High Throughput Compression of Floating Point Numbers on Graphical Processing Units

Ajith Padyana, Pallav Kumar Baruah, Devi Sudheer Kumar
Department of Mathematics and Computer Science
Sri Sathya Sai Institute Of Higher Learning
Prashanthi Nilayam, Puttaparthi, 515134
Email: ajith.padyana@gmail.com, baruahpk@gmail.com,
sudheer3@gmail.com

Ashok. Srinivasan
Department of Computer Science
Florida State University
Tallhassee FL 32306 USA
Email: asriniva@cs.fsu.edu

***Abstract*—In scientific computing applications, with a cluster of nodes working in tandem to perform simulations. These compute intensive tasks, often generate large amounts of data, especially floating-point data that need to be transmitted over the network. Although computation speeds may be high, the overall performance of these applications is affected by the transfer of data. Many CPU cycles are wasted waiting for data to arrive. Moreover, as data sets are constantly growing in size at an exponential rate, bandwidth limitations pose a serious bottleneck in several scientific applications. Limitations in network bandwidth, disk I/O bandwidth and WAN bandwidth are of prime concern.**

However, these limitations can be addressed by fast compression. With a good compression ratio, transferring of compressed data greatly reduces limitations induced by bandwidth. It is imperative however, that the speeds of compression and decompression be greater than the bandwidth, else it will be faster to transmit the data directly without compression. A Graphics Processing Unit (GPU) is a massively parallel co-processor capable of extremely fast data parallel computation. holt‘s Linear Exponential algorithm is a simple prediction based algorithm, which can be used for lossless prediction of floating-point data. To deal with the bandwidth problems, in this work, the effectiveness of holt‘s Linear Exponential Smoothing Algorithm for floating point compression with an implementation on an NVIDIA GPU is investigated.

I. Introduction

There has been tremendous improvement in computing power in high end computing system for over last few years. But I/O has not been keeping pace with computing power. The gap between the computing power and I/O is growing over the years and it is further increasing due to availability of various accelerators which are used for computations. If we can use computation power for reducing the data size it will result in effective increase of overall performance of the systems which is bottlenecked by I/O. For example, in FORGE GPU enabled supercomputing system has peak performance of 153 teraflops/s (Tflop/s) and the maximum possible I/O bandwidth is 16 GB/s [11]. The ratio of computation to I/O is increasing in rapid rate. The total bandwidth on the ROAD RUNNER which was until recently ranked #1 on the top500 supercomputer list, is about a 40 times more, at 432 GB/s. But, the total compute performance is about 100 times faster at 1456.7 Tflop/s.

The increasing core counts and the introduction of accelerators like Cell Processors, GPUs, FPGAs made many applications I/O bound which were not so before. The situations is expected to become worse in the near future. The balance between computation and I/O further reduced because of these introduction of multi-core processors, accelerators.

The obvious alternatives to overcome the I/O bottleneck is either to run application 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 by using abundant existing computing power. For example, if we obtain 50% of compression ratio, then effectively we are doubling the bandwidth available to the applications. But we have to assure that overhead associated with this compression is small, so that it is faster to compress data and store it, rather than to store the uncompressed data directly.

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 various kind of I/O bottleneck for large set of applications.

Our approach is based on prediction. It predicts the next value, based on previous observed values. If the prediction is accurate, then an exclusive-OR of predicted value with the original value is stored. It yields several zeros in most significant bits. We have applied time series based prediction algorithm called holt‘s linear exponential smoothing [3] for our floating point compression.

The outline of the rest of the paper is as follows. In § II we summarize important architectural features of the GPU which are relevant to this work. We then describe our compression algorithm in § III. We then have a section on data sets and experimental design in § IV and § V respectively.§ VI explain about our optimizations and evaluations. We summarize related work in § VII and present our conclusions in § VIII

II. GPU Architecture

Driven by the market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multi threaded, many core processor with tremendous computational horse power and very high memory bandwidth. NVIDIA is the first company to produce GPU. The GPU takes a large load of processing away from CPU freeing up cycles.Graphical processing units are built with integrated transform, lighting, triangle, set-up/clipping, and rendering engines. Rendering engines are capable of handling millions of math-intensive processes per second which are represented in floating point numbers. The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute intensive and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. Not all applications can be ported onto the GPU. Only such applications where data parallelism can be achieved are going to give good result.

A. Programming Difficulty

GPU is a challenging environment for software development, due to its radical departure from general-purpose processor design. It also entails a huge learning curve for writing new applications, using CUDA(Compute Unified Device Architecture). There are hierarchy of memories that are given by the CUDA architecture and utilizing the memory closer to the processors is the challenging task in CUDA programming. Keeping the data that is frequently accessed in the shared memory is the difficult part in programming . Since the memory bandwidth is limited, the computational power of GPU should be used optimally to obtain the good performance.

B. CUDA Programming Model

In November 2006, NVIDIA introduced CUDA(Compute unified device architecture) , a general purpose parallel Computing architecture with a new parallel programming model and instruction set architecture that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language. Other languages or application programming interfaces will be supported in the future, such as FORTRAN, C++, OpenCL, and DirectX Compute. CUDAs parallel programming model is designed to maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions a hierarchy of thread groups, shared memories, and barrier synchronization that are simply exposed to the programmer as a minimal set of language extensions.

III. Time Series Based Floating point Compression Algorithm

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 you have observed y_t and estimated the level L_t and the slope b_t in the series. Then a k-step ahead forecast is $F_{t+k)}$ = L_t + b_{tk}. Holts 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:

$$L_t = \alpha y_t + (1-\alpha)(L_{t-1} + b_{t-1}) \quad (1)$$

$$b_t = \beta * (L_t - L_{t-1}) + (1-\beta) * b_{t-1} \quad (2)$$

for constants α and β between 0 and 1, which must be specified

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.

Now let us see about how Holt‘s linear exponential smoothening can be used for the compression of the floating point numbers. Algorithm 1 and 2 describes the steps for fast floating point compression and decompression using holt‘s exponential smoothening.

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. This prediction method categorizes the algorithm as Holt‘s exponential smoothening prediction algorithm. This is nothing but time series prediction. Once the prediction is done, the original number is XORed 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 counts the number of leading zero bytes and encodes the count in a two-bit value. The resulting two-bit code and the nonzero remainder bytes are written to the compressed stream. The latter are emitted without any form of encoding. In figure 1, we have shown how compressed data is stored.

Algorithm 1 holt‘s Linear exponential for floating point compression

```
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] = Alpha*Input[i] + (1-Alpha)(Level[i-1] + Slope[i-1])
12.     Slope[i] = Beta*(Level[i] - Level[i-1]) + (1-Beta)Slope[i-1]
13. end for
```

Decompression works as follows. It starts by reading the two-bit header. Then the number of nonzero bytes specified

Algorithm 2 holt‘s Linear exponential for floating point uncompression

```
1. Input: Compression representation of Data Code
2. int Pred = 0
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.     Recover X from Compact representation of Data and Code
8.     X = A[i] XOR Pred
9.     Level[i] = Alpha*Input[i] + (1-Alpha)(Level[i-1] + Slope[i-1])
10.    Slope[i] = Beta*(Level[i] - Level[i-1]) + (1-Beta)Slope[i-1]
11.    Pred = Level[i] + Slope[i]
12. end for
13. Store Code and Data compactly.
```

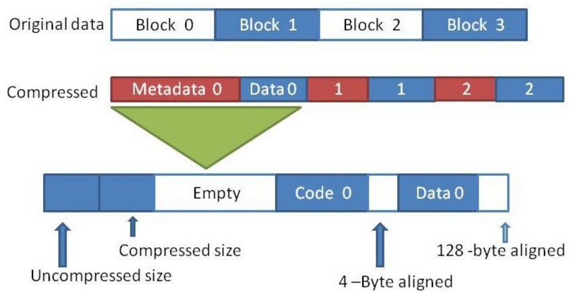

Fig. 1. Constituents of Compressed Data

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. For performing XOR operation reasons, the algorithm interprets all the 32-bit floats as integers and uses integer arithmetic.

IV. Data Used For Compression Algorithm

The data sets used are summarized in table I. The original data were all in double precision. We converted them to single precision numbers.

In order to test the algorithms we considered the sparse matrices, which are available at sparse matrix collection from University of Florida [5]. The main reason behind considering this data set is that, many of the parallel algorithms that work on the sparse matrices consider the matrices from this collection. Also parallel sparse matrix linear algebra packages are not capable of utilizing the potential of a processor for the type of operations that were performed and the type of communication that they are involved in [7].

TABLE I
The 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	lp_truss	
7	para-4	Semiconductor Device
8	2D_27628_bjtcai	Semiconductor Device
9	3D_28984_Tetra	
10	Ohne2	Semiconductor Device
11	c-58	Non Linear Optimizations
12	memplus	Memory Circuit
13	Hamrle3	
14	G3_Circuit	
15	ibm_matrix	
16	apache2	Finite Difference:
17	add20	
18	num_comet	Astro physics simulations
19	a0nsdsil	Lin. Complimentary Problem
20	msg_bt	CFD
21	ted_AB_unscaled	FEM: Thermoelasticity
22	msg_sweep_3D	Neutron Transport
23	num_control	Min. in data assimilation
24	num_brain	Simulation of Brain impact
25	msg_sp	CFD

V. Implementation in GPU

A. Compression Phase

In this phase, the floating-point data is read from a file, sent to the 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 float count is calculated and this value is divided by *FloatsPerBlock*.

$NumBlocks = TotalNumFloats/FloatsPerBlock$

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

B. 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

TABLE II
COMPRESSION RATIO FOR EACH DATA SETS

No	Orig.Size(Bytes)	Compressed Ratio
1	159744	0.44
2	139497932	0.47
3	1638400	0.56
4	1431552	0.56
5	45136128	0.56
6	110592	0.72
7	21304912	0.81
8	1771520	0.83
9	2396160	0.85
10	44254180	0.86
11	1179648	0.91
12	513808	0.90
13	22056968	0.90
14	18491392	0.89
15	4225024	0.92
16	11065344	0.94
17	67584	0.96
18	53673984	0.98
19	798720	1.002
20	133194716	1.009
21	2088960	1.01
22	62865612	1.04
23	79752372	1.05
24	70920000	1.06
25	145052928	1.06

VI. EVALUATION OF HOLT'S COMPRESSION ALGORITHM

First we present the Compression Ratio results. It can be seen in Table II, the scheme yields good compression for some data-sets while poor compression ratios 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), Sparse Matrices (#1), Linear Programming (#5), etc. For applications with random data (#25), 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.

A. 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.

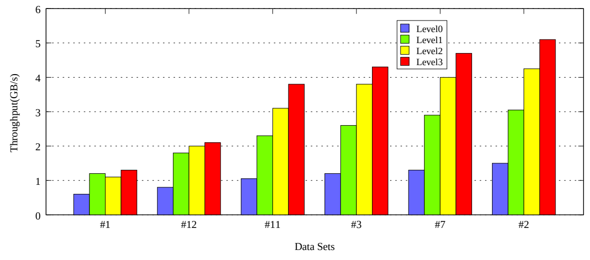

Fig. 2. Compression Throughput comparisons for the four optimization levels

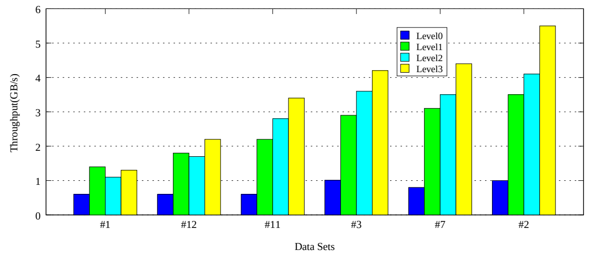

Fig. 3. Decompression Throughput comparisons for the four optimization levels

Compression speed depends on the compression ratio achieved, Greater the compression of data, lesser is the data that needs to be transferred from global memory to shared memory. Compression/Decompression throughput achieved without optimizations was suboptimal 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 2 and 3shows compression and decompression throughputs for data sets #1, #12, #11 #3, #7 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 4 only reflects the computation time and does not include the time taken for data transfers between host and device.

As can be seen from Figure 4, the throughput goes on increasing as the size 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 and more is the performance obtained. With smaller data size, only a few threads are created to compress them in order to avoid increased data

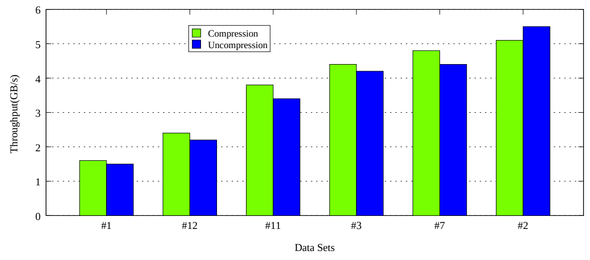

Fig. 4. Compression/Decompression throughput comparisons optimization Level 3

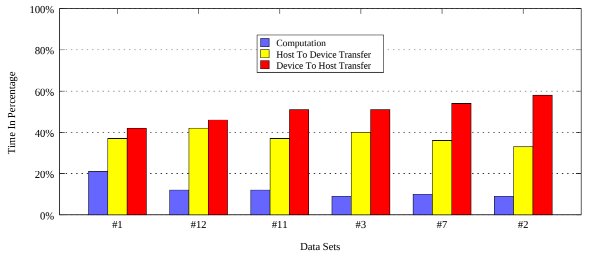

Fig. 5. Time distribution in Compression phase

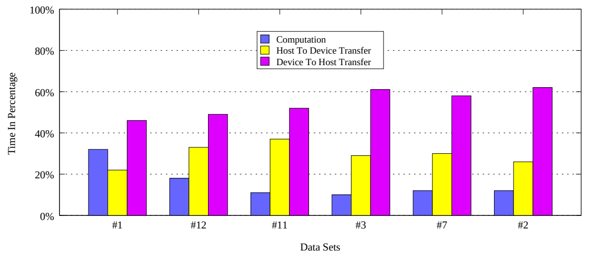

Fig. 6. Time distribution in Decompression phase

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 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.

B. *Time Distribution*

In the results, throughput is calculated only with respect to compression speeds. For any compression to be done by the GPU, data first needs to be transferred and results need to be copied back. 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. Therefore, for greater data size the transfer time between host and device increases. Compared to computation time, transfer time is quite high.

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 [6], 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.

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.

C. *Comparison with Cell BE*

An IBM Cell BE processor comprises of 1 PPU and 8 SPUs, where the PPU offloads the computation to SPUs 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.

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 203 GBps (8 SPUs X 25 GBps) between PPUs and SPUs.

Figures 7 to 9 are comparison of compression ratio and throughput of compression and decompression over two extreme data sets. One which gives a good compression ratio (#2) and another with a poor compression ratio (#25). Compression ratio achieved in Cell is better than the one achieved in GPU. Data fragmentation is lesser with fewer threads. 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.

VII. 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. [8] use extrapolation to predict the next value. The FCM scheme uses certain bits of previous observed values to predict the next value. The DFCM [10] is similar, except that it predicts the difference in values, rather than the values themselves. The FPC algorithm [9] 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.

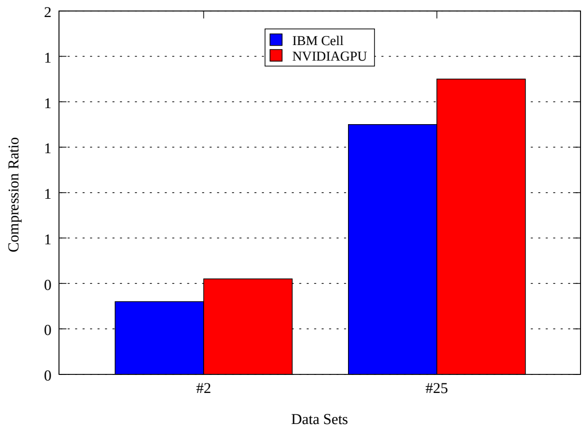

Fig. 7. Compression Ratio in Cell And GPUs

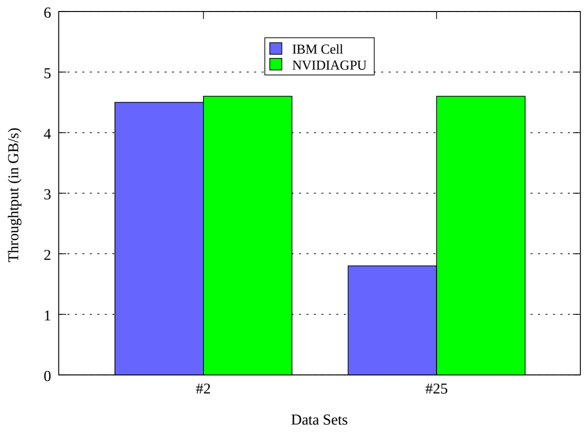

Fig. 8. Compression throughputs in Cell and GPU

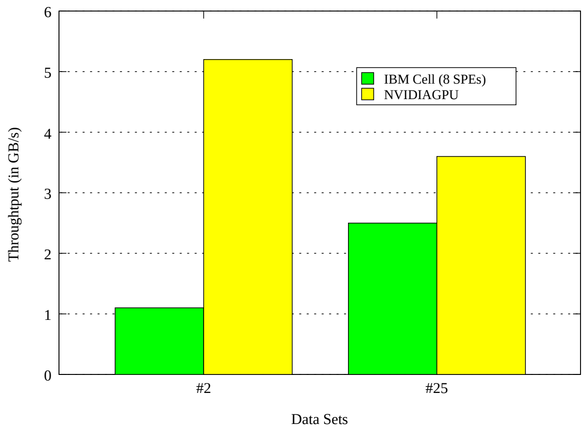

Fig. 9. Decompression throughputs in Cell and GPU

The scheme we have used can be considered a simple special case of any of the above schemes.

VIII. Conclusion and Future Work

We have investigated the effectiveness of the high throughput compression in improving the I/O bandwidth limitations. Various applications are benefiting 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 being in few giga bytes per second. We would like to evaluate many other time series method which would best fit for the compression technique.

In future, we would like to improve our algorithm using various I/O optimizations. We would like to spawn another kernel which could be implemented 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 would also be useful in cases where an application produces a data stream and compression could 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 could be extended to use multiple GPUs which would enhance the performance greatly.

References

[1] http://www.nvidia.com/object/io1258360868914.html

[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] http://www.personal.soton.ac.uk/rchc/Teaching/MATH6011/Forecasting Chap3.htm

[4] http://www.ncsa.illinois.edu/News/11/0518NCSAinstalling.html

[5] University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/

[6] developer.download.nvidia.com/

[7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, Supercomputing (SC), 2007.

[8] 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

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

[10] 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

[11] http://www.ncsa.illinois.edu/News/11/0518NCSAinstalling.html