Fast Floating Point Compression on the Cell BE Processor

Ajith Padyana*, T.V. Siva Kumar, P.K.Baruah
Sri Satya Sai University
Prasanthi Nilayam - 515134 Andhra Pradhesh, India
ajith.padyana@gmail.com, tvsivakumar@gmail.com, baruahpk@gmail.com

Abstract

Bandwidth limitations are a bottleneck in several applications. These limitations may be (i) memory bandwidth, especially on multi core processors (ii) network bandwidth in MPI applications (iii) Bandwidth to disk in I/O intensive applications or (iv) WAN bandwidth in remote sensing applications that send observational data to a central site. Fast compression of floating point numbers can ameliorate bandwidth limitations. Here, it is crucial for the speeds of compression and decompression to be greater than the bandwidth. Otherwise, it will be faster to send the data directly. In this paper, we investigate the effectiveness of a simple Stride Based Compression Algorithm to deal with this problem, on the Cell BE processor. We find that our approach is not fast enough for dealing with the memory bandwidth limitations. However, it can be effective in dealing with the other three limitations.

1 Introduction

The Cell processor contains a PowerPC core, called the PPE, and eight co-processors called SPEs. The SPEs are meant to handle the bulk of the computation bottleneck. They have a combined peak speed of 204.8 GFlops/s in single precision. Each SPE is capable of sending and receiving at 25.6 GB/s.

However, access to main memory is limited by 25.6 GB/s total. If all 8 SPEs access main memory simultaneously, then each sustains bandwidth of just 3.2 GB/s which is much less than 25.6 GB/s in each direction.

In I/O intensive applications, the bandwidth to disk is often a bottleneck, because disk access is relatively slow. WAN bandwidth limits can pose a problem in a different class of applications - when observational or experimental data such as from a satellite, are sent to a remote location for storage or analysis.

The above bandwidth limitations can be ameliorated by sending compressed data, because this reduces the amount of data being transmitted. It incurs an additional computations overhead of compression and decompression. However, with the abundant computation power in emerging architectures, such as cell and GPUs, it appears an attractive alternative .

The aim of this paper is to evaluate the potential of this approach on the Cell Processor. The speed of compression and decompression is critical here, because sending the data directly will be preferable if the overhead of compression is high. We, therefore, use an algorithm that provides fast compression, even though the compression achieved is less than that of popular compression algorithms.

We restrict our focus to compression of 32 bit floating point numbers, because single precision computations is much faster on the current generation of the cell processor.

We have used a very simple algorithm, which sacrifices some compression for speed. In this, it predicts the next value based on recently ob-

*Student Author: Ajith Padyana

served values. If the prediction is fairly accurate, then an exclusive-OR of it with the observed values which yield several zeros in most significant bits. These can be compressed efficiently.

There are two methods for implementation of any compression algorithm (i) Synchronous, where all the data is first produced, then compressed and transmitted, and finally decompressed. (ii) Pipelined implementation, where the data is continuously produced, transmitted and decompressed. We evaluate our approach assuming a pipelined scheme.

We evaluate our scheme on a variety of data, such as sparse matrices from University of Florida collection[2], MPI messages from NAS Parallel benchmark[1], observational data from satellites[1] etc. The Compression achieved by our scheme is consistently worse than that of minigzip. However, our approach yields a throughput of a few GB/s, while the performance of minigzip is an order or two in magnitude lower. This makes our approach effective in dealing with network and disk bandwidth limitations, for those applications where we get significant compression. The low speed of minigzip makes if infeasible for the bandwidth problem that we are addressing.

2 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. [3] use extrapolation to predict the next value. The FCM scheme uses certain bits of previous observed values to predict the next value. The DFCM [6] is similar, except that it predicts the difference in values, rather than the values themselves. The FPC algorithm [1] 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. The scheme we have used can be considered a simple special case of any of the above schemes.

3 Stride-Based Floating Point Compression Algorithm

We first describe a simple Stride based algorithm for compression and decompression and then later mention some modifications to make efficient. The input to the compression algorithm is an array of floats, which we treat as an array of integers. We first try to predict the next, say $i + 1^{th}$, data. This is done assuming that the difference between the $i + 1^{th}$ and the i^{th} data is the same as the difference between the i^{th} data and $i - 1^{th}$ data. These differences are called Strides. The predicted value is exclusive-OR with the actual value. If the most significant bits agree, then we will obtain several leading 0 bits. We wish to store the most significant nonzero bit and all the bits less significant than that bit. We will store the code that tells us how many bits were zero. In order to reduce the size of the code , we can coarse grain this compression by discarding the leading zero bytes, rather than leading zero bits. The decompression algorithm reverse the process. The algorithm is given in the figure for both compression and uncompression.

1. Input : float * Array, Int N
2. int * A = (int *) Array
3. int Stride = 0
4. define A[-1] = 0
5. loop i=0 to N-1
   a) Predicted = A[i-1] + stride
   b) X = A[i] XOR Predicted
   c) Data[i] = trailing non –zero bytes of X
   d) Code[i] = code for number of trailing non zero bytes of X
   e) Stride = A[i] – A[i-1]
6. Store code and Data compactly

Figure 1: Simple Stride based algorithm for Compression

1. Input : Compact representation of data Code
2. int Predicted = 0
3. define A[-1] = 0
4. loop i=0 to N-1
   a) Recover X from Compact representation of Data and Code
   b) X = A[i] XOR Predicted
   c) Stride = A[i] – A[i-1]
   d) Predicted = A[i] + Stride
5. Store code and Data compactly

Figure 2: Simple Stride based algorithm for Uncompression

There are two modifications we make to the above procedure. The first is changing the stride used for prediction. Note that the prediction for each iteration depends on the result of the previous iterations. This data dependence makes vectorization more difficult. So, we have not used vectors in both compression and uncompression. A stride of four improves the compression achieved. Note that 2 bit code can have four possibilities. However, the number of leading zero bytes can have five possibilities : 0-4. we have taken different possibilities : 0, 2, 3, 4 and 0, 1, 2, 3 and we have noticed that i = 3 is very unlikely and in five cases did i=1 and i=2 make a major difference. We have to take care of storing the 2 bit code for each floating point number.

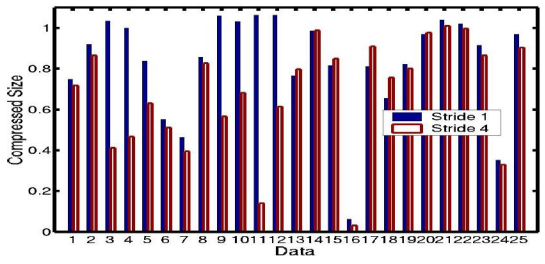

Figure 3: Theoretical compressed size (as a function of the original size) for different strides

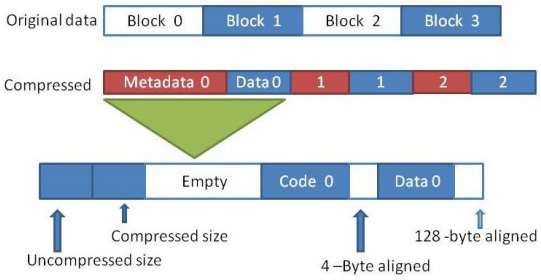

Figure 4: Schematic of data encoding

4 Implementation of Floating Point Compression

Given the data, divide the data into blocks, and compress independently. This will enable the SPE's in a parallel implementation to work on different block independently. Block is chosen such that the data for a block can be brought in single DMA that is the data should be atmost 16KB, and the compressed data can be written back in a single DMA, even in the worst case.

As shown in the figure, Compressed data contains a sequence of pairs, where each pair is the metadata followed by the compressed data for that block. Metadata contains size 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 that block. Followed by aligning the compressed data. There is a significant improvement in performance for DMA puts that are 128 byte aligned compared with 16 byte aligned.

Compression is straight forward. The SPE gets data, compresses it, and writes it back to main memory. Decompression is little more involved, because the location and size of compressed data for each block is unknown. The PPE quickly computes for the starting location of each block using the metadata entries specifying compressed data sizes. SPE gets this index and uses it to determine the starting location for each block. Once the compressed data is received, the SPE uses the uncompressed data size field in the metadata to determine the amount

of data to decompress.

In the parallel implementation, each SPE independently compresses and decompresses blocks of data. The blocks are assigned to SPEs in cyclic manner. For example, if we take 8 SPEs and taking 2 blocks at a time i.e. C = 2, where C is code specifying the number of adjacent blocks that an SPE can handle, then SPE 0 will get blocks 0, 1, 16, 17, 24, 25 and so on. Cyclic distribution is important because we are assuming a pipelined scheme.

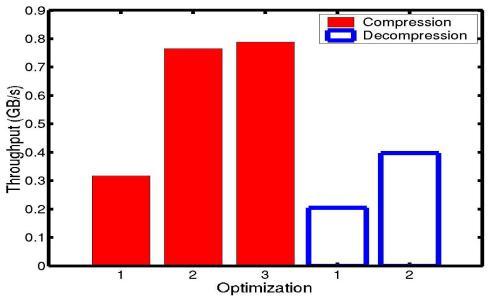

Figure 5: Optimization of the computational phase of Compression and Decompression

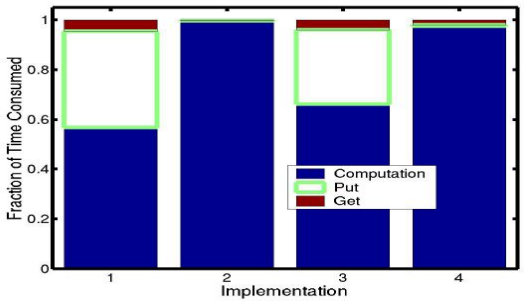

Figure 6: Fraction of time spent in different phases of the implementation (1) Compression, single buffering, (2) Compression, multi buffering, (3) Decompression, single buffering, (4) Decompression, multi buffering

5 Optimization

We manually unrolled the loop four times, we could observe the compiler was able to improve the performance substantially. 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 multi buffering. The code was initially single buffered. We would then put the compressed data back to main memory, and block until that DMA was completed. The DMA *get* meet our expectations. However, DMA *put* times are much greater as shown in the figure .

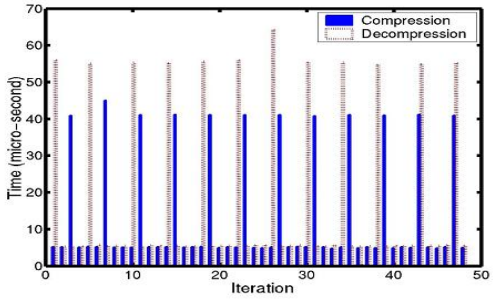

Figure 7: Sample data showing how the cost of DMA *puts* varies in the application, for data of size 16KB

6 Conclusions and Future Work

We have investigated the effectiveness of fast floating point compression in ameliorating different types of bandwidth limitations. Our result indicate that this proposal that this approach would not be effective in dealing with main memory bandwidth limitations.

In future work, we wish to improve the compression algorithm, so that we obtain good compression on a greater variety of applications. We will use time series analysis to perform the better prediction. Now, most of the time taken is for storing the compressed and uncompressed data rather than in the actual prediction. So, we can afford to use the algorithm with better prediction. It will also be useful to evaluate its effectiveness in other types of architectures, such as

GPUs, and also on the latest generations of cell blades.

7 Acknowledgments

We thank IBM for providing access to a Cell blade, and to the Cell center for competence at Georgia Tech. Most of all, we express our gratitude to Sri Sathya Sai Baba, The Chancellor of Sri Sathya Sai University, for bringing all of us together to perform this work.

References

[1] M.Burtscher and P.Ratanaworabhan. High throughput compression of double-precision floating point data. In Proceedings of the IEEE Data Compression Conference (DCC),2007

[2] T.A.Davis .The University of florida sparse matrix collection .Technical report, University of Florida,2007

[3] V.Engelson, D.Fritzson and P.Fritzon. Loseless Compression of high-volume numerical data from simulations. In Proceedings of the IEEE Data Compressions Conference (DCC), pages 574-586,2000

[4] J. Ke, M . Burtscher, and E . Speight. Runtime Compression of mpi messages to improve the performance and scalability of parallel applications. In Proceedings of SC2004, pages 59-65 ,2004

[5] M.Kistler, M.Perrone, and F.Petrini. Cell multiprocessor communication network : Built for speed. IEEE Micro,26:10-23,2006

[6] 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

[7] M.K Velamati, A.Kumar, N.Jayam, G .Senthil Kumar, P.K. Baruah, S Kappor, R Sharma and A Srinivasan . Optimization of collective communication in Intra Cell MPI . In Proceedings of the 14th IEEE International Conference on High Performance Computing (HiPC), pages 488-499,2007.