floating pointcompression

```
1.Input: float* Array, int N
2. int *A = (int *) Array
3. int Level = Input[0], Slope= Input[1] - Input[0]
4. float Alpha = Beta = 0
5. Define A[-1] = 0
6. for i = 0 to N-1 do
7. Pred = Level[i] + Slope[i]
8. X = A[i] XOR Pred
9. Data[i] = trailing non-zero bytes of X
10. Code[i]= code for number of trailing non zero bytes of X
11. Level[i] =α*Input[i] + (1-α)(Level[i-1] + Slope[i-1])
12. Slope[i] = β*(Level[i] - Level[i-1]) + (1-β)Slope[i-1]
13. end for
```

Algorithm 2 holt‘s Linear exponential for floating point decompression

```
1. Input: Compression representation of Data Code
2. int Pred = 0
3. int Level = Input[0], Slope= Inp  ut[1] - Input[0]
4. float Alpha = Beta = 0
5. Define A[-1] = 0
6. for i = 0 to N-1 do
7. Recover X from Compact representation of Data and Code
8. X = A[i] XOR Pred
9. Level[i] =α*Input[i] + (1-α)(Level[i-1] + Slope[i-1])
10. Slope[i] = β*(Level[i] - Level [i-1]) + (1-β)Slope[i-1]
11. Pred = Level[i] + Slope[i]
12. end for
13. Store Code and Data compactly.
```

Decompression works as follows. It starts by reading the two-bit header. Then the number of non-zero bytes specified by the two-bit header is read and zero extended to a full 32-bit number. Then the series of previous values and the predictor are updated using the operations in reverse to that used in compression phase. This lossless reconstruction is possible because XOR is a reversible operation. The algorithm interprets all the 32-bit floats as integers and uses integer arithmetic in order to perform the XOR operations.

3. Data Used For Compression Algorithm

The data sets used are summarized in table 1 They are from the University of Florida sparse matrix collection [5] Many of these sparse matrices are from real parallel applications. The original data were all in double precision. We converted them to single precision numbers. Sparse matrices are not a natural fit for our hardware [6], and so they provide a good test for the effectiveness of our implementation.

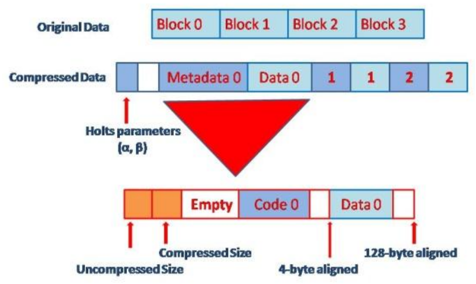

Figure 1: Schematic of Data Encoding

4. GPU Implementation

Compression Phase:
In this phase, the floating-point data is read from a file, sent to the GPU device for compression, and the compressed data is written back to another file. After the uncompressed data is read from a file into an array, the total floating point number count is calculated and this value is divided by ***FloatsPerBlock*.**
NumBlocks = TotalNumFloats/Floats Per Block

The implementation of compression steps are given below:

- Copy Raw Data to Device
- Data Partition
- Perform Compression in Shared Memory
- Write Code and Compressed Data to Global Memory
- Copy Compressed Data back to Host

Decompression Phase
In this phase, the compressed data is read from a file, sent to the device for decompression and the decompressed data is written to another file. The steps involved in this decompression phase are given below:

- Find Block Offsets from Metadata
- Copy Compressed Data to Device
- Copy Compressed Data to Shared Memory
- Perform Decompression in Shared Memory
- Write Decompressed Data to Global Memory
- Copy Decompressed Data back to Host

5. Evaluation of Holt's Compression Algorithm

First we present the compression ratios (ratio of compressed size to original size). It can be seen in Table 2 that the scheme yields good compression for some data-sets but poor compression for a few others. The reason for this is that we achieve good prediction only for data sets which are predictable. This happens for certain types of applications, such as hydrodynamics on a uniform mesh (\#2), certain sparse matrices (\#1), Linear Programming (\#5), etc. For applications with random data (\#19), the data is not predictable with the simple scheme that we are using. In any case, there are applications classes for which we get significant compression.

Comparison of Optimization Levels
Implementation is classified into following Optimization Levels:

- *Level 0:* No Optimization
- *Level 1:* Use of Double Buffering
- *Level 2:* Increased 115 floats for each thread
- *Level 3:* Increased computation threads to 64.

The above optimization levels are not inclusive of each other, although such an inclusion would increase performance significantly, due to want of shared memory. Padding compressed sizes for 4-byte aligned, avoiding bank conflicts and increasing number of threads for memory transfers between shared and global memory has been implemented in levels 1,2 and 3.

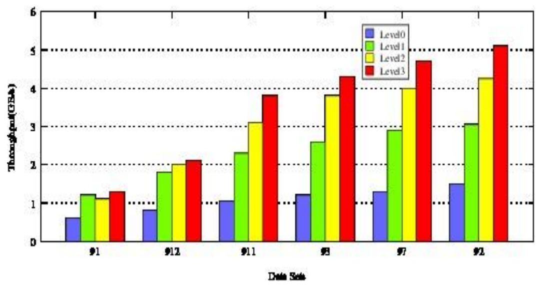

Figure 2: Compression Throughput comparisons for the four optimization levels

Compression speed depends on the compression ratio achieved; greater the compression of data, fewer are the data that need to be transferred from global memory to shared memory. Compression/ Decompression throughput achieved without optimizations was inadequate, yielding a performance of less than 2 GBps. But after tuning the code for performance, a maximum throughput of 5.2 GBps could be achieved.

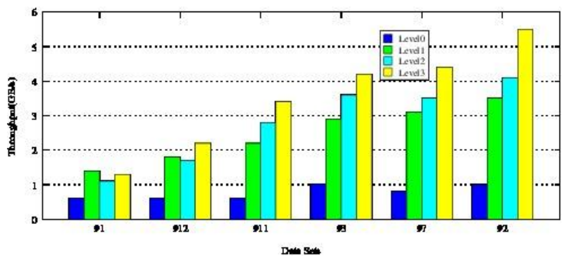

Figure 3: Decompression Throughput comparisons for the four optimization levels

Figure 4 and 5 shows compression and decompression throughputs for data sets \#1, \#10, \#9 \#3, \#6 and \#2. The data sets have been chosen with varying sizes and fairly good compression ratios. The data sets have been plotted with increasing sizes. The throughput shown in Figure 6 only reflects the computation time and does not include the time taken for data transfers between host and device.

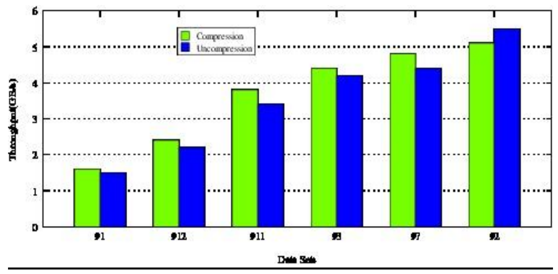

Figure 4: Compression/Decompression throughput comparisons optimization Level 3.

As can be seen from Figure 6 the throughput increases as the sizes of data sets increase. The reason for this is that the number of threads created is directly proportional to data size. GPU power is fully extracted with thousands of threads running in parallel. More the number of threads, greater is the utilization of the GPU memory bandwidth and greater is the performance obtained. With smaller data sizes, only a few threads are created to compress them in order to avoid increased data fragmentation which results in sub-optimal throughputs.A comparison is also made between compression and decompression throughput for optimization Level-4. Compression fairs slightly better as this phase has been optimized more than the decompression phase. Apart from the 64 threads which are busy with computation, more threads are added to improve transfers between shared and global memory. Testing for various block size i.e. number of threads per block, it has been observed that the compression phase works best with 128 threads per block and decompression phase with 192 threads per block. Register usage by threads (i.e. number of variables per thread), warp scheduling and usage of local memory and shared memory determine throughputs achieved with different block sizes.

Time Distribution

In the results, throughput is calculated only with respect to compression speeds. If the GPU is used only to compress or decompress data of an application running on the CPU, then the data first needs to be transferred to the GPU and results need to be copied back to the host. This transfer is limited to 8 GBps peak by the PCIe bus. Although the transfer speed is quoted as 8 GBps, average transfer speed obtained is only around 5 GBps. Compared to computation time, data transfer time is quite high. However, applications running on systems with accelerators typically use accelerators to handle the bulk of the computation. So, data that needs to be computed will typically already be present on the GPU, and does not need to be transferred from the host. Similarly, in decompression, the uncompressed data may not need to be sent back to the host.As can be seen from the Figures 5 and 6, time taken for host to device transfer is lesser than for the reverse case. This is due to two reasons. Since block synchronization is not possible in CUDA [5] each block writes its compressed data to fixed locations in the global memory. The entire global array where compressed data is stored are transferred and written back to host. The size of this array is larger than the input data as it has been allocated for the worst case scenario of zero compression.

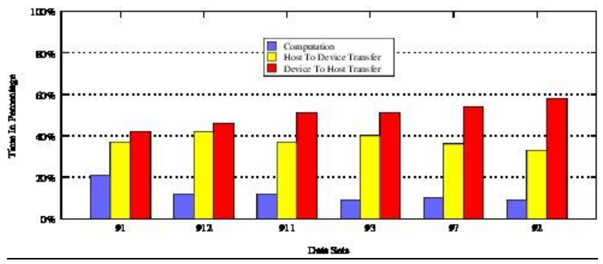

Figure 5: Time distribution in Compression phase

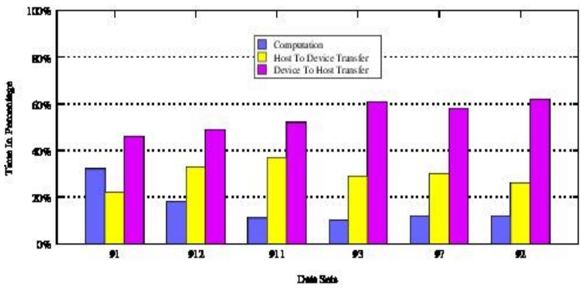

Figure 6: Time distribution in Uncompression phase

In the decompression phase, the time taken to transfer data from host to device is obviously lesser as only compressed data is transferred whereas uncompressed data is transferred from device to host.

Comparison with Cell BE

An IBM Cell BE processor comprises of 1 PPU and 8 SPUs, where the PPU offloads the computation to CPUs to be done in parallel. For SPUs to compute on data, the PPU needs to DMA this data over a bandwidth of 25 GBps. The output is DMAed back to the PPU by the SPU. The PPU is a fully functional processor and so can host an entire OS on it, making the Cell BE an independent device.

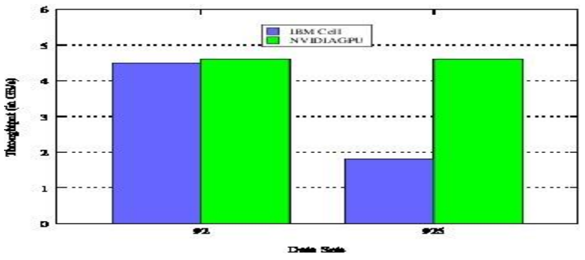

Figure 7: Compression throughput in Cell and GPU

The structure of the GPU however is different. The cores of the GPUs are little more than functional units and contain very little logical units. Therefore, a GPU can be used only as a co-processor to the CPU and cannot be used to run programs independently without CPU collaboration. The CPU in this case can be compared to the PPU in Cell which offloads computation to the GPU. The data needs to be transferred from host to device for computation. The GPU however has an extra level of indirection. After transferring data from host RAM to device RAM (global memory), the data then again needs to be transferred to the shared memory of the multi-processors for computation. This extra transfer can be compared to the transfer between PPU and SPU in Cell discounting the DMA overheads in Cell. However, this transfer is not mandatory but greatly improves performance. The bandwidth within the GPU is 177 GBps (Tesla M2050) which can be compared to the bandwidth of 204.8 GBps (8 SPUs X 25.6 GBps) between PPUs and SPUs.

Figures 7 and 8 throughput of compression and decompression over two extreme data sets. Data fragmentation is lesser with fewer threads.

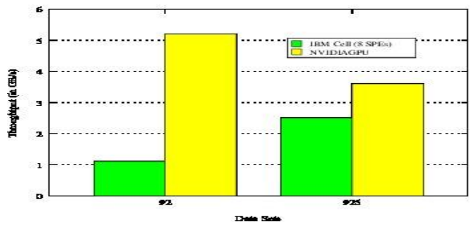

Figure 8: Uncompression throughput in Cell and GPU

The extra overhead of transfer between host and device is not considered for this comparison. Although this comparison cannot be termed completely fair, it does provides a rough estimation. We can also see from these figures that the throughput achieved in both compression phase and decompression in GPU are better than that of CellBE processor.

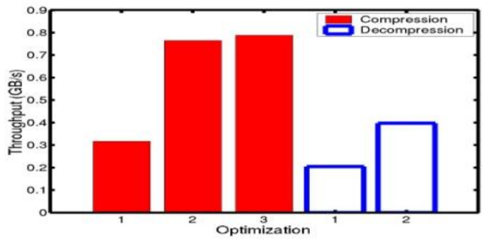

Figure 9: Optimization of the computational phase of compression and decompression. (1) Basic code, (2) with loop unrolling, and (3) vectorized.

6. Implementation of Floating Point Compression on a Single SPE

In this section, we discuss implementation and optimization of the algorithm on a single SPE. However, one important aspect of the implementation is motivated by the need to parallelize it later – we divide the data into blocks of 3968 floats each, and compress each block independently. This will enable the SPEs in a parallel implementation to work on different blocks independently. The block size was chosen such that the data for a block can be brought in through a single DMA, and the compressed data can be written back in a single DMA, even in the worst case.

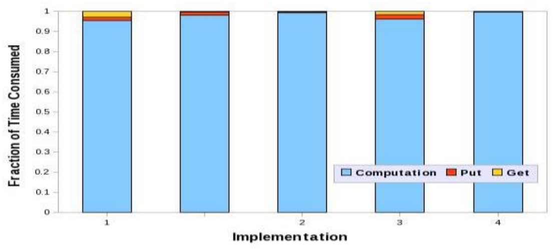

Figure 10: Fraction of time spent in different phases of the algorithm (bottom: computation, middle: put, top: get). (1) Compression, single buffering, (2) Compression, multi-buffering, (3) Compression with multi-buffering and infrequent synchronization, (4) Decompression, single buffering, (5) Decompression, multi-buffering.

This decomposition into blocks is illustrated in fig. 1. The compressed data contains a sequence of pairs, where each pair is the metadata for a block followed by compressed data for that block. The metadata first contains the sizes of the compressed and uncompressed data, followed by a few bytes reserved for possible use later. This is followed by the code for all the data in the block. A few empty bytes may follow, in order to ensure that the compressed data is aligned along integer boundaries. Up to 127 extra bytes may be allocated at the end of the compressed data section of each block, in order to ensure 128-byte alignment of the next block. The space overhead of this is less than one percent of the original data size. Such alignment is useful because it enables DMAs between main memory and SPEs to be 128 byte aligned.

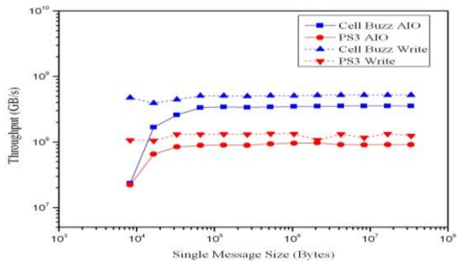

Figure 11: **Throughput to disk using asynchronous (aioWrite) and synchronous IO.**

We observed a significant improvement in performance for DMA puts that are 128 byte aligned compared with 16 byte aligned. Therefore, the marginal increase in space is acceptable.

Compression is fairly straight forward. First, we wish to find suitable parameters _ and _. We use a small fraction of the data and compress it with different values of the parameters to find suitable values. Different SPEs work simultaneously on different values of the parameters, in order to complete the search fast, when we use multiple SPEs. We search in the range [0.95, 1.05], because the optimal is usually in this range. The overhead associated with finding the parameters is around 10% of the compression time. We choose the parameter values that gave best compress ratio, and use it in the actual compression phase. The selected values of the two parameters are written at the beginning of the compressed file.

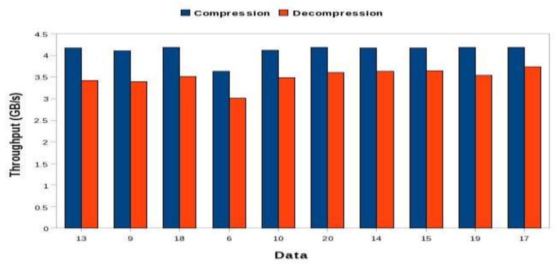

Figure 12: **Total throughput of compression and decompression on 16 SPEs. The data are arranged in decreasing order of compression achieved.**

In the main compression computation, the SPE brings in data, compresses it, and writes it back to main memory. Decompression is a little more involved, because the location and size of compressed data for each block is unknown. The PPE first quickly computes an index for the starting location for each block, using the metadata entry specifying *compressed data size*. (The time taken is less than 2% of the total time in a parallel computation, and an even smaller fraction with one SPE.) The SPE gets this index and uses it to determine the starting location for each block. We allocate two buffers, capable of storing 2048 entries each, for the index on the SPE. This is not sufficient for large data sizes. So the SPE needs to occasionally bring in a new index block. Two buffers are used in order to enable double buffering. So, the SPE rarely needs to wait much for an index to arrive. The location and size of a block can be determined from the index. Once the compressed data is received, the SPE uses the *uncompressed size* field in the metadata to determine the amount of data to decompress.

Figure 9 shows the impact of optimizations on the performance of the computational phases of compression and decompression. The timings obtained are almost identical for all data samples, and do not depend on the compression obtained. The basic compression and decompression code had somewhat modest performance. We manually unrolled the loop eight times, and after this, the compiler was able to improve the performance substantially, as shown in the second bars for compression and for decompression. We also vectorized the compression code by hand, but obtained only a slight improvement in performance. We, therefore, did not vectorize the decompression code by hand.

We next compare the effect of multiple buffering for the input and output data. The code was initially single buffered. That is, before the computational phase for each block, we would request the corresponding data, wait for it, and then compute. We would then put the compressed data back to main memory, and block until that DMA was completed. Figure 10 show that the DMAs consumed a significant fraction of the total time. We next multi-buffered the code, so that data for three input buffers could simultaneously be in flight, and data for four output buffers could simultaneously be in flight. The large number of buffers was primarily to deal with large variance in DMA times in an earlier version of the code [2]. We fixed that problem, related to paging issues, so that the variance is no longer high. Figure 10 shows that the DMA Costs in the multi-buffered code are insignificant. An earlier version of the code had yet inefficiency. After compression of each block, the SPE would send the PPE and mailbox message to inform it that a block had been compressed. The PPE can then save that compressed data to disk. This permits a pipelined compress and store scheme, which overlaps compression and IO. However, the mailbox overhead is somewhat significant even on one SPE, and increases when several SPEs are used. So, instead we have an SPE send a mailbox message only every C iterations, for a suitably large value of C (which depends on data size too). In fact, each SPE does not send a mailbox message. Instead, they synchronize after compressing every C block using our efficient barrier implementation with an overhead of at most 1μs with 16 SPEs, and one SPE sends a mailbox message. This makes the synchronization overhead insignificant, as shown in figure 10.

7. Parallel Implementation on Multiple SPEs

We now describe the parallel implementation, evaluate its performance, and discuss the implications of the results for the four types of bandwidth bottlenecks that we mentioned earlier.

Implementation

In the parallel implementation, each SPE independently compresses and decompresses blocks of data. The blocks are assigned to the SPEs in a block cyclic manner, with a parameter C to the code specifying the number of adjacent blocks that an SPE will handle. For example, if C is 2 and 8 SPEs are involved in the computation, then SPE 0 will get blocks 0, 1, 16, 17, 24, 25, · · ·. A cyclic distribution is important because we are assuming a pipelined scheme. The PPE should transmit compressed data continuously. If the data were partitioned into P pieces on P SPEs, then the early work of SPEs with high ranks would not be used until the end,which is not conducive to pipelining. Each SPE is assigned a different portion of main memory to write all of its output. The amount of memory allocated for each SPE's data is sufficient to hold all its compressed data in the worst case. So SPEs can work independently. The PPE takes data from each SPEs output location cyclically and transmits them, C blocks at a time.

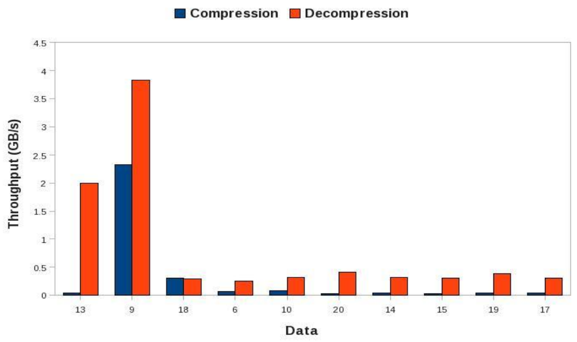

Figure 13: **Throughput of just the compression phase of minigzip, for the case where it gives fastest compression.**

We use a block size C to be greater than one for the following reason, apart from the need for reducing synchronization cost, as mentioned earlier. Figure 11 shows the IO performance in writing 32 MB data to a file, writing a block at a time. We can see that the data sent in each call needs to be around 100 KB for getting close to optimal bandwidth. This is much larger than the the size of a single block. We need to choose C such that the data size for C blocks will normally yields good throughput on transmission. On the other hand, for smaller data files, this may provide enough parallelism. So, we choose smaller values of C for smaller files.

Evaluation

We first present compression results and then performance results. We can see that minigzip performs better than our compression scheme. We get good prediction fairly frequently in certain types of applications, such as hydrodynamics on a uniform mesh (#18), Finite element (#9), etc. For applications with somewhat random data, such as temperature error (#11), the data is not predictable with the simple scheme that we are using. In any case, there are application classes for which we get significant compression. This translates to a corresponding decrease in the time taken in the transmission step.

Table 2: Compression obtained by the Holt's compression algorithm and by minigzip.

Matrix	Original Size(Bytes)	Compressed Size using Holt	Compressed Size using minigzip
1	159744	0.57	0.002
2	503808	0.89	0.14
3	718848	0.49	0.30
4	798720	0.50	0.24
5	1179648	0.66	0.38
6	1638400	0.43	0.05
7	1771520	0.86	0.45
8	9465264	0.73	0.87
9	11065344	0.18	0.006
10	17544800	0.6	0.61
11	31080408	1.03	0.69
12	44254180	0.88	0.56
13	45136128	0.07	0.001
14	62865612	0.79	0.86
15	70920000	0.84	0.89
16	79752372	1.00	0.92
17	99090432	1.02	0.86
18	139497932	0.36	0.10
19	145052928	0.94	0.83
20	149845548	0.68	0.68

We present performance results for compression and decompression using Holt's algorithm in fig. 12, excluding the disk IO cost. The speed of compression and decompression is fairly consistent across all the data, and does not depend on the compression obtained. There is a slight dependence on file size, especially for decompression, with smaller files yielding slightly lower throughputs. We show the speed of minigzip in fig. 13. The timing is for just the compression or decompression phase, excluding thread creation, IO, etc. The throughput varies a lot, but is consistently low; that is, it would be faster to transmit uncompressed data directly, rather than to compress and send it. (Of course, if the goal were to save on storage space, rather than to deal with bandwidth bottlenecks, then minigzip would be a better choice than our scheme.)

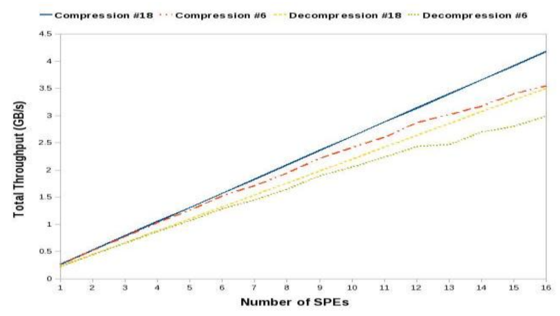

Figure 14: **Scalability of compression and Decompression.**

We finally show scalability with the number of SPEs in fig. 14, using two different data, one with large (# 18) the other small (# 6). As expected compression and decompression scale well, but the larger data scales better than the smaller one.

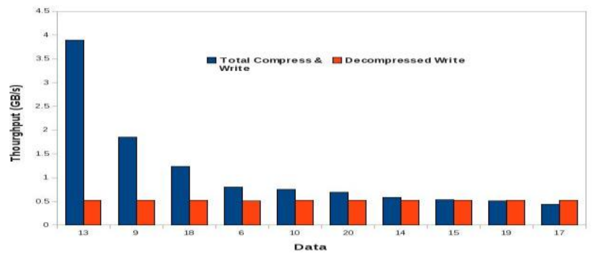

Figure 15: **Compression Throughput Holts Algorithm.**

We now present the total throughput, for compression and disk IO, in 15. This is ultimately the quantity of interest to us, because we are considering this as an alternative to writing uncompressed data. The throughput is primarily limited by disk IO speed, which in turn is primarily determined by the compression achieved. Generally, when the compressed size is less than 80% of the original size, compression is useful for reducing IO cost. Even in these cases, we can see that compression does not decrease the performance by a large amount; there is just a small overhead associated with this. Results for reading from disk and decompressing follow a similar trend.

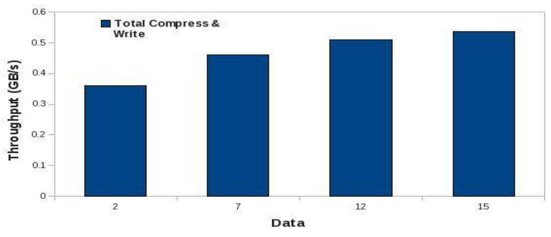

Figure 16: **Impact of file size on compression throughput. The data are organized in increasing order of file size, and compress to a similar extent.**

The throughput for the compressed writes is influenced by the file size for small files, as seen in figure 16. The reason for this is that our pipelined implementation ideally needs several disk IOs of the order of 100 KB for the writing to disk to be efficient. Small data files do not have enough data for this purpose. On the other hand, disk IO may not be as much of the bottleneck for small files.

Potential for dealing with other bandwidth bottlenecks

We have shown above that fast floating point compression can help deal with disk IO bandwidth bottlenecks. Bandwidth bottlenecks arise due to other factors, such as memory bandwidth, LAN bandwidth, and WAN bandwidth limitations.With a pipelined scheme for data transfer and compression or decompression, it compression will be useful for applications that compression well if the compression and decompression speeds are faster than the bandwidth that is a bottleneck.The compression and decompression throughputs are in the range of 3-4 GB/s on 16 SPEs and 1-1.5 GB/s on 6 SPEs (as in a Playstation). The memory bandwidth is around 25 GB/s on the Cell and the order of 10 GB/s on many other platforms. So, compression is not likely to be effective in dealing with this bottleneck, unless faster algorithms or implementations are developed.

We now discuss dealing with network bandwidth limitations in MPI applications. Here, a computation producing data would send the data to another Cell processor. The sender would send compressed data and the receiver would decompress the received data. The network bandwidth depends on the network used.Quad data rate Infiniband, which is not widely deployed, has theoretical bandwidth of 40 Gb/s or, equivalently, 8 GB/s. Compression would not be useful there. However, on most commonly deployed networks, the actual delivered bandwidth will be sufficient for compression to be useful. For example, the measured MPI bandwidth between two nodes on the CellBuzz cluster is 60 MB/s, while the bandwidth between two PS3s at SSSU is around 11 MB/s.

We finally consider sending data over a WAN. There can be a wide difference in performance obtained on a WAN. But this performance is typically much lower than that on a fast network, such as Infiniband. For example, data transfer between the CellBuzz cluster at Georgia Tech and FSU obtains bandwidth around 2 MB/s. Thus compression can be effective.

8. Related Work

There has been much work performed on floating point compression. Many of these are based on predicting the next value based on previous values, and then compressing the result of the difference in the bit patterns of the predicted and actual values. Many of the schemes differ in how the prediction is made and a few other implementation details.
Engelson, et. al.[7]use extrapolation to predict the next value. The FCM scheme uses certain bits of previously observed values to predict the next value. The DFCM [9] is similar, except that it predicts the difference in values, rather than the values themselves. The FPC algorithm [8] uses a combination of FCM and DFCM. It considers both and uses the better choice. It uses one bit of the code to store the choice used.
A preliminary version of this work was reported in [2] as an extended abstract and poster. The algorithm considered there can be considered a simplified version of the Holt`s algorithm, with the parameters fixed as 1. There are also implementation differences which yield better performances now.

9. Conclusion And Future Work

We have investigated the effectiveness of the high throughput compression in improving the I/O

bandwidth limitations. Various applications can benefit from this time series based method for compressing the data. Our approach would be effective in addressing the network bandwidth limitations too since our compression and decompression speeds are a few giga bytes per second. We would like to evaluate other time series method to determine their potential for compression.
In the future, we plan to improve our algorithm using various I/O optimizations. We can spawn another kernel to pack the compressed data on the global memory before the transfer is made to the host. Even if there is a marginal increase in computation time, for a data set with significant compression, the increase in total throughput would be a lot better. In order to mitigate the transfer time between host and device, instead of transferring the entire data before the compression actually begins, a streamed approach could be taken using *cudaStreams* which allows the possibility of overlapping transfer and computation of two different streams.

This approach will also be useful in cases where an application produces a data stream and compression can begin even before the entire data stream is generated. Since it is possible to use multiple GPUs together, each assigned to a different host thread, the implementation can be extended to use multiple GPUs which would enhance the performance greatly.

References

[1] Ajith Padyana, Sudheer, P. K Baruah, Ashok Srinivasan, High Throughput Compression of floating point number in GPUs, In Proceeding of the 2nd IEEE International Conference on Parallel, Distributed and Grid Computing Conference, Dec, 2012.
[2] Ajith Padyana, T.V. Siva Kumar and P.K. Baruah, Fast Floating Point Compression on the Cell BE Processor, In Proceedings of the 15th IEEE International Conference on High Performance Computing (HiPC), 2008.
[3] https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/.
[4] University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.
[5] developer.download.nvidia.com/.
[6] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging MulticorePlatforms, Supercomputing (SC), 2007.
[7] V.Engelson, D.Fritzson and P.Fritzon. Loseless Compression of highvolume numerical data from simulations. In Proceedings of the IEEE Data Compressions Conference (DCC), pages 574-586,2000.
[8] M.Burtscher and P.Ratanaworabhan. High throughput compression of double-precision floating point data. In Proceedings of the IEEE Data Compression Conference (DCC), 2007.
[9] P Ratanaworabhan, J .Ke and M . Burtscher .Fast Loseless Compression of scientific floating point data. In IEEE Data Compression Conference (DCC) pages 133-142, 2006.

Ajith Padyana received his B.Sc (Hons) Mathematics, M.Sc (Mathematics) degrees from Sri Sathya Sai Institute of Higher Learning in 2005 and 2007 respectively. He also completed his M.Tech. Currently he is pursuing PhD and he is an assitant professor in Department of Mathematics and Computer Science, SSSIHL, Muddenahalli, India. His research interests include High Performance Computing, Parallel Programming.

C D Sudheer Kumar received his B.Tech degree. He received his M.Tech degree from Sri Sathya Sai Institute of Higher Learning in 2006. He completed his P.hd in SSSIHL in 2013. Currently he is research scientist in IBM, IRL, Delhi. His research interests include High Performance Computing, Parallel Programming, Programming for Performance, MPI, OpenMP.

Pallav K Baruah received his B.Sc(Hons) Mathematics, M.Sc(Mathematics) degrees from Sri Satya Sai Institute of Higher Learning. Later he went on to do his PhD and received his doctorate in x. Currently he is a professor in Department of Mathematics and Computer Science, SSSIHL, AP, India. His research interests include High Performance Computing, Bioinformatics, Parallel Programming, Boundary Value Problems.

Ashok Srinivasan received his Bachelor's Degree from REC, Tiruchirapalli in the year 1987. Then he went on complete his master in polymer engineering in the University of Akron. Then he completed his PhD in University of California, Santa Barbara in the year 1996. Currently he is working as Associate professor in Florida State University, USA. His research Interests are HPC, Mathematical Software Scalable algorithms.